On Generating Worst-Cases for the Insertion Sort

Julie Tillison* and Ching-Kuang Shene*†

Department of Mathematics and Computer Science
Northern Michigan University
Marquette, MI 49855–5340

e-mail: {jtilliso, shene}@nmu.edu

1 Introduction

Many CS1 texts, for example Nance [2], and data structures and algorithms texts, for example Baase [1] and Sedgewick [3], use insertion sort as an example of elementary sorting algorithms that require $O(n^2)$ comparisons to sort n data items. A worst case that forces the insertion sort to take $O(n^2)$ comparisons is usually mentioned. This worst case, in general, is the reversed-order sequence of the sorted one, and the number of comparisons required is $n(n-1)/2$. It is not difficult to show that $n(n-1)/2$ is the upper bound of all worst cases. On the other hand, the minimum number of comparisons required to sort n data items with insertion sort is $n-1$, and in fact this is a tight lower bound. Knowing these facts, one may easily raise the following questions: (1) are there other non-trivial worst case input sequences? and (2) are there some sequences that can force the insertion sort to have the number of comparisons between $O(n)$ and $O(n^2)$, say $O(n^{1.5})$ and $O(n \log_2 n)$? In this paper, we shall answer these questions with an algorithm that can systematically construct sequences with which the number of comparisons required by the insertion sort is between $O(n)$ and $O(n^2)$.

2 The Algorithm

The idea of our construction is very simple. Let the input sequence be $a_1, a_2, \ldots, a_n$. Suppose the insertion sort algorithm uses n_i comparisons to insert a_i into position $i - n_i$, where $2 \leq i \leq n$. The total number of comparisons is $T = \sum_{i=2}^{n} n_i$. Therefore, a careful choice of n_i can vary T in the range of $O(n)$ and $O(n^2)$. For example, if $n_i = \sqrt{i}$, then we have

$$T = \sum_{i=2}^{n} \sqrt{i} \approx \int_2^n \sqrt{x}dx \in O(n^{1.5})$$

If $n_i = \log_2 i$, then the result is

$$T = \sum_{i=2}^{n} \log_2 i \approx \int_2^n \log_2 x \; dx \in O(n \log_2 n)$$

Suppose $n_i = \alpha_i(i-1)$, $0 < \alpha_i \leq 1$. Let $\alpha = \min_{2 \leq i \leq n}\{\alpha_i\}$ and $\beta = \max_{2 \leq i \leq n}\{\alpha_i\}$. Since $\alpha(i-1) \leq \alpha_i(i-1) = n_i \leq \beta(i-1)$, we have

$$\sum_{i=2}^{n} \alpha(i-1) \leq \sum_{i=2}^{n} \alpha_i(i-1) = T \leq \sum_{i=2}^{n} \beta(i-1)$$

Consequently, we have

$$\frac{\alpha}{2} n(n-1) \leq T \leq \frac{\beta}{2} n(n-1)$$

Therefore, $T \in O(n^2)$, which is a worst case. In particular, $n_i = i - 1$ yields the well-known one. If n_i is a constant, we have linear number of comparisons, and $n_i = 1$ yields the "best" case.

Suppose the actual values of $a_1, a_2, \ldots, a_n$ are unknown. As a_i is considered by the insertion sort algorithm, it will be moved, after n_i comparisons, to either position $i - n_i$ or position $i - n_i + 1$ depending on whether a_i is smaller than the data item currently in position $i - n_i$. Note that a_i will be moved to some other place when $a_{i+1}, \ldots, a_n$ are processed. Let the final position of a_i be j, and let the sorted array be $1, 2, 3, \ldots, n$. It is not difficult to see that the input

*The first author is a student supported by a faculty research grant of Northern Michigan University to the second author.

