Assignment One

Alex Tocci

Mason.Tocci1@Marist.edu

October 5, 2023

1 Stack and Queue

1.1 Creating Node Class

Listing 1: Node Class

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  // Include for std::random_device
5  #include <random>
6
7  using namespace std;
8
9  // Define a Node class for the stack and queue
10 class Node {
11 public:
12     char data;
13     Node* next;
14
15     Node(char value) : data(value), next(nullptr) {}
16 };
```

Lines 1-5: First we must include the necessary C++ libraries for our program to run.
Line 7: This allows us to use standard C++ functions without the prefix 'std::'.
Lines 10-13: Define a C++ class called 'Node' which contains two variables, 'char data' and 'Node* next'. 'Char data' stores the data associated with the node. 'Node* next' points to the next node in the queue or stack or is set to 'nullptr', which indicates that there is no next node.
Line 15: This is a Node class constructor, which initializes the Node object

1.2 Creating Stack Class

Listing 2: Stack Class

```
17 // Create a stack using the Node class
18 class Stack {
19 private:
20     // Point to the top of the stack
21     Node* top;
22
23 public:
24     Stack() : top(nullptr) {}
25
26     // Push a value onto the stack
27     void push(char value) {
28         Node* newNode = new Node(value);
29         newNode -> next = top;
30         top = newNode;
31     }
32
33     // Pop the top value from the stack
34     char pop() {
35         if (isEmpty()) {
36             cerr << "Stack is empty." << endl;
37             return -1;
38         }
39         char value = top->data;
40         Node* temp = top;
41         top = top -> next;
42         delete temp;
43         return value;
44     }
45
46     // Check if the stack is empty
47     bool isEmpty() {
48         return top == nullptr;
49     }
50 };
```

Lines 18-21: Define a class called 'Stack' that contains a 'Node* top' variable which keeps track of the top element in the stack.
Line 24: This is the 'Stack' class constructor which initializes the top pointer to 'nullptr', indicating an empty stack.
Line 27: This function is used to add an element onto the stack.
Line 28: Create a new Node with the given value.
Line 29 : Set the 'next' pointer of the new 'Node' to the current top of the stack.
Line 30: Update the 'top' pointer so it points to the new 'Node', making it the top of the stack.
Line 34: This function is used to remove the top element from the stack.
Line 39: If the stack is not empty, it gets the value stored in the current top Node.
Line 40: Create a temporary pointer to the current top Node.
Line 41: Update the 'top' pointer so it points to the next Node in the stack.
Lines 42-43: Delete the previous top Node and return the popped value.
Lines 47- 49: This function is used to check if the stack is empty. If the 'top' pointer is 'nullptr' it returns true. Meaning the stack is empty.

1.3 Creating Queue Class

Listing 3: Queue Class

```
51 // Create a queue using the Node class
52 class Queue {
53 private:
54     Node* front;
55     Node* back;
56
57 public:
58     Queue() : front(nullptr), back(nullptr) {}
59
60     // Enqueue a value into the queue
61     void enqueue(char value) {
62         Node* newNode = new Node(value);
63         if (isEmpty()) {
64             front = back = newNode;
65         }
66         else {
67             back -> next = newNode;
68             back = newNode;
69         }
70     }
71
72     // Dequeue a value from the queue
73     char dequeue() {
74         if (isEmpty()) {
75             cerr << "Queue is empty." << endl;
76             return -1;
77         }
78         char value = front->data;
79         Node* temp = front;
80         front = front -> next;
81         delete temp;
82         return value;
83     }
84
85     // Check if the queue is empty
86     bool isEmpty() {
87         return front == nullptr;
88     }
89 };
```

Lines 52-55: Define a class called 'Queue' that contains two varables, 'Node* front' and 'Node* back'. The front pointer keeps track of the front element in the queue. The back pointer keeps track of the back element in the queue.
Line 58: Initialize both 'front' and 'back' pointers to 'nullptr', an empty queue.
Line 61: This function is used to add an element to the queue.
Line 62: Create a new Node with the given value.
Lines 63-65: If the queue is empty, make both the 'front' and 'back' pointers point to the new 'Node'. This makes the new 'Node' both the front and back of the queue.
Lines 66-69: If the queue is not empty, the new Node is added to the back of the queue, and the 'back'

pointer is updated to the new 'Node'.
Line 73: This function is used to remove the front element from the queue.
Line 78: If the queue is not empty, get the value stored in the current front Node.
Line 79: Create a temporary pointer to the current front Node.
Line 80: Update the 'front' pointer so it points to the next Node in the queue, removing the current front Node.
Lines 81-83: Delete the previous front Node and return the dequeued value.
Lines 86-89: This function is used to check if the queue is empty. If the 'front' pointer is 'nullptr', it returns true. Meaning the queue is empty.

1.4 Check for palindrome

Listing 4: Check for Palindrome

