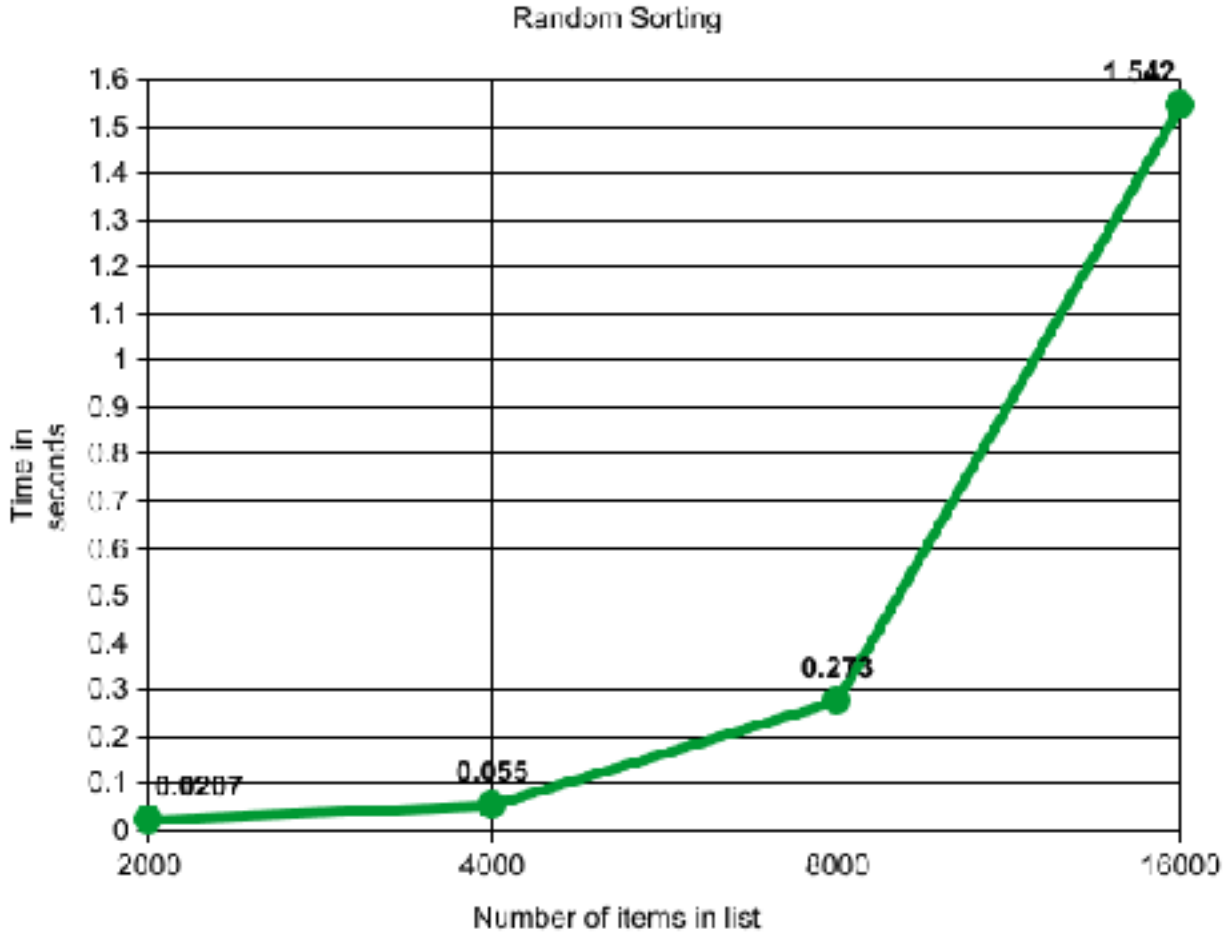
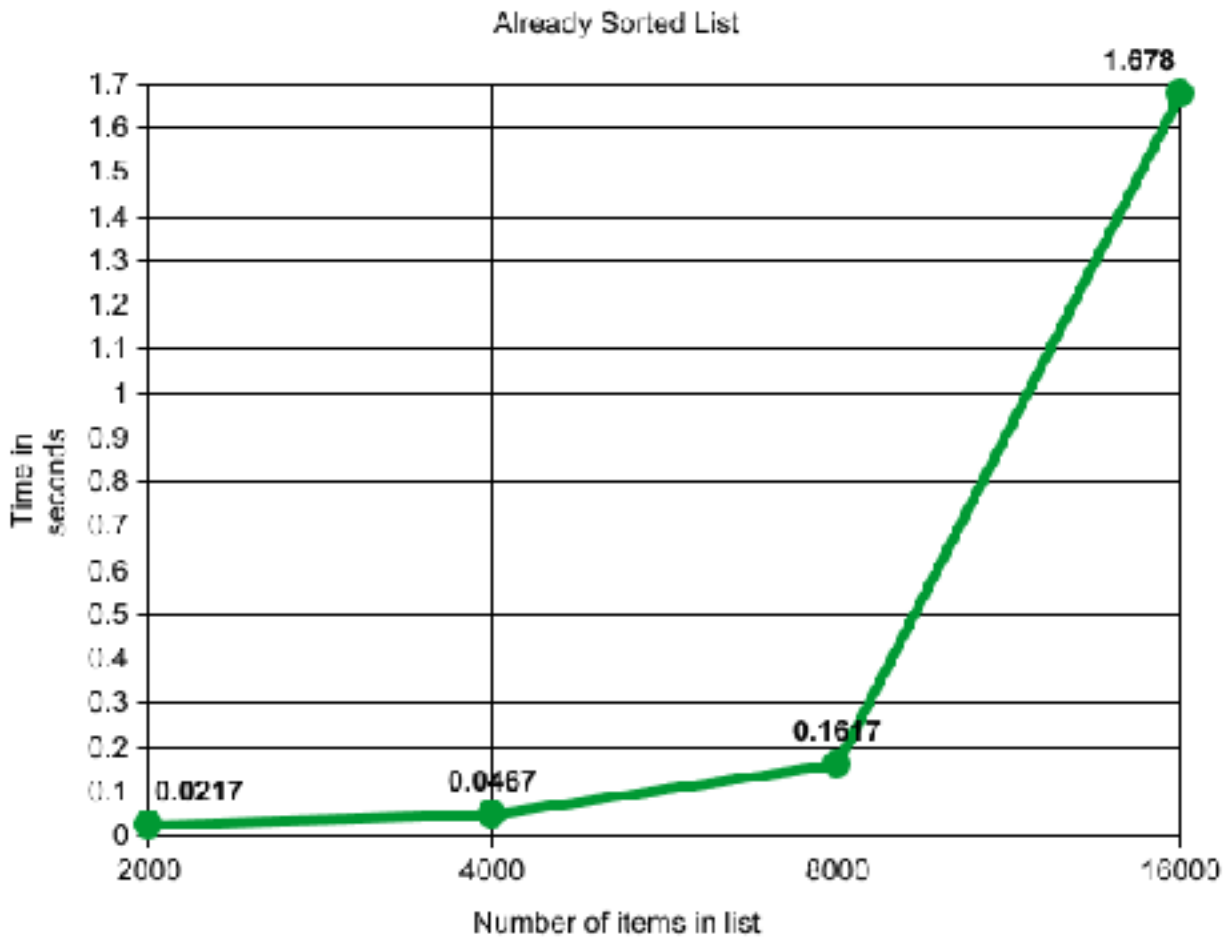
