Programming Notes 9: Multidimensional Arrays & Sorting

As usual, more examples of the notes are in the example program.

1. Multidimensional Arrays

Multidimensional arrays are just like the basic arrays we learned about last week, except they can hold values in more than one dimension. There are a few ways of thinking about it—the simplest is probably to think about a two dimensional array, for example, as simply an array of arrays, as that is how it works in your code. However, you could also think about it that a two dimensional array, again for example, stores a grid of values rather than simply a line of them. Either way can extend to more dimensions than two—for example, a three dimensional array is simply an array of arrays of arrays, or stores values in a three dimensional grid. It's not too complex if you think about it, although a spatial view doesn't work so well with arrays of greater than three dimensions.

2. Using Multidimensional Arrays

Now that you've learned the concept of multidimensional arrays, you should know how to use them. To declare a multidimensional array, it's almost exactly the same as a one-dimensional array: you just add the sizes of however many dimensions you want onto the end. Examples:

array2D[10][10]; Declares an array with dimensions 10 by 10.

array4D[10][10][10][10]; Declares a four dimensional array with dimensions 10 by 10 by 10 by 10.

Using multidimensional arrays is also almost exactly the same as one-dimensional arrays: you still reference each value by specifying the index, except now you must specify the index in each of the dimensions. Remember that I mentioned that for example, 2D arrays are really just arrays of arrays within your code? Well, this comes into play because if you only specify the first index in, again for example, a 2D array, you will get the entire array that goes with that first index. Likewise, if you only specified the first index in a 3D array, you will get a 2D array, and so on. Examples:

array2D[4][4]; Accesses the element at location 4, 4.

array4D[0][0][0]; Accesses the entire array stored at the location 0,0,0 in the 4D array.

Furthermore, passing multidimensional arrays to functions is, yet again, almost exactly the same as one dimensional arrays, although with one big catch. The catch is that you have to specify the length of every dimension except the first. This means that if you want to pass a 2D array into a function, you don't have to know how many arrays will be stored in the first dimension, but you have to know how many values will be in each stored array, or essentially how large the second dimension is. Finally, again, to pass an array into a function, simply pass the name with no brackets at all. Examples:

void func(int[][10]); A function prototype that will take an integer array with any first dimension size, but only a second dimension size of 10.

```
void func(int array[][10]) {
  //do stuff
}
```

This is the implementation of the function "func," which is as you'd expect.

```
int array2D[5][10];
func(array);
```

Finally, this calls the function with the 2D array "array2D."

Finally, *again* as with one-dimensional arrays, you're most often going to want to manipulate multidimensional arrays with loops. Nested loops are particularly useful for this purpose, as you can simply write "array[i][j]" and not have to worry about any fancy math. Example:

```
for(int i = 0; i < 10; i++) {
  for(int j = 0; j < 10; j++) {
    array2D[i][j] = 0;
  }
}
```

Sets all values in the 2D array with dimensions 10 by 10 to 0. In this example, the program will first loop though all 10 values in array2D[0], then array2D[1], and so on, until all 100 values have been passed over.

3. Arrays of c-strings

Another quick note here: an array of c-style strings is technically just a two-dimensional array of characters, but you can think of it of simply an array of c-strings, which is, again, technically an array of arrays. Elements of an array of c-strings can be used in any way you'd otherwise use a c-string, but keep in mind that when you specify the index of the first dimension, you are getting the entire array at that index. Examples:

```
char cstringArr[10][50];
```
Declares an array of 10 c-strings, which each have a size of 50 chars.

```
for(int i = 0; i < 10; i++) {
  cout << cstringArr[i] << endl;
}
```

This will loop though the array of 10 c-strings and output each one. Remember that you can use each element of the first dimension of the c-string array as a c-string.

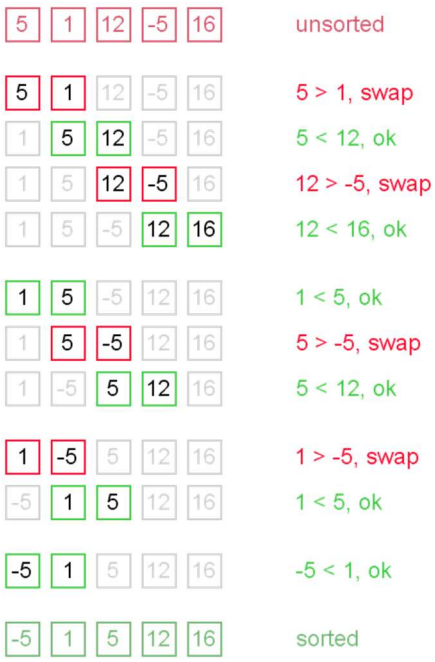
4. Sorting

Moving right along, in the previous few classes I've mentioned ways of sorting arrays. Sorting is simply ordering the values of the array in a particular sequence, for example lowest to highest or vice versa. Now, we'll go over a few different methods of sorting.

5. Bubble Sort

The bubble sort is the simplest of sorts to write code for, but it is also one of the most inefficient. Essentially, the algorithm will loop through the array, and whenever it finds a pair of values where one is larger than the next, it will swap the values and keep going. This means that higher values will "bubble up" in the array, until they find their correct places. Then, it will go back to the beginning and do the same thing again, and again, until the entire array is sorted.

