Assignment One: Stacks, Queues, and Sorting

Aidan Carr

Aidan.Carr1@Marist.edu

October 6, 2023

1 Palindromes

1.1 Introduction

To test whether or a not a word is a palindrome, we must compare the word "frontwards" and backwards. In this Assignment, words are put into stacks and queues letter by letter. They are both popped from the stack and dequeued from the queue and compared. If all letters are equal, then the word is a palindrome. If even one letter does not equal, there is no palindrome.

Given a file of 666 magic items, this assignment traverses the file, adds each line to an array, then goes through each element of the array to check if it is a palindrome or not. The results will be a complete list of 12 palindromes.
One note: when checking for palindromes, we do not look at spaces or capitalization.

1.2 Nodes and Linked Lists

In order to make stacks and queues in the first place, we need some architecture. Both stacks and queues use linked lists as a way to store their data. A linked list, as seen in Figure 1.1, consists of Nodes. Each node is an object that contains a value, and a pointer to the next node. A linked list requires an extra pointer to that first node.

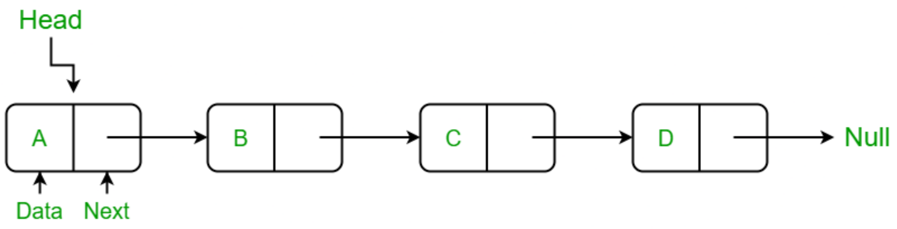

Figure 1.1: Example of a Linked List with four unique Nodes[1]

[1]Image: https://www.geeksforgeeks.org/data-structures/linked-list/singly-linked-list/.

```
1  class Node {
2  
3  public:
4      string itemName;
5      Node* next;
6  
7      //Constructors
8      Node(string itemNameInput){
9          itemName = itemNameInput;
10         next = nullptr;
11     }
12     Node(){
13         itemName = "";
14         next = nullptr;
15     }
16 };
```

Figure 1.2: Node class in C++

Figure 1.2 shows the code for a Node class. Line 4 and 5 are the object's attributes: a name that the node is storing, and a pointer to the next node. When constructed on line 8, the node will have a name of whatever is inputted and **next** points to nothing. If constructed without a name on line 12, the node's name will be an empty string and **next** will still point to nothing.

1.3 Stack

Think of a stack like a stack of blue blocks. Every time you put down a block, that is the new top. And every time you want a block, you take it from the top of the tower. In Figure 1.3, we have a pointer to the top block. With just this one location known, we can add (push) onto the stack and take (pop) from the stack.

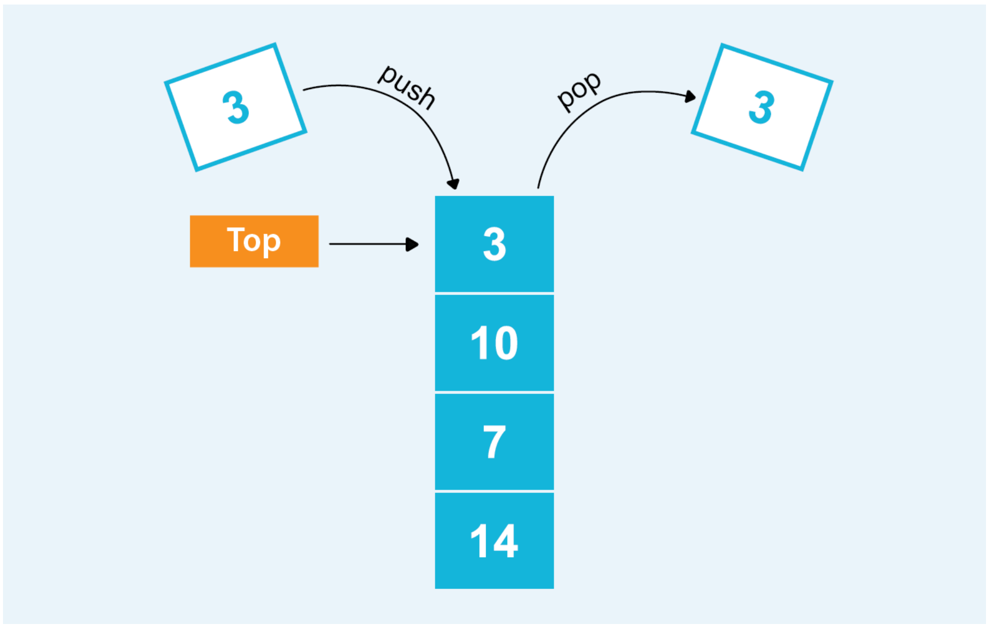

Figure 1.3: A Stack of nodes with a pointer to the top[2]

