Assignment 1 - LaTeX Write Up

Connor Fleischman

October 3, 2024

1 Assignment Results

After implementing each part of Assignment 1 and testing each piece to ensure its functioning properly, here are the results.

Search/Sort Type	Palindrome Count	Selection Sort	Inertion Sort	Merge Sort	Quick Sort
Test 1	15	221445	109385	4316	5215
Test 2	15	221445	112038	4316	5898
Test 3	15	221445	106491	4316	5434
Test 4	15	221445	113176	4316	5271
Test 5	15	221445	113507	4316	4933
Test 6	15	221445	112386	4316	5082
Test 7	15	221445	106824	4316	5433
Test 8	15	221445	114922	4316	5611
Test 9	15	221445	105941	4316	5672
Test 10	15	221445	107550	4316	5844
O(n)	-	O(n^2)	O(n^2)	O(n*log2(n))	O(n*log2(n))
# Items	-	666	666	666	666
Worst Case	-	443556	443556	6246.67	6246.67
Average	-	221445	110222	4316	5439.3
< Worst Case?	-	TRUE	TRUE	TRUE	TRUE

As we can see, after running 10 tests on our data, some patterns have immerged. Firstly, notice that the number of comparisons made during the Selection and Merge Sorts. It remains the same for each test. This is because the implementation of these sorts do not depend on how partially sorted our data is, where as Insertion and Quick sort do. Secondly, the section half of this table calculates the estimated worse case number of comparisons for the given sort. As I have demonstrated using Excel, my sorts are all well below the expected worse case.

2 Prerequisites

These are the modules and the namespace I have used for this assignment. A more detailed list of the include statements can be found on main.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <random>
#include <string>

using namespace std; // Declaring the name space to use std::_
```

```
std::vector<std::string> lines; // Declares the lines a vector of strings
int numComparisons = 0;         // Declares the number of comparisons made
```

3 Main

The Main method of this document houses the function calls for each part of Assignment 1. As we saw in Assignment Results (Sec 1), we first call the calcPalindrome() method (Sec 7) which calculates then number of palindromes in magicItems. Then the globally scoped numComparisons is set to 0 to ensure an accurate count. Next we call the shuffleItems() method (Sec 8) which runs the Knuth Shuffle over the list of magic items and then the selectionSort() function (Sec 9). As shown in the code below, the order of setting numComparisons to 0, shuffling the items, and sorting is repeated throughout Main. However, it's important to note some distinctions. After the insertionSort() method (Sec 10) is called we clear the lines vector (Ln 13/19). This is done so that a freshly shuffled deck of items is given to the sorts. This is only done for mergeSort() and quickSort() (Sec 11, 12) because they use recursion. So the first time they're ran, we have to give an empty vector, fill it with the shuffled items, and sort it. Also note that since recursion is used for these sorts, we cannot have the sort itself return the number of comparisons made, we have to return it outside the function.

```
1 int main() // Comprised of the function calls for Assignment 1
```

```
{
    calcPalindrome(); // Calculates the number of palindromes in magicItems.txt

    numComparisons = 0; // Resets the comparison counter
    shuffleItems();     // Shuffles magicItems.txt using the Knuth Shuffle
    selectionSort();    // Runs a selection sort over the items in magicItems.txt

    numComparisons = 0; // Resets the comparison counter
    shuffleItems();     // Shuffles magicItems.txt using the Knuth Shuffle
    insertionSort();    // Runs an insertion sort over the items in magicItems.txt

    lines.clear();              // Empties the lines vector for the next sort
    numComparisons = 0;         // Resets the comparison counter
    shuffleItems();             // Shuffles magicItems.txt using the Knuth Shuffle
    mergeSort(lines);           // Runs a merge sort over the items in magicItems.txt
    std::cout << "Number of comparisons: " << numComparisons << std::endl; // Outputs the number of comparisons preformed

    lines.clear();              // Empties the lines vector for the next sort
    numComparisons = 0;         // Resets the comparison counter
    shuffleItems();             // Shuffles magicItems.txt using the Knuth Shuffle
    quickSort(lines);           // Runs a quick sort over the items in magicItems.txt
    std::cout << "Number of comparisons: " << numComparisons << std::endl; // Outputs the number of comparisons preformed
```

