Gianna Julio

CMPT435 - Algorithm Analysis and Design
Assignment 2 - Sorting

Assignment2.cpp

Loading Items From File

```
3  #include <fstream>      // file reading
4  #include <iostream>     // file reading
5  #include <iomanip>      // output formatting
6  #include <stdio.h>      // output formatting
7  #include <string>       // strings
8  #include <time.h>       // used for random seed
9  #include <algorithm>    // transform whole strings to lowercase for comparisons
10 #include <ctime>        // measure elapsed time between sorts
11 using namespace std;
12 
13 // initialize array as global variable
14 string items[666];
15 
16 // load magic items into array from file
17 void loadItems()
18 {
19     // initialize file
20     fstream itemFile;
21     itemFile.open("magicitems.txt", ios_base::in);
22 
23     if (itemFile.is_open())
24     {
25         // counter
26         int i = 0;
27 
28         // while file still has items to read
29         while (itemFile.good())
30         {
31             string item; // initialize item string
32             getline(itemFile, item); // get line
33             transform(item.begin(), item.end(), item.begin(), ::tolower); // transform
                   whole string to lowercase
34             items[i] = item; // store item string
35             i++; // increment counter
36         }
37         itemFile.close();
38     }
39 }
```

This function loads all of the items on the magic list into an array of strings. It opens the file, gets each item, saves it to the global array, then closes the file. I added the statement on line 33 to convert the entire string to lowercase in order to be able to accurately sort the strings.

Swapping Elements

```
41 // swap
42 void swap(string* one, string* two)
43 {
44     // store in temporary variable, then swap pointers
45     string temp = *one;
46     *one = *two;
47     *two = temp;
48 }
```

This helper function swaps the two elements passed to it by reassigning the pointers. It is utilized in the shuffle function as well as the selection sort, insertion sort, and quicksort.

Shuffling List Items

```
50 // shuffle elements of the list
51 void shuffleList()
52 {
53     // initialize random seed
54     srand(time(NULL));
55 
56     // for each item in list
57     for (int i = size(items)-1; i > 0; i--)
58     {
59         // choose random item
60         int randIndex = rand() % i;
61 
62         // swap item at current pointer with random item
63         swap(items[i], items[randIndex]);
64     }
65 }
```

The shuffle function works by iterating through the array and choosing a random element to swap it with. The random number seed is initialized in line 54 to ensure that the same random number set is not generated each time.

Selection Sort

```
67 // selection sort
68 int selectionSort(string arr[], int n)
69 {
70     // initialize comparison counter and minimum value index
71     int comparisons = 0;
72     int minIndex = 0;
73 
74     // outer loop - iterate through each element
75     for (int i = 0; i < n; i++)
76     {
77         // the minimum index of the unsorted portion of the array
78         minIndex = i;
79 
```

```
        // inner loop - loop through the unsorted portion of the array (after i)
        for (int j = i+1; j <n; j++)
        {
            // if current value (at j) is less than the store minimum value
            if (arr[j] < arr[minIndex])
            {
                // save current (smaller) value's index as minimum index
                minIndex = j;
            }
            // increment comparision counter
            comparisons++;
        }

        // swap the smallest found element with the first element of the unsorted portion
        // if first element (at i) is not already the smallest
        if (minIndex != i)
        {
            swap(&arr[minIndex], &arr[i]);
        }
    }
    return comparisons;
}
```

The selection sort function utilizes two nested for loops. The outer loop iterates through each element in the list, setting the index of the minimum value to the first element in the unsorted portion of the array. The inner loop iterates through each remaining element in the unsorted portion of the list, which starts after the index of the outer loop's iterator. It compares each element to the minimum, storing the smallest element it finds. The number of comparisons is incremented here as well. If a smaller element is found, the current element is swapped with the smallest element found, its index now stored as the minimum. In summary, the next smallest element is selected and swapped into the next index of the sorted portion of the list.

Insertion Sort

```
// insertion sort
int insertionSort(string arr[], int n)
{
    // initialize comparisions
    int comparisions = 0;

    // outer loop - iterate through each element
    for (int i = 1; i < n; i++)
    {
        // initialize second iterator at the beginning of the unsorted portion of the list (i)
        int j = i;

        // inner loop - iterate through sorted portion of the list, swapping elements until
        // the value is in the correct space and the previous element is smaller than the current
        while (j >= 0 && arr[j-1] > arr[j])
        {
```

```
            swap(arr[j], arr[j - 1]); // swap
            j--; // decrement
            comparisions++; // increment comparision counter
        }
    }
    return comparisions;
}
```

Insertion sort also utilizes two nested loops, the outside a for loop and the inside a while loop. The outer loop iterates through each element of the array, initializing the counter for the while loop at the first index of the unsorted portion of the list, the outer loop's iterator. This iterator is decremented to check the unsorted element against the previous sorted elements, moving the new value backwards until the previous element is smaller than the current element. The comparison count is incremented within this inner while loop. Each element of the list is effectively inserted into its correct position in the sorted portion of the list.

