CS 202 Fundamental Structures of Computer Science II

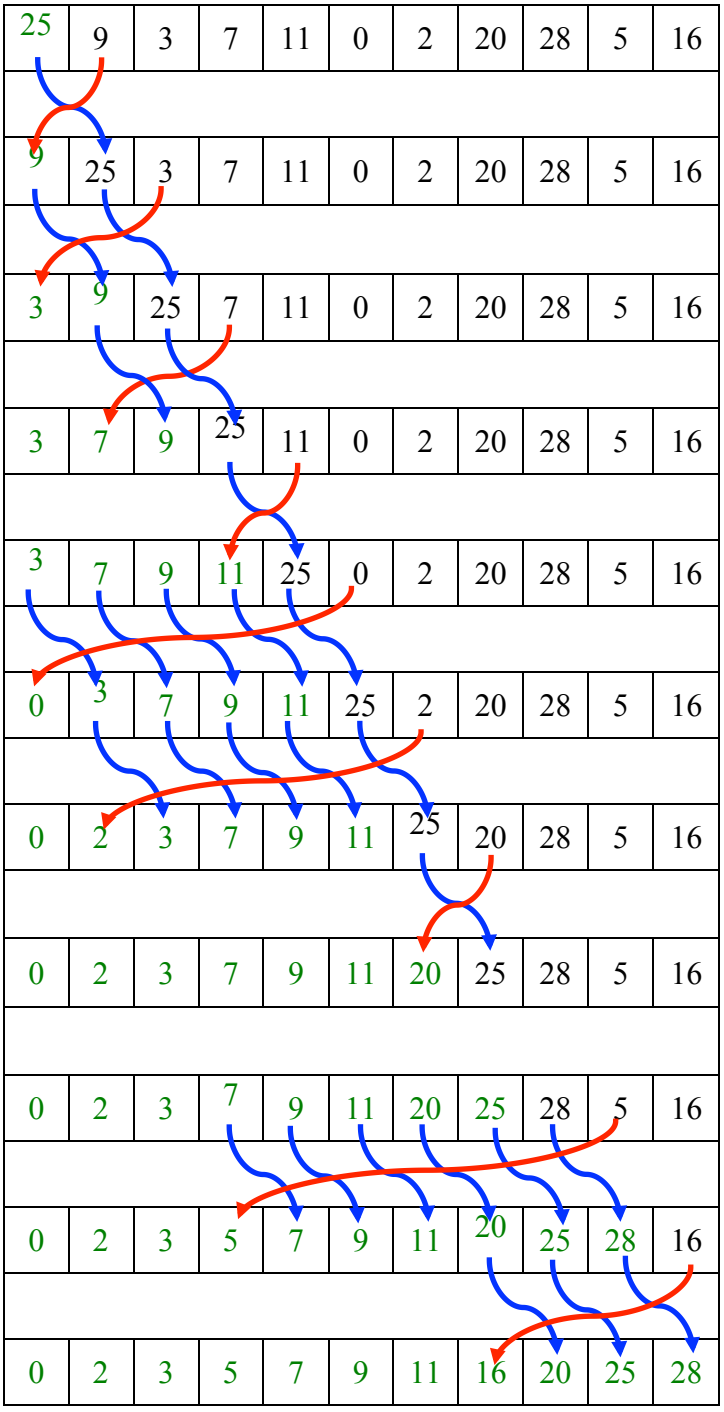
Assignment 1 – Algorithm Efficiency and Sorting

Assigned on: 2 October 2015 (Friday)
Due Date: 16 October 2015 (Friday)

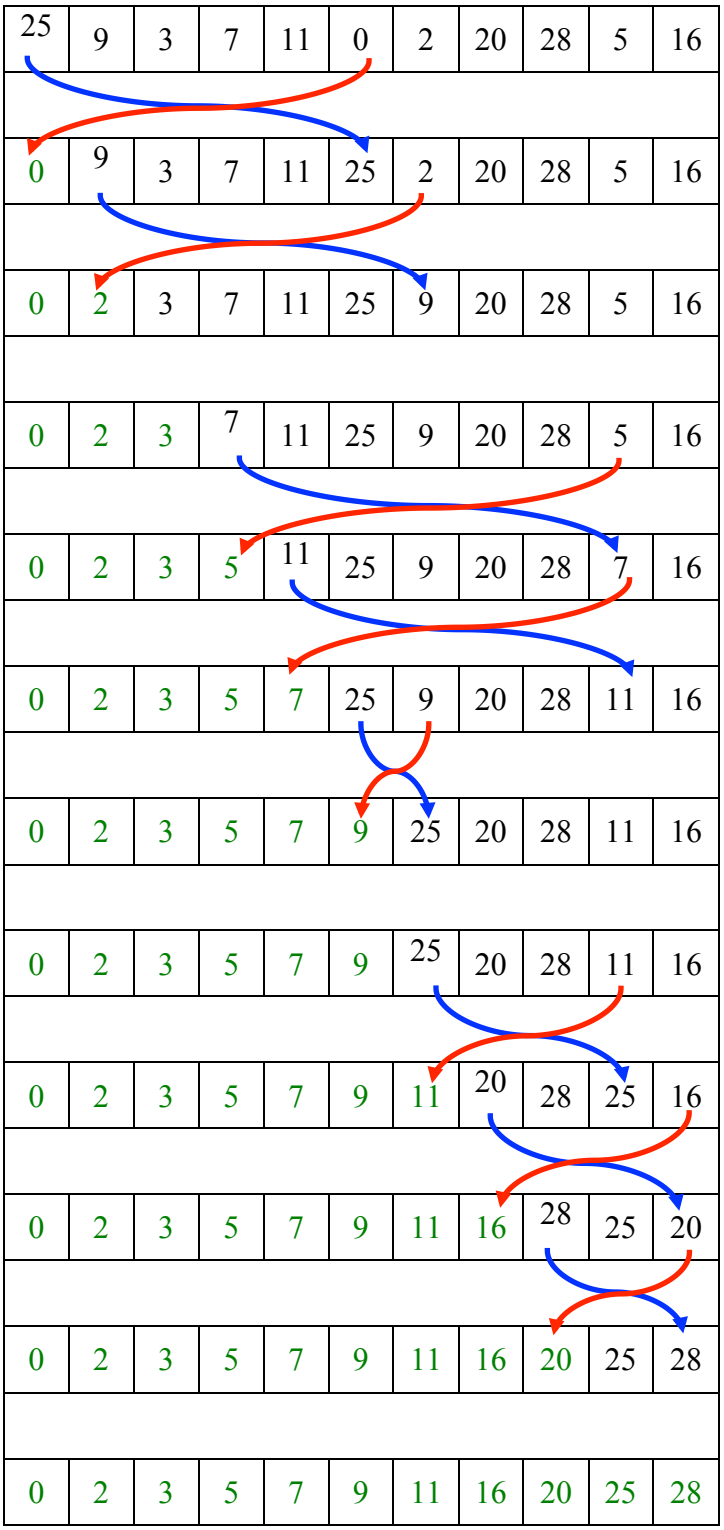
Name: Gülsüm
Surname: Güdükbay
ID: 21401148
Section: 1