See the graphic:

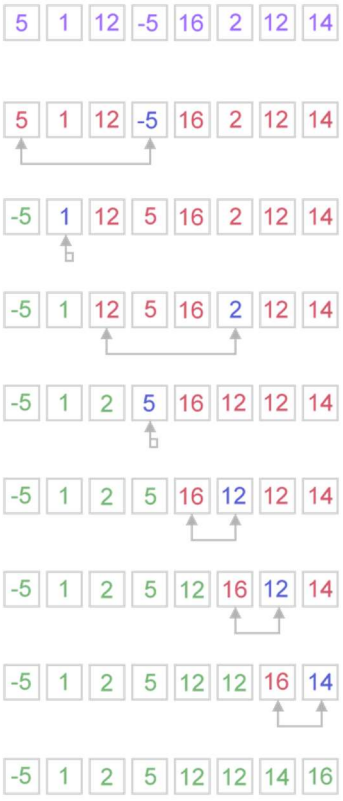


Example of a bubble sort implementation is in the example code.

6. Selection Sort

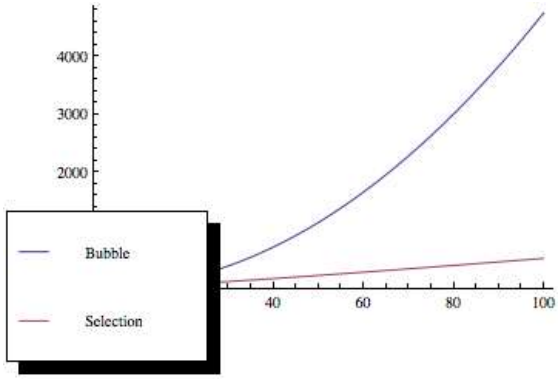
The selection sort is more efficient than the bubble sort and easier to understand, but is a bit more complicated to write the code for. The selection sort algorithm will simply find the lowest value in the array, put it in the first spot, find the next lowest value in the array, but it in the second spot, and so on. To do this, your code will want to first find the lowest value in the entire array, and swap it with the first value in the array. Then your program can do the same thing, except only start searching for the lowest value at the second element, and when you find it, put it in the second slot. Then, start searching from the third value, and so on.

See the graphic:



Example of a selection sort implementation is in the example code.

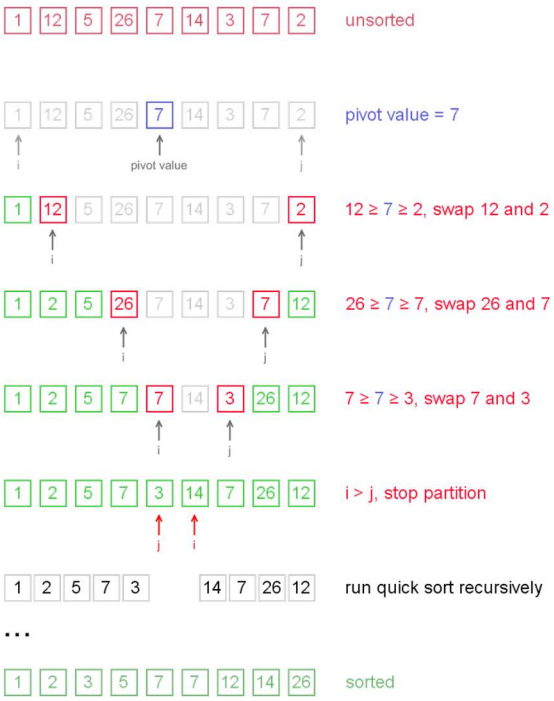
Because the selection sort only has to iterate over the array the number of elements in the array, it quickly becomes much less intensive for your computer compared to a bubble sort, where the number of iterations can be much higher than the number of elements. In fact, the graph of effort required by your computer compared to the array size, per se, looks something like this for a bubble and selection sort:



7. Quicksort

The last sorting algorithm we'll be quickly covering is the quicksort algorithm, which is the most efficient of the three we've seen, but also by far the most complicated to write the code for. For this reason, we'll just be going over the concept, although an example implementation is in the example code. Essentially, the quicksort algorithm sorts the array into two parts (usually called "partitions"), where the first partition holds all the lowest values, and the second holds all the highest values. Then, it does the same operation on each partition, so the lower and higher values are split into partitions within the first and second partition. This process is repeated until the partition size is just one element, and consequently the array has been sorted.

See the graphic:



Example of a quicksort implementation is in the example code.

8. Other Sorting Algorithms

We've only gone over three of many different types of sorting algorithms, so if you'd like to learn more, some other types are:

- Insertion sort – the selection sort, but a bit better
- Modified Bubble sort – the bubble sort, but a bit better
- Exchange sort – similar to the bubble sort, but compares elements to the first value
- Shell sort – similar to the bubble sort, but compares elements that are a certain distance away from each other
- Heap sort – uses heap storage
- Merge sort – similar to quicksort, but combines sorted arrays and can be used with linked lists.