Merge Sort

```
// merge arrays back together for mergeSort
int merge(string arr[], int left, int mid, int right)
{
    // initialize comparision counter
    int comparisons = 0;

    // find size of two subarrays
    int arrayOne = mid - left + 1;
    int arrayTwo = right - mid;

    // create temp arrays
    auto* leftArray = new string[arrayOne];
    auto* rightArray = new string[arrayTwo];

    // copy data to temp arrays leftArray[] and rightArray[]
    for (int i = 0; i < arrayOne; i++)
        leftArray[i] = arr[left + i];
    for (int j = 0; j < arrayTwo; j++)
        rightArray[j] = arr[mid + 1 + j];

    int indexOfArrayOne = 0; // initial index of first sub-array
    int indexOfArrayTwo = 0; // initial index of second sub-array
    int indexOfMergedArray = left; // initial index of merged array

    // merge temp arrays back into array
    while (indexOfArrayOne < arrayOne && indexOfArrayTwo < arrayTwo) {
        // if value in left array is smaller, left value goes into merged array next
        if (leftArray[indexOfArrayOne] <= rightArray[indexOfArrayTwo]) {
            arr[indexOfMergedArray] = leftArray[indexOfArrayOne];
            indexOfArrayOne++;
        }
        // if right is smaller, right vale goes into merged array next
        else {
            arr[indexOfMergedArray] = rightArray[indexOfArrayTwo];
            indexOfArrayTwo++;
```

```
162         }
163         // increment comparison counter and index of merged array
164         comparisons++;
165         indexOfMergedArray++;
166     }
167 
168     // copy remaining elements of left[], if any
169     while (indexOfArrayOne < arrayOne) {
170         arr[indexOfMergedArray] = leftArray[indexOfArrayOne];
171         indexOfArrayOne++;
172         indexOfMergedArray++;
173     }
174 
175     // copy remaining elements of right[], if any
176     while (indexOfArrayTwo < arrayTwo) {
177         arr[indexOfMergedArray] = rightArray[indexOfArrayTwo];
178         indexOfArrayTwo++;
179         indexOfMergedArray++;
180     }
181 
182     // reallocate memory used for subarrays
183     delete[] leftArray;
184     delete[] rightArray;
185 
186     return comparisons;
187 }
188 
189 // merge sort
190 int mergeSort(string arr[], int start, int end)
191 {
192     // initialize comparison count
193     int comparisons = 0;
194 
195     // base case
196     if (start >= end)
197         return comparisons;
198 
199     // find midpoint of array
200     int mid = start + (end - start) / 2;
201 
202     // recursively sort both sides
203     mergeSort(arr, start, mid);
204     mergeSort(arr, mid + 1, end);
205 
206     // merge all subarrays back together after breaking out of recusion
207     // and add comparisions to running total
208     comparisons += merge(arr, start, mid, end);
209 
210     return comparisons;
211 }
```

Merge sort is written over two functions, one that recursively splits the list into halves, and the other that sorts the subarrays as it merges them back together in the correct order. Beginning with the mergeSort function on line 189, the base case of the recursive function breaks out of the recursion when the start value passed into the function is greater than the end value - when the subarrays can no longer be split any smaller and contain only one value. Next, the midpoint is calculated and passed into two recursive calls of mergeSort as the end value of the first call and the start value of the second call, splitting the list into two sub arrays. This continues until the base case is met, at which point the merge function is called, merging all the subarrays back together as the recursion is "unraveled." The merge function determines the size of the two arrays, creates two temporary right and left subarrays, and copies the correct data into them. Then, the values in each temp array are copied into the final merged array, both right and left sub arrays checked for the next smallest value. This is where the comparison counter is incremented. The merge helper function is called for each subarray at the end of the mergeSort function, putting the entire array back together in order.

