CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment One – Data Structures and Sorting

Tyler DeLorey
tyler.delorey1@marist.edu

October 4th, 2024

Contents

1 C++ Basics

1.1 Introduction

I used C++ to complete this assignment. Before I start explaining about stacks and queues, palindromes, and each sort, I just want to justify some of the things I did that are C++ specific, like creating the header files, using vectors, and how I input the magicitems.txt file. This hopefully will explain some things that in my code that may seem odd, but are actually reasonable.

1.2 Header Files

The first thing I want to talk about are how I used header files. Header files in C++ (.h files) typically contain just function declarations and class definitions, while the corresponding .cpp file contains the implementations of those functions and classes. In my case, however, because most of the code is short and simple, I just put the implementations in the .h file and got rid of the corresponding .cpp files to avoid confusion. This is why there is only one .cpp file for this assignment (main.cpp). The five header files for this assignment are Node.h, Stack.h, Queue.h, isPalindrome.h, and Sorts.h. I will be explaining each in-depth in the future sections.

1.3 Makefile

In order to make compiling my code easier, I created a Makefile. By simply running the command "make" (assuming the g++ and make commands are installed on your system), the code will compile automatically and all output will return in the terminal. I also tried to make it compatible for Windows and Linux users, which took me hours to complete. This was difficult mostly due to the "make clean" command I implemented, which removes the unnecessary files created after the "make" command was run. Because these files differ between Windows and Linux, the command would have to run differently depending on the operating system you are using. Figure 1.1 below shows how I implemented Makefile for this assignment (sorry if syntax highlighting looks bad, it's configured for C++ code).

```
# Compiler
CXX = g++

# Compiler flags
CXXFLAGS = -Wall -g

# Target executable
TARGET = main

# Source files
SRCS = main.cpp
```

```
12
13 # Header files
14 HEADERS = Node.h Queue.h Stack.h isPalindrome.h Sorts.h
15
16 # Object files
17 OBJS = $(SRCS:.cpp=.o)
18
19 # Default rule to build and run the executable
20 all: $(TARGET) run
21
22 # Rule to link object files into the target executable
23 $(TARGET): $(OBJS)
24   $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJS)
25
26 # Rule to compile .cpp files into .o files, adding header
     dependencies
27 main.o: main.cpp $(HEADERS)
28   $(CXX) $(CXXFLAGS) -c main.cpp -o main.o
29
30 # Rule to run the executable
31 run: $(TARGET)
32   ./$(TARGET)
33
34 # Clean rule to remove generated files
35 # Uses del if on windows, uses rm on Unix-like systems (and Git
     Bash)
36 clean:
37   rm -f main $(OBJS) || del main.exe $(OBJS)
```

Figure 1.1: Makefile

The Makefile compiles the .cpp files into .o files with the header dependencies, then it runs the executable code. In order for the "make clean" command to work, I have to use the "del" command for Windows and "rm" command for Linux. I also have the specify the correct executable to remove. The executable differs depending on what operating system it is run on. On Windows, the executable is main.exe while it is just main on Linux systems.

1.4 Reading from File

I know this information isn't that at all exciting or necessary (sorry for this), but I had to learn how to implement C++ file IO in order to use the magicitems.txt file in my program. Figure 1.2 below shows how I read from the file.

```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4
5 #include "./isPalindrome.h"
6 #include "./Sorts.h"
```

```
8  using namespace std;
9
10 int main()
11 {
12     // Variable declaration
13     vector<string> magicItems;
14     string line;
15
16     string word;
17
18     int palindromeNum = 0;
19     int comparisonNum = 0;
20
21     // File IO
22     ifstream file;
23
24     // Open file
25     file.open("magicitems.txt");
26     if (!file.is_open())
27     {
28         cout << "File failed to open." << endl;
29         return 1;
30     }
31
32     // Read each line from file and put it into the magicItems
       vector (ArrayList)
33     for (int i = 0; getline(file, line); i++)
34     {
35         magicItems.push_back(line);
36     }
37
38     // Close file
39     file.close();
```

Figure 1.2: File Reading (main.cpp)

On Line 2, I included the file stream library, which provided classes for reading and writing files or data streams. I used the structure ifstream which can read from files on Line 22. I used the structure to open the file (Line 25). After making sure the file is opened and valid (Lines 26-30), I had to read each line from the file to put into an array (Lines 33-36). I used C++ vectors in order to store all of the data from the file.

1.5 Vectors

Vectors in C++ are like ArrayLists in Java. It is a dynamic array that can grow or shrink in size. To use, I had to include the vector library (as shown in Line 3 of Figure 1.2). Then I named the specified it to be a string vector (a vector that holds string) and named it magicItems (Line 13). In order to push items to the vector, I used the push_back function and pushed each line from the file I read (Line 35). After these steps, it mostly functions as any other array. For example, it uses bracket notation, indices, and it is hard to tell apart from a

normal array. If I refer to magicItems as an array in this documentation or comments in the code, I mean it's a vector. I like the use the word array since it's something I'm more familiar with.

2 Stacks and Queues

2.1 Node Class

Before I start explaining how I implemented my own Stacks and Queues in my program, I need to explain my Node class.

```
1  // Prevents multiple inclusions of same header file
2  #ifndef NODE_H
3  #define NODE_H
4
5  #include <iostream>
6
7  using namespace std;
8
9  // Public Node class for the linked list implementation of a stack
       and a queue
10 class Node
11 {
12     public:
13         // Holds a single character for this node
14         char data;
15
16         // Pointer that points to the next node
17         Node* next;
18
19         // Declares parameterized constructor
20         Node(char value)
21         {
22             this->data = value;
23             this->next = nullptr;
24         }
25 };
26
27 #endif
```

Figure 2.1: The Node Class (Node.h)

The Node class contains data and a next pointer, as defined on Lines 14 and 17 respectively. The data variable stores a single character for each Node, while the next pointer points to the next Node. When the Node function is called with a parameter value, the data gets set to that value, and the next pointer initially gets set as null (Lines 20-24). These pointers are adjusted when working with the Stack and Queue classes. With the Node class setup, a linked list can be created, where a Node has a pointer to another Node, and that Node has a pointer to another, and etc. This is how the Stacks and Queues are implemented.