```
90  // Check if a string is a palindrome (ignoring spaces and capitalization)
91  bool isPalindrome(const string& str) {
92      Stack stack;
93      Queue queue;
94
95      // Push characters to the stack and queue (ignoring spaces and converting to lowercase)
96      for (char i : str) {
97          if (!isspace(i)) {
98              stack.push(tolower(i));
99              queue.enqueue(tolower(i));
100         }
101     }
102
103     // Compare characters from the stack and queue
104     while (!stack.isEmpty() && !queue.isEmpty()) {
105         if (stack.pop() != queue.dequeue()) {
106             // Not a palindrome
107             return false;
108         }
109     }
110
111     return true;
112 }
```

Lines 91-93: Create a 'Stack' object and a 'Queue' object.
Line 97: Check if a character is not a space.
Lines 98-100: If 'i' is not a space, push the lowercase characters onto the 'stack' and enqueue the lowercase characters into the 'queue'.
Line 104: This 'while' loop continues as long as both the 'stack' and the 'queue' are not empty.
Line 105-107: Compare the characters at the top of the 'stack' and the front of the 'queue'. If the characters are not equal, it means that the string is not a palindrome, returning false.

1.5 Shuffle

Listing 5: Shuffle

```
113 // Shuffle the array using the Knuth shuffle algorithm with std::random_device seeding
114 void shuffle(string* arr, int size) {
```

```
115     random_device rd;
116     srand(rd());
117 
118     for (int i = size - 1; i > 0; --i) {
119         // Generate a random index between 0 and i
120         int j = rand() % (i + 1);
121         // Swap elements at i and j
122         swap(arr[i], arr[j]);
123     }
124 }
```

Lines 114-116: Declare a random device named 'rd'. Random device is used to generate random numbers. Then, call 'srand(rd())' with 'rd' as the seed value to get the random number
Lines 118-124: This 'for' loop shuffles the elements of the array using the Knuth shuffle. We get a random array index and swap it with the current element in the loop.

2 Sorting

2.1 Selection Sort

Listing 6: Selection Sort

```
125 // Selection sort
126 void selectionSort(string* arr, int size, int& comparisons) {
127     for (int i = 0; i < size - 1; ++i) {
128         int index = i;
129         for (int j = i + 1; j < size; ++j) {
130             // Convert both strings to lowercase for comparison
131             for (char& ch : arr[j]) {
132                 ch = tolower(ch);
133             }
134             for (char& ch : arr[index]) {
135                 ch = tolower(ch);
136             }
137             comparisons++;
138             // Compare the lowercase strings
139             if (arr[j] < arr[index]) {
140                 index = j;
141             }
142         }
143         // If a smaller element was found, swap it with the current element
144         if (index != i) {
145             swap(arr[i], arr[index]);
146         }
147     }
148 }
```

Lines 126-128: This loop takes the minimum element in the unsorted part of the array and puts it in its correct position.
Lines 129-136: This for loop compares each element in the unsorted part of the array with the element at index 'i'. It converts the current element 'arr[index]' and the next element 'arr[j]' to lowercase.
Line 137: Increment 'comparisons' to keep track of the number of comparisons made.
Lines 139-142: Compare the lowercase strings. If the next element is less than the current element, update 'index' to 'j'.
Lines 144-146: After the inner 'for' loop completes, the 'index' variable contains the index of the smallest element in the unsorted part of the array. If 'index' is not equal to 'i', it means that a smaller element was found. If this is true, swap elements 'i' and 'index'.

2.2 Insertion Sort

Listing 7: Insertion Sort

