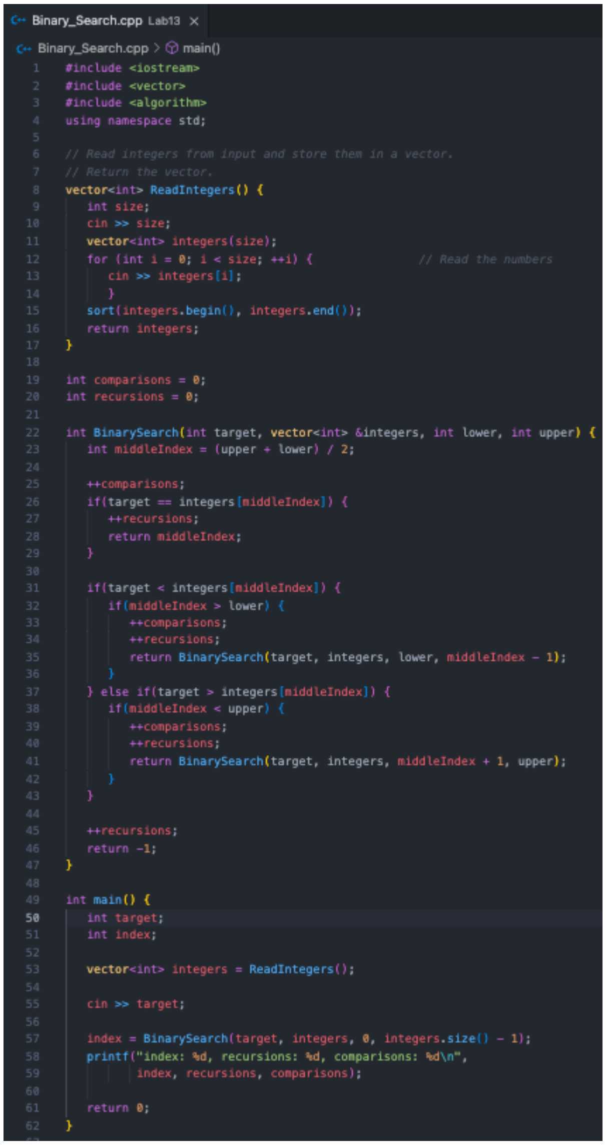
Assignment 1

Lab Explanation

The requirements for this lab were to create a program that would perform binary search on an inputted set of numbers from the user

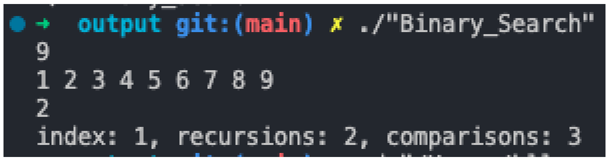
Code



```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Read integers from input and store them in a vector.
// Return the vector.
vector<int> ReadIntegers() {
   int size;
   cin >> size;
   vector<int> integers(size);
   for (int i = 0; i < size; ++i) {              // Read the numbers
      cin >> integers[i];
      }
   sort(integers.begin(), integers.end());
   return integers;
}

int comparisons = 0;
int recursions = 0;

int BinarySearch(int target, vector<int> &integers, int lower, int upper) {
   int middleIndex = (upper + lower) / 2;

   ++comparisons;
   if(target == integers[middleIndex]) {
      ++recursions;
      return middleIndex;
   }

   if(target < integers[middleIndex]) {
      if(middleIndex > lower) {
         ++comparisons;
         ++recursions;
         return BinarySearch(target, integers, lower, middleIndex - 1);
      }
   } else if(target > integers[middleIndex]) {
      if(middleIndex < upper) {
         ++comparisons;
         ++recursions;
         return BinarySearch(target, integers, middleIndex + 1, upper);
      }
   }

   ++recursions;
   return -1;
}

int main() {
   int target;
   int index;

   vector<int> integers = ReadIntegers();

   cin >> target;

   index = BinarySearch(target, integers, 0, integers.size() - 1);
   printf("index: %d, recursions: %d, comparisons: %d\n",
         index, recursions, comparisons);

   return 0;
}
```

