Assignment One

Christian Sarmiento

christian.sarmiento1@marist.edu

October 5, 2024

1 Introduction

Assignment One is focused on creating a program that works with a list of 666 magic items in string format(magicitems.txt). We first find palindromes that exist in the list using both a stack and queue data structure. After finding these palindromes, the list is then sorted four separate times using Insertion, Selection, Merge, and Quick Sort. The number of comparisons are kept track of for each of the sorts. This program was developed using C++.

2 Finding Palindromes

Out of the 666 magic items, there are 15 palindromes (boccob, seussignitingissues, ufotofu, ebuccube, aibohphobia, tacocat, wasitaratisaw, olahhalo, cdcasedividesacdc, dacad, diordroid, robottobor, narcinapanicran, radar, golfflog). We used a stack and a queue to load each character of the string and then popped/dequeued each letter from both data structures and compared. C++ syntax made it interesting to work with this data since there are multiple ways to manipulate data. To clean the data, we needed to make everything lowercase and remove spaces. In C++ this can be done by using these commands which modifies the array in-place:

```
1     // Clean data by removing spaces and normalizing capatilization
2
3     // remove spaces
4     magicItem.erase(std::remove(magicItem.begin(), magicItem.end(), ' '), magicItem.end());
5
6     // lamda function for lowercase conversion
7     std::transform(magicItem.begin(), magicItem.end(), magicItem.begin(), [](unsigned char c
          ) {
8
9     return std::tolower(c);
10
11    });
```

Line 4 deals with actually erasing the spaces from in between the substrings. Specifically, it moves all the spaces to the end and takes them out, directly modifying the string. Line 7 deals with taking each character and converting it to lowercase using a lambda function. The alternative here is to use a for loop and go through each character in the string, but this seemed cleaner.

```
1     // Define stack and queue objects with automatic stack allocation
2     Stack stack;
3     Queue queue;
4
5     // Load each character from the string into a stack and queue
6     for (size_t j=0; j < magicItem.length(); j++) {  // size_t for incrementor since .length
          () returns that type
7
8         // Load each character into the stack and queue
9         stack.push(magicItem[j]);
10        queue.enqueue(magicItem[j]);
11
12    } // for j
13
14    // Check each character in stack and queue and compare
15    for (size_t k=0; k < magicItem.length(); k++) {
16
17        // Get characters from the stack and the queue
18        character1 = stack.pop();
19        character2 = queue.dequeue();
20
21
22        // Check if the character matches or not
23        if (character1 != character2) {
24
25            isPalindrome = false;
26
27        } // if
28
29    } // for k
30
31    // Print if it's a palindrome
32    if (isPalindrome) {
33
34        std::cout << magicItem << std::endl;
35        palindromeCount++;
36
37    } // if
```

After the values are cleaned, we can then check each string to confirm if it is a palindrome or not. First, each character is loaded into a stack and a queue (Line 6). Then, once the strings are loaded into the data structures, they are popped/dequeued from their respective data structure and compared (Lines 18 - 27). Because of how stacks and queues are built, we are checking opposite ends of the string. This will only fail if the string is not a palindrome.

3 Sorting

After the Palindromes are found, we sort the lists alphabetically using Selection, Insertion, Merge, and Quick Sort. To do that, we must first shuffle the array so that we have something to sort.

3.1 Knuth (Fisher-Yates) Shuffle

The Knuth Shuffle is a shuffle algorithm that ensures the array given is randomly shuffled by selecting a random index and swapping with the first value, decrementing the selection window by 1 until all elements have been swapped. Here is my implementation:

```
1         void knuthShuffle(std::vector<std::string>& strings) {
2
3             // Variables
4             int endPos = strings.size() - 1;
5             srand((unsigned) time(NULL));  // setting seed value
6
7             // Shuffle the array
8             for (int i = endPos; i >= 0; i--) {
9
10                int j = rand() % (i + 1);
11                swap(strings[i], strings[j]);
12
13            } // for
14
15        }; // knuthShuffle()
```

As you can see, we set the seed value as the current time to ensure there is a random set of numbers to select from that is unique from run to run. We take the modulo inside of the for loop to ensure that the number we are selecting is always within range of our indexes. Something to note that I thought was interesting about C++ syntax is the fact that there are different ways to pass along parameters, such as including "&" after the type to indicate that we are passing a variable by reference and therefore directly modifying the object being passed. To me, it highlighted the granularity of C++ and why it would be a really good language to write efficient code that is conservative on memory and thus a good candidate for things like video games, GPUs, or self-driving cars.

