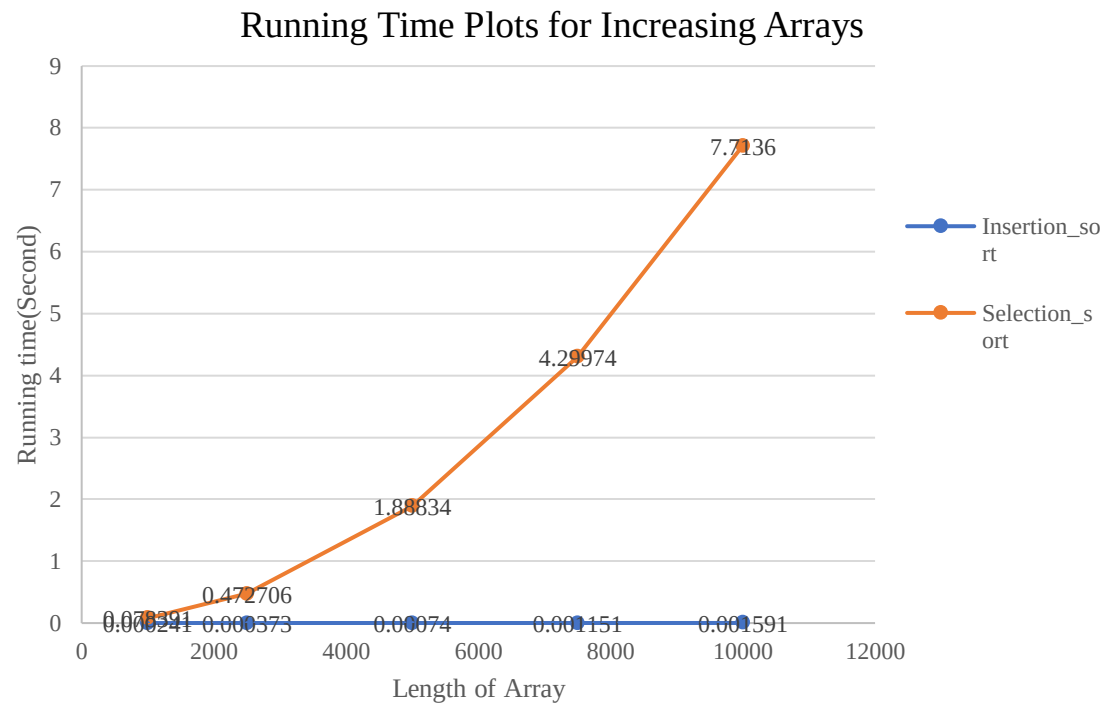
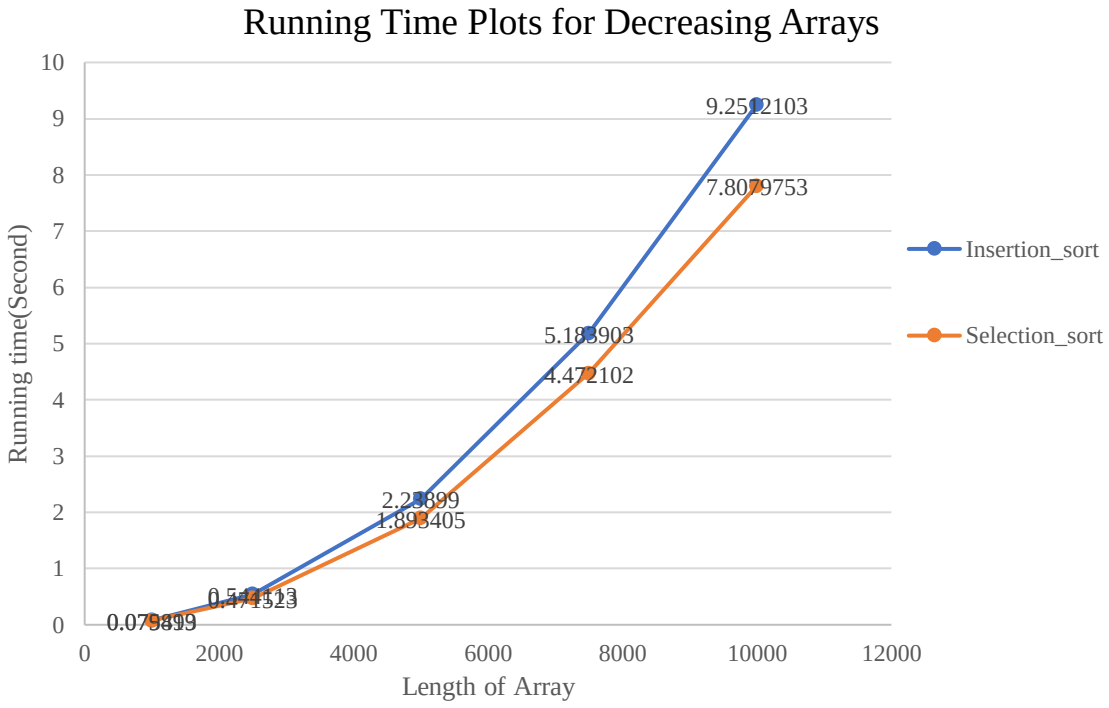