1) Tracing

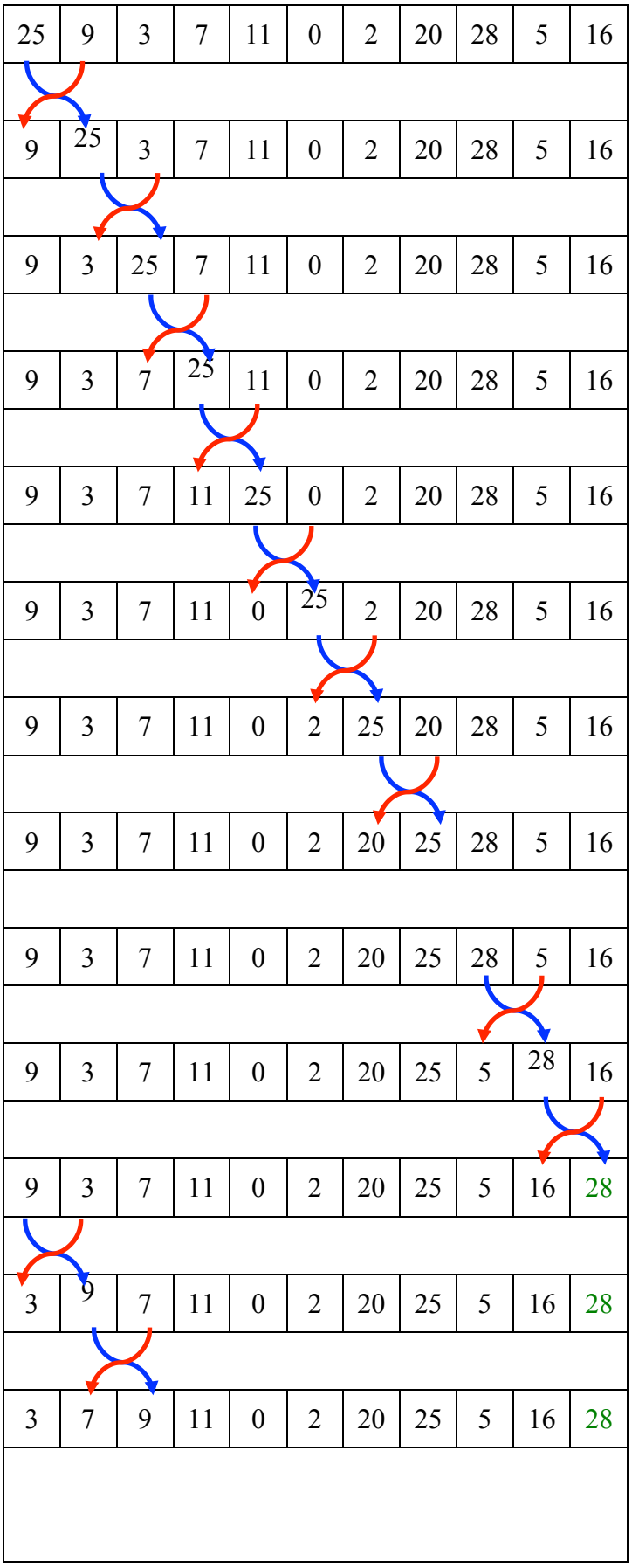
1. Insertion Sort

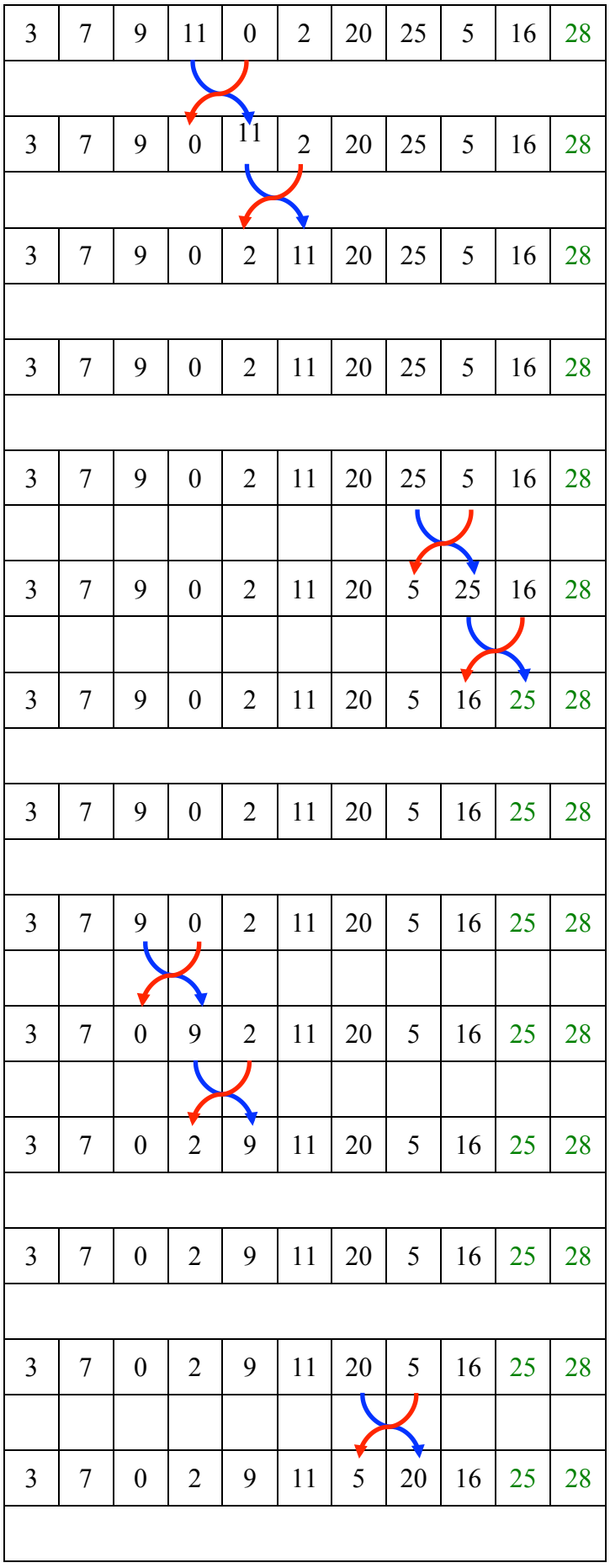


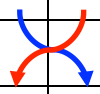
2. Selection Sort



3. Bubble Sort

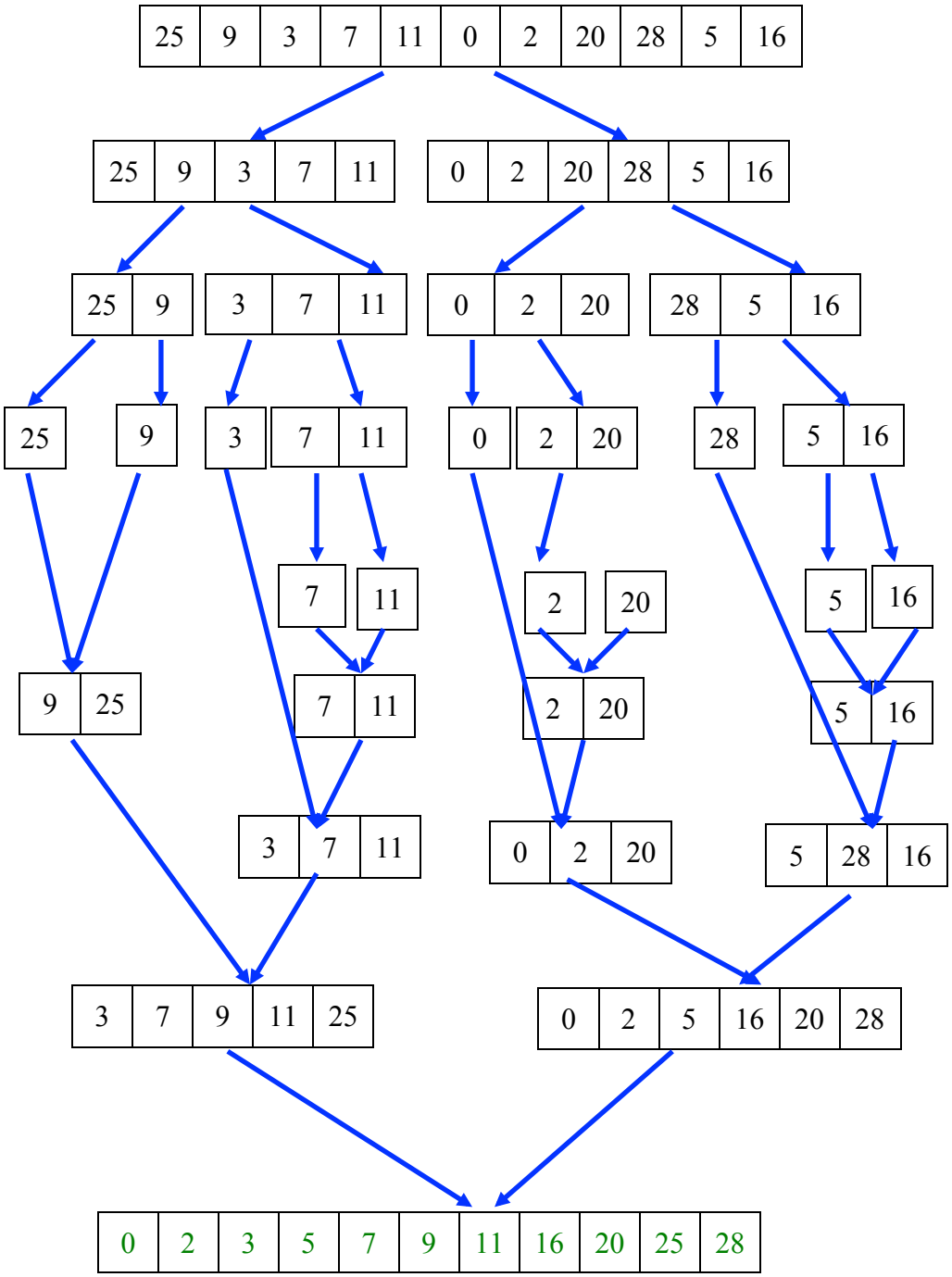



3 7 9 11 0 2 20 25 5 16 28
3 7 9 0 11 2 20 25 5 16 28
3 7 9 0 2 11 20 25 5 16 28
3 7 9 0 2 11 20 25 5 16 28
3 7 9 0 2 11 20 25 5 16 28
3 7 9 0 2 11 20 5 25 16 28
3 7 9 0 2 11 20 5 16 25 28
3 7 9 0 2 11 20 5 16 25 28
3 7 9 0 2 11 20 5 16 25 28
3 7 0 9 2 11 20 5 16 25 28
3 7 0 2 9 11 20 5 16 25 28
3 7 0 2 9 11 20 5 16 25 28
3 7 0 2 9 11 5 20 16 25 28