2.2 Stack Class

The Stack class and its functions are all declared and initialized in my Stack.h file. My Stack has three functions associated with it: push, pop, and isEmpty.

```
#ifndef STACK_H
#define STACK_H

#include <iostream>
#include "./Node.h"

using namespace std;

// Public Stack class for linked list implementation of a stack
class Stack
{
    public:
        Node* top = nullptr;

        // Deconstructor for Stack
        ~Stack()
        {
            while (!isEmpty())
            {
                pop();
            }
        }

        // Push O(1)
        void push(char item)
        {
            Node* newNode = new Node(item);
            newNode->next = top;
            top = newNode;
            return;
        }

        // Pop O(1)
        char pop()
        {
            if (isEmpty())
            {
                throw runtime_error("Attempted to pop empty stack")
    ;
            }

            char poppedItem = top->data;
            Node* temp = top;
            top = top->next;

            // Free memory when popping the Node from the Stack
            delete(temp);

            return poppedItem;
        }
```

```
51          // isEmpty O(1)
52          bool isEmpty()
53          {
54              return (top == nullptr);
55          }
56  };
57
58  #endif
```

Figure 2.2: The Stack Class (Stack.h)

The Stack I implemented is essentially a linked list of Nodes with a pointer named "top" that points to the Node on top of the stack. The three main functions associated with this class have the time complexity of O(1), meaning they run in constant time.

- Line 25: The "push(char item)" function creates a new Node with the provided parameter as its data. The new Node's next pointer is set to the current top of the Stack (or nullptr if the Stack was previously empty). The new Node then becomes the new top of the stack.
- Line 34: The "pop()" function removes the top Node from the Stack and returns its data if it isn't empty. It deletes that Node from memory and sets the new top to the Node referenced by the previous top's next pointer.
- Line 52: The "isEmpty()" function checks whether the Stack is empty by seeing if the top pointer references a valid Node.

In order to avoid memory leaks, I had to also implement a deconstructor for the Stack (Line 15), which pops all Nodes from the Stack when the Stack object is destroyed. The deconstructor ensures that all Nodes are popped and deallocated from memory.

2.3 Queue Class

The Queue has a ton of similarities to the Stack class, so sorry if this information seems very repetitive. The Queue class and its functions are all declared and initialized in my Queue.h file. My Queue has three functions associated with it: enqueue, dequeue, and isEmpty.

```
1  #ifndef QUEUE_H
2  #define QUEUE_H
3
4  #include <iostream>
5  #include "./Node.h"
6
7  using namespace std;
```

```cpp
// Public Queue class for linked list implementation of a queue
class Queue
{
    public:
        Node* head = nullptr;
        Node* tail = nullptr;

        // Deconstructor for Queue
        ~Queue()
        {
            while (!isEmpty())
            {
                dequeue();
            }
        }

        // Enqueue O(1)
        void enqueue(char item)
        {
            Node* newNode = new Node(item);

            if (isEmpty())
            {
                head = newNode;
            }
            else
            {
                tail->next = newNode;
            }

            tail = newNode;
            return;
        }

        // Dequeue O(1)
        char dequeue()
        {
            if (isEmpty())
            {
                throw runtime_error("Attempted to dequeue empty queue");
            }

            char dequeuedItem = head->data;
            Node* temp = head;
            head = head->next;

            // Free memory when dequeuing the Node from the Queue
            delete(temp);

            return dequeuedItem;
        }
```

```
61          // isEmpty O(1)
62          bool isEmpty()
63          {
64              return (head == nullptr);
65          }
66  };
67
68  #endif
```

Figure 2.3: The Queue Class (Queue.h)

Like with the Stack class, the three main functions associated with Queue have the time complexity of O(1), meaning they run in constant time. Unlike the Stack however, I implemented the Queue class using two pointers to keep track of the Nodes in the linked list instead of just one. There is a head pointer which keeps track of the head of the list, and there is a tail pointer which keeps track of the tail of the list.

- Line 26: The "enqueue(char item)" function creates a new Node with the provided parameter as its data. If the Queue is not empty, the current tail's next pointer is set to the new Node, and the new Node becomes the new tail. If the queue is empty, the new Node becomes both the head and tail of the queue.
  The reason why I included the tail pointer was to make the enqueue function O(1) instead of O(n). If I only had a pointer to the head, I would have to traverse the entire linked list to get to the end of the list, which would be an O(n) operation.
- Line 44: The "dequeue()" function removes the head Node from the Queue and returns its data if it isn't empty. It deletes that Node from memory and sets the new head to the Node referenced by the previous head's next pointer.
- Line 62: The isEmpty() function checks whether the Queue is empty by seeing if the head pointer references a valid Node.

Like with the Stack class, I had to implement a deconstructor for the Queue (Line 17) to avoid memory leaks, which dequeues all Nodes from the Queue when the Queue object is destroyed. The deconstructor ensures that all Nodes are dequeued and deallocated from memory.