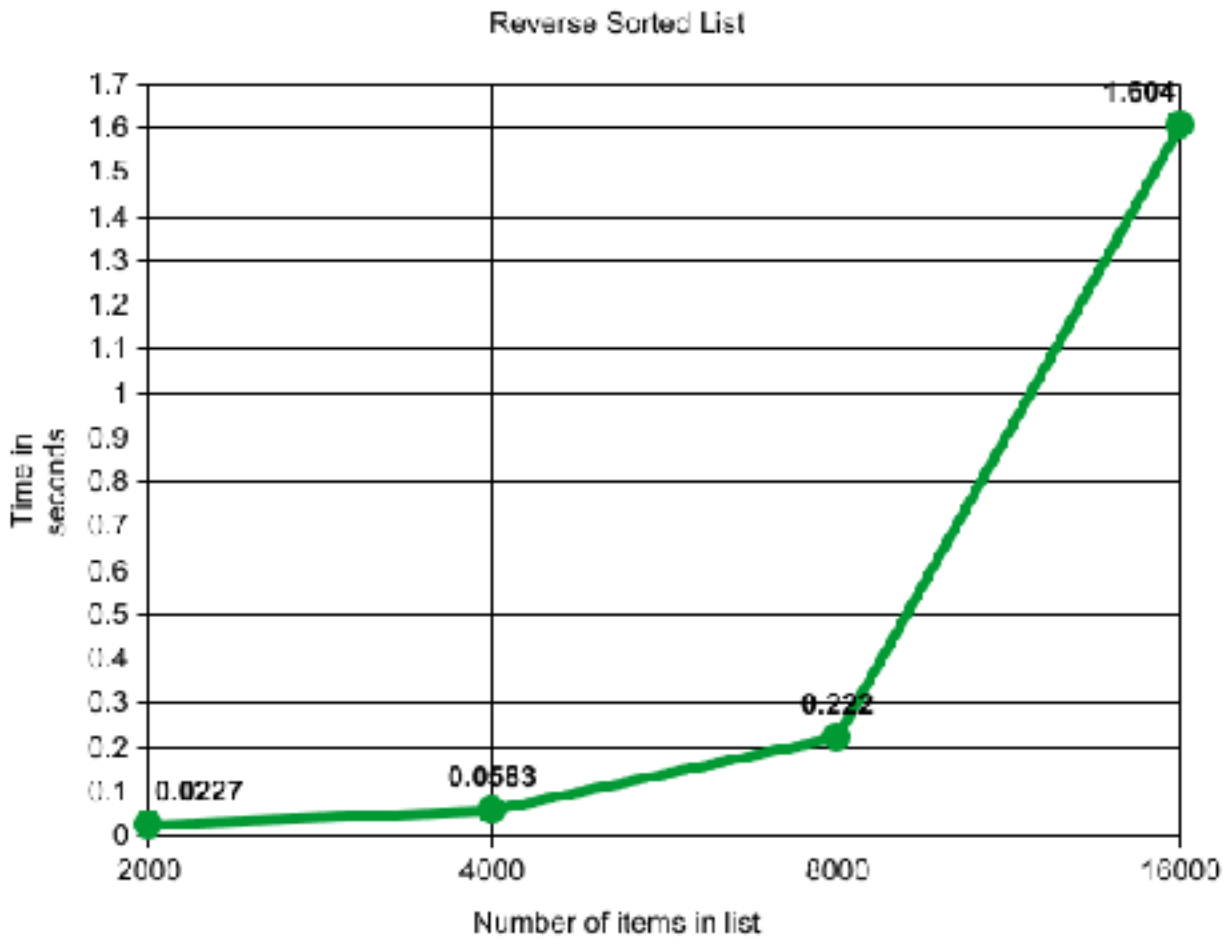