```
149 // Insertion sort
150 void insertionSort(string* arr, int size, int& comparisons) {
151     for (int i = 1; i < size; ++i) {
152         int j = i;
153         while (j > 0) {
154             // Convert both strings to lowercase for comparison
155             for (char& ch : arr[j]) {
```

```
156                     ch = tolower(ch);
157                 }
158                 for (char& ch : arr[j - 1]) {
159                     ch = tolower(ch);
160                 }
161                 comparisons++;
162                 // Compare the lowercase strings
163                 if (arr[j] < arr[j - 1]) {
164                     swap(arr[j], arr[j - 1]);
165                     --j;
166                 }
167                 else {
168                     break;
169                 }
170             }
171     }
172 }
```

Lines 150-152: This loop inserts each element into its correct position based on previously sorted elements.
Lines 154-160: Convert the elements at index 'j' and the index 'j - 1' to lowercase.
Line 161: Increment 'comparisons' to keep track of the number of comparisons made.
Line 163-166: Compare the lowercase strings. If 'arr[j]' is less than 'arr[j - 1]', swap the two elements and decrement 'j' to continue sorting.
Lines 167-169: If the comparison is false, break out of the inner loop.

2.3 Merge Sort

2.3.1 Merge Function

Listing 8: Merge Function

```
173 // Function to merge two sorted subarrays
174 void merge(string* arr, int left, int mid, int right, int& comparisons) {
175     // Calculate the sizes of the subarrays
176     int sizeLeft = mid - left + 1;
177     int sizeRight = right - mid;
178 
179     // Create temporary arrays for the subarrays
180     string* leftArray = new string[sizeLeft];
181     string* rightArray = new string[sizeRight];
182 
183     // Copy data to temporary arrays
184     for (int i = 0; i < sizeLeft; ++i) {
185         leftArray[i] = arr[left + i];
186     }
187     for (int i = 0; i < sizeRight; ++i) {
188         rightArray[i] = arr[mid + 1 + i];
189     }
190 
191     // Merge the two subarrays back into array
192     // Index for the left subarray
193     int i = 0;
```

```
194     // Index for the right subarray
195     int j = 0;
196     // Index for the merged array
197     int k = left;
198 
199     while (i < sizeLeft && j < sizeRight) {
200         // Convert characters to lowercase for case-insensitive comparison
201         for (char& ch : leftArray[i]) {
202             ch = tolower(ch);
203         }
204         for (char& ch : rightArray[j]) {
205             ch = tolower(ch);
206         }
207 
208         comparisons++;
209         // Compare and merge based on lowercase strings
210         if (leftArray[i] <= rightArray[j]) {
211             arr[k] = leftArray[i];
212             ++i;
213         }
214         else {
215             arr[k] = rightArray[j];
216             ++j;
217         }
218         ++k;
219     }
220 
221     // Copy the remaining elements of leftArray[], if any
222     while (i < sizeLeft) {
223         arr[k] = leftArray[i];
224         ++i;
225         ++k;
226     }
227 
228     // Copy the remaining elements of rightArray[], if any
229     while (j < sizeRight) {
230         arr[k] = rightArray[j];
231         ++j;
232         ++k;
233     }
234 
235     // Delete temporary arrays
236     delete[] leftArray;
237     delete[] rightArray;
238 }
```

Line 174: This function, is a part of the merge sort algorithm and involves merging two sorted subarrays into a single sorted array.
Line 176-177: Calculate the sizes of the two subarrays.
Lines 180-181: Create temporary arrays for the left and right subarray.
Lines 184-189: Two 'for' loops copy the data from the original array into the subarrays.
Lines 193-197: Merge the subarrays back into the original array. Initialize, three index variables: 'i' for the

left subarray, 'j' for the right subarray, 'k' for the merged array.
Lines 201-206: Elements in the left and right subarrays are converted to lowercase.
Lines 201-206: Increment 'comparisons' to keep track of the number of comparisons made.
Lines 210-219: Elements are compared based on their lowercase strings. The smaller element between the left subarray and right subarray is copied into the merged array. The index of the smaller element is incremented. Index 'k' is also incremented.
Lines 222-233: The two 'while' loops copy the remaining elements in the left and right subarrays to the merged array.
Lines 236-238: Delete the left and right subarrays.

2.3.2 mergeSort Function

Listing 9: mergeSort Function