3 Palindromes

3.1 Introduction

The established Stack and Queue classes are used in order to check if words are palindromes or not. Figure 3.1 below shows partly how this is done.

```
41      // Checking for palindromes in the magic items array
42      cout << "PALINDROMES:" << endl;
43      for (int i = 0, arraySize = magicItems.size(); i < arraySize; i
        ++)
44      {
45          word = magicItems[i];
46
47          if (isPalindrome(word))
48          {
49              cout << word << endl;
50              palindromeNum++;
51          }
52      }
53      cout << "Number of Palindromes: " << palindromeNum << endl <<
        endl;
```

Figure 3.1: Palindrome Checking (main.cpp)

To check if each word in the magicItems array is a palindrome, the whole list has to be looped through (Line 43). Each word is passed through a "isPalindrome" function to check if it a palindrome (this function will be further discussed). If the function returns true, the word will be displayed and the palindrome counter will increase (Lines 49 and 50). After the loop completes, the program displays the number of palindromes on Line 53.

3.2 isPalindrome: Pushing and Enqueuing Letters

The isPalindrome function is located in the isPalindrome.h header file. Below is the first half of the function, which focuses on pushing letters onto the Stack and enqueuing letters to the Queue.

```
1  #ifndef ISPALINDROME_H
2  #define ISPALINDROME_H
3
4  #include <iostream>
5  #include "./Stack.h"
6  #include "./Queue.h"
7
8  using namespace std;
9
10 bool isPalindrome(string word)
11 {
12     // Create Stack and Queue for the word
13     Stack myStack;
14     Queue myQueue;
15
16     char sLetter;
17     char qLetter;
```

```
19      char letter;
20      int wordSize = word.size();
21      int sizeWithoutSpaces = 0;
22
23      bool palindrome = true;
24
25      // Add each letter of word to a Stack and Queue (ignoring
        spaces and capitalization)
26      for (int index = 0; index < wordSize; index++)
27      {
28          letter = toupper(word[index]);
29
30          if (letter != ' ')
31          {
32              myStack.push(letter);
33              myQueue.enqueue(letter);
34              sizeWithoutSpaces++;
35          }
36      }
```

Figure 3.2: Pushing and Enqueuing Letters (isPalindrome.h)

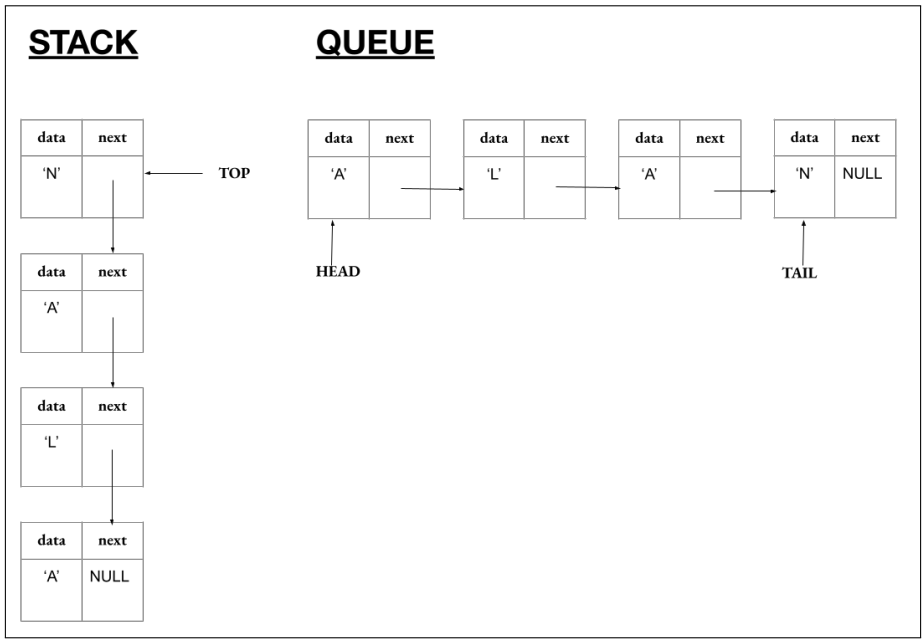

Figure 3.3: Stack and Queue Example for isPalindrome.h

Before pushing and enqueuing letters into the Stack and Queue, the classes need to be initialized. I created a Stack and a Queue named myStack and myQueue for the word passed into the function on Lines 13 and 14. The for-loop on Line

26 iterates through each letter in the word. In this program, palindromes ignore capitalization and spaces, so the program automatically converts each character to uppercase (Line 28) and checks if the character is not a space (Line 30). If the character is not a space, it is pushed onto the Stack and enqueued in the Queue (Lines 32 and 33). Each letter is stored as the data in a new Node. Once the for-loop is complete, the Stack and Queue will be full of Nodes representing the uppercase letters. The program will also have the size of the Stack and Queue, tracked by the "sizeWithoutSpaces" variable that increments each time a push and enqueue occurs (Line 34). Figure 3.3 above is an example of what the Stack and Queue would look like at the step if the parameter passed into the function was the word 'Alan'.