Random Sorting
Time in seconds
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1.542
0.273
0.055
0.0207
2000
4000
8000
16000
Number of items in list


Already Sorted List
Time in seconds
1.7
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.0217
0.0467
0.1617
1.678
2000
4000
8000
16000
Number of items in list



3. From your timing data analyze the rate of growth of SelectionSort using the doubling rule.

Type	2000	4000	8000	16000
Random	0.0207	0.055	0.273	1.542
Sorted	0.0217	0.0461	0.1517	1.678
Reversed	0.0227	0.0583	0.222	1.604
Average	0.0217	0.053	0.219	1.608

We can expect a function with a linear rate of growth to double in time as *n* doubles. As such, if the time for SelectionSort to sort a list of 2000 was around 0.0217 each time, we could expect 16000 inputs to take, on average, 0.1736 seconds. Instead, 16000 inputs took around 1.6 seconds with this function, which is almost 10 times longer. From 2000 inputs to 4000 inputs, the time it took complete the sort grew by a factor of almost 2.5, which is already larger than the factor of 2 we would expect from a linear-time algorithm. Further, from 4000 to 8000 inputs the time grew by a factor of 4.13. And from 8000 to 16000, it grow by 7.34, which is a substantial increase. In order words, the rate of rate of growth increased as the input size increased. This leads us to conclude that

SelectionSort is at least an O(n^2) function or of even higher order. But how can we be sure? By abstracting the math a little bit, we can say that the growth factor in time from 2000 to 4000 was ~2; ~4 from 4000 to 8000; and ~8 from 8000 to 16000. As *n* doubled, so did the rate at which the rate of growth grew. In other words, the time it takes to complete the sort grows by a factor of *n*^2, so this function's rate of growth is O(n^2).

4. Analyze the SelectionSort code: use number of array accesses as your cost model, and describe the rate of growth in tilde notation.

Right away from looking at the code for SelectionSort we can see that it's rate of growth is at least O(n^2). This is because this function uses a double for loop the size of *n* to compare each element in the unsorted array with every other element. Since we know that each element in the array must be compared to every element, and a comparison can be counted as one operation, that alone is n squared operations. In addition to the n squared comparisons, the function also exchanges elements that must be exchanged in order for the array to be sorted. Because a place exchange is an operation on two different array elements, and in the worst case every single element must be exchanged with another, that is another 1/2n operations. So the rate of growth of this function is O(n^2 + 1/2n) by looking at the code, as the two steps in the code mentioned above are the only ones that require a significant number of unique operations.

5. How does the order of the input data affect SelectionSort? Explain why, by referring to the algorithm or the code (how many times is less and exch called?).

The order of the input data does not affect SelectionSort substantially. As we saw above, the exchanges that must be made on an unsorted array still cost must less than the comparisons themselves, which is why in Big O notation we would simply consider this function to have a rate of growth of O(n^2). Regardless of whether or not the items in the array are sorted, the comparison function **less** is still called on every element of the array with every other element - that is still n squared calls. SelectionSort calls **exch** regardless of whether or not the array is already sorted, so that is 1/2n array accesses anyway (even if SelectionSort didnt call **exch** when the elements were already sorted, 1/2n fewer operations is still trivial to the order of growth of *n* squared, so it still wouldn't make a difference). This is a drawback of sorts like SelectionSort which don't have ways to check whether most of the list to be sorted already has been.