

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	Insertion 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
$O(n)$	-	$O(n^2)$	$O(n^2)$	$O(n*\log_2(n))$	$O(n*\log_2(n))$
# 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 emerged. 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, whereas Insertion and Quick sort do. Secondly, the section half of this table calculates the estimated worst case number of comparisons for the given sort. As I have demonstrated using Excel, my sorts are all well below the expected worst 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

```
1 #include <iostream>
2 #include <fstream>
3 #include <vector>
4 #include <algorithm>
5 #include <random>
6 #include <string>
7
8 using namespace std; // Declaring the name space to use std::
9
10
11 std::vector<std::string> lines; // Declares the lines a
    vector of strings
12 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
```

```

2 {
3     calcPalindrome(); // Calculates the number of palindromes
                        // in magicItems.txt
4
5     numComparisons = 0; // Resets the comparison counter
6     shuffleItems();    // Shuffles magicItems.txt using the
                        // Knuth Shuffle
7     selectionSort();   // Runs a selection sort over the
                        // items in magicItems.txt
8
9     numComparisons = 0; // Resets the comparison counter
10    shuffleItems();     // Shuffles magicItems.txt using the
                        // Knuth Shuffle
11    insertionSort();    // Runs an insertion sort over the
                        // items in magicItems.txt
12
13    lines.clear();
                        // Empties the lines vector for the next
sort
14    numComparisons = 0;
                        // Resets the comparison counter
15    shuffleItems();
                        // Shuffles magicItems.txt using the Knuth
                        // Shuffle
16    mergeSort(lines);
                        // Runs a merge sort over the items in
magicItems.txt
17    std::cout << "Number of comparisons: " << numComparisons
<< std::endl; // Outputs the number of comparisons
preformed
18
19    lines.clear();
                        // Empties the lines vector for the next
sort
20    numComparisons = 0;
                        // Resets the comparison counter
21    shuffleItems();
                        // Shuffles magicItems.txt using the Knuth
                        // Shuffle
22    quickSort(lines);
                        // Runs a quick sort over the items in
magicItems.txt
23    std::cout << "Number of comparisons: " << numComparisons
<< std::endl; // Outputs the number of comparisons
preformed

```

```

24
25     numComparisons = 0; // Resets the comparison counter
26
27     return 0; // Finishes main method
28 }

```

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.

```
1 class Stack // Defines the Stack class
2 {
3 private:
4     Node *top; // Identifies top as a Node
5
6 public:
7     Stack() // Creates a Stack
8     {
9         top = nullptr; // Sets top to a null pointer
10    }
11
12    void push(std::string item) // Enters an item into the
stack
13    {
14        Node *newNode = new Node(item); // Creates a newNode
for the item
15        newNode->next = top; // Sets the newNode's
next to previous' top
16        top = newNode; // Sets the new top
to the newNode
17    }
18
19    std::string pop() // Removes an item from the top of the
stack
20    {
21        Node *temp = top; // Creates a
temporary node to store the top
22        std::string toBePopped = top->data; // Creates
toBePopped which is the data of the top element
23
24        top = top->next; // Sets the new top to what the
current top's next pointer is
25        delete temp; // Deletes the old top
26
27        return toBePopped; // Returns the old top's data
28    }
29
30    bool isEmpty() // Returns if "the stack is empty" is true
31    {
32        return top == nullptr; // If the top points to null
return True
33    }
34 };
```

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 is empty. 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         }
27     }
28 }
```

```

27     }
28
29     std::string dequeue() // Removes an item from the front
of the queue
30     {
31         Node *temp = head; // Creates a
temporary node to store the current head
32         std::string toBePopped = head->data; // Creates
toBePopped which is the data of the current head element
33
34         head = head->next; // Sets the new head to the the
next pointer of the current head
35         delete temp; // Deletes the current head
36
37         return toBePopped; // Returns the dequeued data
38     }
39     bool isEmpty() // Returns if 'queue is empty' is true
40     {
41         return head == nullptr; // If the head points to null
return True
42     }
43 };

```

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 dequeuing 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.