3.3 isPalindrome: Popping and Dequeuing Letters

After adding the valid letters as Nodes in the Stack and Queue, it is reasonable to pop and dequeue each letter one at a time and compare them. A palindrome is spelled the same forwards as it is backwards. Since the Stack is essentially reversing the word, the letter popped from the Stack should always match the letter dequeued from the Queue if the word is a palindrome.

```
38      // Pop from Stack, dequeue from Queue, compare letters to check
         if its palindromic
39      for (int index = 0; index < sizeWithoutSpaces; index++)
40      {
41          sLetter = myStack.pop();
42          qLetter = myQueue.dequeue();
43
44          if (sLetter != qLetter)
45          {
46              palindrome = false;
47              break;
48          }
49      }
50
51      return palindrome;
52  }
53
54  #endif
```

Figure 3.4: Popping and Dequeuing Letters (isPalindrome.h)

The loop on Line 39 iterates for each letter there is in the Stack and the Queue. The letter at the top of the Stack is popped and the letter at the head of the Queue is dequeued on Lines 41 and 42. Like stated earlier, if these letters don't match, the word is not a palindrome and the loop is broken out of (Lines 44-49) to return false. If the letters match, the word could possibly be a palindrome, but it has to keep iterating through the entire loop. If the letters always match throughout the entire loop, the function returns true, meaning that the word was indeed a palindrome. Looking back at Figure 3.3, the word 'Alan' is obviously

not a palindrome because 'N' (popped value from Stack) and 'A' (dequeued value from Queue) are not equal. For the case of magicitems.txt, out of the 666 "words" in the array, only 15 of them were palindromes. They are displayed in Table 1.

Palindromes
Boccob
Seuss Igniting Issues
UFO tofu
Ebuc Cube
Aibohphobia
Taco cat
Was It A Rat I Saw
Olah Halo
CD case divides ACDC
Dacad
Dior Droid
Robot Tobor
Narc in a panic ran
radar
Golf flog

Table 1: Palindromes in the magicitems.txt file

4 Sorting

4.1 Introduction

Now that the palindrome program is all figured out, it is time to explain the sorts. This assignment tasked us to do four different sorts: Selection Sort, Insertion Sort, Merge Sort, and Quick Sort. The sorts are implemented in the Sorts.h file and are run on the magicItems array. The figure below shows the sort functions being called in main, with the number of comparisons being displayed for each sort.

```
55      // Shuffles magic items
56      shuffle(magicItems);
57
58      // Performs Selection Sort O(n^2)
59      comparisonNum = selectionSort(magicItems);
60      cout << "Number of Comparisons doing Selection Sort: " <<
        comparisonNum << endl;
```

```
62      // Shuffles magic items
63      shuffle(magicItems);
64
65      // Performs Insertion Sort O(n^2)
66      comparisonNum = insertionSort(magicItems);
67      cout << "Number of Comparisons doing Insertion Sort: " <<
        comparisonNum << endl;
68
69      // Shuffles magic items
70      shuffle(magicItems);
71
72      // Performs Merge Sort O(n * log(n))
73      comparisonNum = mergeSort(magicItems);
74      cout << "Number of Comparisons doing Merge Sort: " <<
        comparisonNum << endl;
75
76      // Shuffles magic items
77      shuffle(magicItems);
78
79      // Performs Quick Sort O(n * log(n))
80      comparisonNum = quickSort(magicItems);
81      cout << "Number of Comparisons doing Quick Sort: " <<
        comparisonNum << endl;
82
83      cout << endl;
```

Figure 4.1: Calling Sort Functions (main.cpp)

Before each sort is called, the array is passed through a shuffle function which randomly rearranges the contents in the array. Then the array is passed through the sort functions, which returns the number of comparisons the sort had to make. Then, the comparison count is displayed in the terminal.
QUICK NOTE: The sorts work in ASCII-order, meaning all of the lowercase strings will be after the uppercase ones. All sorts are case sensitive. In the magicitems.txt file, there are only about nine strings that are lowercase, so the results shouldn't look super jarring.

4.2 Shuffle Function

Before calling each sort, the shuffle function is called, which shuffles the entire array. I did this by implementing a shuffle routine based on the Knuth, or Fisher-Yates, shuffle.

```
1  #ifndef SORTS_H
2  #define SORTS_H
3
4  #include <iostream>
5  #include <vector>
6  #include <random>
7  #include <climits>
```

```
8
9  using namespace std;
10
11 // Shuffles array based on Knuth shuffle
12 // Pass by reference (changes to myItems affects magicItems)
13 void shuffle(vector<string> &myItems)
14 {
15     int size = myItems.size();
16     int randomIndex = 0;
17     string temp;
18
19     for (int i = size - 1; i > 0; i--)
20     {
21         // Picks random index with random number generation
22         random_device rd;
23         mt19937 gen(rd());
24         uniform_int_distribution<> distrib(0, i);
25
26         randomIndex = distrib(gen);
27
28         // Swaps current value with random index value
29         temp = myItems[i];
30         myItems[i] = myItems[randomIndex];
31         myItems[randomIndex] = temp;
32     }
33
34 }
```

Figure 4.2: Shuffle Function (Sorts.h)

The shuffle function iterates through the entire array from the end to the beginning (Line 19). In each iteration, a random index from non-shuffled part of the array is chosen (Lines 22-26), and the item at that random index is swapped with the item at the current index (Lines 29-31). This makes it so the entire array is shuffled when the loop completes.
In order to compute the random index, instead of using the rand() function, I included the random library (Line 6). I realized that the rand() function isn't all that random because each time I compiled the code I would always get the same "random" results. With the random library, using the structures provided, it makes the random index TRULY random.