3	7	0	2	9	11	5	20	16	25	28
										
3	7	0	2	9	11	5	16	20	25	28
										
3	0	7	2	9	11	5	16	20	25	28
										
3	0	2	7	9	11	5	16	20	25	28
3	0	2	7	9	11	5	16	20	25	28
										
3	0	2	7	9	5	11	16	20	25	28
3	0	2	7	9	5	11	16	20	25	28
										
0	3	2	7	9	5	11	16	20	25	28
										
0	2	3	7	9	5	11	16	20	25	28
0	2	3	7	9	5	11	16	20	25	28
0	2	3	7	9	5	11	16	20	25	28
										
0	2	3	7	5	9	11	16	20	25	28
0	2	3	7	5	9	11	16	20	25	28
0	2	3	7	5	9	11	16	20	25	28

0	2	3	7	5	9	11	16	20	25	28
0	2	3	5	7	9	11	16	20	25	28
0	2	3	5	7	9	11	16	20	25	28

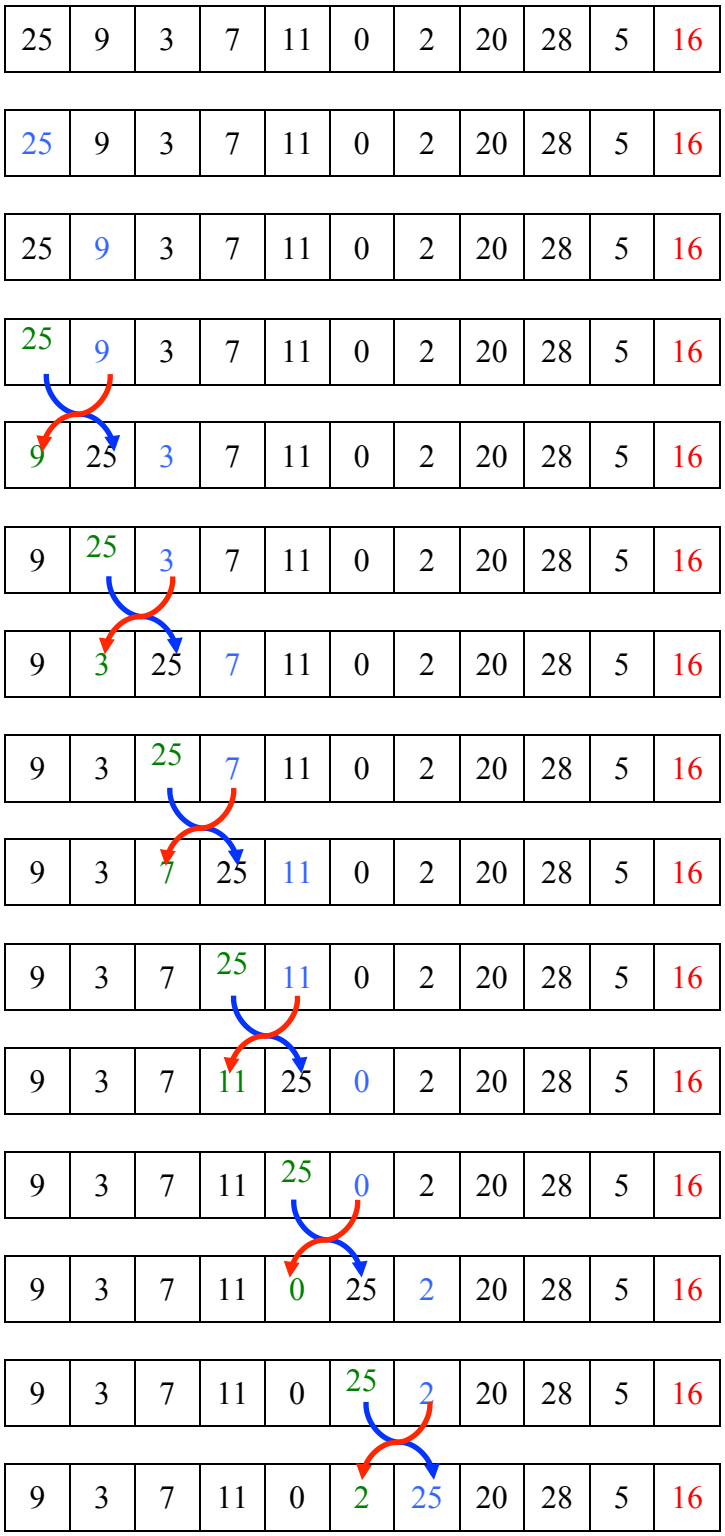
4. Merge Sort

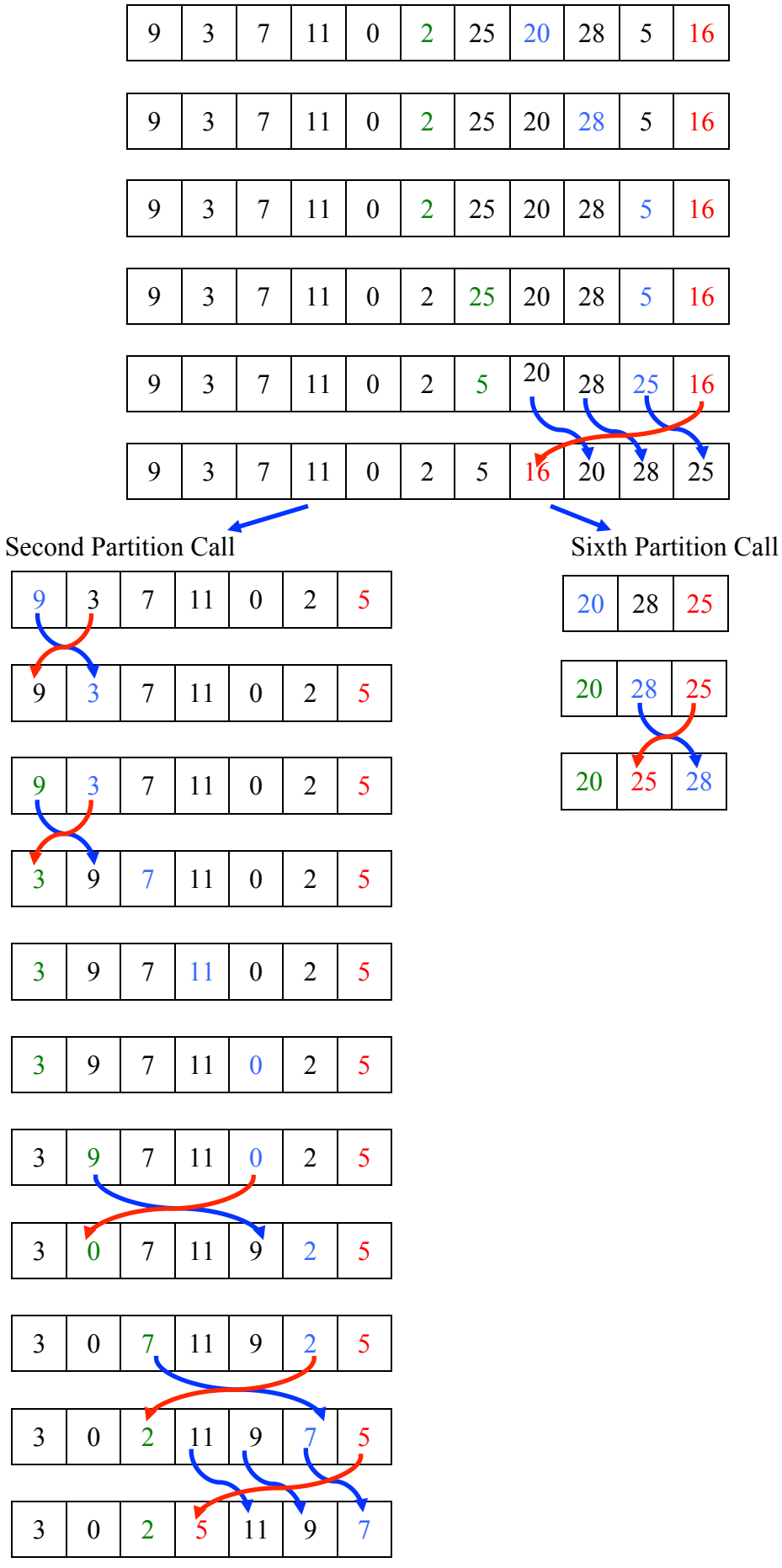


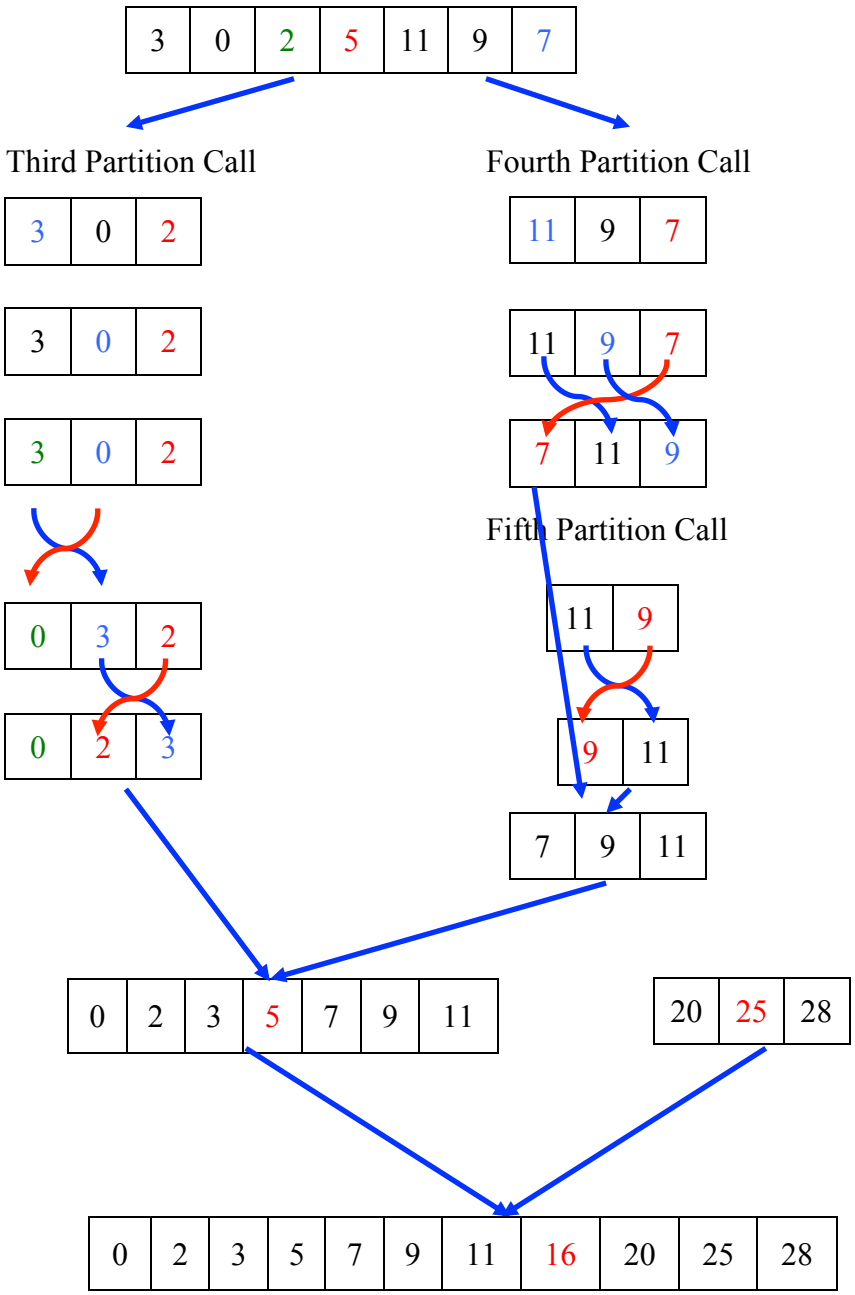
5. Quick Sort

Pivot is shown in red.