```
    numComparisons = 0; // Resets the comparison counter

    return 0; // Finishes main method
}
```

4 Node

First I created instances of a Node class which extends a Stack(Sec 5) and a Queue (Sec 6). This node class is comprised of a data portion which is a string (Ln 4), and a next pointer to reference the next node in order (Ln 4/10). The method creates a node (Ln 7) with an input of some string ”item” which data is set to (Ln 7/9), then the next pointer is set to NULL (Ln 10).

```
1 class Node // Defines the Node class
2 {
3 public:
4     std::string data; // Identifies the data section of the node
5     Node *next;        // Identifies the pointer to the next node
6 
7     Node(std::string item) // Creates a node which takes in some item
8     {
9         this->data = item;    // Sets this node's data point to item
10        this->next = nullptr; // Sets this node's next point to null
11    }
12 };
```

5 Stack

Then I made a Stack class which extended the Node (Sec 4) class. The stack has a pointer to its first element (Ln 4) called top which when a stack is made is set to NULL (Ln 9). I next implemented push(), pop(), and isEmpty() methods to the stack (Ln 12, 19, 30). These all work how one might expect, push takes a string and makes it the new top, setting the old top to the new ones next. Pop takes the top element, returns the data, sets the tops next to the new top, and deletes the old top. And isEmpty checks if the stack is empty and returns True if.

```
class Stack // Defines the Stack class
{
private:
    Node *top; // Identifies top as a Node

public:
    Stack() // Creates a Stack
    {
        top = nullptr; // Sets top to a null pointer
    }

    void push(std::string item) // Enters an item into the stack
    {
        Node *newNode = new Node(item); // Creates a newNode for the item
        newNode->next = top;            // Sets the newNode's next to previous' top
        top = newNode;                  // Sets the new top to the newNode
    }

    std::string pop() // Removes an item from the top of the stack
    {
        Node *temp = top;                    // Creates a temporary node to store the top
        std::string toBePopped = top->data; // Creates toBePopped which is the data of the top element

        top = top->next; // Sets the new top to what the current top's next pointer is
        delete temp;     // Deletes the old top

        return toBePopped; // Returns the old top's data
    }

    bool isEmpty() // Returns if "the stack is empty" is true
    {
        return top == nullptr; // If the top points to null return True
    }
};
```

6 Queue

Next I made a Queue class also extending the Node (Sec 4) class. The queue has pointers to both its head and tail (Ln 4/5). When the queue is made these are set to NULL (Ln 10/11). Then, like the stack, I define methods for enqueueing, dequeueing, and checking to see if the queue isEmpty. The enqueue() method (Ln 14) takes in a string and sets it to the head and tail if the queue is empty or sets it to the new tail if not empty (Ln 18-26). The dequeue() method (Ln 29) is the same as the pop() method in the Stack (Sec 5). It returns the data of the removed element, however with queues, elements are removed from the front, so the head is dequeued, not the tail. The isEmpty() method is also the same (Ln 39).

```
1  class Queue // Defines the Queue class
2  {
3  private:
4      Node *head; // Identifies the head as a Node
5      Node *tail; // Identifies the tail as a Node
6  
7  public:
8      Queue() // Creates a Queue
9      {
10         head = nullptr; // Sets head to a null pointer
11         tail = nullptr; // Sets tail to a null pointer
12     }
13 
14     void enqueue(std::string item) // Enters an item to the queue
15     {
16         Node *newNode = new Node(item); // Creates a new node for the item
17 
18         if (isEmpty()) // If the queue is empty
19         {
20             head = tail = newNode; // Set the head and tail both to be newNodes
21         }
22         else
23         {
24             tail->next = newNode; // Set the tail's next pointer to the newNode
25             tail = newNode;       // Set the tail to the newNode
26         }
```

