ACE Report for Coursework 1

[Answer for Q1]

Algorithm 1: InsertionSort(Target)
Input: an array of integers Target[0..len -1] **Output:** an increasing array of integers from Target
len ← length(Target)
for i from 0 to len - 1 **do**
ToCompare ← Target[i]
j ← i – 1
while j >= 0 **and** Target[j] > ToCompare
Target[j+1] ← Target[j]
j ← j – 1
{**Assertion A:** Each time this point is reached, ToCompare is at most equal to Target[j+1..i], and they have been shifted back one place}
end while
Target[j+1] ← ToCompare
{**Assertion B:** At i-th time this point is reached, Target[0..i] are sorted in increasing order and all data hasn't been destroyed}
end for

[Answer for Q2]

Proof:

The Assertion B, denoted as **aB**, implies that **Target[0..i-1]** is a sorted permutation of the original **Target[0 .. i-1]**, at the beginning of each time the outer loop iteration. B is true in the beginning because **Target[0]** is sorted. However, in order to prove **aB** is true for each iteration, the analysis through the executions within the outer loop shall be conducted. So Assertion A, denoted as **aA**, would be applied. **aA** indicates that **Target[j..i]** are each greater or equal to **ToCompare**.

Since **-1 < 0** (The first iteration with **i=0**) and **Target[0] = ToCompare** (The second iteration with **i=1**) caused the end, **aA** is true with these initialization. The inner loop keeps **aA** true because the operation **Target[j+1] ← Target[j]** shifts back a value in **Target**, which is greater than **ToCompare**, into **Target[j+1]**. This loop didn't destroy data in Target because **Target[i]** was copied into **ToCompare**. If **ToCompare** is reallocated into a place in **Target**, it's true that **Target[0..i]** contains the first **i** elements of the original one. When inner loop terminates, there are four key facts:

1. **Target[0..j]** is sorted and is at most equal to ToCompare.
2. **Target[j+1..i]** is sorted and at least equal to ToCompare.
3. **Target[j +1] = Target[j +2]** if the loop is executed at least once.
4. **Target[j+1] = ToCompare** if the loop did not execute at all.

Note: **Equality was considered due to the boundary situations.**

With these four points, **Target[j+1] ← ToCompare** didn't destroy any data and then offered a sorted permutation **Target[0..i]**. Since **aA** was supported after the inner loop terminates, **aA** would be true until outer loop ends, in other words, **i = len-1**. Finally, **Target[0..len-1]** is sorted. The algorithm is correct.

[Answer for Q3]

Algorithm 1: InsertionSort(Target) **Input:** an array of integers Target[0..len -1] **Output:** an sorted array of integers from Target		Costs	Times
1	len ← length(Target)	C_1	1
2	**for** i from 0 to len - 1 **do**	C_2	len
3	ToCompare ← Target[i]	C_3	len
4	j ← i – 1	C_4	len
5	**while** j >= 0 **and** Target[j] > ToCompare	C_5	$\sum_{i=0}^{len-1}(t_i)$
6	Target[j+1] ← Target[j]	C_6	$\sum_{i=0}^{len-1}(t_i - 1)$
7	j ← j – 1	C_7	$\sum_{i=0}^{len-1}(t_i - 1)$
8	{**Assertion A:** Each time this point is reached, ToCompare is at most equal to Target[j+1..i-1], and they have been shifted back one place}	0	0
9	**end while**	0	**0**
10	Target[j+1] ← ToCompare	C_{10}	len
11	{**Assertion B:** At i-th time this point is reached, Target[0..i] are sorted in increasing order and all data hasn't been destroyed}	C_{11}	0
12	**end for**	0	**0**

Note: t_i stands for the number of of times, which inner loop runs for with the value i, $t_i \in [1, i]$.

So in general,

$$T(len) = C_1 \times 1 + C_2 \times (len) + C_3 \times (len) + C_4 \times (len) + C_5 \times \sum_{i=0}^{len-1}(t_i) + C_6 \times \sum_{i=0}^{len-1}(t_i - 1) + C_7 \times \sum_{i=0}^{len-1}(t_i - 1) + C_8 \times 0 + C_9 \times 0 + C_{10} \times (len) + C_{11} \times 0 + C_{12} \times 0$$

Equivalently,

$$T(len) = C_1 + C_2 \times (len) + C_3 \times (len) + C_4 \times (len) + C_5 \times \sum_{i=0}^{len-1}(t_i) + C_6 \times \sum_{i=0}^{len-1}(t_i - 1) + C_7 \times \sum_{i=0}^{len-1}(t_i - 1) + C_{10} \times (len)$$

And equivalently,

$$T(len) = \sum_{i=0}^{len-1}((C_5 + C_6 + C_7) \times t_i - C_6 - C_7) + (C_2 + C_3 + C_4 + C_{10}) \times (len) + C_1$$

So now we focused on $\sum_{i=0}^{len-1}((C_5 + C_6 + C_7) \times t_i - C_6 - C_7)$, **abstracted as** $\sum_{i=0}^{len-1}(C \times t_i - C')$

Extend it to $(C \times t_0 - C') + (C \times t_1 - C') + \cdots + (C \times t_{len-1} - C')$

$$= C \times (t_0 + t_1 + \cdots + t_{len-1}) + (len) \times C', t_i \in [1, i].$$