[2]Image: https://medium.com/geekculture/stack-in-python-33617350b6d2.

A stack is a linked list, but flipped to be standing up. We are constantly adding to and removing from the front of this linked list. Each blue block is a node with its value and a pointer to the next node. When we reach the bottom node in the stack, its next value points to nothing.

To implement this into a program, a Stack class is created. In Figure 1.4, on line 4, the only attribute is given. The node pointer `top` points to the top of the stack. This value is constantly changing which we will see in the following methods.

```
1  class Stack{
2  
3  public:
4      Node* top;
5  
6      //Constructor
7      Stack(){
8          top = nullptr;
9      }
10 
11     //METHODS
12     //return true if Empty
13     bool isEmpty(){
14         if (top == nullptr){
15             return true;
16         }
17         return false;
18     }
19 
20     //push/add to the top
21     void push(Node* newTop){
22         newTop->next = top;
23         top = newTop;
24     }
25 
26     //retrieve from the top
27     string pop(){
28         //cant pop if there is nothing to pop
29         if (isEmpty()){
30             return "EMPTY STACK";
31         }
32         string popping = top->itemName;
33         top = top->next;
34         return popping;
35     }
36 };
```

Figure 1.4: Stack class in C++

The constructor on line 7 in Figure 1.4 will only point the top of the stack to nothing; it is empty. The method `isEmpty()` on line 13 checks to see if the top points to any node. If not, the stack is empty and returns true. If there is a top, it returns false.

The `push()` method on line 21 receive a pointer to a node (a node's memory address). It makes this node's next value point to the top. Now, this node that had been pushed can become the new top.

The `pop()` method on line 27 returns the stored name of the top node. First, we check if the stack is empty, because we can't pop nothing. We first look at the top node's `next` pointer. We make this node (the second to the top) our stack's new `top`. Then, we pop our old top node by returning its name on line 34.

1.4 Queue

Think of a queue like a line to your favorite roller coaster like in Figure 1.5. The first person in the park goes right to the front of the line. Everyone else files behind them. Every new person stands behind the previous person who got in line. We refer to this final person in line at the tail and the action of getting into line as enqueuing.

Figure 1.5: A queue for a super awesome roller coaster[3]

The front of the line is the head. When taking people out of the queue, we remove them from the head because they are next up have been waiting in line for the longest amount of time. This action is calling dequeuing.

A queue, similar to a stack, is another linked list. This time, we not only store the start of the list, but the end of it as well. We do this because we interact with the front and back of the queue. It also saves time to know where each end is, rather then combing though every node to find an end. We can see this below in Figure 1.6.

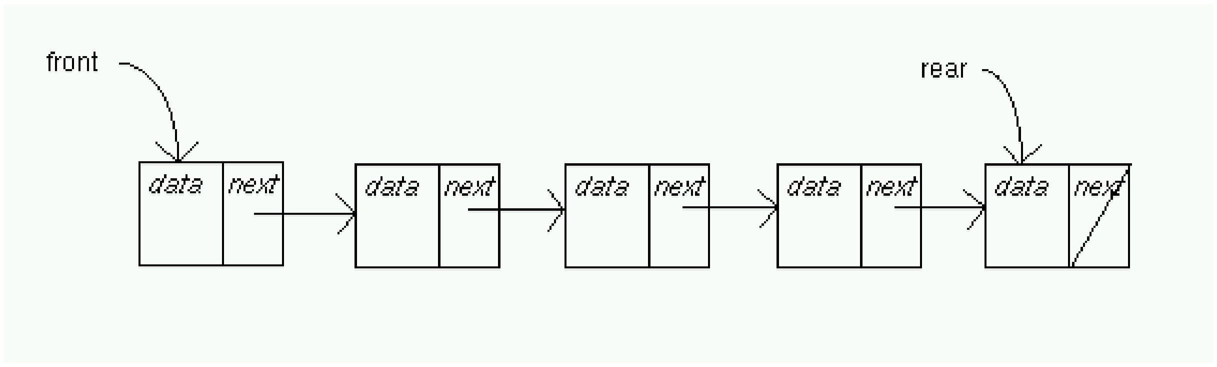

Figure 1.6: A queue as a linked list[4]

[3]Image: https://www.flickr.com/photos/geffawoof/183124579.
[4]Image: https://walker.cs.grinnell.edu/courses/195.fa01/lab.queues.html.