4.3 Selection Sort

Selection sort works by finding the smallest value in the unsorted section of the array, and swapping it with the first element of that section.

```
36 // Returns number of comparisons
37 // Pass by reference (changes to myItems affects magicItems)
38 int selectionSort(vector<string> &myItems)
39 {
```

```
40      int comparisonNum = 0;
41      int size = myItems.size();
42      string temp;
43
44      // Array from 0 to i is sorted
45      for (int i = 0; i < size - 1; i++)
46      {
47          int minIndex = i;
48
49          // Find minimum element in array
50          for (int j = i + 1; j < size; j++)
51          {
52              // Checks if current word is less than the current
    smallest word
53              if (myItems[j] < myItems[minIndex])
54              {
55                  minIndex = j;
56              }
57              comparisonNum++;
58          }
59
60          // Swap minimum element with element at i
61          if (minIndex != i)
62          {
63              temp = myItems[minIndex];
64              myItems[minIndex] = myItems[i];
65              myItems[i] = temp;
66          }
67      }
68
69      return comparisonNum;
70  }
```

Figure 4.3: Selection Sort (Sorts.h)

The outer loop (Line 45) iterates through the entire array, excluding the last index, with the variable i tracking the current position to be sorted. The inner loop (Line 50) iterates through the remaining unsorted part of the array to find the minimum value (Line 53). The index of this minimum value is stored in the minIndex variable (Line 55). After the inner loop completes, if the minimum value is not already at position i, the element at minIndex is swapped with the element at i (Lines 61-66), placing the smallest element in its correct position. Here is a sort result of running Selection Sort five times:

Input Size (n)	**Number of Comparisons**	**Time Complexity**
666	221,445 221,445 221,445 221,445 221,445	$O(n^2)$

Table 2: Selection Sort Results (Average Num of Comparisons: 221,445)

The time complexity of Selection Sort is $O(n^2)$ because there are nested loops. The outer loop iterates through the array (n - 1) times, and the inner loop iterates through the array (n - i - 1) times, meaning as the variable "i" increases, the inner loop iterates through less and less because more and more of the array is being sorted. When multiplying (n - 1) and (n - i - 1), and ignoring constant factors and only looking at the most dominant function of n, it results to n^2. Selection Sort will always have the same number of comparisons no matter how the list is sorted beforehand. The inner loop first makes n - 1 comparisons, then n - 2, then n - 3, all the way down to 1. This can all be summed up to $(n-1)(n-1+1)/2$, or $n(n-1)/2$, and when plugging in the input size of the magicItems array (666) for n, it results to 221445.

4.4 Insertion Sort

Insertion sort works by building a sorted section of the array one element at a time, repeatedly picking the next element and inserting it into its correct position relative to the already sorted elements.

```
72 // Returns number of comparisons
73 // Pass by reference (changes to myItems affects magicItems)
74 int insertionSort(vector<string> &myItems)
75 {
76     int comparisonNum = 0;
77     int size = myItems.size();
78 
79     // Traverses array from second element to last element (first
       element is sorted in an array of size 1)
80     for (int i = 1; i < size; i++)
81     {
82         string key = myItems[i];
83 
84         int end = i - 1;
85 
86         comparisonNum++;
87 
88         // Find place for the current key value
89         while (end >= 0 && myItems[end] > key)
90         {
91             // Shift element to the right if not correct position
92             myItems[end + 1] = myItems[end];
93             end--;
94 
95             comparisonNum++;
96         }
```

```
98          // Insert the key into correct position
99          myItems[end + 1] = key;
100     }
101
102     return comparisonNum;
103 }
```

Figure 4.4: Insertion Sort (Sorts.h)

The outer loop (Line 80) iterates through the entire array, excluding the first index (it's technically already sorted in an array of size 1), with the variable i tracking the current position to be sorted. The inner loop (Line 89) helps find the correct index to place the current "key" value. It achieves this by shifting rest of the element to the right if the position is not correct (Lines 92-93). Once the inner loop is completed, that means it found the correct position, and the key value is inserted there (Line 99). Here is a sort result of running Insertion Sort five times:

Input Size (n)	**Number of Comparisons**	**Time Complexity**
666	120,103 108,403 109,636 112,684 108,552	$O(n^2)$

Table 3: Insertion Sort Results (Average Num of Comparisons: 111,876)

Like Selection Sort, the time complexity of Insertion Sort is $O(n^2)$ because there are nested loops. The outer loop iterates through the array (n - 1) times. Depending on where the correct position is for the key value, the inner loop can run up to i times in a worst case scenario. Since the variable i will eventually be n - 1, there is essentially an equation that will have n multiplied by n (in a worst case scenario), which results in n^2, and since that's all that is cared about (ignoring constant factors and it's the most dominant function of n), the time complexity of Insertion Sort is $O(n^2)$. Calculating the number of comparisons is quite difficult because unlike Selection Sort, the number of comparisons actually depend on the original order of the array. No matter what, the first loop is run (n - 1) times. On average, the inner loop would be ran (i / 2) times. The total comparisons can be defined as the summation $\sum_{i=1}^{n-1} i$. Since this is essentially adding up the first n - 1 integers, this can be rewritten as $(n-1)(n-1+1)/2$, or $n(n-1)/2$. This is all multiplied by $1/2$ because on average, the key value is being compared with half of the previous elements (hence the i / 2). Multiplying these numbers results in approximately $n^2/4$. Plugging in 666 for n results in 110,889. This is approximately the average number of comparisons Insertion Sort performs with an input array of size 666. Selection Sort and Insertion Sort have the same big-Oh time complexity, but Insertion Sort does half the amount of comparisons that Selection Sort does.