The element at the counter j is shown in blue.
The element at the counter i is shown in green.

First Partition Call




9 3 7 11 0 2 25 20 28 5 16
9 3 7 11 0 2 25 20 28 5 16
9 3 7 11 0 2 25 20 28 5 16
9 3 7 11 0 2 25 20 28 5 16
9 3 7 11 0 2 5 20 28 25 16
9 3 7 11 0 2 5 16 20 28 25
Second Partition Call
Sixth Partition Call
9 3 7 11 0 2 5
9 3 7 11 0 2 5
9 3 7 11 0 2 5
3 9 7 11 0 2 5
3 9 7 11 0 2 5
3 9 7 11 0 2 5
3 9 7 11 0 2 5
3 0 7 11 9 2 5
3 0 7 11 9 2 5
3 0 2 11 9 7 5
3 0 2 5 11 9 7
20 28 25
20 28 25
20 25 28

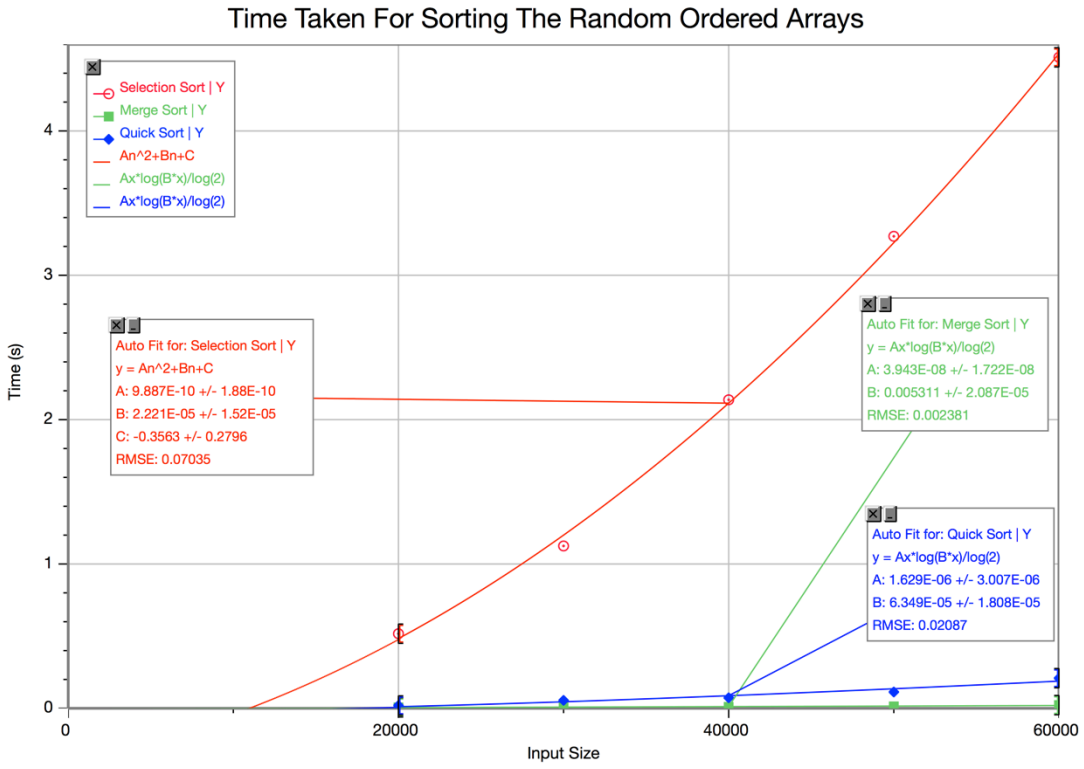

3 0 2 5 11 9 7
Third Partition Call
Fourth Partition Call
3 0 2
3 0 2
3 0 2
11 9 7
11 9 7
7 11 9
Fifth Partition Call
0 3 2
0 2 3
11 9
9 11
7 9 11
0 2 3 5 7 9 11
20 25 28
0 2 3 5 7 9 11 16 20 25 28

