# Insertion Sort 4

## 4.1 Insertion Sort

The problem of sorting is defined as:

- **Input**: $n$ integers in array $A[1..n]$
- **Output**: $A$ sorted in increasing order

Here we present INSERTION-SORT, which is an efficient algorithm for sorting a small number of elements. Insertion sort works the way many people sort a hand of playing cards. We start with an empty left hand and the cards face down on the table. We then remove one card at a time from the table and insert it into the correct position in the left hand. To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left. At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

We present our pseudocode for INSERTION-SORT, which takes as a parameter an array $A[1..n]$ containing a sequence of length $n$ that is to be sorted. Note that in the code, $n = A.length$. The algorithm sorts the input numbers **in place**. That is, it rearranges the numbers within the array $A$, with at most a constant number of them stored outside the array at any time. The input array $A$ contains the sorted output sequence when the INSERTION-SORT procedure is finished.

```
INSERTION-SORT($A$):
  for $j \leftarrow 2$ to $A.length$
      $key = A[j]$
      // Insert $A[j]$ into the sorted sequence $A[1..j-1]$
      $i = j - 1$
      while $i > 0$ and $A[i] > key$
          $A[i+1] = A[i]$
          $i = i - 1$
      $A[i+1] = key$
```

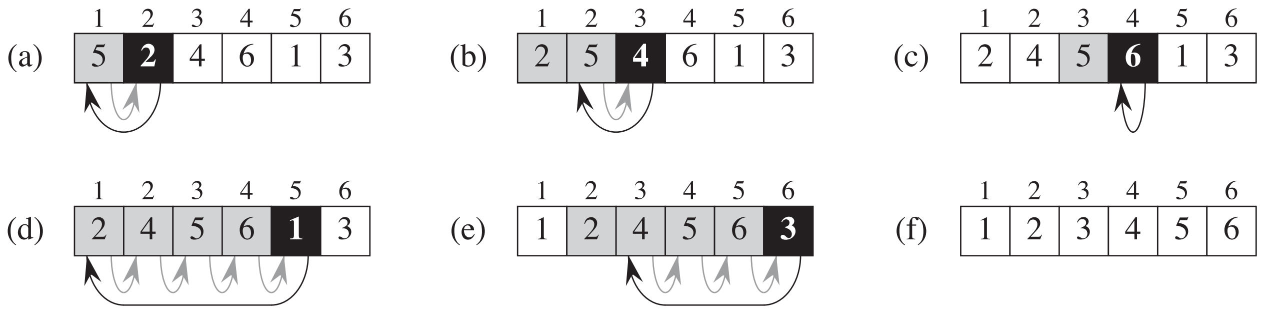

**Figure 4.1:** From CLRS, the operation of INSERTION-SORT on the array $A = [5, 2, 4, 6, 1, 3]$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)-(e)** The iterations of the **for** loop of lines 1-8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

These notes were adapted from CLRS Chapter 2

Now, we must ask ourselves two questions: does this algorithm work, and does it have good performance?

## 4.2 Correctness of Insertion Sort

Once you figure out what INSERTION-SORT is doing, you may think that it's "obviously" correct. However, if you didn't know what it was doing and just got the above code, maybe this wouldn't be so obvious. Additionally, for algorithms that we'll study in the future, it won't always be obvious that it works, and so we'll have to prove it. To warm us up for those proofs, let's carefully go through a proof of correctness of INSERTION-SORT.

We'll prove the correctness of INSERTION-SORT formally using a **loop invariant**:

> At the start of each iteration of the **for** loop of lines 1-8, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

To use loop invariants, we must show three things:

1. **Initialization**: It is true prior to the first iteration of the loop.
2. **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration
3. **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Note that this is basically **mathematical induction**: the initialization is the base case, and the maintenance is the inductive step).

In the case of insertion sort, we have:

**Initialization:** Before the first iteration (which is when $j = 2$), the subarray $A[1..j-1]$ is just the first element of the array, $A[1]$. This subarray is sorted, and consists of the elements that were originally in $A[1..1]$.

**Maintenance:** Suppose $A[1..j-1]$ is sorted. Informally, the body of the for loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$ and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4-7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order. Incrementing $j$ for the next iteration of the for loop then preserves the loop invariant.

**Termination:** The condition causing the for loop to terminate is that $j > n$. Because each loop iteration increases $j$ by 1, we must have $j = n + 1$ at that time. By the initialization and maintenance steps, we have shown that the subarray $A[1..n+1-1] = A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

## 4.3 Running Time of Insertion Sort

The running time of an algorithm on a particular input is the number of primitive operations or "steps" executed. We assume that a constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the $i^{\text{th}}$ line takes time $c_i$, where $c_i$ is a constant.

```
INSERTION-SORT(A)                                  cost    times
1  for $j = 2$ to $A.length$                       $c_1$   $n$
2      $key = A[j]$                                $c_2$   $n-1$
3      // Insert $A[j]$ into the sorted
           sequence $A[1..j-1]$.                   0       $n-1$
4      $i = j - 1$                                 $c_4$   $n-1$
5      while $i > 0$ and $A[i] > key$              $c_5$   $\sum_{j=2}^{n} t_j$
6          $A[i+1] = A[i]$                         $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
7          $i = i - 1$                             $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
8      $A[i+1] = key$                              $c_8$   $n-1$
```

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes $c_i$ steps to execute and executes $n$ times will contribute $c_i n$ to the total running time.

To compute $T(n)$, the running time of INSERTION-SORT on an input of $n$ values, we sum the products of the *cost* and *times* columns. Let $t_j$ represent the number of times the **while** loop test is executed on the $j^{\text{th}}$ iteration of the **for** loop.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + +c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

So if I asked you how long an algorithm takes to run, you'd probably say, "it depends on the input you give it." In this class we mostly consider the "worst-case running time", which is the longest running time for any input of size $n$.

For insertion sort, what are the worst and best case inputs? The best case input is a sorted array, because you never have to move elements, and the worst case input is a reverse sorted array (i.e., decreasing order).

In the best case, $t_j = 1$ for all $j$, and thus we can rewrite $T(n)$ as:

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

We can express this running time as $an + b$ for constants $a$ and $b$ that depend on the statement costs $c_i$; it is thus a **linear function** of $n$.

In the worst case, we must compare each element $A[j]$ with each element in the entire sorted subarray $A[1..j-1]$, so $t_j = j$ for all $j$. Substituting in $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$, we can rewrite $T(n)$ as:

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

We can express this worst-case running time as $an^2 + bn + c$ for constants $a$, $b$, and $c$ that again depend on the statement costs $c_i$; it is thus a **quadratic function** of $n$.