```

1 bool isPalindrome(const std::string line) // Defines the
isPalindrome method
2 {
3     Stack stack; // Implements a stack
4     Queue queue; // Implements a queue
5
6     std::string parsedInput; // Creates a variable to hold
the parsed input of a line
7
8     for (char n : line) // For n in line

```

```

9      {
10         if (std::isalnum(n)) // Checking if character n is
alphanumeric
11         {
12             char lowerChar = std::tolower(n); // Converts
character n to lowercase
13             parsedInput.push_back(lowerChar); // Sets
parsedInput as alphanumeric lowercase
14         }
15     }
16
17     for (char n : parsedInput) // For n in parsedInput
18     {
19         std::string firstChar(1, n); // Let n be the first
character
20         stack.push(firstChar);          // Push the nth
character of the line on the stack
21         queue.enqueue(firstChar);      // Queue nth character
22     }
23
24     while (!stack.isEmpty() && !queue.isEmpty()) // While the
stack and queue are not empty
25     {
26         if (stack.pop() != queue.dequeue()) // If the word
forwards (stack) is not equal to the word backwards (queue
)
27         {
28             return false;
29         }
30     }
31
32     return true;
33 }
34
35 void calcPalindrome() // Defines the calcPalindrome method
36 {
37     std::ifstream file("magicItems.txt"); // Opens and reads
the file
38     std::string line;                      // Defines a line
39     int palindromeCount = 0;              // Defines a
palindrome counter
40
41     while (std::getline(file, line)) // While there is a new
line in the file
42     {

```



```

43     if (isPalindrome(line)) // If the new line is a
        palindrome
44     {
45         palindromeCount++; // Increment palindromeCount
46     }
47 }
48
49 file.close(); // Closes the file
50
51 std::cout << "Number of palindromes: " << palindromeCount
    << std::endl; // Output the number of palindromes
52 }

```

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.

```

1 void shuffleItems() // Defines the shuffleItems method
2 {
3     std::ifstream file("magicItems.txt"); // Opens and reads
        the file
4     std::string line; // Declares line as
        a string
5
6     while (std::getline(file, line)) // While there is a new
        line in the file
7     {
8         lines.push_back(line); // Add the new line to the
        vector lines
9     }
10
11     file.close(); // Closes the file
12
13     std::random_device randNum; // Creates
        a random number generator
14     std::default_random_engine engine(randNum()); // "Seeds"
        the random number (essentially starting it)
15
16     for (int i = lines.size() - 1; i > 0; i--) // For every
        item in the lines vector except the last

```

```

17     {
18         std::uniform_int_distribution<int> distr(0, i); //
        Take a random number from 0 - i
19         int j = distr(engine); //
        Set j as that number
20
21         std::swap(lines[i], lines[j]); // Swap the positions
        of i and j
22     }
23 }

```

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.

```

1 void selectionSort() // Defines the selectionSort method
2 {
3     int n = lines.size(); // n = the length of the
        lines vector
4     for (int i = 0; i < n - 1; i++) // For every item in the
        vector
5     {
6         int toCompare = i; // Sets the item to
        be compared to to i
7         for (int j = i + 1; j < n; j++) // For every item j
            in lines after i
8         {
9             numComparisons++; // Increment
            numComparisons
10            if (lines[j] < lines[toCompare]) // If the next
                item in lines is less than the item being compared
11            {
12                toCompare = j; // Set the new item to be
                compared to to j
13            }
14        }
15
16        std::swap(lines[i], lines[toCompare]); // Swap the
        positions of the previously compared element and the

```

```

current
17     }
18     std::cout << "Number of comparisons: " << numComparisons
    << std::endl; // Output the number of comparisons
19
20     lines.clear(); // Clears the sorted vector
21     lines.shrink_to_fit(); // Shrinks the size of the vector
    to the number of items in it, 0
22 }

```

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