3.2 Selection Sort

```
1         int selectionSort(std::vector<std::string>& items) {
2
3             // Variables
4             int numComparisons = 0;
5             int arrayLength = items.size();
6
7             // Anchor on an element
8             for (int i=0; i <= arrayLength - 2; i++) {
9
10                // Find the minimum
11                int currentMin = i;
12                for (int j=i+1; j <= arrayLength - 1; j++) {
13
14                    // Swap only if the number is less than the current mininum
15                    numComparisons++;
16                    if (items[j] < items[currentMin]) {
17
18                        currentMin = j;
19
20                    } // if
21
22                } // for j
23
24                // Swap after finding the minimum
25                swap(items[i], items[currentMin]);
26
27            } // for i
28
29            return numComparisons;
30
31        } // selectionSort()
```

Selection Sort works by selecting a value by iterating through the array (Line 8) and finding and swapping with the minimum element (Line 11 - Line 25) until the list is fully sorted. We start two down from the length of the array since the inner loop starts +1 from where the outer loop is at. This avoids going off-bound from where the list actually is. The inner loop (Line 12) fully focuses on finding that minimum element, so we swap after we find it, or in other words, after the inner loop (Line 25).

3.3 Insertion Sort

```
1         int insertionSort(std::vector<std::string>& items) {
2
3             // Variables
4             std::string currentElement = " ";
5             int numComparisons = 0;
6
7             // Anchor on an element
8             for (int i=0; i < items.size() - 1; i++) {
9
10                // Move elements if they are bigger than the current element
11                currentElement = items[i];
12                int j = i - 1;
13                numComparisons++;
14                while (j >= 0 && items[j] > currentElement) {
15
16                    numComparisons++;
17                    items[j+1] = items[j];
18                    j--;
19
20                } // while
21
22                // Move current element into proper place
23                items[j+1] = currentElement;
24
25            } // for i
26
27            return numComparisons;
28
29        } // insertionSort()
```

Insertion Sort works by essentially partitioning the array in-place between sorted and unsorted, adding an element each time from the unsorted to the sorted portion by inserting it where it belongs according to the sort. We must iterate through each element and anchor on one of the elements to serve as the sorted portion, with the first iteration being a sub array of size 1 (Line 8). Elements are then sorted using the inner loop (Line 14) only if the element before it is greater than the anchor we have. This happens until we have gone through everything before the anchor (Line 18) and then we repeat the process until the array has been exhausted of usable anchors.

3.4 Merge Sort

```
        int mergeHelper(std::vector<std::string>& array, int left, int mid, int right) {

            .
            . //    Variable definitions
            .

            // Put the respective halves into the temp arrays
            for (int i=0; i < leftHalf; i++) {

                leftArray[i] = array[left + i];

            } // for i
            for (int j=0; j < rightHalf; j++) {

                rightArray[j] = array[mid + 1 + j];

            } // for j

            // Merge the arrays together
            int k = 0, l = 0, m = left;
            while (k < leftHalf && l < rightHalf) {

                numComparisons++;
                if (leftArray[k] <= rightArray[l]) {

                    array[m] = leftArray[k];
                    k++;

                } // if
                else {

                    array[m] = rightArray[l];
                    l++;

                } // else

            } // while

            // Copy remaining elements if there are any left
            while (k < leftHalf) {

                array[m] = leftArray[k];
                k++;
                m++;

            } // while

            while (l < rightHalf) {

                array[m] = rightArray[l];
                l++;
                m++;

            } // while

            return numComparisons;

        } // mergeHelper

        int mergeSort(std::vector<std::string>& items, int leftPos, int rightPos) {

            // Variables
            int totalComparisons = 0;
```

```
64
65              // Check to make sure there are still items to sort
66              if (leftPos < rightPos) {
67
68                  // Get the midpoint
69                  int midpoint = leftPos + (rightPos - leftPos) / 2;
70
71                  // Sort the list using recursion
72                  totalComparisons += mergeSort(items, leftPos, midpoint);
73                  totalComparisons += mergeSort(items, midpoint + 1, rightPos);
74
75                  // Merge the sorted arrays
76                  totalComparisons += mergeHelper(items, leftPos, midpoint, rightPos);
77
78              } // if
79
80              return totalComparisons;
81
82          } // mergeSort()
```

Merge Sort makes use of recursion to sort a list using divide and conquer. First, the sort will divide the array into two halves by finding the midpoint (Line 60) and then recursively calling the sort (Lines 72, 73) until there is nothing more to divide. Then, using loops in the mergeHelper() function (Line 1), we merge the divided arrays together while ensuring that they are sorted in the process. Specifically, each half of the array is sorted and then put together in the same way it was divided.