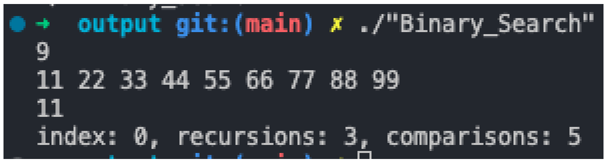
I added recursion to the binary search method as well as adding a base case. Increments to the recursion variable were placed around the function to ensure the logic of counting the number of recursive calls were correct. The binary search method works by simply cutting the search pool in half. The pool to use is determined whether the middle value of the search pool is greater than or less than the target value

Test 1



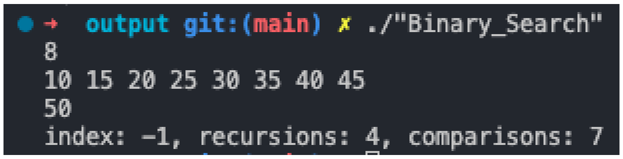
```
● → output git:(main) ✗ ./"Binary_Search"
  9
  1 2 3 4 5 6 7 8 9
  2
  index: 1, recursions: 2, comparisons: 3
```

Test 2



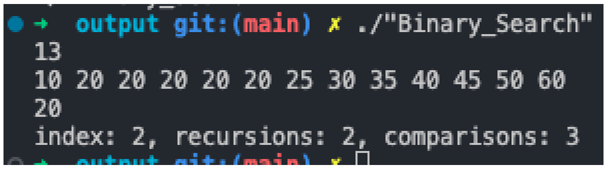
```
● → output git:(main) ✗ ./"Binary_Search"
  9
  11 22 33 44 55 66 77 88 99
  11
  index: 0, recursions: 3, comparisons: 5
```

Test 3



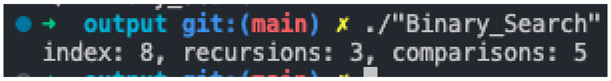
```
● → output git:(main) ✗ ./"Binary_Search"
  8
  10 15 20 25 30 35 40 45
  50
  index: -1, recursions: 4, comparisons: 7
```

Test 4

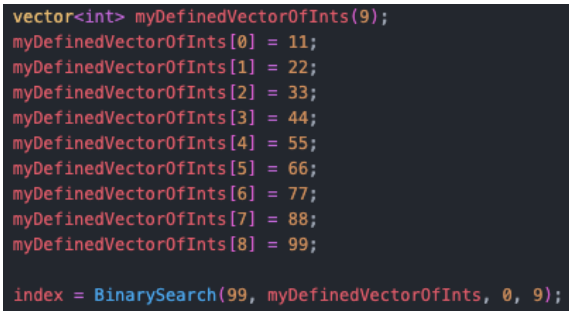


```
● → output git:(main) ✗ ./"Binary_Search"
  13
  10 20 20 20 20 20 25 30 35 40 45 50 60
  20
  index: 2, recursions: 2, comparisons: 3
○ → output git:(main) ✗ □
```

Test 5



```
● → output git:(main) ✗ ./"Binary_Search"
  index: 8, recursions: 3, comparisons: 5
```