4.5 Merge Sort

Merge Sort takes a different approach than Selection Sort and Insertion Sort. Merge Sort does a divide and conquer technique by recursively splitting the array into halves, and then merges the halves back together, sorting them on the way, to produce a fully sorted array. I will first explain the divide step of Merge Sort.

```
160 // Returns number of comparisons
161 // Pass by reference (changes to myItems affects magicItems)
162 int mergeSort(vector<string> &myItems, int begin = 0, int end =
        INT_MIN)
163 {
164     int comparisonNum = 0;
165
166     // If end parameter wasn't clarified, set it to the last index
        of the array (myItems[end] returns last element)
167     // This means the function is dealing with the entire list
168     if (end == INT_MIN)
169     {
170         end = myItems.size() - 1;
171     }
172     // Base case
173     else if (begin >= end)
174     {
175         return comparisonNum;
176     }
177
178     // Gets the mid point for splitting the array
179     int mid = (begin + end) / 2;
180
181     // Divides array into left and right sides
182     comparisonNum += mergeSort(myItems, begin, mid);
183     comparisonNum += mergeSort(myItems, mid + 1, end);
184
185     // Conquers each section
186     comparisonNum += merge(myItems, begin, mid, end);
187
188     return comparisonNum;
189 }
```

Figure 4.5: mergeSort function (Sorts.h)

The mergeSort function essentially gets the mid point to split the array into two halves (Line 179), and then recursively calls itself twice with parameters signifying that it's the left and right sides of the original array (Lines 182 and 183). The function will keep being called recursively until the array is of size 1. In this case, where the beginning index is greater than or equal to the ending index, it will terminate (Line 175). This is the base case for the function.

Something interesting that I decided to do for this program was to not have any extra parameters in the original call of the function in main.cpp (Line

73 of Figure 4.1 shows the program calling mergeSort(magicItems) with no other parameters). I wanted to do this to increase readability for the main function, instead of also including the begin and end index parameters in the function call, which would be 0 and n - 1 respectively. To do this, I had to use default parameters. A problem came with this because I couldn't set the default parameter for the variable "end" to myItems.size() - 1 because an error would occur stating that myItems may not appear in that context. I think the error occurs because myItems wouldn't be accessible whenever the function was declared because it isn't static. I came up with a workaround to this error. I set the "end" variable to INT_min as a default parameter, and if that variable was equal to INT_MIN, it would be changed to equal myItems.size() - 1 (Lines 168-171) because that's the last index of the whole array.

Now that I covered the dividing step of Merge Sort, next up the conquer step. I implemented this by creating a merge function (called on Line 186 of mergeSort).

```
105 // Merges left and right arrays into a larger sorted array
106 int merge(vector <string> &myItems, int begin, int mid, int end)
107 {
108     int comparisonNum = 0;
109 
110     // Tracks both sides of the array (index and size)
111     int lIndex = begin, rIndex = mid + 1;
112     int lSize = mid - begin + 1;
113     int rSize = end - mid;
114 
115     // Size of merged array
116     int n = lSize + rSize;
117 
118     vector<string> tempItems(n);
119 
120     // Merges items into temp array
121     for (int k = 0; k < n; k++)
122     {
123         comparisonNum++;
124 
125         // If right side has been completely traversed, copy
    remaining left half items into tempItems
126         if (rIndex > end)
127         {
128             tempItems[k] = myItems[lIndex];
129             lIndex++;
130         }
131         // If left side has been completely traversed, copy
    remaining right half items into tempItems
132         else if (lIndex > mid)
133         {
134             tempItems[k] = myItems[rIndex];
135             rIndex++;
136         }
```

```
137         // If current element in left half is less than current
    element in right half, copy left element into tempItems
138         else if (myItems[lIndex] < myItems[rIndex])
139         {
140             tempItems[k] = myItems[lIndex];
141             lIndex++;
142         }
143         // If current element in right half is less than or equal
    to current element in left half, copy right element into
    tempItems
144         else
145         {
146             tempItems[k] = myItems[rIndex];
147             rIndex++;
148         }
149     }
150
151     // Copies sorted tempItems to myItems
152     for (int k = 0; k < n; k++)
153     {
154         myItems[begin + k] = tempItems[k];
155     }
156
157     return comparisonNum;
158 }
```

Figure 4.6: Merge Function (Sorts.h)

The merge function begins by initializes some variables that keep track of the current index for both the left and right portions of the array (Line 111). It also keeps track the size of each portion (Lines 112, 113) and adds them up to get the size of the merged array (Line 116). To actually merge the two portions successfully, there needs to be another array that holds the values temporary (Line 118). Because I'm using a temporary vector, this means that the sort is not in-place (I'm using more than a constant amount of space in order to complete the sort). After declaring the tempItems vector of size n, a loop iterates through each of these indexes to see what value should be put in what places (Line 121). The four cases are as follows:

- Lines 126-130: If the right portion was completely ran through, then copy the remaining left portion items into the temporary vector.
- Lines 132-135: If the left portion was completely ran through, then copy the remaining right half items into the temporary vector
- Lines 138-142: If the current element in the left portion is less than the current element in the right portion, then choose the element in the left portion to copy to the temporary vector
- Lines 144-148: If the current element in the right portion is less than the current element in the left portion, then choose the element in the right portion to copy to the temporary vector.

After the loop is complete, the tempItems vector should be completely full of values that are sorted. However, these sorted values are no good if they aren't copied back over to the original vector (myItems). Lines 152-155 focus on copying the values from tempItems back over to myItems in the correct positions. Here is a sort result of running Merge Sort five times:

Input Size (n)	**Number of Comparisons**	**Time Complexity**
666	6,302 6,302 6,302 6,302 6,302	$O(n * log(n))$

Table 4: Merge Sort Results (Average Num of Comparisons: 6,302)

