APS106: FUNDAMENTALS OF COMPUTER PROGRAMMING
LAB # 7 - THURSDAY, MARCH 13, 12:00 - 2:00

This lab will test your ability to declare arrays, initialize arrays, and compute arrays using function calls. The extension of your source file must be .c, not .cpp (e.g. lab7.c).

BACKGROUND: Recently in class, you may have learned how an array can be sorted using the *Bubble Sort* algorithm. Another simple method to sort an array is the *Insertion Sort*. We illustrate this sorting algorithm by an example shown in Figure 1.

Suppose we are given the array [d c a b]. One variation of insertion sort starts by extracting the element in position 1 of the array (recall that array indexing in C starts at 0). As shown in Figure 1(a), in our example we extract the 'c'. We then compare 'c' to each of the elements located *above* its initial position in Figure 1(a), shifting the elements that are greater than 'c' down one by one, until we find the *correct* position for 'c'. (Equivalently, you can think of the elements "above" as the elements "to the left" and "below" as the elements "to the right" of a given position.) "Correct" position in this case means that the character above our chosen one is smaller than or equal to it (or you've reached the beginning of the array and there is no character above it), while the character below is greater than or equal to it (or you've reached the end of the array and there is no character below it). In our example, we compare 'c' to 'd'. Since 'c' is smaller than 'd', we shift 'd' down. At this point, we can't compare 'c' to anything else since we are at the beginning of the array. We therefore insert 'c' in the first position.

Next, we pick the character in the third position of the array. In this case, it is 'a', as demonstrated in Figure 1(b). We extract this character and then sequentially compare it to each character above the extracted one: 'a' is smaller than 'd', so we shift 'd' down; 'a' is smaller than 'c' so we shift 'c' down; now we are at the beginning of the array so we insert 'a' there.

Finally, in Figure 1(c), we complete the sort by inserting the 'b' into the correct place: we extract it from its initial position; compare it to 'd', shift 'd' down; compare it to 'c', shift 'c' down; compare it to 'a', find that we cannot shift 'a', and therefore insert 'b' between 'a' and 'c'.

Thus, on iteration k (k is >=1), we extract the element in position k of the array and find the correct relative position of the extracted element in the first k+1 positions of the array.

The insertion sort is an *in-place* sort, in that we sort the original array in-place, and do **not** require other (temporary) array.

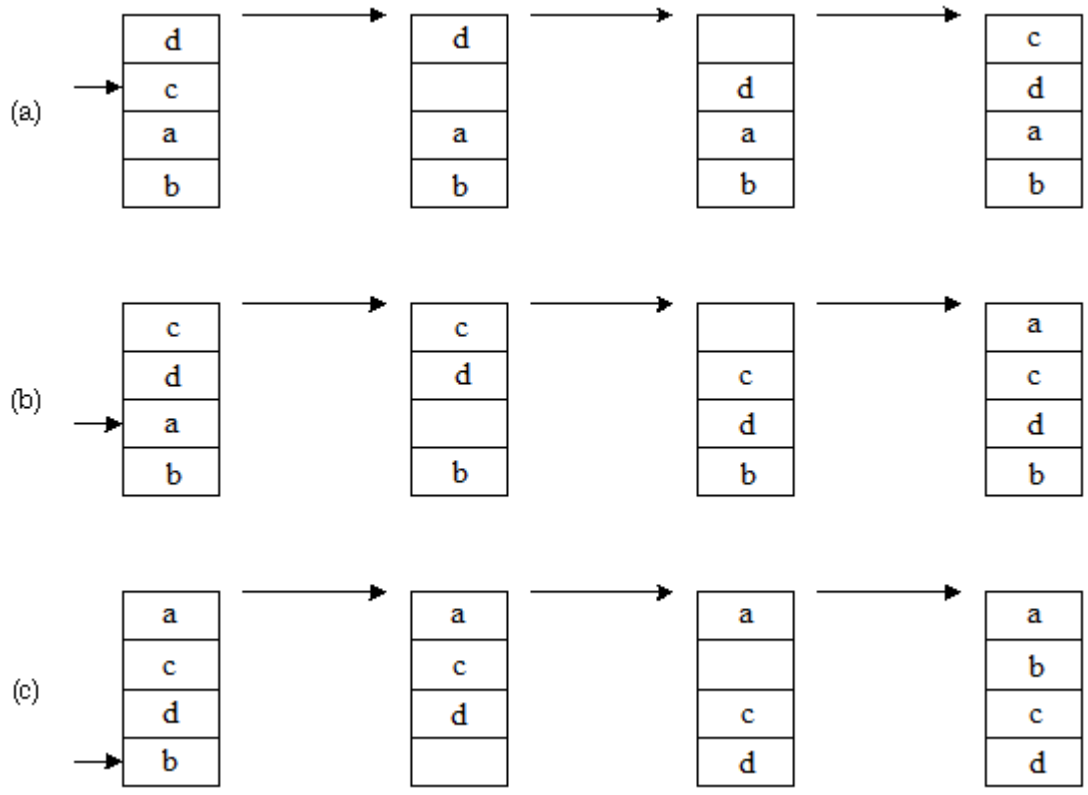

Figure 1: An example of *Insertion Sort*

PROBLEM: Write a function that implements the insertion sort algorithm described above. Assume that you have to sort an array of 10 characters. Your program must contain the following two functions:

```
void getText(char a[], int n);  /* get n characters from user */

void insertionSort(char a[], int n);  /* sort array a[] of n elements and
                                         print the sorted array */
```

Write a main() function to test your program. The array a[10] should be declared in main() and passed as an argument to the other functions.

HINTS:

Make sure you understand the logic of the sorting algorithm by working through the above example. Writing pseudo-code is also a good idea.

Assume that the user enters only lower case characters.

Use the scanf("%c", &a[i]) function to read the characters one by one.

Start sorting with position 1 of the array, as shown in the example above.

Your insertionSort() function should print the original array first, then sort the array, and finally print the sorted array. Use the printf("%c", a[i]) function to print the characters.