```
    }

    std::string dequeue() // Removes an item from the front of the queue
    {
        Node *temp = head;                    // Creates a temporary node to store the current head
        std::string toBePopped = head->data; // Creates toBePopped which is the data of the current head element

        head = head->next; // Sets the new head to the the next pointer of the current head
        delete temp;       // Deletes the current head

        return toBePopped; // Returns the dequeued data
    }
    bool isEmpty() // Returns if 'queue is empty' is true
    {
        return head == nullptr; // If the head points to null return True
    }
};
```

7 Palindromes

We are asked to calculate the number of palindromic strings there were in a given list of 666 magic items. After running the search through the vector of strings, and entering each item into a Stack (Sec 5) and Queue (Sec 6). Then comparing popping and dequeueing each element of the item, comparing for equality, and if equal then checking the next element. Until, if all are true then the item is a palindrome else go to next element. We then find that the number of palindromes in the magicItems.txt file is 15.

```
bool isPalindrome(const std::string line) // Defines the isPalindrome method
{
    Stack stack; // Implements a stack
    Queue queue; // Implements a queue

    std::string parsedInput; // Creates a variable to hold the parsed input of a line

    for (char n : line) // For n in line
```

```
    {
        if (std::isalnum(n)) // Checking if character n is alphanumeric
        {
            char lowerChar = std::tolower(n); // Converts character n to lowercase
            parsedInput.push_back(lowerChar); // Sets parsedInput as alphanumeric lowercase
        }
    }

    for (char n : parsedInput) // For n in parsedInput
    {
        std::string firstChar(1, n); // Let n be the first character
        stack.push(firstChar);       // Push the nth character of the line on the stack
        queue.enqueue(firstChar);    // Queue nth character
    }

    while (!stack.isEmpty() && !queue.isEmpty()) // While the stack and queue are not empty
    {
        if (stack.pop() != queue.dequeue()) // If the word forwards (stack) is not equal to the word backwards (queue)
        {
            return false;
        }
    }

    return true;
}

void calcPalindrome() // Defines the calcPalindrome method
{
    std::ifstream file("magicItems.txt"); // Opens and reads the file
    std::string line;                     // Defines a line
    int palindromeCount = 0;              // Defines a palindrome counter

    while (std::getline(file, line)) // While there is a new line in the file
    {
```

```
        if (isPalindrome(line)) // If the new line is a palindrome
        {
            palindromeCount++; // Increment palindromeCount
        }
    }

    file.close(); // Closes the file

    std::cout << "Number of palindromes: " << palindromeCount << std::endl; // Output the number of palindromes
}
```

8 List Shuffle

Here we define the shuffling method which is used to randomize the magicItems vector before each sort. This is called the Knuth sort, where we take a random index in the vector and swap the first item and that random item. Do this for each element in the vector.

```
void shuffleItems() // Defines the shuffleItems method
{
    std::ifstream file("magicItems.txt"); // Opens and reads the file
    std::string line;                     // Declares line as a string

    while (std::getline(file, line)) // While there is a new line in the file
    {
        lines.push_back(line); // Add the new line to the vector lines
    }

    file.close(); // Closes the file

    std::random_device randNum;                   // Creates a random number generator
    std::default_random_engine engine(randNum()); // "Seeds" the random number (essentially starting it)

    for (int i = lines.size() - 1; i > 0; i--) // For every item in the lines vector except the last
```

```
    {
        std::uniform_int_distribution<int> distr(0, i); // Take a random number from 0 - i
        int j = distr(engine);                                // Set j as that number

        std::swap(lines[i], lines[j]); // Swap the positions of i and j
    }
}
```

9 Selectio nSort

Now we move to the first of our sorting methods. The Selection sort which goes through our randomized vector, takes the first element and looks for an element less than itself. Once it has looked through the whole vector, it then swaps the current and the lowest items, then it goes onto the next items until it reaches the end of the list, sorting it.