Unlike Selection Sort and Insertion Sort, the time complexity of Merge Sort is $O(n * log(n))$. The reason why is it isn't $O(n^2)$ is because there aren't nested loops. Instead, there are recursive calls. The n portion of $n * log(n)$ is because the merge function runs through the first loop n times (can ignore the second loop at Line 152 because it also runs n times, and $n + n = 2n$ is irrelevant because asymptotic running time ignores constant factors). The $log(n)$ comes from the recursive calls. Let's say there is an array of size 8. The amount of divides it would take to get the array to the size of one is 3 ($8 \rightarrow 4 \rightarrow 2 \rightarrow 1$). The amount of divides it would take to get an array of size 32 to an array of size 1 is 5 ($32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$). This is basically what $\log_2(n)$ is, where $\log_2(8) = 3$ and $\log_2(32) = 5$. Since asymptotic running time ignores constant factors, $log(n)$ is the time complexity of the dividing step, and n is the time complexity of the conquering step, resulting in $O(n*log(n))$. Also, like Selection Sort, the amount of comparisons are always the same, no matter how the array was previously sorted before being passed into the functions. It is actually easier to count the number of comparisons with Merge Sort than it is with Selection Sort and Insertion Sort. It is just $n * \log_2(n)$. This is because Merge Sort can be written as the recurrence relation $T(n) = T(n/2) + T(n/2) + n$. The two $T(n/2)$ represent the dividing and the n represents the conquering. Trying to solve this equation eventually results in $T(n * \log_2(n))$. Plugging in 666 for n results in approximately 6,247. This is very close to the number of comparisons I got in my program. Merge Sort is faster and does less comparisons then both Selection and Insertion Sort. However, Merge Sort is not an in-place sort, while Selection Sort and Insertion Sort are, because I used the tempItems vector to store some information in the merge function.

4.6 Quick Sort

Quick Sort takes a very similar approach compared to Merge Sort. Quick Sort also does a divide and conquer, but instead of doing it in two different steps, it does it at the same time. I will first explain the quickSort function before I get into the partition one.

```
230 int quickSort(vector<string> &myItems, int begin = 0, int end =
        INT_MIN)
231 {
232     int comparisonNum = 0;
233 
234     // If end parameter wasn't clarified, set it to the last index
        of the array (myItems[end] returns last element)
235     // This means the function is dealing with the entire list
236     if (end == INT_MIN)
237     {
238         end = myItems.size() - 1;
239     }
240     // Base case
241     else if (begin >= end)
242     {
243         return comparisonNum;
244     }
245 
246     // Selects random index for the contenders
247     random_device rd;
248     mt19937 gen(rd());
249     uniform_int_distribution<> distrib(begin, end);
250     int contender1 = distrib(gen);
251     int contender2 = distrib(gen);
252     int contender3 = distrib(gen);
253 
254     // Find best pivot value out of the three contenders
255     int pivotValue = 0;
256     if ((contender1 > contender2 && contender2 > contender3) || (
        contender3 > contender2 && contender2 > contender1))
257     {
258         pivotValue = contender2;
259     }
260     else if ((contender2 > contender3 && contender3 > contender1)
        || (contender1 > contender3 && contender3 > contender2))
261     {
262         pivotValue = contender3;
263     }
264     else
265     {
266         pivotValue = contender1;
267     }
268 
269     // Choosing the pivot element to be myItems[randomPivot]
270     auto parResult = partition(myItems, pivotValue, begin, end);
```

```
272     // Gets pivot index and comparison num
273     int r = parResult.first;
274     comparisonNum += parResult.second;
275 
276     // Runs quick sort on first and second parts of the array
277     comparisonNum += quickSort(myItems, begin, r - 1);
278     comparisonNum += quickSort(myItems, r + 1, end);
279 
280     return comparisonNum;
281 }
282 
283 #endif
```

Figure 4.7: quickSort function (Sorts.h)

Quick Sort works by choosing a pivot value. The arrays are recursively divided (like Merge Sort) into two different sections split up by a value from the partition function (will be explained later). Conquering occurs by partitioning those two sections until the smallest sub-arrays are sorted. To choose a pivot value, I used three contenders to avoid getting the absolute worst case possible: when the pivot value is on either end of the array. This could result in $O(n^2)$ time complexity, which isn't wanted. The contenders are random indexes in the array (Lines 247-252). To not get the absolute worst pivot, I compared the three contenders and chose the one in the middle to be the pivot (Lines 255-267). After the pivot index is chosen, the partition function is called (Line 270), which returns the partition index that splits the array into two. To split the array, the quickSort function is recursively called with the two parts (excluding the partition index), kind of like how Merge Sort recursively called the two halves of that array.
Lines 236-244 of the quickSort function are exactly the same as the mergeSort function, so the explanation of why the code looks the way it does is in the Merge Sort section. The initial end index of the array is myItems.size() - 1, and if the array is of size 1, then it terminates (Base Case).
Now that I explained the quickSort function, it is time to talk about the partition function.

```
191 // Returns both the partition point and the comparison count in a
        pair
192 pair<int, int> partition(vector<string> &myItems, int p, int begin,
         int end)
193 {
194     int comparisonNum = 0;
195 
196     // Indicates the currently found pivot position
197     int l = begin - 1;
```

```
199     // Swap myItems[p] and myItems[end] (moves pivot value to the
        end)
200     string temp = myItems[p];
201     myItems[p] = myItems[end];
202     myItems[end] = temp;
203
204     for (int i = begin; i < end; i++)
205     {
206         comparisonNum++;
207
208         // If current element is less than pivot element
209         if (myItems[i] < myItems[end])
210         {
211             // Swap myItems[l] and myItems[i] to move smaller
        element to the left partition
212             l++;
213
214             temp = myItems[l];
215             myItems[l] = myItems[i];
216             myItems[i] = temp;
217         }
218     }
219
220     // Swap the pivot and partition indexes to put pivot in the
        correct position
221     temp = myItems[end];
222     myItems[end] = myItems[l + 1];
223     myItems[l + 1] = temp;
224
225     return make_pair(l + 1, comparisonNum);
226 }
```