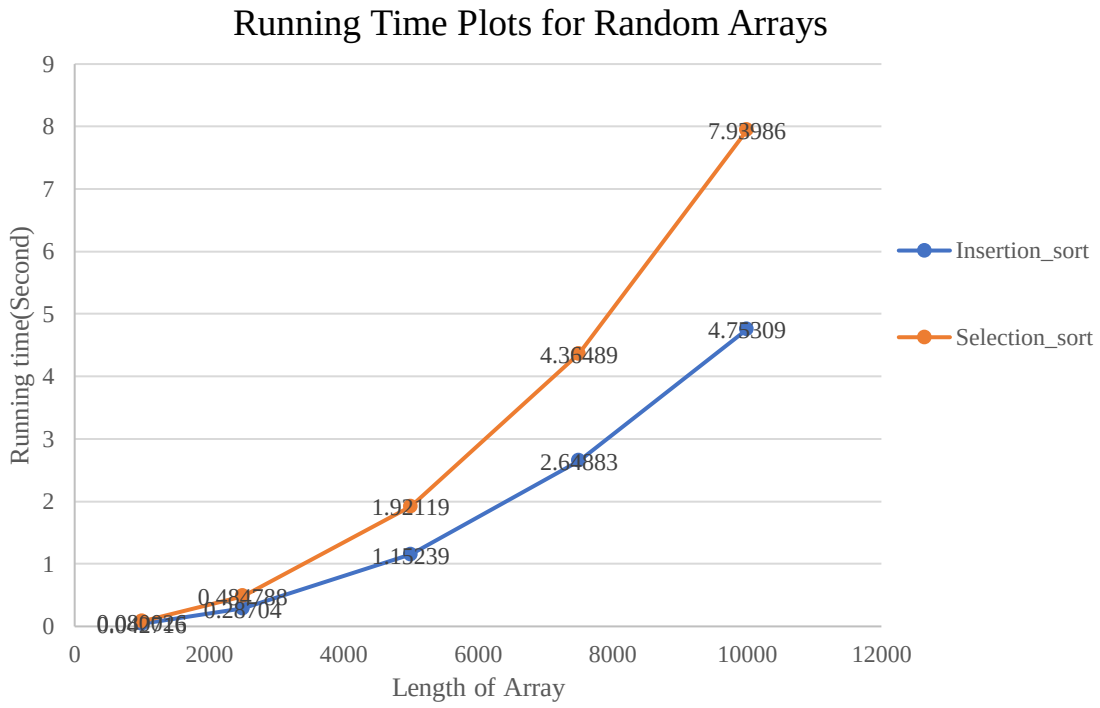