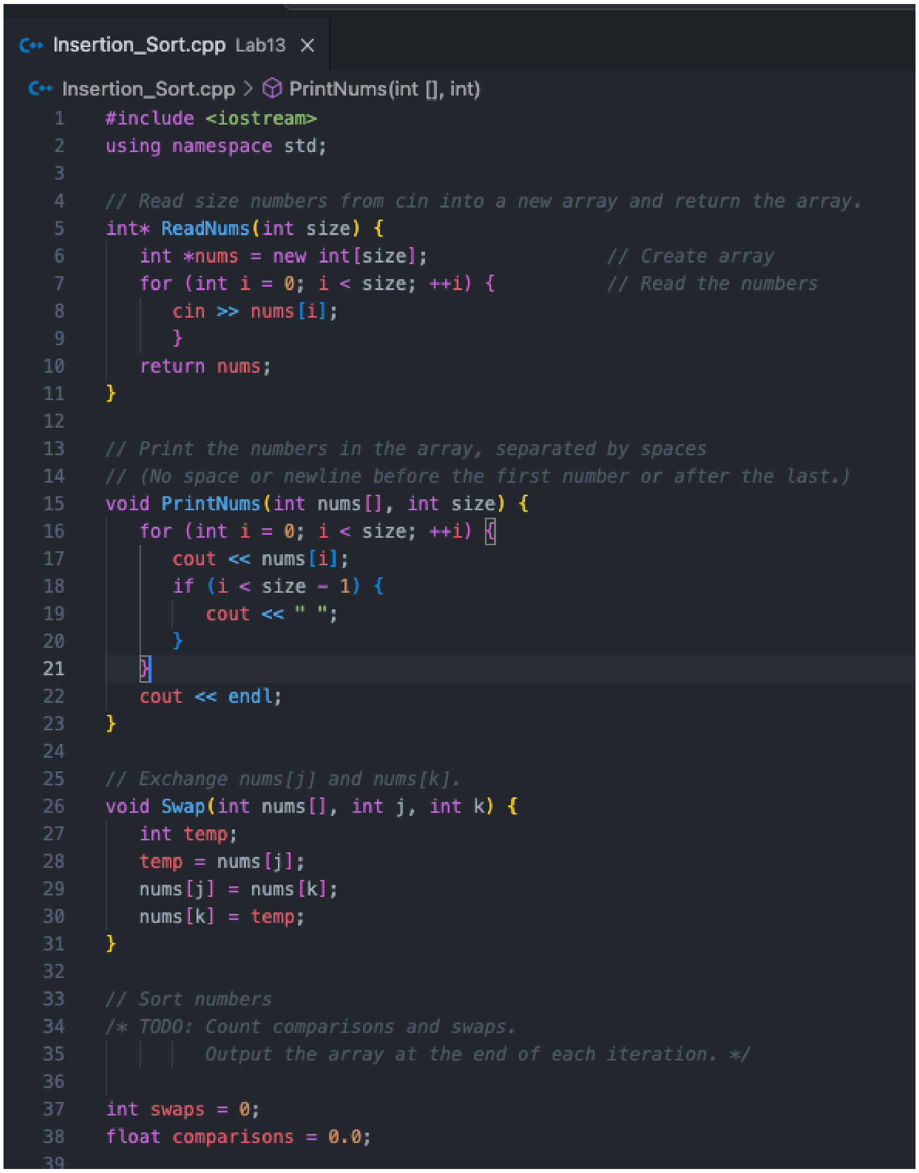
```
vector<int> myDefinedVectorOfInts(9);
myDefinedVectorOfInts[0] = 11;
myDefinedVectorOfInts[1] = 22;
myDefinedVectorOfInts[2] = 33;
myDefinedVectorOfInts[3] = 44;
myDefinedVectorOfInts[4] = 55;
myDefinedVectorOfInts[5] = 66;
myDefinedVectorOfInts[6] = 77;
myDefinedVectorOfInts[7] = 88;
myDefinedVectorOfInts[8] = 99;

index = BinarySearch(99, myDefinedVectorOfInts, 0, 9);
```