```
239 // Merge sort
240 void mergeSort(string* arr, int left, int right, int& comparisons) {
241     if (left < right) {
242         // Find the middle point
243         int mid = left + (right - left) / 2;
244 
245         // Recursively sort the first and second halves
246         mergeSort(arr, left, mid, comparisons);
247         mergeSort(arr, mid + 1, right, comparisons);
248 
249         // Merge the sorted halves
250         merge(arr, left, mid, right, comparisons);
251     }
252 }
```

Lines 240-243: If the left index of the current subarray is less than the right index of the current subarray, find the middle point. Calculate the midpoint by taking the average of 'left' and 'right'. The midpoint is the place where the array will be split.
Lines 246-247: The function recursively calls itself to sort the two halves of the current subarray. The first recursive call sorts the left half of the subarray and the second recursive call sorts the right half of the subarray. These recursive calls split the subarrays into smaller and smaller arrays until they contain only one element.
Line 250: Once the left and right halves of the subarray are sorted, the 'merge' function is called to merge them back together into a single sorted subarray.

2.4 Quick Sort

2.4.1 Partition Function

Listing 10: Partition Function

```
253 // Partition function for quick sort
254 int partition(string* arr, int low, int high, int& comparisons) {
255     // Choose the rightmost element as the pivot
256     string pivot = arr[high];
257     // Index of the smaller element
258     int i = (low - 1);
259 
260     for (int j = low; j <= high - 1; ++j) {
```

```
261             // Convert both strings to lowercase for comparison
262             for (char& ch : arr[j]) {
263                 ch = tolower(ch);
264             }
265             for (char& ch : pivot) {
266                 ch = tolower(ch);
267             }
268 
269             comparisons++;
270             // Compare the lowercase strings
271             if (arr[j] < pivot) {
272                 ++i;
273                 swap(arr[i], arr[j]);
274             }
275         }
276 
277         // Swap the pivot element
278         swap(arr[i + 1], arr[high]);
279         return (i + 1);
280     }
```

Lines 256-258: Select the far right element in the given subarray as the pivot. 'i' is initialized to 'low - 1' and keeps track of the index of the smaller element during partitioning.
Line 260: This loop compares each element with the pivot and sorts them.
Lines 262-267: Inside the loop, both the current element and the pivot are converted to lowercase.
Line 269: Increment 'comparisons' to keep track of the number of comparisons made.
Lines 271-274: Compare the lowercase strings of the current element and the pivot. If the current element is less than the pivot, increment 'i' to mark the position of the smaller element and swap the element at index 'i' with the element at index 'j'.
Lines 278-280: Swap the pivot with the element at index 'i + 1' which places the pivot in its correct sorted position. Return the index '(i + 1)', the correct position of the pivot element in the sorted array.

2.4.2 quickSort Function

Listing 11: quickSort Function

