CMPT 435L ALGORITHMS
Assignment 1

October 4th 2024

Jonathan "Grant" Bever

Link to github repo

PART 1: Using Stacks and Queues to identify palindromes

In order to use stacks and queues to identify palindromes we must first understand what they are and how their built-in operations are perfect for this task. While both data structures are quite similar in the fact that they consist of nodes (elements of the data structure that have a piece of data and a pointer to the next node), their operations are what separates them.

Stacks are Last In First Out, meaning the last node added is the first to go, think a stack of plates, you put a plate on top and you pull from the top. this is in contrast to Queues which are First In First out, for this think of a line of people, people join the line from the back and exit from the front. These data structures operations are as follows:

Stacks:	**Queues:**
Push() Adds a node to top of stack	enQueue() Adds node to top of queue
Pop() Removes node from top of stack	deQueue() Removes node from end of queue

There is also isEmpty() which checks if a node is empty

Refer to the page below for my implementation for this:

The first letter of a palindrome is the same as the last so if we put the same word into a stack and queue and the letters we got when we pushed and popped until isEmpty() are the same, then that word is a palindrome.

Listing 1: Stack Class

```
export class Stack {
  top: node | null;

  constructor() {
    this.top = null;
  }

  // LAST IN
  push(data: string): void {
    const newNode = new node(data);
    newNode.next = this.top;
    this.top = newNode;
  }

  // FIRST OUT
  pop(): string | null {
    if (this.top === null) return null;
    const poppedNode = this.top;
    this.top = this.top.next;
    return poppedNode.data;
  }

  isEmpty(): boolean {
    return this.top === null;
  }
}
```

Listing 2: Queue Class

```
export class Queue {
  front: node | null;
  rear: node | null;

  constructor() {
    this.front = null;
    this.rear = null;
  }

  // FIRST IN
  enqueue(data: string): void {
    const newNode = new node(data);
    if (this.rear === null) {
      this.front = this.rear = newNode;
    } else {
      this.rear.next = newNode;
      this.rear = newNode;
    }
  }

  // FIRST OUT
  dequeue(): string | null {
    if (this.front === null) return
        null;
    const dequeuedNode = this.front;
    this.front = this.front.next;
    if (this.front === null) this.rear
        = null;
    return dequeuedNode.data;
  }

  isEmpty(): boolean {
    return this.front === null;
  }
}
```

With these classes in place we can go ahead and create a simple function that takes in magicitems.txt, turns it into an array that's readable for stacks and queues (no spaces or caps) and feed it word by word into the algorithm

1 Palindrome Detection Code

```
1  for (let i=0;i<dataItems.length;i++){
2      const word = dataItems[i];
3      const {stack,queue}=
           processItemsWithStackQueue(word);
4
5  //Compare the stack and queue to check for
       a palindrome
6      if (compareStackQueue(stack, queue)) {
7        console.log(`${word} is a palindrome
             `);
8        palendromes.push(word);
9      }
10 };
```

```
1  export function processItemsWithStackQueue
       (word: string):{stack:Stack,queue:
       Queue}{
2    const stack = new Stack();
3    const queue = new Queue();
4  //Splits into letters -> pushes and
       enqueues it
5    for (const letter of word) {
6      stack.push(letter);
7      queue.enqueue(letter);
8    }
9    // back to index
10   return { stack, queue };
11 }
```

Line 3 on the left chunk of code sends the word to the function on the right where the word is pushed and enqueued character by character then sent back only to get sent to the function below where all the cool stuff happens.

```
1  function compareStackQueue(stack: Stack, queue: Queue): boolean {
2    while (!stack.isEmpty() && !queue.isEmpty()) {
3      if (stack.pop() !== queue.dequeue()) {
4        return false;
5      }
6    }
7    return true;
8  }
```

This function is super simple but can be a bit hard to read because it deals with double negatives, lets dissect lines 2 and 3 because that's where everything happens.

Line 2 sets the while loop under the conditions of the loop while the stack isn't empty and the queue isn't empty aka while there are still elements in the data structure. Line 3 pops and dequeues an element, if that element is not the same then it moves on to the next word, if the element is the same then it keeps looping until all items in the array have been checked and all 15 palindromes are found.

PART 2: Shuffling and Sorting

Now that all the palindromes have been found let's move on to sorting.
In this assignment, we will be implementing the selection sort, insertion sort, merge sort, and quick sort but before that...

Knuth Shuffle

In order to test our sorts we first need to shuffle the array. So we will be making a version of the Knuth Shuffle. The Knuth shuffle is a simple yet thorough shuffle that works by randomly selecting items from the array and swaps them into a "shuffled zone" that starts at the end of the array.

```
1 export function shuffle(items: any) {
2     for (let i = items.length - 1; i > 1; i--) {
3         const j = Math.floor(Math.random() * (i + 1));
4         [items[i], items[j]] = [items[j], items[i]];
5     }
6     return items;
7 }
```

We see this working at line 2 where it starts the for loop at length - 1 because "the shuffled zone" starts at length and increases as the index decreases. Through random selection on line 3, and swapping on line 4, shuffled items are brought into that zone and the index decrements until the entire array is shuffled.