```

1 void insertionSort() // Defines the insertionSort method
2 {
3     int n = lines.size(); // n = the length of lines
    vector
4     for (int i = 1; i < n; i++) // For every item i in the
    vector
5     {
6         int j = i; // Let j = i
7         while (j > 0 && lines[j - 1] > lines[j]) // While j >
    0 and the previous element is larger than the current
8         {
9             numComparisons++; // Increment
    numComparisons
10             std::swap(lines[j], lines[j - 1]); // Swap the
    positions of the current and previous elements
11             j--; // Decrement j
12         }
13     }
14     std::cout << "Number of comparisons: " << numComparisons
    << std::endl; // Output the number of comparisons
15
16     lines.clear(); // Clears the sorted vector
17     lines.shrink_to_fit(); // Shrinks the size of the vector
    to the number of items in it, 0
18 }

```

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]
```

```

24         i++; // Increment i
25     }
26     else
27     {
28         temp[k].assign(list[j]); // assign the value at
temp[k], list[j]
29         j++; // Increment j
30     }
31 }
32 for (int k = 1; k < n; k++) // for every element of list
besides list
33 {
34     list[k].assign(temp[k]); // assign the value at list[
k], temp[k]
35 }
36
37 temp.clear(); // clear temp
38 }
39
40 void mergeSort(std::vector<std::string> list) // Defines the
mergeSort method taking a list to be sorted
41 {
42     int n = list.size(); // n = the length of inserted list
43
44     if (n > 1) // if the size of list is 2 or more
45     {
46         int m = floor(n / 2);
// m = floor of (n/2)
47         std::vector<std::string> left(list.begin(), list.
begin() + m); // left = new vector<string> consisting
of original list to m
48         std::vector<std::string> right(list.begin() + (m + 1)
, list.end()); // right = new vector<string> consisting of
list[m] to end of list
49
50         mergeSort(left); // recurse on left side
51         mergeSort(right); // recurse on right side
52
53         list.clear();
// clear the unsorted list
54         list.reserve(left.size() + right.size());
// reserve space in unsorted list for sorted left and
right
55         list.insert(list.end(), left.begin(), left.end());
// insert items from left into list

```

```

56     list.insert(list.end(), right.begin(), right.end());
    // insert items from right into list
57
58     mergePartitions(list, m); // merges all sublist
    around the merge point
59 }
60 }

```

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 than 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.

```

1  int quickPartitions(std::vector<std::string> list, int p) //
    Partitions a vector of strings, list, around a point, p
2  {
3      int n = list.size() - 1; // sets n to the size of the
        list - 1
4      std::swap(list[p], list[n]); // swaps the position of p
        and n in list
5      int l = 0; // sets l = 0
6
7      for (int i = 1; i < (n - 1); i++) // for every element in
        the list
8      {
9          numComparisons++; // Increment numComparisons
10         if (list[i] < list[n]) // if the element at list[i] <
            the element at list[j]
11         {
12             l++; // Increment l
13             std::swap(list[l], list[i]); // swaps the
                position of l and i in the list
14         }
15     }
16
17     std::swap(list[n], list[l + 1]); // swaps the position of
        n and l + 1 in the list
18

```

```

19     return l + 1; // returns l + 1
20 }
21
22 void quickSort(std::vector<std::string> list) // Defines the
    quickSort method
23 {
24     int n = list.size(); // n = the length of inserted list
25
26     if (n > 1) // if the size of list is 2 or more
27     {
28         std::random_device randNum;
29         // Creates a random number generator
30         std::default_random_engine engine(randNum());
31         // "Seeds" the random number (essentially starting it)
32         std::uniform_int_distribution<int> distr(0, n - 1);
33         // Take a random number from 0 - (n-1)
34         int p = distr(engine);
35         // Set p as that number
36
37         int r = quickPartitions(list, p);
38         // make r the partitioned element of list
39         at p
40         std::vector<std::string> left(list.begin(), list.
41         begin() + (r - 1)); // left = new vector<string>
42         consisting of original list to (r - 1)
43         std::vector<std::string> right(list.begin() + (r + 1)
44         , list.end()); // right = new vector<string> consisting
45         of original list[(r - 1)] to end of list
46
47         quickSort(left); // recurse on left side
48         quickSort(right); // recurse on right side
49
50         list.clear();
51         // clear the unsorted list
52         list.reserve(left.size() + right.size());
53         // reserve space in unsorted list for sorted left and
54         right
55         list.insert(list.end(), left.begin(), left.end());
56         // insert items from left into list
57         list.insert(list.end(), right.begin(), right.end());
58         // insert items from right into list
59     }
60 }

```

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.