```
void selectionSort() // Defines the selectionSort method
{
    int n = lines.size();            // n = the length of the lines vector
    for (int i = 0; i < n - 1; i++) // For every item in the vector
    {
        int toCompare = i;               // Sets the item to be compared to to i
        for (int j = i + 1; j < n; j++) // For every item j in lines after i
        {
            numComparisons++;                 // Increment numComparisons
            if (lines[j] < lines[toCompare]) // If the next item in lines is less than the item being compared
            {
                toCompare = j; // Set the new item to be compared to to j
            }
        }

        std::swap(lines[i], lines[toCompare]); // Swap the positions of the previously compared element and the
```

```
current
    }
    std::cout << "Number of comparisons: " << numComparisons << std::endl; // Output the number of comparisons

    lines.clear();          // Clears the sorted vector
    lines.shrink_to_fit(); // Shrinks the size of the vector to the number of items in it, 0
}
```

10 Insertion Sort

The second sorting method was Insertion sort which takes the list and compares each element one by one. Meaning the first element is compared to the second, if its greater than the second, swap, then go to the third element, if its less than the second swap, if its less than the first, swap again. This continues until the array is sorted

```
void insertionSort() // Defines the insertionSort method
{
    int n = lines.size();       // n = the length of lines vector
    for (int i = 1; i < n; i++) // For every item i in the vector
    {
        int j = i;                                    // Let j = i
        while (j > 0 && lines[j - 1] > lines[j]) // While j > 0 and the previous element is larger than the current
        {
            numComparisons++;                    // Increment numComparisons
            std::swap(lines[j], lines[j - 1]); // Swap the positions of the current and previous elements
            j--;                                 // Decrement j
        }
    }
    std::cout << "Number of comparisons: " << numComparisons << std::endl; // Output the number of comparisons

    lines.clear();          // Clears the sorted vector
    lines.shrink_to_fit(); // Shrinks the size of the vector to the number of items in it, 0
}
```

11 Merge Sort

Merge sort is the first sort in this assignment to use recursion. Recursion is the best! Why? Because it recurs. Unlike the above sorts, merge sort takes in a vector of strings called list (Ln 1). When this function is called from the Main method, it is given the list shuffled from before (Sec 8). However, when the sort recurs, it then uses a broken-up version of the original vector. Merge sort works by taking the original list, and breaking it down into halves over and over again until each half is of length one, then it will rejoin each half, but just before it will compare if the left half is greater or less than the right half and combine accordingly to make a sorted list. Divide 'n Conquer.

```
1  void mergePartitions(std::vector<std::string> list, int m) // Merges a vector of strings, list, around a point, m
2  {
3      int i = 1;                          // set i = 1
4      int j = m + 1;                      // set j = mergePoint + 1
5      int n = list.size() - 1;            // set n = length(list) - 1
6      std::vector<std::string> temp(n); // creates a new temporary vector with n elements
7
8      for (int k = 1; k < n; k++) // for every element of list besides list
9      {
10         numComparisons++; // Increment numComparisons
11         if (j > n)        // if after the mergePoint > the n
12         {
13             temp[k].assign(list[i]); // assign the value at temp[k], list[i]
14             i++;                     // Increment i
15         }
16         else if (i > m) // if i > the mergePoint
17         {
18             temp[k].assign(list[j]); // assign the value at temp[k], list[j]
19             j++;                     // Increment j
20         }
21         else if (list[i] < list[j]) // if the element at i of list < the element at j of list
22         {
23             temp[k].assign(list[i]); // assign the value at temp[k], list[i]
```