†This work was also supported in part by a NSF grant CCR-9410707.

```
PROCEDURE  GenSequence(VAR  Out:IntARRAY;
                            Num:IntARRAY;
                            N:INTEGER);
VAR
   Work       : IntARRAY;
   i, j, Temp : INTEGER;
BEGIN
   FOR i := 1 TO N DO
      Work[i] := i;
   FOR i := 1 TO N DO
      IF (Num[i] >= 1) AND (Num[i] <= i-1) THEN
         BEGIN
            Temp := Work[i];
            FOR j := i-1 DOWNTO i-Num[i] DO
               Work[j+1] := Work[j];
            Work[i-Num[i]] := Temp;
         END;
   FOR i := 1 TO N DO
      Out[Work[i]] := i;
END;
```

Figure 1: Input Sequence Generation Algorithm

Table 1: Results Generated for $n = 10$

n_i	*Generated Data*									
$i-1$	10	9	8	7	6	5	4	3	2	1
$0.5(i-1)$	1	10	2	9	3	8	4	7	5	6
$0.3(i-1)$	1	2	3	10	4	5	9	6	7	8
3	8	9	10	1	2	3	4	5	6	7
1	10	1	2	3	4	5	6	7	8	9
$\sqrt{i}$	10	1	9	2	3	4	5	8	6	7
$\log_2 i$	10	1	2	9	3	4	5	8	6	7

value to a_i is j. Figure 1 is an algorithm that implements the above observation.

In the algorithm, `Out` is the generated array, `Num[i]` corresponds to n_i, and `N` is the number of data items. This procedure uses a local array `Work` to store the indices of $a_1, a_2, \ldots, a_n$. Initially, `Work[i]` is `i`, giving the initial position of a_i. The second `FOR` loop simulates an insertion sort, however, without the actual comparisons. As mentioned earlier, there are two possible positions for `Work[i]`: (1) `Work[i-Num[i]]` and (2) `Work[i-Num[i]+1]`. Our algorithm implements the first case. It is not difficult to modify the algorithm to implement the second. Or, one can even use a random number to choose `Work[i-Num[i]]` or `Work[i-Num[i]+1]`. The third `FOR` stores the actual values $(1, 2, \ldots, n)$ into the output array `Out`.

Note that since the maximum number of comparisons required to process a_i is $i-1$, $0 \leq n_i \leq i-1$ must hold. Therefore, in the algorithm shown above, if `Num[i]` is not in this range, `Work[i]` will not be processed.

Table 1 shows the results generated by the above algorithm for $n = 10$ and various forms of n_i. Note that since `Work[i]` is always moved to `Work[i-Num[i]]`, the generated sequence for $n_i = 1$ is not the usual reversed-order sequence. Storing `Work[i]` into `Work[i-Num[i]+1]` yields the desired result; however, for the case of $n_i = i-1$, the generated sequence becomes 1, 10, 9, 8, 7, 6, 5, 4, 3, 2.

The number of shifts is easy to determine. In Figure 1, data items `Work[i-Num[i]]`, …, `Work[i-1]` are always shifted one position to the right. Hence, the number of shifts is equal to the number of comparisons. If `Work[i]` is inserted into `Work[i-Num[i]+1]`, the number of shifts required to process `Work[i]` is `Num[i]-1`.

3 Conclusion

In this paper we have successfully presented an algorithm that can systematically generate input sequences for insertion sort with the number of comparisons in the range of $O(n)$ and $O(n^2)$. We believe that our results will be very helpful to students in a CS2 class.

Our technique can be easily adopted to bubble sort as well. As a natural next step, we hope similar algorithms could be developed for other sorting algorithms such as quicksort and shellsort, although the worst case complexity of shellsort is still unknown.

References

[1] Sara Baase, *Computer Algorithms*, second edition, Addison-Wesley, 1988.

[2] Douglas W. Nance, *Understanding Turbo Pascal: Programming and Problem Solving*, West Publishing Co., St. Paul, MN, 1994.

[3] Robert Sedgewick, *Algorithms*, second edition, Addison-Wesley, 1988.