Figure 4.8: Partition function (Sorts.h)

The main goal of the partition function is to ensure that all elements less than the pivot are moved to the left side and all elements greater than the pivot are moved to the right side. The function starts of by declaring the variable "l" which indicates the current partition position (Line 197). It then swaps the pivot index value and the value at the end of the array, which essentially just moves the pivot value to the end (Lines 200-202). Then a loop runs for every index in that sub-array range (Line 204), where if the current element is less than the pivot value, it will swap the value at the current iterated index in the loop with the partition index value (Lines 209-218). This ensures that the smaller element is in the left side of the partition. After the loop is complete, the pivot and partition indexes are swapped to put the pivot in the correct position (Lines 221-223).
Quick Note: the reason why the function returns a pair is because even though I need the partition index to actually make the sort work, I also need the comparison count. This is why the function return type is ¡int, int¿ (it's returning two separate values).

Here is a sort result of running Quick Sort five times:

Input Size (n)	Number of Comparisons	Time Complexity
666	6,468 6,245 7,084 6,541 6,329	$O(n * log(n))$

Table 5: Quick Sort Results (Average Num of Comparisons: 6,533)

Exactly like Merge Sort, the time complexity of Quick Sort is $O(n * log(n))$. There are recursive calls instead of nested loops. The n portion of $n * log(n)$ is because the partition function runs through the loop approximately n times. The $log(n)$ comes from the recursive calls in the quickSort function. Also, like Insertion Sort, but unlike Merge Sort and Selection Sort, the amount of comparisons are different each time the function gets called. This is because it depends on the pivot values chosen, which are random in this case. Because of this, it is very difficult to come up with an equation to help find the average number of comparisons Quick Sort does. But, according to the table above, the average number of comparisons were 6,533. This is approximately equal to $n * log(n)$ with n = 666 in this case, like Merge Sort. Using the Merge Sort test table and the Quick Sort test table, it looks like Quick Sort does slightly more comparisons than Merge Sort. However, I think Quick Sort is overall faster because a lot less data is moving around in the partition function compared to data moving around in the merge function. Like Selection Sort and Insertion Sort, Quick Sort is an in-place sort. Each of these sorts have their own pros and cons, which is very interesting.

4.7 Running Sorts on Already Sorted Array

The table results of all the previous sorts are inferring that we are passing in an array that is randomly shuffled. What happens if I pass in an array that is already sorted? The end of the main.cpp file explores this topic.

```
85      // Performs every sort again, but when array is already sorted
86      comparisonNum = selectionSort(magicItems);
87      cout << "Number of Comparisons doing Selection Sort on a sorted
         array: " << comparisonNum << endl;
88
89      comparisonNum = insertionSort(magicItems);
90      cout << "Number of Comparisons doing Insertion Sort on a sorted
         array: " << comparisonNum << endl;
```

```
92      comparisonNum = mergeSort(magicItems);
93      cout << "Number of Comparisons doing Merge Sort on a sorted
        array: " << comparisonNum << endl;
94
95      comparisonNum = quickSort(magicItems);
96      cout << "Number of Comparisons doing Quick Sort on a sorted
        array: " << comparisonNum << endl;
97
98  }
```

Figure 4.9: Calling Sorts on Already Sorted Array (main.cpp)

Here are the results I found:

Sort Type	Number of Comparisons	Time Complexity
Selection Sort	221,445 221,445 221,445 221,445 221,445	$O(n^2)$
Insertion Sort	665 665 665 665 665	$O(n)$
Merge Sort	6,302 6,302 6,302 6,302 6,302	$O(n \log n)$
Quick Sort	5,822 5,935 5,575 5,661 5,749	$O(n \log n)$

Table 6: Sort Results on Already Sorted Array (Input Size (n) = 666)

When looking at Selection Sort and Merge Sort, it appears that nothing has changed at all. Those are the same numbers as in the previous tables for those two sorts. Looking at Quick Sort, it seems like the average comparison count is less than it previously was, which I find fascinating. I'm not completely sure why this is, considering the pivot value will always be random. However, the time complexity is still the same, so it's not much better than before. However, when looking at Insertion Sort, there is a clear and huge difference. The number of comparisons is only n - 1, meaning the time complexity of insertion sort on an already sorted list is just $O(n)$. The reason why it is so fast is because insertion sort only does comparisons on the unsorted section of the array, so if it's all sorted, the while loop will always fail, and no shifting will occur.

5 Conclusion

```
==8196==
==8196== HEAP SUMMARY:
==8196==     in use at exit: 0 bytes in 0 blocks
==8196==   total heap usage: 35,408 allocs, 35,408 frees, 1,288,605 bytes allocated
==8196==
==8196== All heap blocks were freed -- no leaks are possible
==8196==
==8196== For lists of detected and suppressed errors, rerun with: -s
==8196== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 5.1: Valgrind Output (running command "valgrind ./main")

In conclusion, this assignment was a great learning experience for me. This is my first time diving deep with C++ and so far, I'm having a great time. I struggled a little bit in the beginning, and also struggled during the implementation of Merge Sort and Quick Sort, but I got through it. With this assignment I also learned how to create a Makefile and also how to use Valgrind. The Makefile took a lot of my time because I tried to make it compatible with Windows and Linux. I had to read a lot of documentation and look up a ton of videos to find out how the file works and how to implement it specifically for my program. Valgrind was a very useful tool for me to use because it checks if memory leaks occur. One of the problems about C++ I was worried about was it doesn't automatically do garbage collection. However, with the help of Valgrind, I got my program from having more than 100,000 bytes of leakage down to 0. I can't wait to do more with C++ and algorithms as a whole. This is a ton of fun.