**Bubble Sort** is probably one of the oldest, easiest, straight-forward, but inefficient sorting algorithms. It is the algorithm introduced as a sorting routine in most introductory courses on Algorithms. Bubble Sort works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest of the list is "bubbled" to the end of the list whereas the smaller values sink to the bottom. It is similar to selection sort although not as straight forward. Instead of "selecting" maximum values, they are bubbled to a part of the list. An implementation in C:

```c
void bubbleSort(int a[], int array_size)
{
    int i, j, temp;
    for (i = 0; i < (array_size - 1); ++i)
    {
        for (j = 0; j < array_size - 1 - i; ++j )
        {
            if (a[j] > a[j+1])
            {
                temp = a[j+1];
                a[j+1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

```
8 6 10 3 1 2 5 4  } pass 0
6 8 3 1 2 5 4 1 0  } pass 1
6 3 1 2 5 4 8 1 0  } pass 2
3 1 2 5 4 6 8 1 0  } pass 3
1 2 3 4 5 6 8 1 0  } pass 4
1 2 3 4 5 6 8 1 0  } pass 5
1 2 3 4 5 6 8 1 0  } pass 6
1 2 3 4 5 6 8 1 0  } pass 7
```

The above tabulated clearly depicts how each bubble sort works. Note that each pass results in one number being bubbled to the end of the list.
The **Exchange Sort**, similar to its cousin, the bubble sort, in that it compares two elements, swapping them if necessary. The difference is the exchange sort compares the first item in the array with each of the remaining elements, making any necessary swaps. After the first pass through the array is complete, the exchange sort then takes the second element and compares it with each following element of the array, swapping when necessary. An implementation in C:

```c
void exchangeSort(int a[], int array_size) {
    int i, j, temp;
    for (i = 0; i < (array_size - 1); ++i) {
        for (j = i + 1; j < array_size; ++j) {
            if (a[i] > a[j]) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting):

- **Pass 0**: 8 6 10 3 1 2 5 4
- **Pass 1**: 1 8 10 6 3 2 5 4
- **Pass 2**: 1 2 10 8 6 3 5 4
- **Pass 3**: 1 2 3 10 8 6 5 4
- **Pass 4**: 1 2 3 4 10 8 6 5
- **Pass 5**: 1 2 3 4 5 10 8 6
- **Pass 6**: 1 2 3 4 5 6 10 8
- **Pass 7**: 1 2 3 4 5 6 8 10

The above tabulated clearly depicts how each exchange sort works. Note that each pass results in the lowest number being put in the first element, then the next lowest in the second element, etc.
The idea of **Selection Sort** is rather simple. It basically determines the minimum (or maximum) of the list and swaps it with the element at the index where its supposed to be. The process is repeated such that the nth minimum (or maximum) element is swapped with the element at the n-1th index of the list. The below is an implementation of the algorithm in C.

```c
void selectionSort(int a[], int array_size) {
    int i;
    for (i = 0; i < array_size - 1; ++i) {
        int j, min, temp;
        min = i;
        for (j = i+1; j < array_size; ++j) {
            if (a[j] < a[min])
                min = j;
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

Consider the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

<table>
<thead>
<tr>
<th>Pass</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8 6 10 3 1 2 5 4</td>
</tr>
<tr>
<td>1</td>
<td>1 6 10 3 8 2 5 4</td>
</tr>
<tr>
<td>2</td>
<td>1 2 10 3 8 6 5 4</td>
</tr>
<tr>
<td>3</td>
<td>1 2 3 10 8 6 5 4</td>
</tr>
<tr>
<td>4</td>
<td>1 2 3 4 8 6 5 10</td>
</tr>
<tr>
<td>5</td>
<td>1 2 3 4 5 6 8 10</td>
</tr>
<tr>
<td>6</td>
<td>1 2 3 4 5 6 8 10</td>
</tr>
<tr>
<td>7</td>
<td>1 2 3 4 5 6 8 10</td>
</tr>
</tbody>
</table>

At pass 0, the list is unordered. Following that is pass 1, in which the minimum element 1 is selected and swapped with the element 8, at the lowest index 0. In pass 2, however, only the sublist is considered, excluding the element 1. So element 2, is swapped with element 6, in the 2nd lowest index position. This process continues till the sub list is narrowed down to just one element at the highest index (which is its right position).
The **Insertion Sort** algorithm is a commonly used algorithm. Even if you haven’t been a programmer or a student of computer science, you may have used this algorithm. Try recalling how you sort a deck of cards. You start from the beginning, traverse through the cards and as you find cards misplaced by precedence you remove them and insert them back into the right position. Eventually what you have is a sorted deck of cards. The same idea is applied in the Insertion Sort algorithm. The following are two implementations in C.

```c
void insertionSort(int a[], int array_size)
{
    int i, j, temp;
    for (i = 1; i < array_size; ++i)
    {
        temp = a[i];
        for (j = i; j > 0 && a[j-1] > temp; j--)
            a[j] = a[j-1];
        a[j] = temp;
    }
}

void insertionSort(int a[], int array_size)
{
    int i, j, temp;
    for (i = 1; i < array_size; ++i)
    {
        if (a[i] < a[i - 1]) {
            temp = a[i];
            j = i;

            while (temp < a[j-1]) {
                a[j] = a[j - 1];
                j--;
            }  // end while
            a[j] = temp;
        }  // end if
    }  // end for
}
```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

```
8 6 1 0 3 1 2 5 4  pass 0
6 8 1 0 3 1 2 5 4  pass 1
6 8 1 0 3 1 2 5 4  pass 2
3 6 8 1 0 1 2 5 4  pass 3
1 3 6 8 1 0 2 5 4  pass 4
1 2 3 6 8 1 0 5 4  pass 5
1 2 3 5 6 8 1 0 4  pass 6
1 2 3 4 5 6 8 1 0  pass 7
```

The pass 0 is only to show the state of the unsorted array before it is given to the loop for sorting. Now try out the deck-of-cards-sorting algorithm with this list and see if it matches with the tabulated data. For example, you start from 8 and the next card you see is 6. Hence you remove 6 from its current position and "insert" it back to the top. That constituted pass 1. Repeat the same process and you'll do the same thing for 3 which is inserted at the top. Observe in pass 5 that 2 is moved from position 5 to position 1 since its < (6,8,10) but > 1. As you carry on till you reach the end of the list you'll find that the list has been sorted. It didn't take a course to tell you how to sort a deck of cards, did it; you prolly figured it out on your own. Amazed at the computer scientist in you? ;)}