```
            i++;                        // Increment i
        }
        else
        {
            temp[k].assign(list[j]); // assign the value at temp[k], list[j]
            j++;                        // Increment j
        }
    }
    for (int k = 1; k < n; k++) // for every element of list besides list
    {
        list[k].assign(temp[k]); // assign the value at list[k], temp[k]
    }

    temp.clear(); // clear temp
}

void mergeSort(std::vector<std::string> list) // Defines the mergeSort method taking a list to be sorted
{
    int n = list.size(); // n = the length of inserted list

    if (n > 1) // if the size of list is 2 or more
    {
        int m = floor(n / 2);                  // m = floor of (n/2)
        std::vector<std::string> left(list.begin(), list.begin() + m);       // left = new vector<string> consisting of original list to m
        std::vector<std::string> right(list.begin() + (m + 1), list.end()); // right = new vector<string> consisting of list[m] to end of list

        mergeSort(left);  // recurse on left side
        mergeSort(right); // recurse on right side

        list.clear();                                          // clear the unsorted list
        list.reserve(left.size() + right.size());              // reserve space in unsorted list for sorted left and right
        list.insert(list.end(), left.begin(), left.end());     // insert items from left into list
```

```
        list.insert(list.end(), right.begin(), right.end()); // insert items from right into list

        mergePartitions(list, m); // merges all sublist around the merge point
    }
}
```

12 Quick Sort

Our final sort was Quick sort. Similarly to the mergeSort() method (Sec 11), quick sort also uses recursion to sort the magic items. However, instead of splitting the vector down the middle, we now take a random value called the partition point, and use that to split the vector. Everything larger then the partition point to the right and everything lower to the left. Then quick sort recurs on these smaller left and right portion, similarly to mergeSort() (Sec 11) until sorted.

```
int quickPartitions(std::vector<std::string> list, int p) // Partitions a vector of strings, list, around a point, p
{
    int n = list.size() - 1;     // sets n to the size of the list - 1
    std::swap(list[p], list[n]); // swaps the position of p and n in list
    int l = 0;                   // sets l = 0

    for (int i = 1; i < (n - 1); i++) // for every element in the list
    {
        numComparisons++;      // Increment numComparisons
        if (list[i] < list[n]) // if the element at list[i] < the element at list[j]
        {
            l++;                         // Increment l
            std::swap(list[l], list[i]); // swaps the position of l and i in the list
        }
    }

    std::swap(list[n], list[l + 1]); // swaps the position of n and l + 1 in the list

```

```cpp
    return l + 1; // returns l + 1
}

void quickSort(std::vector<std::string> list) // Defines the quickSort method
{
    int n = list.size(); // n = the length of inserted list

    if (n > 1) // if the size of list is 2 or more
    {
        std::random_device randNum;                                // Creates a random number generator
        std::default_random_engine engine(randNum());              // "Seeds" the random number (essentially starting it)
        std::uniform_int_distribution<int> distr(0, n - 1);        // Take a random number from 0 - (n-1)
        int p = distr(engine);                                     // Set p as that number

        int r = quickPartitions(list, p);                // make r the partitioned element of list at p
        std::vector<std::string> left(list.begin(), list.begin() + (r - 1)); // left = new vector<string> consisting of original list to (r - 1)
        std::vector<std::string> right(list.begin() + (r + 1), list.end());  // right = new vector<string> consisting of original list[(r - 1)] to end of list

        quickSort(left);  // recurse on left side
        quickSort(right); // recurse on right side

        list.clear();                                              // clear the unsorted list
        list.reserve(left.size() + right.size());                  // reserve space in unsorted list for sorted left and right
        list.insert(list.end(), left.begin(), left.end());         // insert items from left into list
        list.insert(list.end(), right.begin(), right.end());       // insert items from right into list
    }
}
```

13 Final Thoughts

This assignment was a large challenge in the beginning. I am new to C++ completely and I have been using YouTube, StackOverflow, and tons of other sources to put this assignment together. It was very fun learning how to implement these ideas which I use in other languages in a new one. I hope all code compiles and runs flawlessly but please report any issues, I will fix them.