Now that we know what a queue looks like conceptually, let's see the code.

```
 1 class Queue{
 2 
 3 public:
 4     Node* head;
 5     Node* tail;
 6 
 7     //Constructor
 8     Queue(){
 9         head = nullptr;
10         tail = nullptr;
11     }
12 
13     //METHODS
14     //return true if Empty
15     bool isEmpty(){
16         if (tail == nullptr || head == nullptr){
17             return true;
18         }
19         return false;
20     }
21 
22     //enqueue/add to back of queue
23     void enqueue(Node* newTail){
24         //add a head if it is the first in line
25         if (isEmpty()){
26             head = newTail;
27         }
28         else {
29             //put new tail behind the old tail
30             tail->next = newTail;
31         }
32         //make this queue's tail this new tail
33         tail = newTail;
34     }
35 
36     //dequeue/retrieve from front of queue
37     string dequeue(){
38         //cant pop if there is nothing to pop
39         if (isEmpty()){
40             return "EMPTY STACK";
41         }
42         string dequeuing = head->itemName;
43         head = head->next;
44         return dequeuing;
45     }
46 };
```

Figure 1.7: Queue class in C++

At the top of Figure 1.7 on lines 4 and 5 are the attributes. Both point to nodes at either the `head` or `tail` of our queue. When constructed on line 8, the object's attributes both point to nothing. Similar to the Stack class, the queue's `isEmpty()` method return's true when the tail or head are undefined.

The `enqueue()` method takes a node pointer as a parameter. If the stack is empty, this node becomes both the head and tail. Otherwise, the previous tail's `next` points to this node, and this node becomes the new tail.

The `dequeue()` method returns an error message if the queue isn't populated. Otherwise, we store the previous head's name as `dequeuing`, appoint the new `head` pointer, and return that stored value.

1.5 File Reading

Now that we have our classes out of the way, its time to start the main function. Within this main function are a few components, one being the file reading. At the top of our file (outside of any function) two global, constant variables have been declared (See Figure 1.8). This is our file name and the number of items/lines in this file.

```
//GLOBAL variables
const int _NUM_OF_ITEMS = 666;
const string _FILE_NAME = "magicitems.txt";
```

Figure 1.8: Global variables in the code

Now, going into the main function. In Figure 1.9, we first open the file in line 1 and 2, and create an array called `magicItems` in line 3. Then, on line 9, we loop through every line of the file and assign the string value to an element in `magicItems`.

```
1      ifstream itemsFile;
2      itemsFile.open(_FILE_NAME);
3      string magicItems[_NUM_OF_ITEMS];
4
5      string currentLine;
6      if (itemsFile.is_open()){
7
8          //assign each line to an element in the array
9          for (int i = 0; i < _NUM_OF_ITEMS; i++){
10             std::getline(itemsFile, currentLine);
11             magicItems[i] = currentLine;
12         }
13     }
14
15     else {}
```

Figure 1.9: Copying the given file into an array

1.6 Palindrome Checking

Before we see the algorithm for checking for Palindromes, let's look at Figure 1.10 to see how we loop through the array to check each word. We loop through on line 3, and on line 6 we see that if the string is a palindrome, we will print it out.

```
1      //go through every magic item, check if PALINDROME
2      string item;
3      for (int i = 0; i < _NUM_OF_ITEMS; i++){
4          item = magicItems[i];
5
6          if (isPalindrome(item)){
7              cout << item << "\n";
8          }
9      }
```