Assignment 2

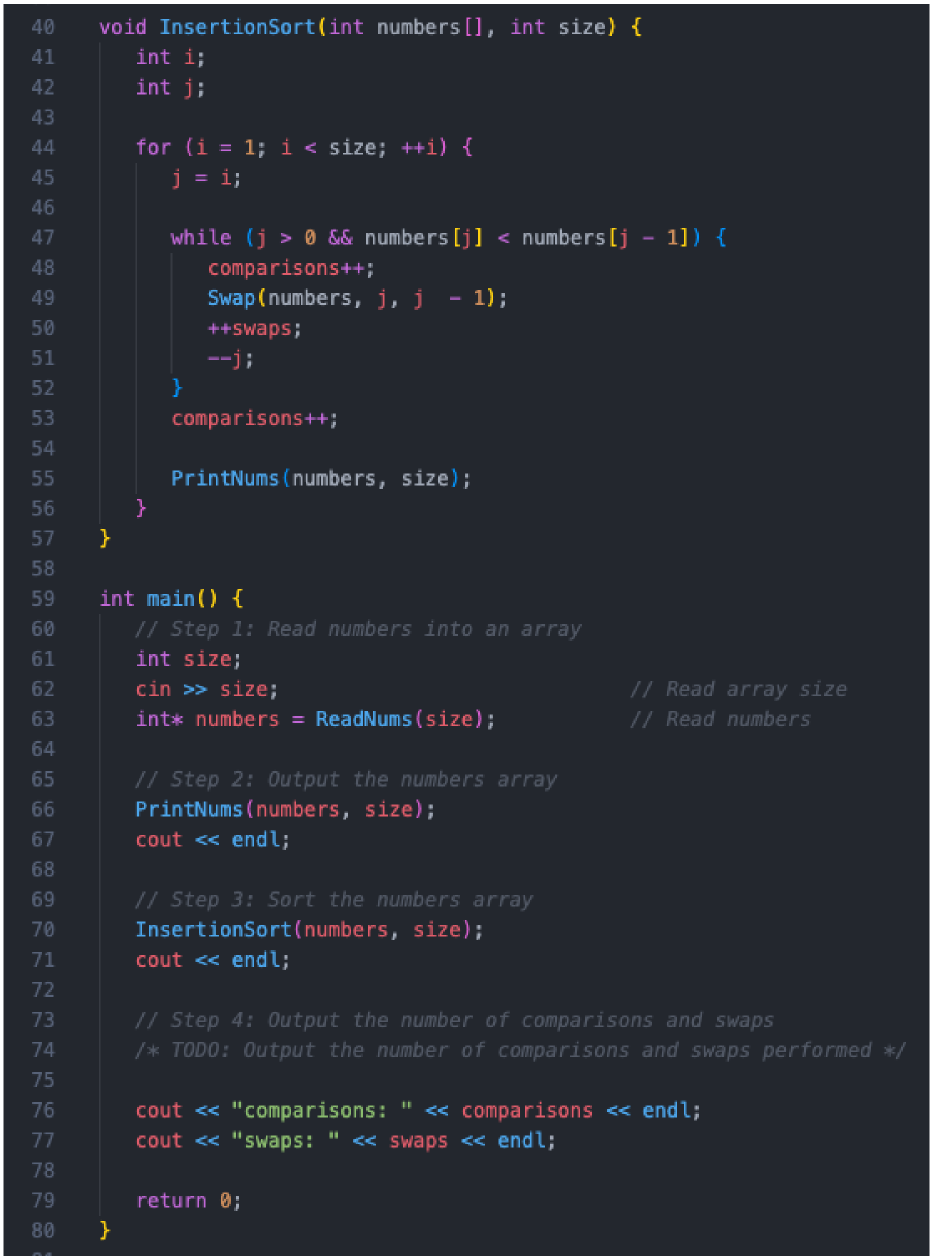
Lab Explanation

The requirements for this lab were to create a program that would sort an inputted list from the user using insertion sort.