2) Programming

Sample Output (For Random Ordered Arrays)

```
 INPUT SIZE: 20000
SelectionSort took 535.443 milliseconds, 2104435 comparisons and 56922
moves.
MergeSort took 5.27 milliseconds, 260480 comparisons and 287232 moves.
QuickSort took 20.209 milliseconds, 2072206 comparisons and 6.27632e+06
moves.

 INPUT SIZE: 30000
SelectionSort took 1300.91 milliseconds, 4643669 comparisons and 85200
moves.
MergeSort took 10.01 milliseconds, 407655 comparisons and 447232 moves.
QuickSort took 45.653 milliseconds, 4569841 comparisons and 1.37992e+07
moves.

 INPUT SIZE: 40000
SelectionSort took 2120.66 milliseconds, 8163572 comparisons and 113838
moves.
MergeSort took 11.007 milliseconds, 560616 comparisons and 614464 moves.
QuickSort took 88.099 milliseconds, 8079067 comparisons and 2.43569e+07
moves.

 INPUT SIZE: 50000
SelectionSort took 3249.75 milliseconds, 12681241 comparisons and 142404
moves.
MergeSort took 15.721 milliseconds, 716654 comparisons and 784464 moves.
QuickSort took 122.14 milliseconds, 12557788 comparisons and 3.78231e+07
moves.

 INPUT SIZE: 60000
SelectionSort took 4740.33 milliseconds, 18194421 comparisons and 170763
moves.
MergeSort took 15.845 milliseconds, 875308 comparisons and 954464 moves.
QuickSort took 212.09 milliseconds, 18166310 comparisons and 5.46786e+07
moves.
```

1. Random Ordered Arrays

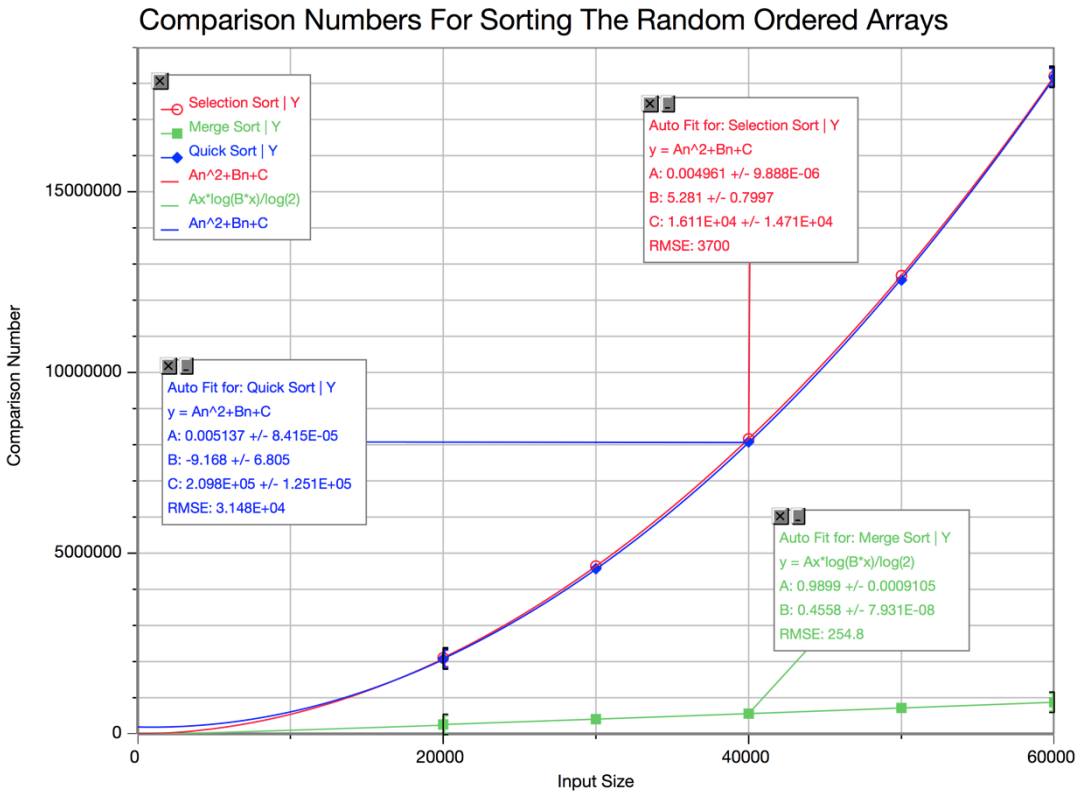


Graph 1.1 - Time Taken For Sorting The Random Ordered Arrays

Time taken for SelectionSort is in $O(n^2)$

Time taken for MergeSort is in O(nlogn)

Time taken for QuickSort is in O(nlogn)

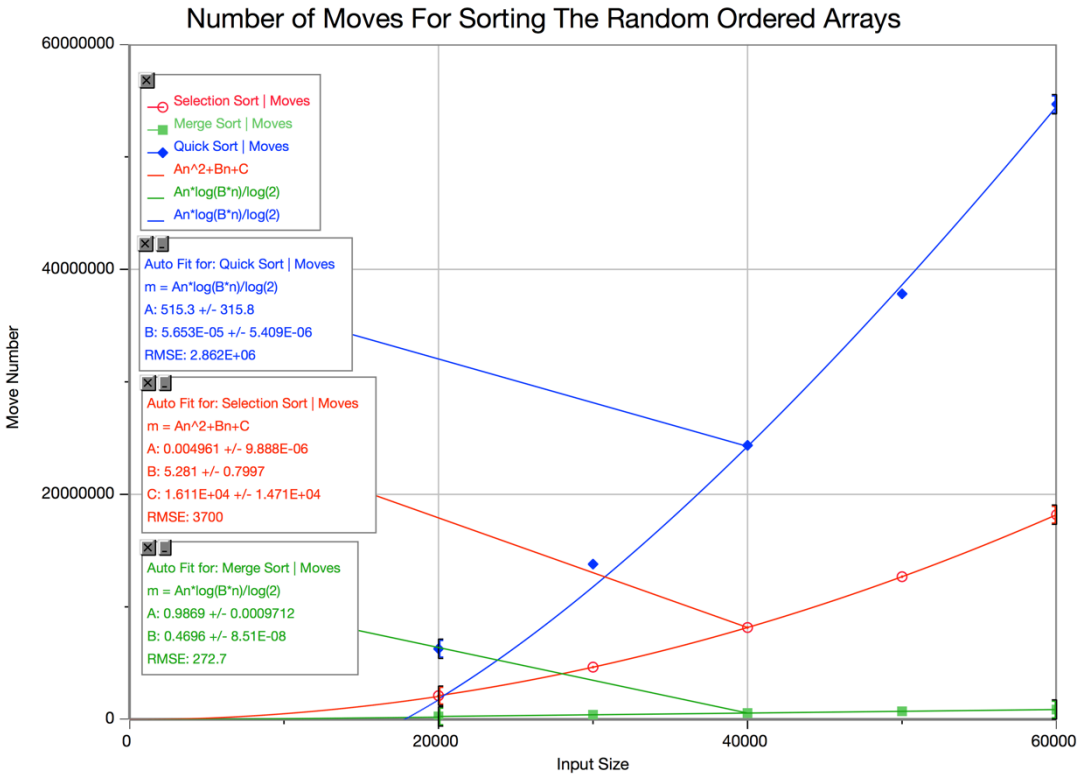


Graph 1.2 – Comparison Numbers For Sorting The Random Ordered Arrays

Comparisons for SelectionSort is in $O(n^2)$

Comparisons for MergeSort is in O(nlogn)

Comparisons for QuickSort is in $O(n^2)$



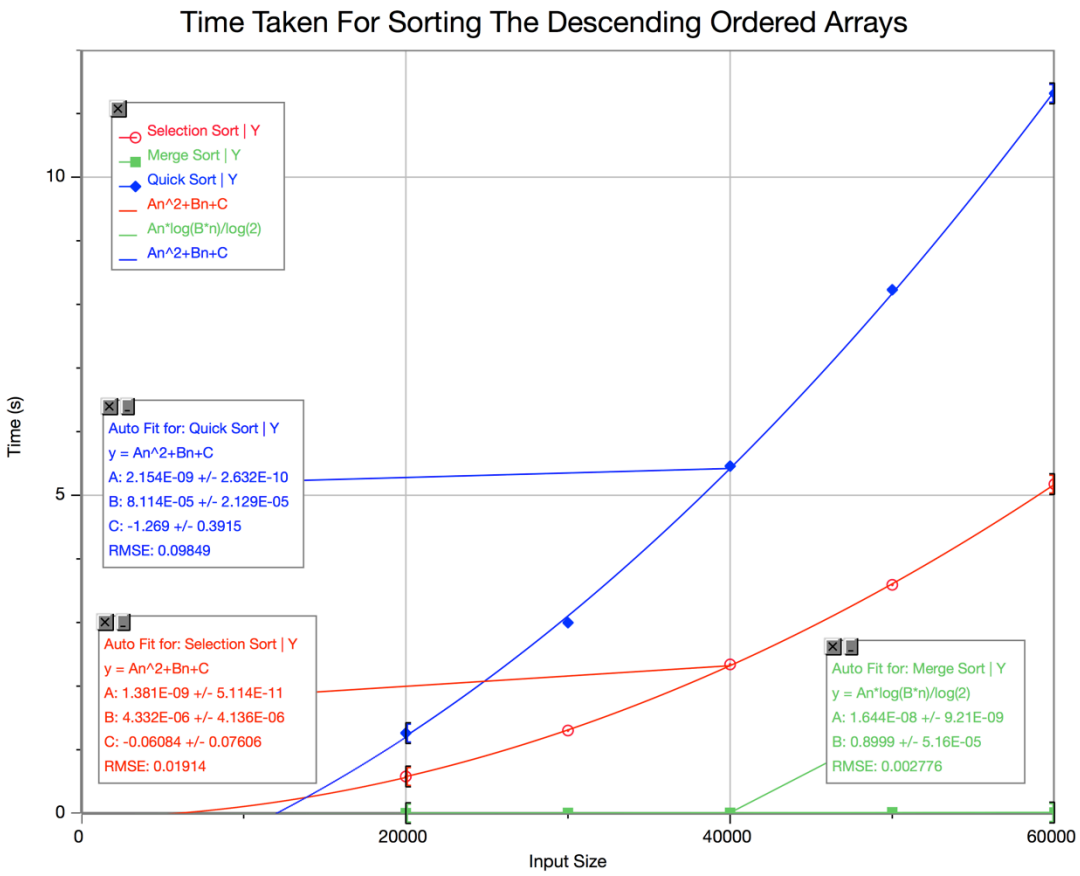
Graph 1.3 – Number of Moves For Sorting The Random Ordered Arrays