Quick Sort

```
// quick sort - pivot point always at start
int partition(string arr[], int start, int end, int* comparisons)
{
    // store value at pivot point for comparisons
    string pivot = arr[start];

    // find smallest element
    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }

    // put pivot value in the correct position by swapping smallest and pivot
    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);

    // sort left and right parts of the pivot element
    int i = start, j = end;

    while (i < pivotIndex && j > pivotIndex)
    {
        (*comparisons)++;

        // if previous values are less (on the correct side of the pivot), keep them in
            place and keep iterating
        while (arr[i] <= pivot)
        {
            i++;
            (*comparisons)++;
        }

        // if next values are greater (on the correct side of the pivot), keep them in
            place and keep iterating
        while (arr[j] > pivot)
        {
            j--;
            (*comparisons)++;
```

```
        }

        // if incorrect, swap
        if (i < pivotIndex && j > pivotIndex)
        {
            swap(arr[i++], arr[j--]);
            (*comparisons)++;
        }
    }

    return pivotIndex;
}

// quick sort
int quickSort(string arr[], int start, int end)
{
    int comparisons = 0;

    // base case
    if (start >= end)
        return comparisons;

    // partition the array - pass in address of comparisons
    int p = partition(arr, start, end, &comparisons);

    // Sorting the left part
    quickSort(arr, start, p - 1);

    // Sorting the right part
    quickSort(arr, p + 1, end);

    return comparisons;
}
```

Quicksort also utilizes a helper function called partition. Similarly to merge sort, the quickSort function has a base case that checks if the start position passed into the function is greater than the end position, meaning that the subarrays can no longer be partitioned any smaller. The partition helper function is called next, which splits the array around a pivot point, which I chose as the first value in the array. The smallest element is found and swapped with the pivot value. The remaining elements are sorted onto either side of the pivot. Each value is compared to the pivot,ensuring that smaller elements are on the left and larger elements are on the right. If an element is on the incorrect side of the pivot point, it is swapped. Returning to the quickSort function, it is recursively called and partitioned until each subarray is sorted around a pivot point, eventually ”unraveling” and resulting in a fully sorted array.

Displaying Formatted Output

```
// display formatted summary of sorts
void display(string sortType, int comparisons, double time)
{
    // display heading - left justified, width of 34, filler char of '-', title, end line
    cout << left << setw(34) << setfill('-') << sortType + " " << endl;
```

```
288     // display label left aligned in 25 spaces, followed by value right aligned (default)
            with 8 spaces
289     printf("%-25s %8d\n", "Number of comparisions:", comparisons);
290     // display label left aligned in 25 spaces, followed by value right aligned (default)
            with 5 spaces (8-3=5 characters for ' ms')
291     printf("%-25s %5.f ms\n", "Elapsed time:", time);
292     // extra spacing
293     cout << endl;
294 }
```

The display function outputs the formatted results of each sort. Line 287 outputs the name of each sort left aligned and padded with dashes to create a heading. Line 289 outputs the label "Number of comparisons" left justified along with the actual value of the number of comparisons, right justified. Line 291 works similarly, outputting the left aligned label "Elapsed time" as well as the time it took for the sort to complete. Line 293 is included to ad extra spacing between each call of the display function.

Main Function

```
int main()
{
    loadItems();

    // SELECTION SORT ------------------------------------------------------------------
    shuffleList(); // shuffle
    clock_t begin = clock(); // begin timer
    int selectionComparisons = selectionSort(items, 666); // call sort
    clock_t end = clock(); // end timer
    display("Selection Sort", selectionComparisons, end-begin); // display results

    // INSERTION SORT ------------------------------------------------------------------
    shuffleList(); // shuffle again
    begin = clock(); // restart timer
    int insertionComparisons = insertionSort(items, 666); // call sort
    end = clock(); // end timer
    display("Insertion Sort", insertionComparisons, end-begin); // display results

    // MERGE SORT ----------------------------------------------------------------------
    shuffleList(); // shuffle again
    begin = clock(); // restart timer
    int mergeComparisions = mergeSort(items, 0, 666); // call sort;
    end = clock(); // end timer
    display("Merge Sort", mergeComparisions, end-begin); // display results

    // QUICK SORT ----------------------------------------------------------------------
    shuffleList(); // shuffle again
    begin = clock(); // restart timer
    int quickComparisions = quickSort(items, 0, 666); // call sort
    end = clock(); // end timer
    display("Quick Sort", quickComparisions, end-begin); // display results
}
```

The main function calls the loadItems function first to load the array with all of the magic items. The shuffle function is called before each sort, and a timer is reset as well. After each sort function is called, the timer is ended and the display function is called, passing the type of sort, number of comparisons, and the elapsed time.

Sample Output

```
Selection Sort -------------------
Number of comparisions:     221445
Elapsed time:                60 ms

Insertion Sort -------------------
Number of comparisions:     106463
Elapsed time:                77 ms

Merge Sort -----------------------
Number of comparisions:        665
Elapsed time:                20 ms

Quick Sort -----------------------
Number of comparisions:        661
Elapsed time:                 5 ms
```

This is a sample of the output after the program is run to display the formatting and sample data values for each sort type.