The maximum is when each one element be as much as possible, so

$$(C \times (t_0 + t_1 + \cdots + t_{len-1}) + (len) \times C') \leq (C \times (0 + 1 + \cdots + (len - 1)) + (len) \times C')$$

And then $\left(C \times \left(0 + 1 + \cdots + (len - 1)\right) + (len) \times C'\right) = \frac{(0+len-1)\times(len)}{2} + (len) \times C'$

Now we could form the inequaility

$$T(len) = \sum_{i=0}^{len-1}((C_5 + C_6 + C_7) \times t_i - C_6 - C_7) + (C_2 + C_3 + C_4 + C_{10}) \times (len) + C_1$$

$$\leq \frac{(0 + len - 1) \times (len)}{2} + (len) \times C' + C_1$$

According to the Big-Oh definition,

$$T(len) = O\left(\frac{(0 + len - 1) \times (len)}{2} + (len) \times C' + C_1\right)$$

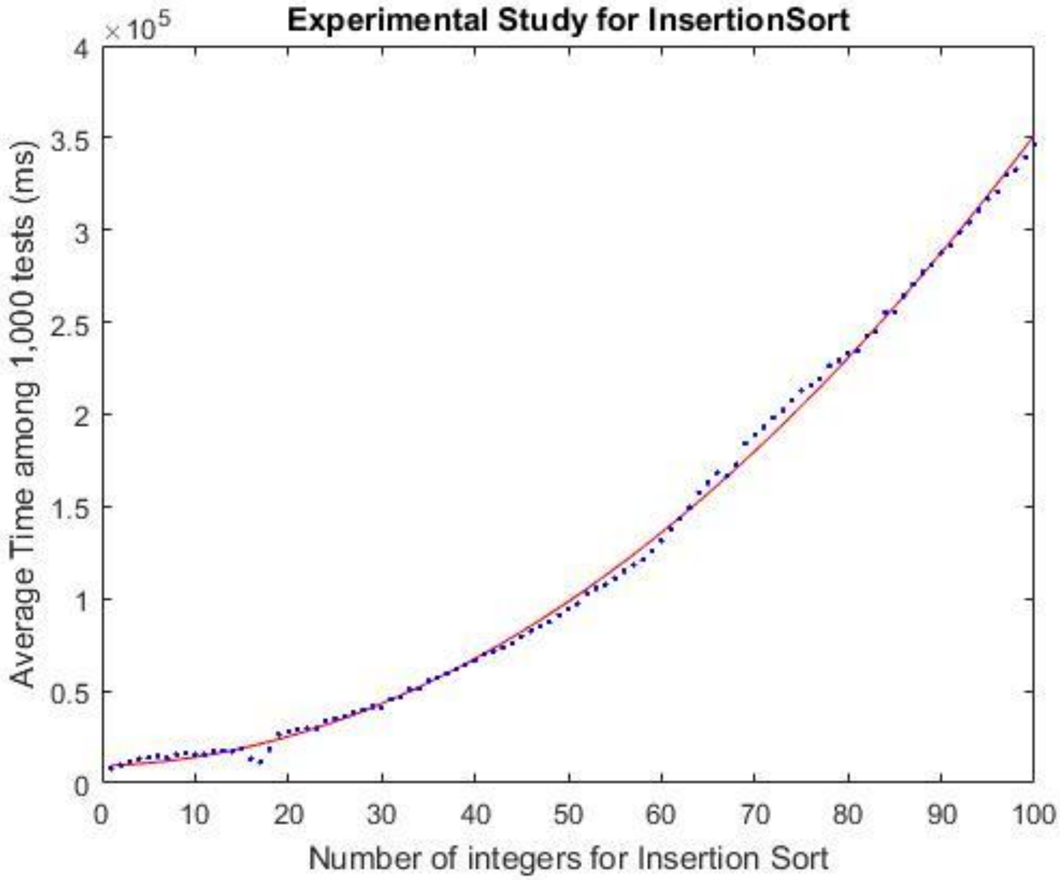
$$T(len) = O\left(\frac{(len - 1) \times len}{2} + (len) \times C' + C_1\right)$$

$$T(len) = O(len^2)$$

So, it is $\boldsymbol{T(n) = O(n^2)}$

[Answer for Q4]

Java Implementation enabled an input to identify how many numbers the program would generate each time. And the shell would pass the input and also repeat the same input for a few times. In experiments of this report, the shell would **pass the input from 1 to 100** and **repeat each input for 1000 times.** All the time data would be collected in "out.csv". Experimental study could be re-generated by executing **"./experiment.sh"** but it would took a few hours to do so. And then curves fitting has been conducted through Analysis.m, according to the average time of each input.



As figure shown, **blue points** stand **for the average time among 1000 trials with each input**, and **red line** stands for **the fitting curve**. The results supported the theoretical analysis in previous section. Combined with breakdown analysis, **the key observation is that Insertion sort is sensitive with the original permutation of Target.** For worst case, if Target is permutated as decreasing order, each time inserting a number would cause the highest cost. And this situation could be abstracted mathematically as the previous section did.

Please note: Related illustration are available in README.md