Moves for SelectionSort is in $O(n^2)$

Moves for MergeSort is in O(nlogn)

Moves for QuickSort is in $O(n^2)$

2. Descending Ordered Arrays

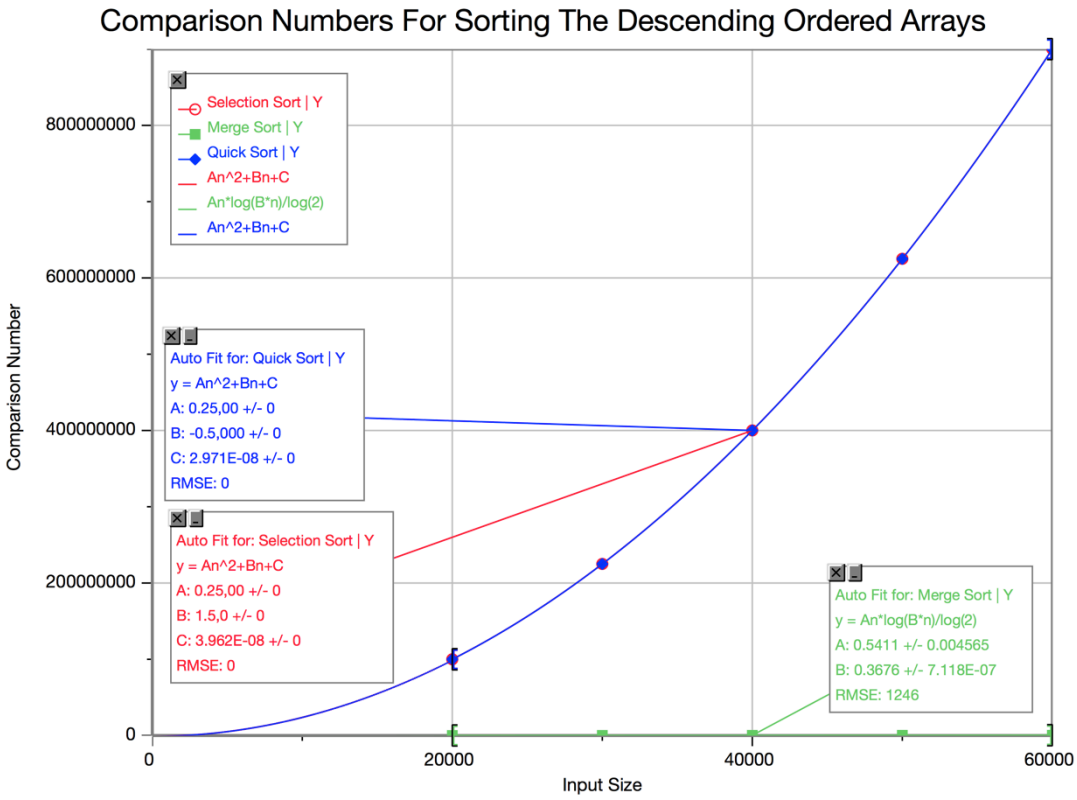


Graph 2.1 - Time Taken For Sorting The Random Ordered Arrays

Time taken for SelectionSort is in $O(n^2)$

Time taken for MergeSort is in O(nlogn)

Time taken for QuickSort is in $O(n^2)$

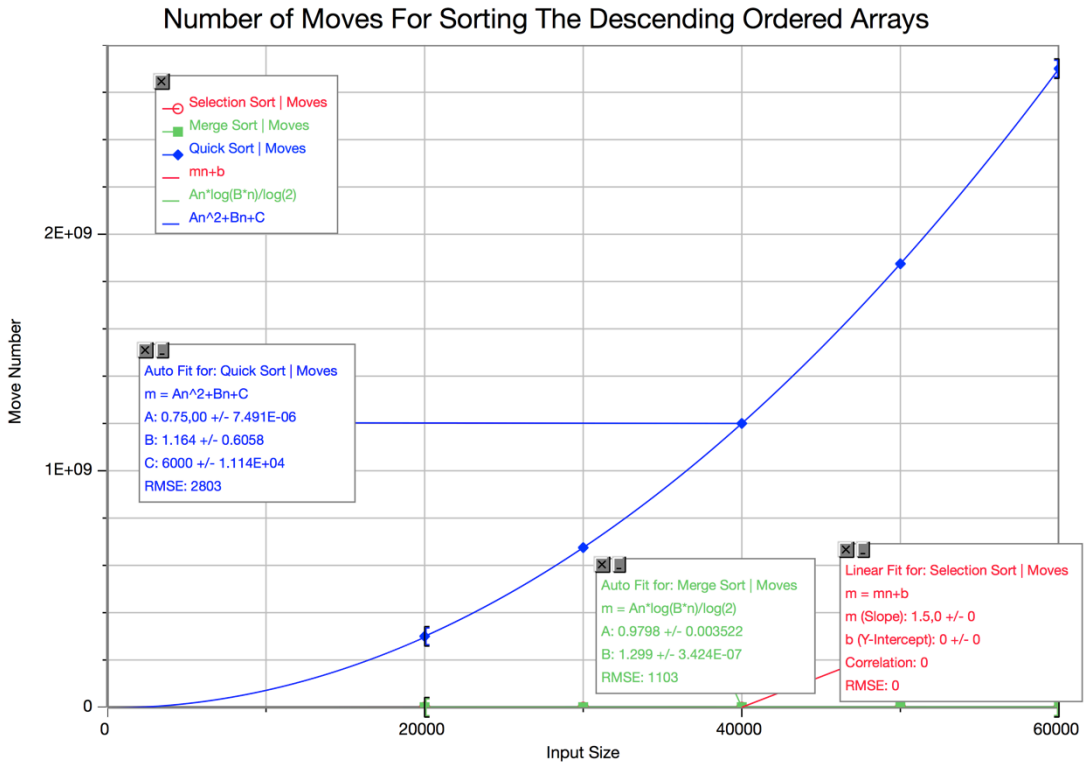


Graph 2.2 – Comparison Numbers For Sorting The Random Ordered Arrays

Comparisons for SelectionSort is in $O(n^2)$

Comparisons for MergeSort is in O(nlogn)