Figure 1.10: Copying the given file into an array

```cpp
1  bool isPalindrome(string item){
2      Stack myStack;
3      Queue myQueue;
4      string character;
5      //arrays are for storage purposes only
6      Node myStackNodes[item.length()];
7      Node myQueueNodes[item.length()];
8  
9      //create 2 nodes for each character (appropriately)
10     for (int letterIndex = 0; letterIndex < item.length(); letterIndex++){
11 
12         if (item[letterIndex] != ' '){
13             //convert current character to an uppercase string
14             character = (1,toupper(item[letterIndex]));
15 
16             //make 2 nodes
17             myStackNodes[letterIndex].itemName = character;
18             myQueueNodes[letterIndex].itemName = character;
19 
20             //add each character to stack and queue
21             myStack.push(&myStackNodes[letterIndex]);
22             myQueue.enqueue(&myQueueNodes[letterIndex]);
23         }
24     }
25 
26     bool isPalindrome = true;
27     string dequeued = "";
28     string popped = "";
29 
30     //compare letter by letter for palindrome
31     while (!myQueue.isEmpty() && isPalindrome){
32         //search until all letters checked, or palindrome is proven wrong
33 
34         dequeued = myQueue.dequeue();
35         popped = myStack.pop();
36 
37         //compare the single character Nodes
38         if (dequeued != popped){
39             isPalindrome = false;
40         }
41     }
42     return isPalindrome;
43 }
```

Figure 1.11: The Palindrome function checks if the string is a palindrome using stacks and queues

Figure 1.11 above represents the palindrome checker. We start by creating a Stack object and a Queue object. On line 10, we loop through the string character by character. If the character is not a space, we create 2 nodes on line 17 and 18. Then, on lines 21 and 22, we pop one node and enqueue the other one. The stack and queue are full now.

Next, we'll set a boolean variable `isPalindrome` to true. And we will move through the string character by character once again. This time, either popping the stack or dequeuing the queue. One at a time we compare them on line 38. If any characters do not match, there is no palindrome. If we get through the whole stack and whole queue with no mismatches, we have a palindrome! We return our already true `isPalindrome` value.

And that's the palindrome section done! We used Nodes, Stacks, and Queues to store strings and characters, a file reader to read our magic items, and a loop to go through and print out each palindrome! Check out the results in Figure 1.12 below.

Palindromes
Boccob
Ebuc Cube
Olah Halo
radar
Robot Tobor
Dacad
UFO tofu
Dior Droid
Taco cat
Golf flog
Was It A Rat I Saw
Aibohphobia

Figure 1.12: The 12 palidromes printed from magicitems.txt

2 Sorting

2.1 Introduction

Given a set of items, there are many ways to sort it. Of these several ways, there are four methods that will be used and described in-depth.

This section will explain the algorithms and complexity of:

- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Using the same list of magic items from the first section, each given algorithm will sort the list alphabetically after it has been randomly shuffled.

2.2 Comparisons and Swapping

Let's first set up some global variables. In Figure 2.1 below, the number of items and the file name are constant variables. `_comparisons` will be used to count the number of times we have to compare two strings. This will metric will help us get a rough calculation for "Big Oh" or O().

```
//GLOBAL variables
const int _NUM_OF_ITEMS = 666; //CONSTANT number of magic items
const string _FILE_NAME = "magicitems.txt";
int _comparisons = 0; //count comparisons for each sorting method (resets on shuffle)
```

Figure 2.1: Global variables in the code

We will also need to be able to swap two elements. This function is simple, yet crucial. In Figure 2.2, we see an input of the array, and two index positions. Two switch them, we put the first string in a temporary variable, move the second to the first position, then put the temporary value into the second position.

```
void swap(string items[], int position1, int position2){
    string temp = items[position1];
    items[position1] = items[position2];
    items[position2] = temp;
}
```

Figure 2.2: Swap two elements of an array

```
 1 bool isLessThan(string first, string second){
 2 
 3     //find correct length to avoid out of bound error when comparing
 4     int length1 = first.length();
 5     int length2 = second.length();
 6     int length = (length1<length2)? length1 : length2;
 7 
 8     //+1 comparison
 9     _comparisons++;
10 
11     //compare letter by letter until an alphabetically 'smaller' string is found (disregard
       CAPs)
12     for (int i = 0; i < length; i++){
13         if (toupper(first.at(i)) < toupper(second.at(i))){
14             return true;
15         }
16         else if (toupper(first.at(i)) > toupper(second.at(i))){
17             return false;
18         }
19     }
20 
21     //tie goes to the shorter string
22     return (length1<=length2)? true : false;
23 }
```

Figure 2.3: Checks if first string is less than second string

Figure 2.3 shows the function when comparing two strings. We return true if the first string comes before the second alphabetically, and return false if the second string comes first. On line 8, we increment our number of comparisons because this function counts as one comparison. On line 12, we check the both strings using the length of the smaller string. Then, going through each letter, the code compares the values of the upper char types, disregarding capitalization.

