INSERTION SORT

The basic step in this method is to insert a new record into a sorted sequence of *i* records in such a way that the resulting sequence of size *i* + 1 is also ordered.
Function *insert* accomplishes this insertion.

```
void insert(element e, element all, int i)
    {/* insert e into the ordered list a[1 : i] such that the resulting list a[1: i+1] is also
        ordered, the array a must have space allocated for at least i+2 elements */
        a[0] = e;
        while (e.key < a[i].key)
        {
            a[i+1] = a[i] ;
            i--;
        }
        a[i+1] = e;
    }
```

Program: Insertion into a sorted list

The use of *a*[0] enables us to simplify the while loop, avoiding a test for end of list (*i* < 1). In insertion sort, begin with the ordered sequence *a* [1] and successively insert the records *a* [2], *a* [3], ... , *a [n]*. Since each insertion leaves the resultant sequence ordered, the list with *n* records can be ordered making *n* - 1 insertions.
The details are given in function *insertionSort*.

```
void insertionSort(element all, int n)
    {    /* sort a[1: n] into nondecreasing order */
        int j;
        for (j = 2i j <= n : j++)
        {
            element temp = a[j];
            insert (temp, a, j-1);
        }
    }
```

Program: Insertion sort

Analysis of insertion Sort: In the worst case insert (e, a, i) makes i + 1 comparisons before making the insertion. Hence the complexity of Insert is O(i). Function insertionSort invokes insert for i = j - 1 = 1, 2, ... , n - 1. So, the complexity of insertionSort is

$$O(\sum_{i=1}^{n-1} (i+1)) = O(n^2).$$

The average time for insertionSort is $O(n^2)$

Example: Assume that n = 5 and the input key sequence is 5, 4, 3, 2, 1. After each iteration we have

j	[1]	[2]	[3]	[4]	[5]
–	**5**	4	3	2	1
2	**4**	**5**	3	2	1
3	**3**	**4**	**5**	2	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

Example: Assume that n = 5 and the input key sequence is 2, 3, 4, 5, 1. after each iteration we have

j	[1]	[2]	[3]	[4]	[5]
–	**2**	3	4	5	1
2	**2**	**3**	4	5	1
3	**2**	**3**	**4**	5	1
4	**2**	**3**	**4**	**5**	1
5	**1**	**2**	**3**	**4**	**5**

RADIX SORT

Radix sort is the method that many people intuitively use or begin to use when alphabetizing a large list of names. (Here the radix is 26, the 26 letters of the alphabet.) Specifically, the list of names is first sorted according to the first letter of each name. That is, the names are arranged in 26 classes, where the first class consists of those names that begin with "A," the second class consists of those names that begin with "B," and so on. During the second pass, each class is alphabetized according to the second letter of the name. And so on. If no name contains, for example, more than 12 letters, the names are alphabetized with at most 12 passes.

The radix sort is the method used by a card sorter. A card sorter contains 13 receiving pockets labelled as follows:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, 12, R (reject)

Each pocket other than R corresponds to a row on a card in which a hole can be punched. Decimal numbers, where the radix is 10, are punched in the obvious way and hence use only the first 10 pockets of the sorter. The sorter uses a radix reverse-digit sort on numbers. That is, suppose a card sorter is given a collection of cards where each card contains a 3-digit number punched in columns 1 to 3. The cards are first sorted according to the unit's digit. On the second pass, the cards are sorted according to the tens digit. On the third and last pass, the cards are sorted according to the hundreds digit.

Illustration with an example:

Suppose 9 cards are punched as follows:

348, 143, 361, 423, 538, 128, 321, 543, 366

Given to a card sorter, the numbers would be sorted in three phases, as pictured in

Input	0	1	2	3	4	5	6	7	8	9
348									348	
143				143						
361		361								
423				423						
538									538	
128									128	
321		321								
543				543						
366							366			

(a) First pass

Input	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366					543					
366							366			
348					348					
538				538						
128			128							

(b) Second pass

Input	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

(c) Third pass