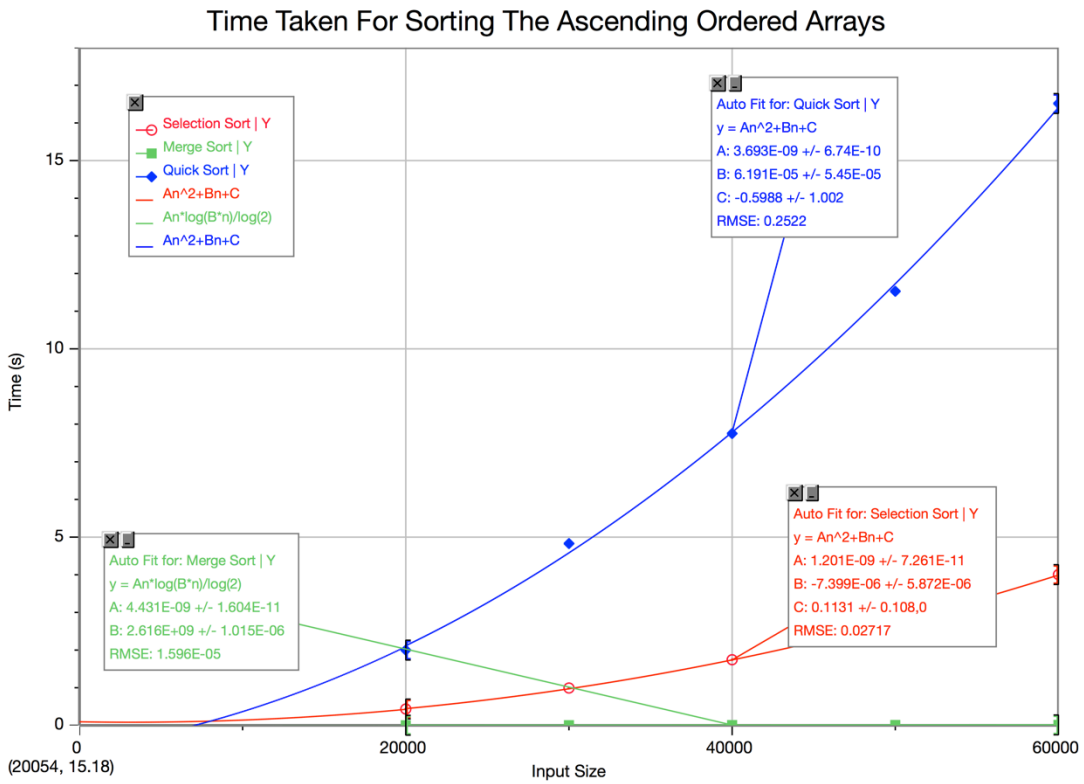
Comparisons for QuickSort is in $O(n^2)$



Graph 2.3 – Number of Moves For Sorting The Random Ordered Arrays

Moves for SelectionSort is in O(n)
Moves for MergeSort is in O(nlogn)
Moves for QuickSort is in $O(n^2)$

3. Ascending Ordered Arrays

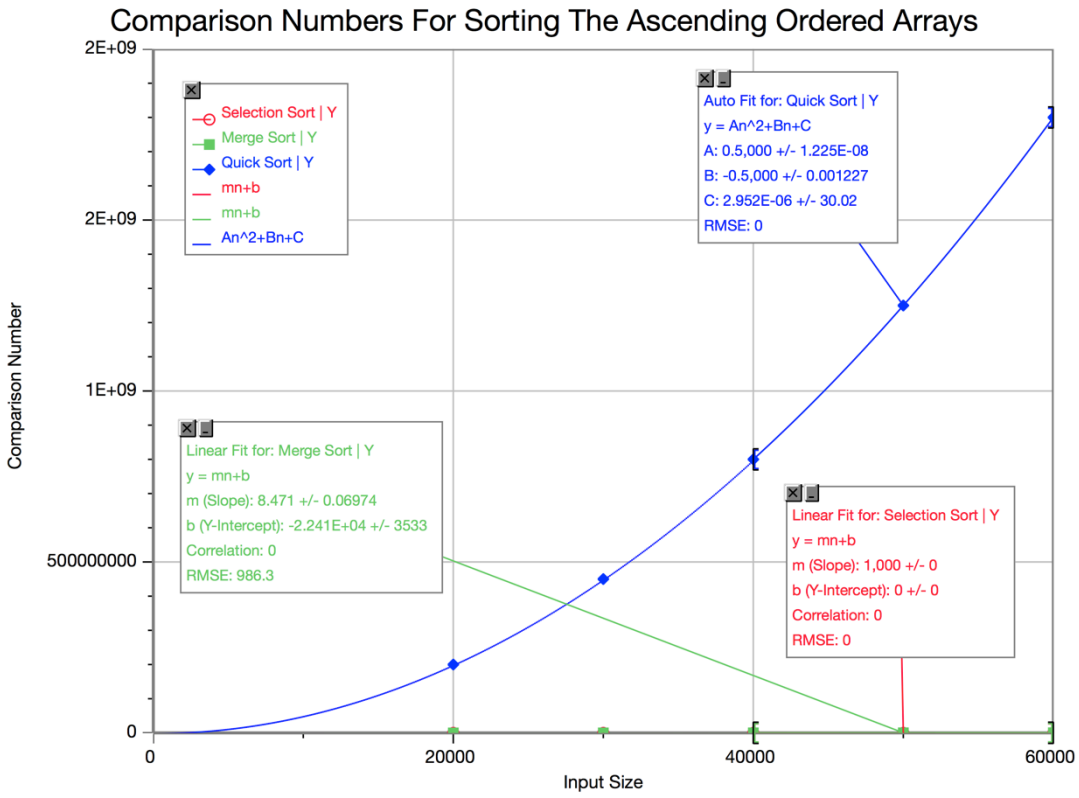


Graph 3.1 - Time Taken For Sorting The Random Ordered Arrays

Time taken for SelectionSort is in $O(n^2)$

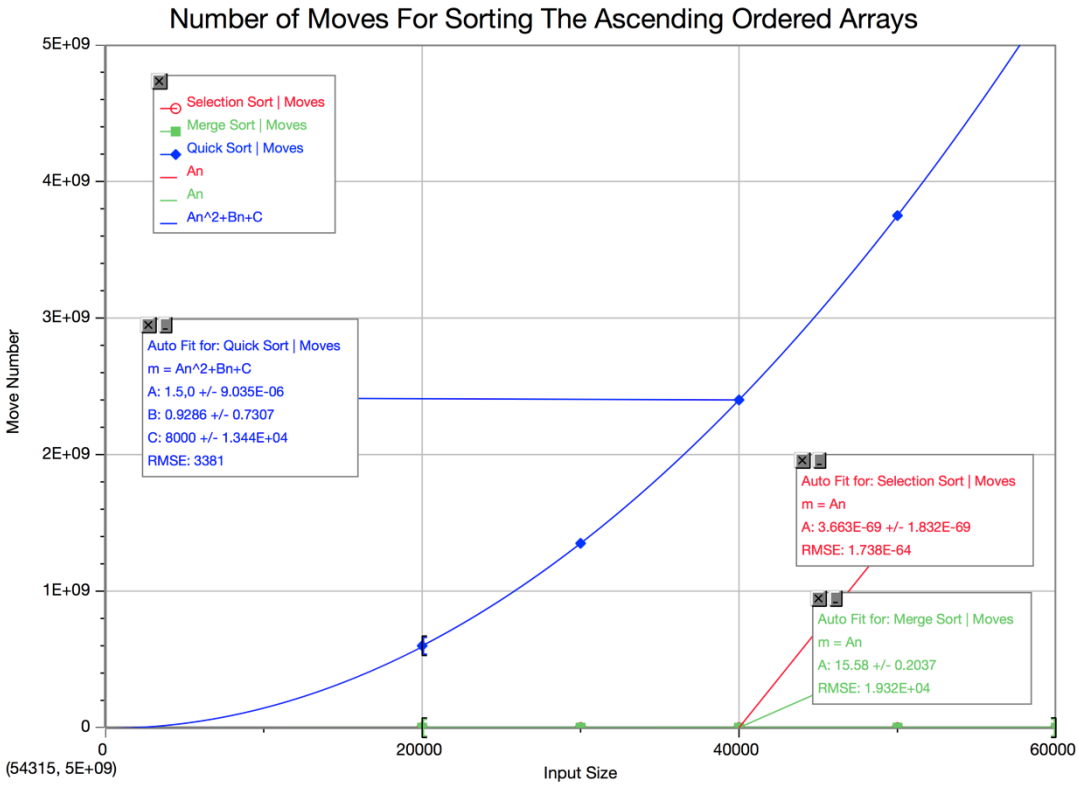
Time taken for MergeSort is in O(nlogn)

Time taken for QuickSort is in $O(n^2)$



Graph 3.2 – Comparison Numbers For Sorting The Random Ordered Arrays

Comparisons for SelectionSort is in O(n)
Comparisons for MergeSort is in O(0)
Comparisons for QuickSort is in $O(n^2)$



Graph 3.3 – Number of Moves For Sorting The Random Ordered Arrays

Moves for SelectionSort is in O(0)
Moves for MergeSort is in O(n)
Moves for QuickSort is in $O(n^2)$

3) Interpretation

Selection Sort was the least efficient in terms of the time complexity for the random ordered array, since it was in $O(n^2)$. The most efficient sorting algorithms in terms of the time complexity for the random ordered array were Merge Sort and Quick Sort, since they were O(nlogn).

Merge Sort was the most efficient in terms of the comparison number for the random ordered array, since it had O(nlogn) complexity. The other sort algorithms had $O(n^2)$.

Merge Sort was the most efficient in terms of the move number for the random ordered array, since it had O(nlogn) complexity. The other sort algorithms had $O(n^2)$.