The function returns true when the first string is found to be smaller alphabetically on line 14, or false on line 17 if the second string is smaller. If we reach line 22, this means we hit the end of a string. The result will point to the smaller string because these fall first alphabetically.

2.3 Shuffle

Before any sorting begins, there needs to be an "anti-sort" function where we can perform our sorts on. Everyone knows what shuffling a deck of cards looks like. It is a set of random moves done before any hand of a card game is played. The goal is to get the cards into a unique, random order.

Figure 2.4: An experienced casino dealer shuffles a deck of cards like it's nobody's business[5]

Our goal is to complete this algorithm in a complexity of O(n). So, Figure 2.5 shows the code for shuffling which involves a simple loop through all items on line 5. Then swaps the current string with a string at a random position in the array.

The comparisons variable is also reset to 0 on line 12 to start a new count for the next sorting algorithm.

```
1  void shuffle(string items[]){
2      srand(time(NULL)); //set RNG seed based on current time
3  
4      int length = _NUM_OF_ITEMS;
5      for (int i = length-1; i > 0; i--){
6          int randomIndex;
7          randomIndex = rand() % i;
8          swap(items,i,randomIndex);
9      }
10 
11     //reset comparisons count after each shuffle
12     _comparisons = 0;
13 }
```

Figure 2.5: The `shuffle()` function shuffles the given array

[5]Image: https://phys.org/news/2017-02-shuffle-cards-maths.html.

2.4 Selection Sort

Selection sort is one of the most basic sorting algorithms. In short, this algorithm loops through the array many times. Each time, the function loops through to find the smallest value. It takes this value and swaps with with the string that is in that correct position.

```
1  void selectionSort(string items[]){
2      int minPosition;
3  
4      for (int i = 0; i <= _NUM_OF_ITEMS - 2; i++){
5          minPosition = i;
6  
7          for (int j = i+1; j <= _NUM_OF_ITEMS - 1; j++){
8  
9              //compare for alphabetical order
10             if (isLessThan(items[j],items[minPosition])){
11                 minPosition = j;
12             }
13         }
14         swap(items, i, minPosition);
15     }
16 }
```

Figure 2.6: Selection Sort Algorithm

Going through the code in Figure 2.6, we start by looping through all array items execpt the last in line 4. We say that, for now, the value of position `i` is the minimum value on line 5. We then compare the minimum value to every other element to the left on line 7. If we find a smaller value on lines 10 and 11, we make this position or new `minPosition`, until we reach the end of the array.

When this happens, we swap the position `i` with the smallest position we found. Now, we increment `i` and the first i positions are sorted. We repeat until we have reached the end of the array.

With a nested for loop in this code on lines 4 and 7, this algorithm has a complexity of:

$$O(n^2)$$

Because for every loop through n for each position, we must loop once again through n to find the correct minimum value. Every other process is O(n) so they will not be included in the complexity. You can see one example number of comparisons when using selection sort in Figure 2.7.

Selection Sort		
n	comparisons	n^2
666	221445	443556

Figure 2.7: Number of comparisons for selection sort, O(n^2)

2.5 Insertion Sort

Insertion sort is another basic sorting algorithm. This algorithm loops through each item, then checks to see where it belongs in a sorted set to the left of it. When it finds something smaller than it, it will swap with the previously compared to value.

```
1  void insertionSort(string items[]){
2  
3      for (int i = 1; i < _NUM_OF_ITEMS; i ++){
4          int sortedIndex = i - 1;
5          int sortingIndex = i;
6  
7          //find item[i]'s spot in the already sorted part of list
8          while(sortedIndex >= 0 && isLessThan(items[sortingIndex],items[sortedIndex])){
9              swap(items, sortedIndex, sortingIndex);
10             sortedIndex --;
11             sortingIndex --;
12         }
13     }
14 }
```

Figure 2.8: Insertion Sort Algorithm

Going through the code in Figure 2.8, we start by looping through all items in line 3. Starting at index 1 this is our sorting index, we then compare it to the values to the left (in the first case only one) this is the sortED index. On line 8, we enter a loop that asks if the sorting is less than the sortED index's value. If it is, we swap the elements. Then we lower the sorted and sorting and keep comparing.

We keep going until either we find a smaller value to the left that we leave in place or we have reached the beginning of the array.

