**Sorting Algorithms**

Sorting is a fundamental operation in computer science. Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabetically, with character data.

There are many sorting algorithms. The particular algorithm one chooses depends on the properties of the data and the operations one may perform on the data. Accordingly, we will want to know the complexity of each algorithm; that is, we will want to know the running time *f(n)* of each algorithm as a function of the number *n* of input items.

Sorting frequently applies to a file of records. Each record in a file *F* can contain many fields, but there may be one particular field whose values uniquely determine the records in the file. Such a field *K* is called a primary key, and the values *k1, k2, …* in such a field are called keys or key values. Sorting the file *F* usually refers to sorting *F* with respect to a particular primary key.

Let *A* be a list of *n* elements *A1, A2, ….. An* in memory. Sorting *A* refers to the operation of rearranging the contents of *A* so that they are increasing order (numerically or lexicographically), that is so that

*A1 ≤ A2 ≤ A3 ≤ …≤ An*

Since *A* has *n* elements, there are *n!* ways that the contents can appear in *A*. These ways correspond precisely to the *n!* permutations of *1, 2, …, n*. Accordingly, each sorting algorithms must take care of these *n!* possibilities.

**Example**

Suppose an array DATA contains 8 elements as follows:

DATA: 77, 33, 44, 11, 88, 22, 66, 55

After sorting, DATA must appear in memory as follows:

DATA: 11, 22, 33, 44, 55, 66, 77, 88

Since DATA consists of 8 elements, there are 8! = 40,320 ways that the numbers 11, 22, …, 88 can appear in data.

**Complexity of Sorting Algorithms**

The complexity of a sorting algorithm measures the running time as a function of the number *n* of items to be sorted. We note that each sorting algorithm *S* will be made up of the following operations, where *A1, A2, …, An* contain items to be sorted and *B* is an auxiliary location:

1. Comparisons, which test whether *Ai* < *Aj* or test whether *Ai* < *B*
2. Interchanges, which switch the contents of *Ai* and *Aj* or of *Ai* and *B*
3. Assignments, which set *B = A* and then set *Aj = B* or *Aj = Ai*

Normally, the complexity function measures only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

There are two main cases whose main complexity we will consider: the worst case and the average case. In studying the average case, we make the probabilistic assumption that all the *n!* permutations of the given *n* items are equally likely.

I**nsertion Sort**

[Video](https://www.youtube.com/watch?v=ROalU379l3U)



This sorting algorithm is frequently used when *n* is small. For example, this algorithm is very popular with bridge players when they are first sorting their cards.

declare A[0..n-1] // This array is declared as zero-based for

//the purpose of this algorithm

function insertionSort(IN A[], In n) // n is the size of A

declare i, j, x, j

i ← 1

**while** i < n

j ← i

**while** j > 0 **and** A(j-1) > A(j)

CALL swap(A(j), A(j-1))

j ← j - 1

**end**

i ← i + 1

**end**

**return A[]**

**end**

procedure insertionSort(INOUT A[], In n) // n is the size of A

declare i, j, x, j

i ← 1

**while** i < n

j ← i

**while** j > 0 **and** A(j-1) > A(j)

CALL swap(A(j), A(j-1))

j ← j - 1

**end**

i ← i + 1

**end**

**end**

procedure swap(INOUT element1, INOUT element2)

declare temp

temp ← element1

element1 ← element2

element2 ← temp

end

**Example:**

The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the key under consideration is underlined. The key that was moved (or left in place because it was biggest yet considered) in the previous step is marked with an asterisk.

3 7 4 9 5 2 6 1

3\* 7 4 9 5 2 6 1

3 7\* 4 9 5 2 6 1

3 4\* 7 9 5 2 6 1

3 4 7 9\* 5 2 6 1

3 4 5\* 7 9 2 6 1

2\* 3 4 5 7 9 6 1

2 3 4 5 6\* 7 9 1

1\* 2 3 4 5 6 7 9

**Complexity of Insertion Sort**

The number of *f(n)* of comparisons in the insertion sort algorithm can easily be computed. First, the worst case occurs when the array is in reverse order and the inner loop must use the maximum number of *j - 1* comparisons. Hence,

*f(n)* 1 + 2 + … + (n - 1) = = *O(n2)*

Furthermore, one can show that, on the average, there will be approximately (j - 1)/2 comparisons in the inner loop. Accordingly, for the average case,

= *O(n2)*

Thus, insertion sort is a very slow algorithm when *n* is very large.

**Selection Sort**

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

Assume that the array A=[7,5,4,2] needs to be sorted in ascending order.

The minimum element in the array i.e. 2 is searched for and then swapped with the element that is currently located at the first position, i.e. 7. Now the minimum element in the remaining unsorted array is searched for and put in the second position, and so on.

Let’s take a look at the algorithm.

procedure selectionSort(INOUT A[], IN n)

declare minimum, i, j,

// minimum is a temporary variable to store the

// position of minimum element. It reduces the

// effective size of the array by one in each

// iteration.

for i ← 0 to (n - 1) by 1

// assuming the first element to be the minimum of the unsorted

// array.

minimum = i

// gives the effective size of the unsorted array.

For j ← i+1 to (n - 1) by 1

if(A(j) < A(minimum)) //finds the minimum element

minimum = j

endif

end

// putting minimum element on its proper position.

CALL swap (A(minimum), A(i))

end

end

At ith iteration, elements from position 0 to i−1 will be sorted.



**Complexity of the Selection Sort Algorithm:**

First note that the number of *f(n)* of comparisons in the selection sort algorithm is independent of the original order of the elements. That is, there are *n - 1* comparisons during Pass 1 to find the smallest element, there are *n- 2* comparisons during Pass 2 to find the second smallest element, and so on. Accordingly,

*f(n)* (n - 1) + (n -2) + … 2 + 1 = = *O(n2)*

**Remark:** The number of interchanges and assignments does depend on the original order of the elements in the array *A*, but the sum of these operations does not exceed a factor of n2.