Code



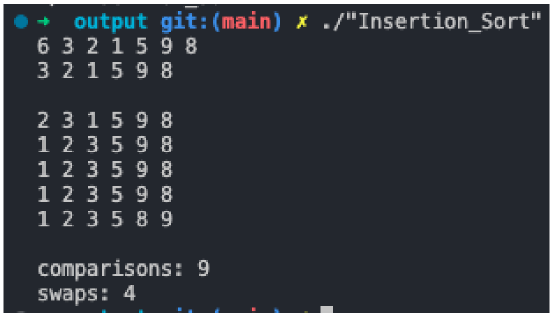
```cpp
#include <iostream>
using namespace std;

// Read size numbers from cin into a new array and return the array.
int* ReadNums(int size) {
   int *nums = new int[size];                 // Create array
   for (int i = 0; i < size; ++i) {           // Read the numbers
      cin >> nums[i];
      }
   return nums;
}

// Print the numbers in the array, separated by spaces
// (No space or newline before the first number or after the last.)
void PrintNums(int nums[], int size) {
   for (int i = 0; i < size; ++i) {
      cout << nums[i];
      if (i < size - 1) {
         cout << " ";
      }
   }
   cout << endl;
}

// Exchange nums[j] and nums[k].
void Swap(int nums[], int j, int k) {
   int temp;
   temp = nums[j];
   nums[j] = nums[k];
   nums[k] = temp;
}

// Sort numbers
/* TODO: Count comparisons and swaps.
         Output the array at the end of each iteration. */

int swaps = 0;
float comparisons = 0.0;
```



```cpp
void InsertionSort(int numbers[], int size) {
    int i;
    int j;

    for (i = 1; i < size; ++i) {
        j = i;

        while (j > 0 && numbers[j] < numbers[j - 1]) {
            comparisons++;
            Swap(numbers, j, j  - 1);
            ++swaps;
            --j;
        }
        comparisons++;

        PrintNums(numbers, size);
    }
}

int main() {
    // Step 1: Read numbers into an array
    int size;
    cin >> size;                                // Read array size
    int* numbers = ReadNums(size);              // Read numbers

    // Step 2: Output the numbers array
    PrintNums(numbers, size);
    cout << endl;

    // Step 3: Sort the numbers array
    InsertionSort(numbers, size);
    cout << endl;

    // Step 4: Output the number of comparisons and swaps
    /* TODO: Output the number of comparisons and swaps performed */

    cout << "comparisons: " << comparisons << endl;
    cout << "swaps: " << swaps << endl;

    return 0;
}
```

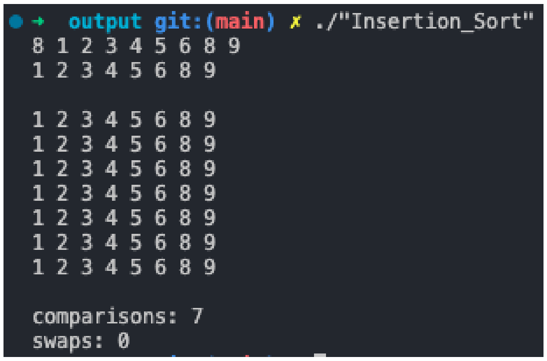
The insertion sort works by breaking a list into 2 sections. Sorted and unsorted. Incrementally, items are appended to the end of the sorted list. When an item is appended to the sorted side, the sorted side is searched linearly from the end to the beginning to see if any values must be swapped to keep the sorted side sorted. This repeats until there are no more items in the unsorted list.

Test 1



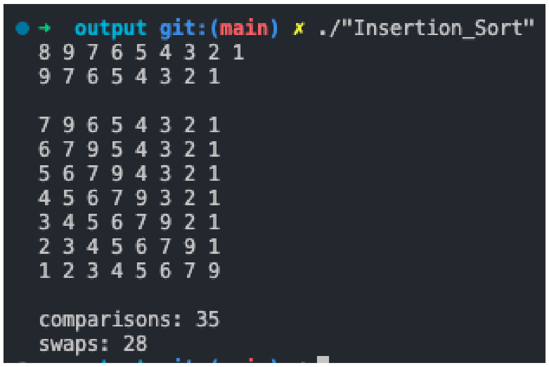
```
● ➜  output git:(main) ✗ ./"Insertion_Sort"
  6 3 2 1 5 9 8
  3 2 1 5 9 8

  2 3 1 5 9 8
  1 2 3 5 9 8
  1 2 3 5 9 8
  1 2 3 5 9 8
  1 2 3 5 8 9

  comparisons: 9
  swaps: 4
```