Running Time Plots for Increasing Arrays
Running time(Second)
Length of Array
Insertion_sort
Selection_sort
7.7136
4.29974
1.88834
0.472706
0.000373
0.00074
0.001151
0.001591
0
2000
4000
6000
8000
10000
12000


Running Time Plots for Decreasing Arrays
Running time(Second)
Length of Array
Insertion_sort
Selection_sort
9.2512103
7.8079753
5.183903
4.472102
2.23899
1.893405
0
2000
4000
6000
8000
10000
12000



Prose explanation:

It can first be observed that the performance of selection sort is roughly identical in the three types of input, which I think is because no matter what the input is like, selection sort always goes through the same process. The *while* loop inside the *for* loop always go through the whole sub-array after the index *k* in the for loop, making up for $\frac{n^2-n}{2}$ loops, and inside every turn of the *while* loop the *if* statement is executed regardless of whether the *minimalindex* is updated, which makes up for aa total $\frac{n^2-n}{2}$ compares. After each *while* loop ends, there's always one movement of values, swapping the value in index *k* and index *minimalindex*, making up for a total *n* movement of values. Therefore, the execution of selection sort is steady, comprised of $\frac{n^2-n}{2}$ compares and *n* movement.

While the performance of selection sort is roughly the same, insertion sort shows obviously different efficiency in dealing with different types of input. This is because the *while* loop in the insertion sort function is executed only if *arr[j-1]>cur*, which is dependent on the feature of the input.

In the first graph where the input array is already sorted, the *while* loop in the insertion sort is never executed because every *arr[j]* <= *cur,* so the movement of value inside the body of the *while* loop is never executed. In this case, the only work that insertion sort does is looping through the *for* loop and assign the value of *arr[j]* to itself, which is very fast because the *cur* and *arr[j]* are already in alias in this case. Therefore, when the input is

increasing already, insertion sort does n loops through the array and n compares and n movements, while selection sort does $\frac{n^2-n}{2}$ loops inside which it executes $\frac{n^2-n}{2}$ compares, and n movement of values. Insertion sort does significantly less work than selection sort, which explains why in the first graph insertion sort is drastically faster than selection sort.

In the second graph where the input array is decreased, insertion sort performs slightly less efficient than selection sort. In this type of input, each *arr[j-1] > cur* is satisfied in each while loop, so the *while* loop is executed a total $\frac{n^2-n}{2}$ times, with a total $\frac{n^2-n}{2}$ compares. Further, in each loop, there's a movement of values from *arr[j-1]* to *arr[j],* and one movement in each *for* loop, making up for a total $\frac{n^2-n}{2}+n$ movements. Therefore, in the second case, insertion sort does the same time of loops and compares as selection sort but does more movements of values than selection sort. This explains the difference of two sorts functions in the second case.

Last, in the third graph where the input array is random, insertion sort performs more efficiently than selection sort to a moderate degree. The reason for this, I think, lies in that random array is the intermediate state between increasing and decreasing. In a random array, it is very likely that some sections of it is already increasing so that the *while* loop in the insertion sort need not be executed. Thus, overall the insertion sort will probably do less times of *while* loops, less times of compare accordingly, but more times of movements of values, because anyway the insertion sort will do some *while* loops and do some more movements of values besides the n movements in the *for* loop. The reason why insertion sort performs better in the random input case is probably that moving values of *int* type is much cheaper and faster than comparing two values of *int* type. This can be confirmed by comparing the pattern in the first and second graph: when the times of compares drops, the efficiency increases dramatically, while when the times of movements increases to roughly the same proportion, the decrease of efficiency is relatively small. This also implies that when moving values is much more expensive than comparing values, selection sort will probably be faster than insertion sort.

To sum up:

1. Increasing:
   Insertion sort: n loops, n compares, n movements
   Selection sort: $\frac{n^2-n}{2}$ loops, $\frac{n^2-n}{2}$ compares, n movements
   Result: Insertion sort significantly faster than selection sort
2. Decreasing:
   Insertion sort: $\frac{n^2-n}{2}$ loops, $\frac{n^2-n}{2}$ compares, $n+\frac{n^2-n}{2}$ movements
   Selection sort: $\frac{n^2-n}{2}$ loops, $\frac{n^2-n}{2}$ compares, n movements

Result: Insertion sort slightly slower than selection sort

3. Random:

   Insertion sort: less than $\frac{n^2-n}{2}$ loops, less than $\frac{n^2-n}{2}$ compares, less than $n+\frac{n^2-n}{2}$, more than n movements

   Selection sort: $\frac{n^2-n}{2}$ loops, $\frac{n^2-n}{2}$ compares, n movements

   Result: Insertion sort moderately faster than selection sort