With a nested loop in this code on lines 3 and 8, this algorithm has a complexity of:

$$O(n^2)$$

Because for every loop through n for each value, we must loop through the sorted items to the left to find a correct spot. Every other process is O(n) so they will not be included in the complexity. Figure 2.9 shows an output using insertion sort.

Insertion Sort		
n	comparisons	n^2
666	109424	443556

Figure 2.9: Number of comparisons for insertion sort, O(n^2)

2.6 Merge Sort

Merge sort is a quicker sorting algorithm that divides the given array in half. It then divides those arrays in half and so on until each array is only one element long. Then, each array is merge with another until the whole thing is sorted. Check out the visual in Figure 2.10

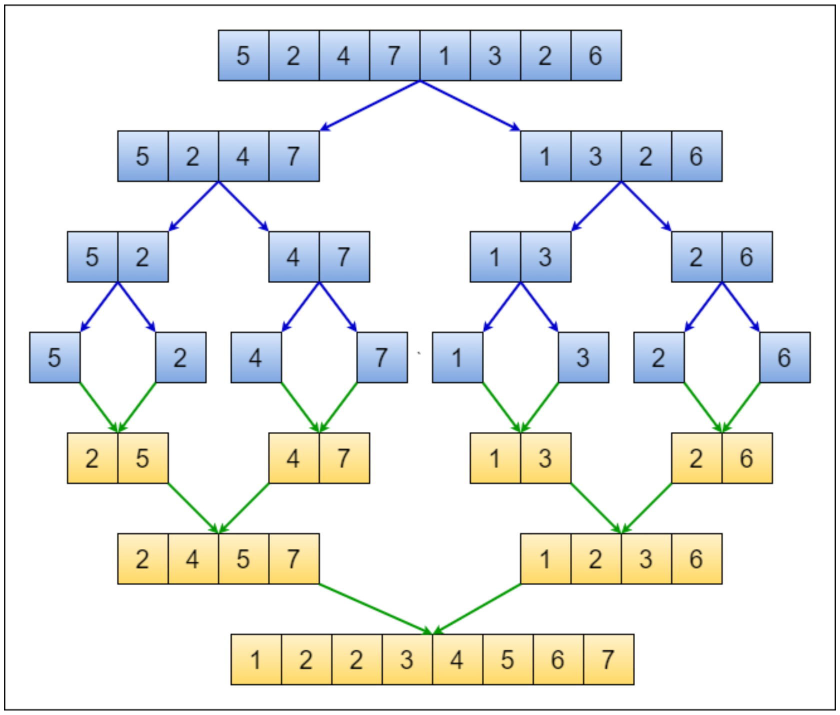

Figure 2.10: Visual representation of how an array of length 8 is sorted in merge sort[6]

Using the code in Figure 2.11 on the next page, merge sort proves that it is a recursive algorithm: an algorithm that calls itself.[7]

On line 3, we check if the length is greater than 1, this is the base case. Which we will come back to. Lines 5 through 22 create two arrays: one consisting of the elements in the first half of the larger, given array, and one consisting of the elements of the second half. Look at Figure 2.10 from the top row to the next row.

Merge sort is then recursively called for each half, halving each array until there are n arrays of size 1 (the base case!).

Now that we have divided the array, it is time to sort. We've reached the base case so we combine the arrays (moving from blue to yellow in the above figure). This brings us to line 28 of Figure 2.11. For the length of the given array, we check if the first element in each belongs first or not. Then we put it in place. Say this was in `half1`, next, we check the first element of `half2` and the second element of `half1` because the first element has been placed into the array. Lines 32-39 are in place to make sure that if we run through all of `half1`'s elements or `half2`'s elements, it adds the rest of the elements from the other one into the final array.

We repeat this process of merging `half1` and `half2` until merge sort is done calling itself and we have a sorted list like the final yellow row in Figure 2.10.

