Insertion Sort Algorithm

If the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place. This is called insertion sort. An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted).

Insertion sort is an example of an incremental algorithm; it builds the sorted sequence one number at a time. This is perhaps the simplest example of the incremental insertion technique, where we build up a complicated structure on n items by first building it on n − 1 items and then making the necessary changes to fix things in adding the last item. The given sequences are typically stored in arrays. We also refer the numbers as keys. Along with each key may be additional information, known as satellite data.

Algorithm: Insertion Sort -

It works the way you might sort a hand of playing cards:

1. We start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].
2. Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].
3. To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.

Note that 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.

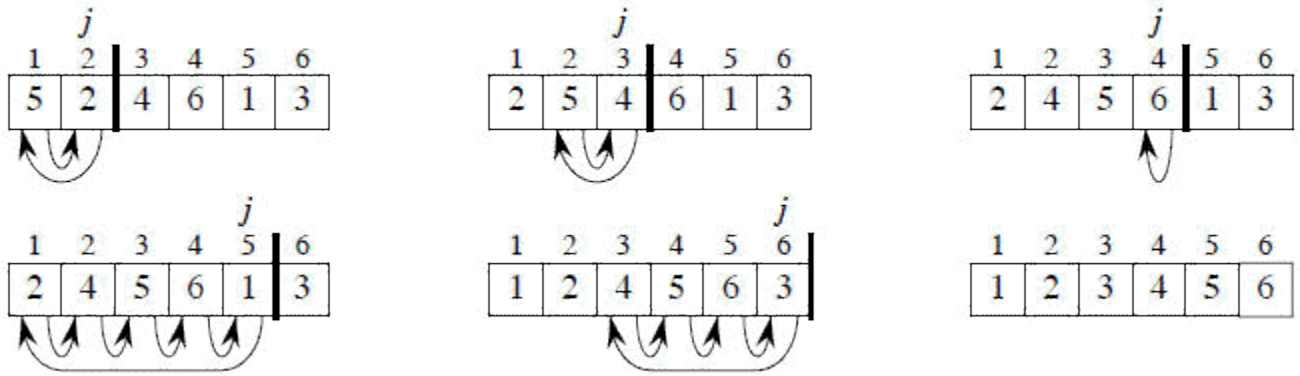
Pseudocode

We use a procedure INSERTION_SORT. It takes as parameters an array $A[1..n]$ and the length n of the array. The array $A$is sorted in place: the numbers are rearranged within the array, with at most a constant number outside the array at any time.

```
INSERTION_SORT (A)
1.    FOR j ← 2 TO length[A]
2.          DO  key ← A[j]
3.              {Put A[j] into the sorted sequence A[1 . . j − 1]}
4.              i ← j − 1
5.              WHILE i > 0 and A[i] > key
```

```
6.                    DO A[i +1] ← A[i]
7.                        i ← i − 1
8.              A[i + 1] ← key
```

Example: Following figure shows the operation of INSERTION-SORT on the array A= (5, 2, 4, 6, 1, 3). Each part shows what happens for a particular iteration with the value of *j* indicated. *j* indexes the "current card" being inserted into the hand.



Read the figure row by row. Elements to the left of *A*[*j*] that are greater than *A*[*j*] move one position to the right, and *A*[*j*] moves into the evacuated position.

Analysis

Since the running time of an algorithm on a particular input is the number of steps executed, we must define "step" independent of machine. We say that a statement that takes c_i steps to execute and executed n times contributes $c_i n$ to the total running time of the algorithm. To compute the running time, T(n), we sum the products of the cost and times column . That is, the running time of the algorithm is the sum of running times for each statement executed. So, we have

$T(n) = c_1 n + c_2 (n - 1) + 0 (n - 1) + c_4 (n - 1) + c_5 \sum_{2 \le j \le n} (t_j) + c_6 \sum_{2 \le j \le n} (t_j - 1) + c_7 \sum_{2 \le j \le n} (t_j - 1) + c_8 (n - 1)$

In the above equation we supposed that t_j be the number of times the while-loop (in line 5) is executed for that value of *j*. Note that the value of *j* runs from 2 to (n − 1). We have

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

A. Best-Case

The best case occurs if the array is already sorted. For each j = 2, 3, ..., n, we find that *A*[*i*] less than or equal to the key when *i* has its initial value of (j − 1). In other words, when $i = j - 1$, always find the key *A*[*i*] upon the first time the WHILE loop is run.

