Lab 8: Searching and Sorting

Searching

"It’s very common for programs not only to store and process data stored in arrays, but also to search arrays for specified items." Tony Gaddis, Starting Out in C++

You learned earlier in the class that an "algorithm" is a step by step list of instructions for completing some task. Search algorithms are one of the simplest and most common varieties of algorithms in computer science. Any sorting algorithm should check elements of some collection (like an array) in some order until it has found the item it is searching for or has exhausted the collection.

Linear Search in the simplest common searching algorithm. To search an array, the stored elements are checked in ascending order starting from index 0. The function should return the first index that it finds its search value in. If the the algorithm reaches index=SIZE, then the entire list has been checked and the search value is not in the list. Consider this implementation:

```
//Code provided on page 464 of Starting out with C++ by Tony Gaddis

int linearSearch (const int arr[], int size, int value){
   int index = 0;
   int position = -1;
   bool found = false;

   while(index < size && !found){
      if(arr[index] == value) {
         found = true;
         position = index;
      }
      else{
         index++;
      }
   }
   return position;
}
```

Take a moment to walk through the code in the **linearSearch** function. The parameters are an array that will be searched, the size of that array, and a value to search for. The while loop will continue until either a number is found in **arr** that has the same value as the integer **value** or the last index in **arr** has been checked without finding the search value. If the value is not found then the function will return -1. If the value is found in **arr,** then the function will return the index of the matching number. -1 is not a valid index in C++ arrays so this value is used as an indicator that the value is not in the array.

Linear search is not the most efficient algorithm for searching but it is the easiest to implement.

Sorting

Another class of simple algorithm is sorting algorithms. These algorithms change the order of the values stored in an array (or any other collection) so that they are strictly ascending or descending. You’ve probably seen one of these algorithms operating when you sorted the items on a shopping website by price, or sorted a list of files alphabetically by name.

Bubble Sort is a very simple and common sorting. Like **Linear Search,** there are many sorting algorithms that are more efficient, but **Bubble Sort** remains popular because it is so easy to implement.

The basic concept of **Bubble Sort** is to compare each element in an array to it’s "neighbor" two at a time and swap the two elements if they’re out of order. On each pass through the array the largest element in an array is pushed into a the last position (or smallest element for descending order). Read through this implementation carefully.

```
//Code provided on page 464 of Starting out with C++ by Tony Gaddis

void bubbleSort(int array[], int size){
   int maxElement;
   int index;

   for(maxElement = size-1; maxElement >0; maxElement--){
      for(index = 0; index < maxElement; index++){
         if(array[index] > array[index+1]) {
            swap(array[index], array[index + 1]);
         }
      }
   }
}

void swap(int &a, int &b) {
   int temp = a;
   a = b;
   b = temp;
}
```

First take a look at the **swap** function. The parameters **a** and **b** are passed by reference so making changes to them inside of the function swap will change the variables that are passed as arguments by the calling function (in this case **bubbleSort**). After **swap** returns, the value that was stored in **a** will now be stored in **b** and the value that was stored in **b** will now be stored in **a**.

Now take a look at the function **bubbleSort**. The two nested for loops tell us that we are going to iterate over the values in **array** several times. The outer loop sets **maxElement** to the last index on the first run, and shifts if back by one position every iteration. The inner loop compares each value starting at index 0 to its neighbor (so 0 to 1, then 1 to 2, …) until the higher number is the same as **maxElement.** If they are out of order, they get swapped using the **swap** funciton.

That’s a lot to understand all at once so lets sort a short array manually using Bubble Sort. We will use an array with the numbers: {2, 3, 4, 1}. This array has a size of 4 so **maxElement** will start at 3. Underlining indicates that two numbers are being compared.

maxElement: 3
array = $\underline{2\ 3}$ 4 1 correct order, no swap

maxElement: 3
array = 2 $\underline{3\ 4}$ 1 correct order, no swap

maxElement: 3
array = 2 3 $\underline{4\ 1}$ out of order, swap and finish inner loop

maxElement: 2
array = $\underline{2\ 3}$ 1 4 correct order, no swap

maxElement: 2
array = 2 $\underline{3\ 1}$ 4 out of order, swap and finish inner loop

maxElement: 1
array = $\underline{2\ 1}$ 3 4 out of order, swap and finish inner loop

maxElement: 0
array = 1 2 3 4 finish outer loop

Notice that the inner loop ran for 3 iterations on the first iteration of the outer loop, then 2 on the 2nd iteration of the outer loop, then only once on the last iteration of the outer loop. Also notice notice that every time the inner loop terminates the biggest element in the remaining values is stored in the index **maxElement.** The largest values in the array "rise" through the array as the sorting progresses like a bubble. This is where **Bubble Sort** gets its name.