3.5 Quick Sort

```
1         int choosePivot(std::vector<std::string>& array, int left, int mid, int right) {
2
3             // Variables
4             int pivotPostion = 0;
5
6             // Find middle value out of left, mid, and right indexes
7             if (array[left] < array[mid] && array[mid] < array[right])
8                 pivotPostion = mid;
9
10            else if (array[mid] < array[left] && array[left] < array[right])
11                pivotPostion = left;
12
13            else
14                pivotPostion = right;
15
16            return pivotPostion;
17
18        } // choosePivot()
19
20        std::vector<int> partition(std::vector<std::string>& array, int startPos, int endPos
              , int pivotPoint) {
21
22            // Variables
23            int l = startPos - 1;
24            int numComparisons = 0;
25            std::vector<int> output(2);
26
27            // Swap elements at the partition
28            swap(array[pivotPoint], array[endPos]);
29
30            // Sort through partitions
31            for (int i = startPos; i <= endPos - 1; i++) {
32
33                numComparisons++;
34                if (array[i] < array[endPos]) {
35
36                    l++;
37                    swap(array[l], array[i]);
38
39                } // if
40
41            } // for i
42
43            // Final swap
44            swap(array[endPos], array[l+1]);
45
46            // Output
47            output[0] = l + 1;
48            output[1] = numComparisons;
49
50            return output;
51
52        } // partition()
53
54        int quickSort(std::vector<std::string>& items, int leftPos, int rightPos) {
55
56            //Variables
57            int totalComparisons = 0;
58            int pivot = 0;
59            int midpoint = 0;
60            int partitionPoint = 0;
61            std::vector<int> response(2);
62
```

```
63              // Base case
64              if (leftPos < rightPos) {
65
66                  // Choose pivot
67                  midpoint = leftPos + (rightPos - leftPos) / 2;
68                  pivot = choosePivot(items, leftPos, midpoint, rightPos);
69
70                  // Partition the array
71                  response = partition(items, leftPos, rightPos, pivot);
72                  partitionPoint = response[0];
73                  totalComparisons += response[1];
74
75                  // Recursive sort
76                  totalComparisons += quickSort(items, leftPos, partitionPoint - 1);
77                  totalComparisons += quickSort(items, partitionPoint + 1, rightPos);
78
79              } // if
80
81              return totalComparisons;
82
83          } // quickSort()
```

Quick Sort works similar to Merge Sort, but instead of dividing it in half, a pivot value is selected (Line 68) which serves as the partition point for the sort. Each partitioned section is then sorted through a series of recursive calls (Lines 76, 77), which works by ensuring that all the elements that are less than the partition point stay to the left and all the elements greater stay to the right (Line 31), resulting in a sorted array. This goes for both sorts, but I thought it should be noted that the C++ implementation of these recursive sorts were weird to implement since you can't use list slicing on vectors in C++ the way you would in other languages, which made it interesting to figure out how to split up the arrays while still modifying the array directy. The fix to that was passing along the start and end indexes of the arrays as parameters to the functions so that the arrays can be modified through the parameters instead of passing along a sliced subset of the arrays.

3.6 Results

I tested my sorts by running them on 20 trial runs each, averaging them out. For the purposes of clarity in presenting the data, I am going to display on the 1st, 10th, and 20th alongside the average.

Sort	Expected Value - $\frac{E(O(n))}{2}$	Test 1	Test 10	Test 20	Average
Selection Sort $O(n^2)$	221778	221445	221445	221445	221445
Insertion Sort $O(n^2)$	221778	108914	108868	108868	108870
Merge Sort $O(n \log_2 n)$	3123.33	3111	3237	3277	3229
Quick Sort $O(n \log_2 n)$	3123.33	5696	5696	5696	5696

Most of these sorts perform as expected, especially Selection and Merge Sort. Insertion Sort performs better than expected and Quick Sort performs a little worse than expected, most likely due to the selection of pivot values. We expect the $O(n^2)$ sorts (Selection, Insertion) to perform the way they do due to the nested loops that each of them utilize to complete the sorts. This results in a way bigger run time in comparison to the sorts that divide and conquer their way to the sort (Merge, Quick). Division is expressed through logs, which makes it a way faster option than going through each element in the array. Our data represents the notion.

4 References

- Rosetta Code: Knuth Shuffle
- Rosetta Code: Insertion Sort
- Digital Ocean - RNG in C++
- Geeks for Geeks - C++ Classes and Objects
- Stackover Flow - Quick Sort: Choosing a Pivot Value
- ChatGPT-4o - Help with basic C++ syntax and debugging help with prompts like:
  - "How does list slicing work in C++? How does it differ from JavaScript or Python?
  - "What is the difference between signed and unsigned variables?"
  - "How can i define a parameter so that I can directly modify an object? How can I define a parameter so that it is just a copy I am working with?"