[6]Image: https://medium.com/javarevisited/visualizing-designing-and-analyzing-the-merge-sort-algorithm-904ceb78a592.
[7]Pseudocode courtesy of: https://pseudoeditor.com/guides/merge-sort.

```
void mergeSort(string items[], int length){
    //Base case: when length 1, start conquering, until then, divide
    if (length > 1){

        int firstHalfLength = length/2;
        int secondHalfLength = length - firstHalfLength;
        string half1[firstHalfLength];
        string half2[secondHalfLength];

        //fill in half1
        for (int i = 0; i < firstHalfLength; i++){
            half1[i] = items[i];
        }

        //fill in half2
        for (int i = 0; i < secondHalfLength; i++){
            half2[i] = items[firstHalfLength+i];
        }

        //sort each half
        mergeSort(half1, firstHalfLength);
        mergeSort(half2, secondHalfLength);

        int sortingPos = 0;
        int firstPos = 0;
        int secondPos = 0;

        while (sortingPos < length){

            //if one half is completely put into sorted array,
            //put the rest of the other half into sorted array
            if (firstPos == firstHalfLength){
                items[sortingPos] = half2[secondPos];
                secondPos ++;
            }
            else if (secondPos == secondHalfLength){
                items[sortingPos] = half1[firstPos];
                firstPos ++;
            }

            //sort 2 ordered arrays into one larger array
            else if (isLessThan(half1[firstPos],half2[secondPos])){
                items[sortingPos] = half1[firstPos];
                firstPos ++;
            }
            else {
                items[sortingPos] = half2[secondPos];
                secondPos ++;
            }

            //move to next position to be sorted
            sortingPos ++;
        }
    }
}
```

Figure 2.11: Merge Sort Algorithm

The recursive Merge sort algorithm has a complexity of:

$$O(nlog(n))$$

We first start by dividing the array in half, then each in half again and again until we get to arrays of size one. This is how a logarithm function works in base 2. This is shown in lines 5-22. And for every merge sort, we must also traverse both arrays on line 28 to put all the elements into one sorted array. This complexity is O(n) because it hits each element once. Combining these two complexities, we get O($nlog(n)$)

Figure 2.12 shows one example number of comparisons when sorting with merge sort.

Merge Sort		
n	comparisons	$nlog(n)$
666	5430	6246.7

Figure 2.12: Number of comparisons for merge sort, O($nlog(n)$)

2.7 Quick Sort

Quick sort is the final sorting algorithm. This fast method also divides the given array into parts (like merge sort), but not equal halves. In Figure 2.13, we randomly select an element in the array (blue box on top). We then check each element and compare it to this pivot value. If it is less than, we put it on the left. If it is greater we put it on the right. These right and left sides are then quick sorted themselves until the whole array is solved!

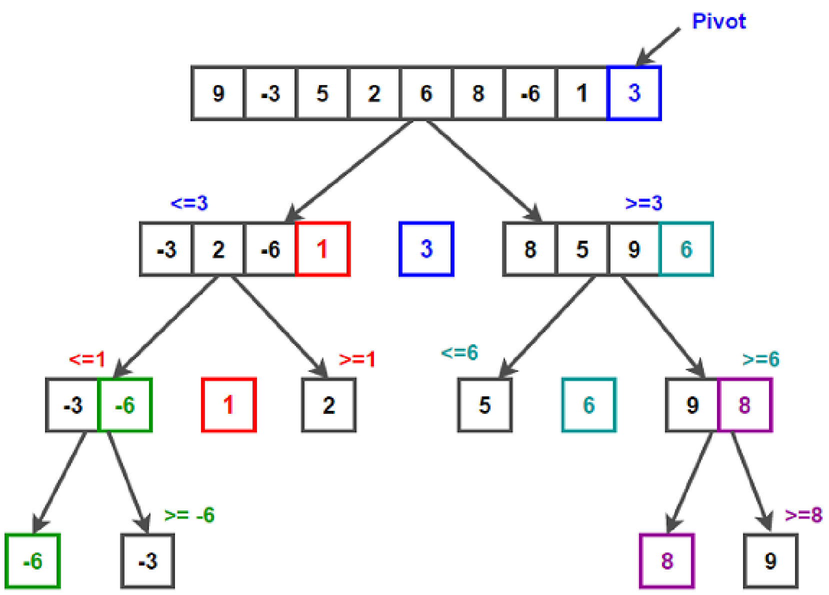

Figure 2.13: Visual representation of how an array of length 8 is sorted in quick sort[8]