Test 2



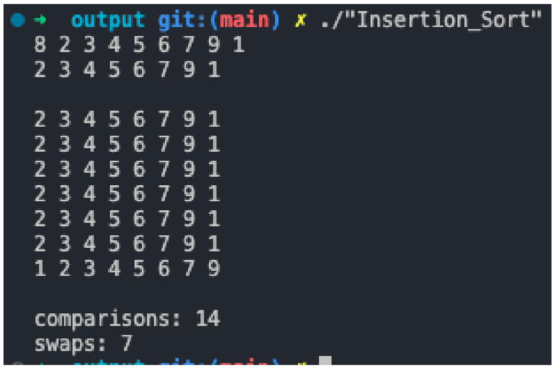
```
● ➜  output git:(main) ✗ ./"Insertion_Sort"
  8 1 2 3 4 5 6 8 9
  1 2 3 4 5 6 8 9

  1 2 3 4 5 6 8 9
  1 2 3 4 5 6 8 9
  1 2 3 4 5 6 8 9
  1 2 3 4 5 6 8 9
  1 2 3 4 5 6 8 9
  1 2 3 4 5 6 8 9
  1 2 3 4 5 6 8 9

  comparisons: 7
  swaps: 0
```

Test 3



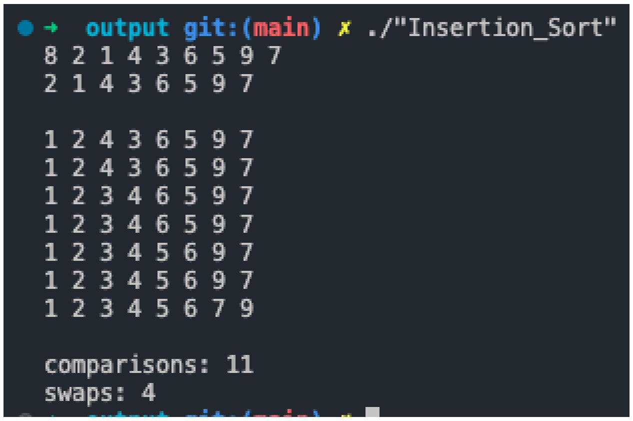
```
● ➜  output git:(main) ✗ ./"Insertion_Sort"
  8 9 7 6 5 4 3 2 1
  9 7 6 5 4 3 2 1

  7 9 6 5 4 3 2 1
  6 7 9 5 4 3 2 1
  5 6 7 9 4 3 2 1
  4 5 6 7 9 3 2 1
  3 4 5 6 7 9 2 1
  2 3 4 5 6 7 9 1
  1 2 3 4 5 6 7 9

  comparisons: 35
  swaps: 28
```

Test 4



```
● ➜  output git:(main) ✗ ./"Insertion_Sort"
  8 2 3 4 5 6 7 9 1
  2 3 4 5 6 7 9 1

  2 3 4 5 6 7 9 1
  2 3 4 5 6 7 9 1
  2 3 4 5 6 7 9 1
  2 3 4 5 6 7 9 1
  2 3 4 5 6 7 9 1
  2 3 4 5 6 7 9 1
  1 2 3 4 5 6 7 9

  comparisons: 14
  swaps: 7
```

Test 5



```
● → output git:(main) ✗ ./"Insertion_Sort"
  8 2 1 4 3 6 5 9 7
  2 1 4 3 6 5 9 7

  1 2 4 3 6 5 9 7
  1 2 4 3 6 5 9 7
  1 2 3 4 6 5 9 7
  1 2 3 4 6 5 9 7
  1 2 3 4 5 6 9 7
  1 2 3 4 5 6 9 7
  1 2 3 4 5 6 7 9

  comparisons: 11
  swaps: 4
● → output git:(main) ✗
```