Now that our array is shuffled we can move on to our first sort, the selection sort.

Selection Sort

Selection sort goes through an array and finds the minimum value in the unsorted portion and swaps it out keeps going until the whole thing is sorted.

```
export function selectionSort(items: []) {
    let comparisons = 0;

    for (let i = 0; i < items.length - 1; i++) { // i loop
        let smallNum = i;
        for (let j = i + 1; j < items.length; j++) { // j loop
            comparisons++;
            if (items[j] < items[smallNum]) {
                smallNum = j;
            }
        } // end i loop
        if (smallNum !== i) {
            [items[i], items[smallNum]] = [items[smallNum], items[i]];
        }
    } // end j loop
    console.log("Selection sort had", comparisons, "comparisons");
    return items;
}
```

The selection sort is O(n^2) we can easily see this since it is a nested loop. For a n of 666, the number of comparisons I get is 221445, unlike the other sorts this is the only result. For each sort we can calculate what the number of comparisons should be. For Selection sort the algorithm is $\frac{(n-1)n}{2}$ If we plug n = 666 into this we get 221445, meaning this is correct.

Insertion Sort

Insertion sort works left to right examining each item and comparing it to the items on its left, then we insert the item in the correct position in the array.

```
export function insertionSort(items: string[]){
    let comparisons = 0;
    for(let i = 1; i < items.length; i++){
        let selected = items[i];
        let j = i - 1;
        while (j >= 0 && (items[j] > selected)) {
            comparisons++;
            items[j + 1] = items[j];
            j--;
        }
        items[j + 1] = selected; }}
```

The amount of comparisons for any given insertion sort is heavily reliant on how sorted the array is, since we shuffle the array every time it runs we will get a different result each time. Because of the nested loops insertion sort is $O(n^2)$ however if the array is already sorted then it is only O(n). While we cannot get an exact figure we can use averages to find out what we should expect. The average case about equals $n^2/4$, plugging n = 666 into this we get 110889 which is about the same as what we get from this function.

Merge Sort

The next two sorts work with a "divide-and-conquer" methodology, in theory, you just split it up and sort it while you put it back together, but the implementation, as you will see, is much easier said than done.

Listing 3: Merge Sort Function

```
1  let comparisons = 0;
2
3  export function mergeSort(items: string[]): string[] {
4      if (items.length <= 1) {
5          return items;
6      }
7      const middle = Math.floor(items.length / 2);
8      const left = mergeSort(items.slice(0, middle));
9      const right = mergeSort(items.slice(middle));
10
11     const merged = merge(left, right);
12     return merged;
13 }
14 function merge(left: string[], right: string[]): string[] {
15     let sortedItems: string[] = [];
16     let leftIndex = 0;
17     let rightIndex = 0;
18
19     while (leftIndex < left.length && rightIndex < right.length) {
20         comparisons++;
21         if (left[leftIndex].toLowerCase() < right[rightIndex].toLowerCase()) {
22             sortedItems.push(left[leftIndex]);
23             leftIndex++;
24         } else {
25             sortedItems.push(right[rightIndex]);
26             rightIndex++;
27         }
28     }
29
30     return sortedItems
31            .concat(left.slice(leftIndex))
32            .concat(right.slice(rightIndex));
33 }
```

Merge sort works recursively that's why on lines 8 and 9 the function calls itself, this is part of the splitting up portion of the algorithm. On line 14 we make a new function that does the second part of the algorithm where it puts the whole thing back together. It is during this process that all the comparisons (line 21) and the switching (line 22 and line 25) happen. Like insertion sort the amount of comparisons heavily depends on the sortedness of the array, which is why as we shuffle we will get a different result every time.

Merge sort is $O(n \log_2 n)$ and with n = 666 we get an average of 6237, which is a bit above my average of 5420.

Quick Sort

Quick sort is also a "divide-and-conquer" method but is much simpler than merge sort. For this sort, the algorithm chooses a pivot point somewhere in the middle (line 5) and swaps the numbers to the left and right of that point until it's sorted (lines 8 - 13). then puts it all together (line 16).

```
1  export function quickSort(items: string[]): string[] {
2      if (items.length <= 1) {
3          return items;
4      }
5      const pivot = items[Math.floor(items.length / 2)];
6      const left = [];
7      const right = [];
8      for (const item of items) {
9          comparisons++;
10         if (item < pivot) {
11             left.push(item);
12         } else if (item > pivot) {
13             right.push(item);
14         }
15     }
16     return quickSort(left).concat(pivot, quickSort(right)); }
```

Finding the running time for quick sort is tricky, like merge sort the running time is $O(n \log_2 n)$ for the best/average case where the pivot is in the middle, with bad luck quick sort has the potential to be $O(n^2)$. Calculating with $O(n \log_2 n)$ we find an estimation of 6237. When running my version of quickSort my averages are in the low 7000s which, while is in the range to be fine, could indicate an off-by-one error.

Running Time Table

Sort	AVG Comparisons	Time Complexity
Selection Sort	221445	$O(n^2)$
Insertion Sort	113325	$O(n^2)$
Quick Sort	7241	$O(n \log_2 n)$
Merge Sort	5421	$O(n \log_2 n)$