Merge Sort was the most efficient in terms of the time complexity for the descending ordered array, since it was in O(nlogn). The least efficient sorting algorithms in terms of the time complexity for the random ordered array were Merge Sort and Quick Sort, since they were $O(n^2)$.

Merge Sort was the most efficient in terms of the comparison number for the descending ordered array, since it had O(nlogn) complexity. The other sort algorithms had $O(n^2)$.

Merge Sort was the most efficient in terms of the move number for the descending ordered array, since it had O(nlogn) complexity. The worse algorithm in terms of the move number was Quick Sort, which had $O(n^2)$.

Merge Sort was the most efficient in terms of the time complexity for the ascending ordered array, since it was in O(nlogn). The most efficient sorting algorithms in terms of the time complexity for the random ordered array were Merge Sort and Quick Sort, since they were $O(n^2)$.

Merge Sort was the most efficient in terms of the comparison number for the ascending ordered array, since it had O(0) complexity, therefore 0 moves. The worse algorithm in terms of comparisons was Quick Sort, which was in $O(n^2)$.

Selection Sort was the most efficient in terms of the move number for the ascending ordered array, since it had O(0) complexity, therefore 0 moves. The worse algorithm was Quick Sort, which was in $O(n^2)$.

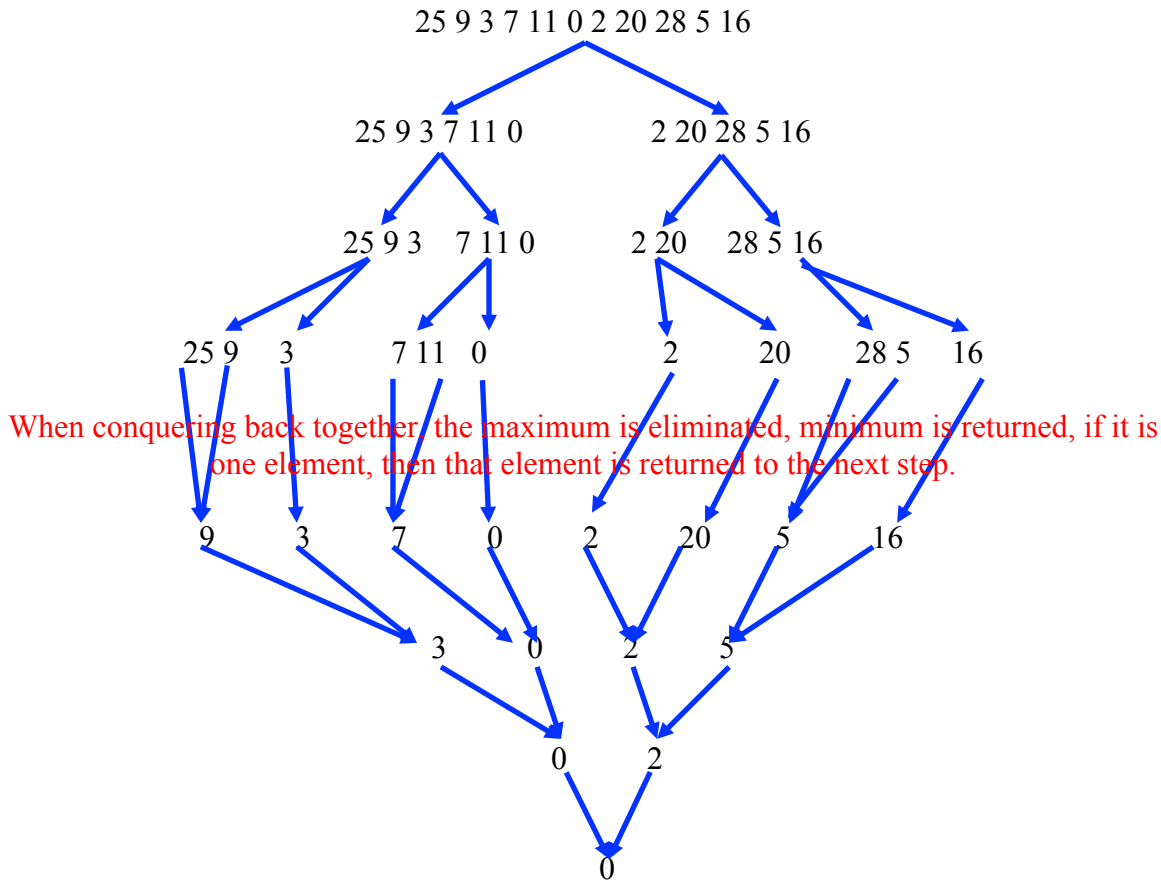
4) Algorithm Analysis

1. Recursive algorithm of finding the minimum element of an array

```
findMin(arr: integer, first: integer, last: integer)

        //First base case
        if( array has one element)
                return (first element of array)
        //Second base case
        else if( array has two elements)
                return min(first element of array, second element of array)
        //Recursive step
        else{
                first = findMin(first half of arr)
                second = findMin(second half of arr)
                return min(first, second)
        }
//End of findMin
```

2. Proof for the algorithm being correct

Tracing of the algorithm for the array given in the first question of the homework:



0 is the minimum of the inputted array, therefore the algorithm is correct.

3. Time Complexity of the algorithm

Base case: T(1) = 1 or T(2) = c, where c is a constant.

$$T(n) = 2T\left(\frac{n}{2}\right) + 2$$

$$T(n) = 2(2T\left(\frac{n}{4}\right) + 2) + 2$$

$$T(n) = 4T\left(\frac{n}{4}\right) + 6$$

$$T(n) = 2^{k-1}T(2) + 2^k - 2$$

$$T(n) = \frac{3n}{2} - 2$$

$$T(n) = O(n)$$

5) Asymptotic Analysis and Growth Rate Functions

1. Proving that log(n!) = O(n log(n)):

Big Oh notation indicates the upper bound of a function. If we use the logarithm rule
log(a*b) = log(a) + log(b),
we can infer that:
log(n!) becomes log(n)+log(n-1)+log(n-2)+…+log(1).

Noting that every element of the sum is less than or equal to log(n), so:
log(n)+log(n-1)+log(n-2)+…+log(1) <= log(n)+ log(n)+ log(n)+… log(n)
Therefore, log(n!) = O(n log(n)) is proven true.

2. Proving that the solution for the recurrence relation

$$T(1) = 0$$
$$T(n) = 2T\left(\frac{n}{2}\right) + n, n \geq 2$$

is

$$T(n) = nlog_2(n)$$

by induction:

Proving that the equation is true for n = 2:

$$T(2) = 2\log_2 2 = 2T(1) + 2$$
$$2 = 2$$

Assuming true for n = k:

$$T(k) = 2T\left(\frac{k}{2}\right) + k = klog_2(k)$$

Using the assumption, proving true that the relation holds for n = k+1:

$$T(k+1) = 2T\left(\frac{k+1}{2}\right) + k + 1$$
$$T(k+1) = 2\left(2T\left(\frac{k+2}{4}\right) + \frac{k}{2} + 1\right) + k + 1$$
$$T(k+1) = 4T\left(\frac{k+2}{4}\right) + 2k + 3$$
$$T(k+1) = 2\left(4T\left(\frac{k+4}{8}\right) + k + 3\right) + 2k + 3$$
$$T(k+1) = 8T\left(\frac{k+4}{8}\right) + 4k + 9$$
$$T(k+1) = 8T\left(\frac{(k+1)+3}{8}\right) + 4(k+1) + 1$$

$k + 1 = 13$ for the base case. After doing the algebra, it is found that

$$T(k+1) = (k+1)\log_2(k+1)$$

which shows that the relation holds for n = k+1, therefore all integer values of n ≥ 2.

Therefore, the solution for the recurrence relation

$$T(1) = 0$$
$$T(n) = 2T\left(\frac{n}{2}\right) + n, n \geq 2$$

is proven to be

$$T(n) = nlog_2(n)$$

for n ≥ 2, using the principles of mathematical induction.

3. Showing that $f(n) = 4n^5 + 3n^2 + 1$ is order of $O(n^5)$ by giving appropriate c and n_0 values.

Graphing $f(n) = 4n^5 + 3n^2 + 1$ (shown in red) and $g(n) = 7n^5$ (shown in blue), the c and n_0 values are 7 and 5 respectively. The two graphs intersect at n = 1.08. So for n >= 1.08, $g(n) = 7n^5$ is greater than $f(n) = 4n^5 + 3n^2 + 1$.

Ignoring c, the multiplicative term in the function, the Big Oh notation of the time complexity of f(n) is found to be $O(n^5)$.

GRAPH OF g(n) AND f(n) COMPARED