Therefore, $t_j = 1$ for $j = 2, 3, \ldots, n$ and the best-case running time can be computed using equation (1) as follows:

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

$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 (n - 1) + c_8 (n - 1)$

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

This running time can be expressed as $an + b$ for constants a and b that depend on the statement costs c_i. Therefore, $T(n)$ it is a linear function of n.

The punch line here is that the while-loop in line 5 executed only once for each j. This happens if given array A is already sorted.

$T(n) = an + b = O(n)$

It is a linear function of n.

B. Worst-Case

The worst-case occurs if the array is sorted in reverse order i.e., in decreasing order. In the reverse order, we always find that $A[i]$ is greater than the key in the while-loop test. So, we must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 .. j - 1]$ and so $t_j = j$ for $j = 2, 3, \ldots, n$. Equivalently, we can say that since the while-loop exits because i reaches to 0, there is one additional test after $(j - 1)$ tests. Therefore, $t_j = j$ for $j = 2, 3, \ldots, n$ and the worst-case running time can be computed using equation (1) as follows:

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

And using the summations, we have

$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{2 \le j \le n} [n(n+1)/2 + 1] + c_6 \sum_{2 \le j \le n} [n(n - 1)/2] + c_7 \sum_{2 \le j \le n} [n(n - 1)/2] + c_8 (n - 1)$

$T(n) = (c_5/2 + c_6/2 + c_7/2) n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8) n - (c_2 + c_4 + c_5 + c_8)$

This running time can be expressed as $(an^2 + bn + c)$ for constants a, b, and c that again depend on the statement costs c_i. Therefore, $T(n)$ is a quadratic function of n.

Here the punch line is that the worst-case occurs, when line 5 executed j times for each j. This can happens if array A starts out in reverse order

$T(n) = an^2 + bn + c = O(n^2)$

It is a quadratic function of n.

Worst-case and average-case Analysis :

We usually concentrate on finding the worst-case running time: the longest running time for any input size n. The reasons for this choice are as follows:

- The worst-case running time gives a guaranteed upper bound on the running time for any input. That is, upper bound gives us a guarantee that the algorithm will never take any longer.
- For some algorithms, the worst case occurs often. For example, when searching, the worst case often occurs when the item being searched for is not present, and searches for absent items may be frequent.

- Why not analyze the average case? Because it's often about as bad as the worst case.

  Example: Suppose that we randomly choose n numbers as the input to insertion sort.

  On average, the key in $A[j]$ is less than half the elements in $A[1 .. j - 1]$ and it is greater than the other half. It implies that on average, the while loop has to look halfway through the sorted subarray $A[1 .. j - 1]$ to decide where to drop key. This means that $t_j = j/2$.

  Although the average-case running time is approximately half of the worst-case running time, it is still a quadratic function of n.

  • Stability -

  Since multiple keys with the same value are placed in the sorted array in the same order that they appear in the input array, Insertion sort is stable.

  • Extra Memory -

  This algorithm does not require extra memory.

- For Insertion sort we say the worst-case running time is $\theta(n^2)$, and the best-case running time is $\theta(n)$.
- Insertion sort use no extra memory it sort in place.
- The time of Insertion sort is depends on the original order of a input. It takes a time in $\Omega(n^2)$ in the worst-case, despite the fact that a time in order of n is sufficient to solve large instances in which the items are already sorted.

Implementation

```
void insertionSort(int numbers[], int array_size)
{
  int i, j, index;

  for (i = 1; i < array_size; i++)
  {
        index = numbers[i];
        j = i;
        while ((j > 0) && (numbers[j − 1] > index))
        {
              numbers[j] = numbers[j − 1];
              j = j − 1;
        }
        numbers[j] = index;
  }
}
```

Source:

http://www.learnalgorithms.in/#