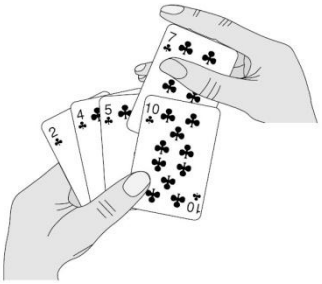
Insertion Sort



Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. In this algorithm each iteration removes an element from the input data and inserts it into the correct position in the list being sorted. The choice of element being removed from the input is random and this process repeated until all the input elements have been gone through. When people manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort. However, insertion sort provides several advantages:

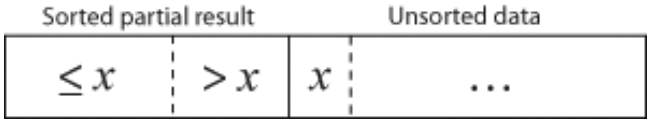
- Simple implementation: Bentley shows a three-line C version, and a five-line optimized version
- Efficient for (quite) small data sets
- More efficient in practice than most other simple quadratic (i.e., $O(n2)$) algorithms such as selection sort or bubble sort
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is $O(nk)$ when each element in the input is no more than k places away from its sorted position
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount $O(1)$ of additional memory space
- Online; i.e., can sort a list as it receives it

Algorithm

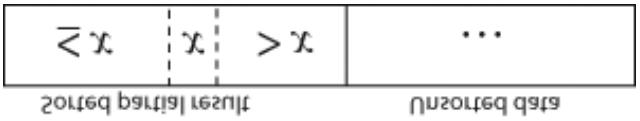
The most common variant of insertion sort, which operates on arrays, can be described as follows:

1. Suppose there exists a function called Insert designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.
2. To perform an insertion sort, begin at the left-most element of the array and invoke Insert to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:



Becomes



Pseudo code

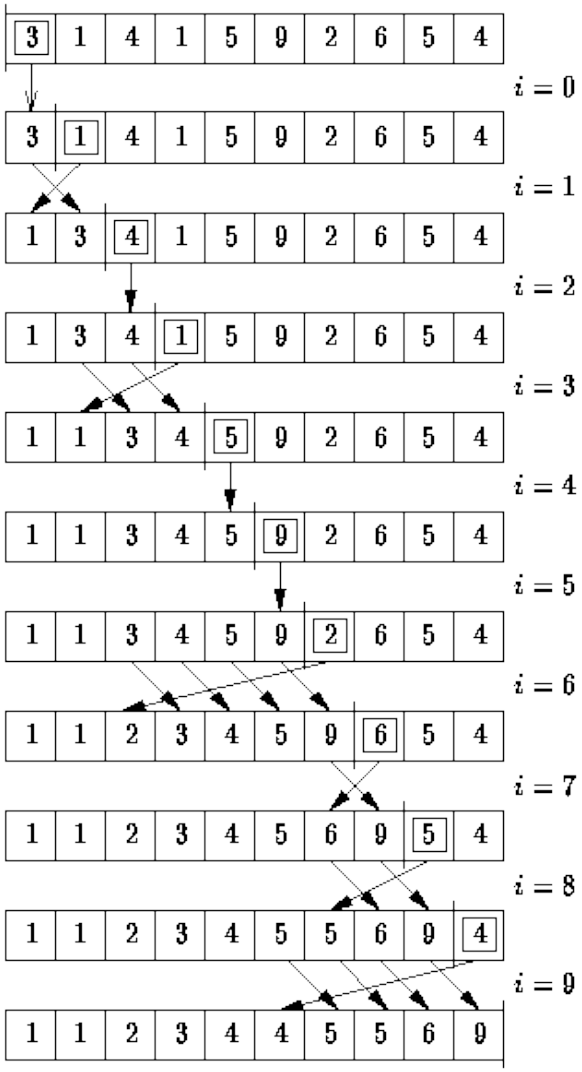
```
for i ← 1 to length(A)
        j ← i
        while j > 0 and A[j − 1] > A[j]
                swap A[j] and A[j − 1]
                j ← j – 1
```

.

Program

```
Void InsertionSort(int A[], int n){
        Int i, j, V;
        For (i = 2; i <= n − 1; i + +){
                V = A[i];
                J = I;
                While(A[j − 1] > V && j >= 1){
                        A[j] = A[j − 1];
                        j − −;
                }
                A[j] = v;
        }
}
```

Example



Analysis

Worst Case

Worst case occurs when for every I the inner loop has to move all elements $A[1] \ldots \ldots A[i-1]$ (which happens when $A[i] = key$ is smaller than all of them) , that takes O(i-1) time

$$T(n) = O(1) + O(2) + \cdots \ldots O(n-1) = O(1+2+3+4+\cdots..n-1) = O(n^2)$$

Average Case

For the Average case the inner loop will insert $A[i]$ in the middle of $[1], \ldots . A[i-1]$. this takes $O(i/2)$ Time.

$$T(n) = \sum_{i=1}^{n} O\left(\frac{i}{2}\right) = O(n^2)$$

Performance

Worst case	$O(n^2)$
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case Space Complexity	$O(n^2)$ total, $O(1)$ Auxiliary