[8]Image: https://cs.brown.edu/courses/csci0111/bridge-to18/sorting-lists.html.

```
void quickSort(string items[], int length){

    srand(time(NULL)); //set RNG seed based on current time

    if (length > 1){
        int splitIndex;
        int comparisonsWithOutRands = _comparisons;

        //choose 3 pivot candidates
        if (length >= 3){
            int rand1 = rand() % length;
            int rand2 = (rand1 + 1) % length;
            int rand3 = (rand1 + 2) % length;

            string item1 = items[rand1];
            string item2 = items[rand2];
            string item3 = items[rand3];

            //pick middle valued index
            if ( (isLessThan(item2,item1) && isLessThan(item1,item3)) || (isLessThan(item3,
    item1)) && (isLessThan(item1,item2)) ){
                splitIndex = rand1;
            }
            else if ( (isLessThan(item1,item2) && isLessThan(item2,item3)) || (isLessThan(
    item3,item2)) && (isLessThan(item2,item1)) ){
                splitIndex = rand2;
            }
            else {
                splitIndex = rand3;
            }
        }

        //choose 1 pivot index if there aren't enough candidates
        else {
            splitIndex = rand() % length;
        }

        //sort into left, center(r), and right
        int r = partition(items, splitIndex, length);


        //Create left and right half arrays
        int firstHalfLength = r;
        int secondHalfLength = length - r - 1;
        string left[firstHalfLength];
        string right[secondHalfLength];

        //fill in left (0,r-1)
        for (int i = 0; i < firstHalfLength; i++){
            left[i] = items[i];
        }

        //fill in half2 (r+1,length-1)
        for (int i = 0; i < secondHalfLength; i++){
            right[i] = items[firstHalfLength+i+1];
        }

        //merge sorted each half
        quickSort(left, firstHalfLength);
        quickSort(right, secondHalfLength);

        //merge sorted arrays into items[]
        for (int i = 0; i < firstHalfLength; i++){
            items[i] = left[i];
        }
        //item[r] stays item[r]
        for (int i = r+1; i < length; i++){
            items[i] = right[i-r-1];
        }
    }
}
```

Figure 2.14: Quick Sort Algorithm

Using the code in Figure 2.14 on the previous page, merge sort proves that it is a recursive algorithm, it divides, then conquers, then divides again, and conquers again until we are done.

On line 3, we check if the length is greater than 1, this is the base case. Which we will come back to. Then, if there are more than 3 elements, we pick three random elements and choose the middle value to be our pivot in lines 10 through 29. If there's less, we pick one pivot.

Next, we partition the array about the pivot. Look at Figure 2.13 from the top row to the next row, this is what partitioning is doing. This is done in the `partition()` function in Figure 2.15. We put our pivot value at the end on line 3. We then go through each element to check if it is greater than or less than the pivot on line 7 and 9. We move higher numbers to the right side (this is done with the `higherPos` variable) and keep the smaller ones on the left with line 12. We then swap that pivot value with a high right value close to that middle.

From here we move back out to `quickSort()` where we have our partitioned array. And just like merge sort, we will now sort the right side and the left side seperately on lines 57 and 58 of Figure 2.14 half of the larger, given array, and one consisting of the elements of the second half. Quick sort is then recursively called for each half, halving each array until there are n arrays of size 1.

We then combine the sorted `left` and `right` arrays with the middle pivot value into a larger one on lines 61 through 67. This is combination step is repeated as we exit the recursive calls and end up with a fully sorted array.

```
 1 int partition(string items[], int splitIndex, int length){
 2 
 3     swap(items, splitIndex, length-1);
 4 
 5     //This variable tracks the left-most index of elements that belong to the right of pivot
      value
 6     int higherPos = -1;
 7     for (int checking = 0; checking < length-1; checking++){
 8 
 9         if(isLessThan(items[checking], items[length-1])){
10             higherPos++;
11             //move higher values to the right
12             swap(items, higherPos, checking);
13         }
14     }
15 
16     //swap pivot and value the should be right of pivot
17     swap(items, length-1, higherPos+1);
18     return higherPos + 1;
19 }
```

Figure 2.15: Partition Algorithm[9]

[9]Pseudocode courtesy of: Jeff Erickson's *Algorithms*.

Are you sweating? I'm sweating. The recursive quick sort algorithm has a complexity of:

$$O(nlog(n))$$

Here's why. We start off by dividing the array into two parts and one pivot item. By diving the array roughly in half we are essentially limiting the complexity to $log(n)$ for this step. After spliting, we go through each of our n elements to check where they belong (left or right). This part has a complexity of n. Meaning, when we combine our two parts, we get O($nlog(n)$).

Because of the randomness of quick sort, the worst possible case is n^2. We avoid this case because of the fact that we picked 3 candidate options for pivots on lines 10 through 29 in Figure 2.14.

The following Figure 2.16 is an output of one case where quick sort was used.

Quick Sort		
n	comparisons	$nlog(n)$
666	6848	6246.7

Figure 2.16: Number of comparisons for quick sort, O($nlog(n)$)