```
281 // Quick sort
282 void quickSort(string* arr, int low, int high, int& comparisons) {
283     if (low < high) {
284         // Find the pivot element such that
285         // element smaller than pivot are on the left and
286         // elements greater than pivot are on the right
287         int pivotIndex = partition(arr, low, high, comparisons);
288 
289         // Recursively sort the subarrays
290         quickSort(arr, low, pivotIndex - 1, comparisons);
291         quickSort(arr, pivotIndex + 1, high, comparisons);
292     }
293 }
```

Line 283-287: If low is less than the high, find a pivot element and partition the subarray.
Lines 290-293: After partitioning the subarray, the 'quickSort' function is called recursively twice to sort

the left and right subarrays. The first recursive call sorts the left subarray and the second recursive call sorts the right subarray. The recursive calls continue until all subarrays contain zero or one element.

3 Main Function

3.1 Test Stack and Queue

Listing 12: Test Stack and Queue

```
294 int main() {
295     // Test the stack
296     Stack testStack;
297     testStack.push('A');
298     testStack.push('B');
299     testStack.push('C');
300 
301     cout << "Stack contents: ";
302     while (!testStack.isEmpty()) {
303         cout << testStack.pop() << " ";
304     }
305     cout << endl;
306 
307     // Test the queue
308     Queue testQueue;
309     testQueue.enqueue('A');
310     testQueue.enqueue('B');
311     testQueue.enqueue('C');
312 
313     cout << "Queue contents: ";
314     while (!testQueue.isEmpty()) {
315         cout << testQueue.dequeue() << " ";
316     }
317     cout << endl;
```

Lines 296-299: Create an instance of a stack and push three characters onto it. The elements are added to the top of the stack in the order they are pushed.
Lines 301-305: The 'while' loop will continue as long as the stack is not empty. Within the loop, pop the top element from the stack and print it to the standard output.
Lines 309-311: Create an instance of a queue and enqueue three characters onto it. The elements are added to the back of the queue in the order they are enqueued.
Lines 313-317: The 'while' loop will continue as long as the queue is not empty. Within the loop, dequeue the front element from the queue and print it to the standard output.

3.2 Read 'magicitems.txt' and store items into an array

Listing 13: Store Items in Array

```
318 // Read all lines of magicitems.txt and put them in an array
319 // Open the magicitems.txt file
320 ifstream file("magicitems.txt");
321 
322 // Handle failure
323 if (!file)
324 {
325     cerr << "Failed to open magicitems.txt" << endl;
326     return 1;
```

```
327 }
328
329 // Count the number of lines in the file
330 int magicItemsSize = 0;
331 string line;
332 while (getline(file, line))
333 {
334     magicItemsSize++;
335 }
336
337 // Close and reopen the file to read from the beginning
338 file.close();
339 file.open("magicitems.txt");
340
341 // Create a dynamically allocated array
342 string* magicItemsArray = new string[magicItemsSize];
343
344 // Read the file line by line and store each line in the array
345 int index = 0;
346 while (getline(file, line))
347 {
348     magicItemsArray[index] = line;
349     index++;
350 }
351
352 // Close the file
353 file.close();
```

Line 320: Open the file 'magicitems.txt' for reading.
Lines 323-327: Check if the file was successfully opened. If the file cannot be opened, print an error message and exit the program.
Lines 330-335: Read the file line by line and count the number of lines in the file. Increment the 'magicItemsSize' variable for each line read.
Lines 338-339: Close and reopen the file to read from the beginning.
Line 342: Create a dynamically allocated array of strings named 'magicItemsArray'. Set the size equal to 'magicItemsSize'.
Lines 345-350: Read the file line by line and store each line in the array.
Line 353: Close the file.

3.3 Check for Palindromes

Listing 14: Check for Palindromes

```
354 cout << "\nPalindromes: " << endl;
355 // Check each line in the array for palindromes and print it if it is
356 for (int i = 0; i < magicItemsSize; ++i) {
357     if (isPalindrome(magicItemsArray[i])) {
358         cout << magicItemsArray[i] << endl;
359     }
360 }
```

Lines 354-360: The 'for' loop iterates over each line in the 'magicItemsArray'. Inside the loop, check if the

current line in the array is a palindrome. If 'isPalindrome' returns true, it means that the current line is a palindrome and will be printed.

3.4 Selection Sort

Listing 15: Selection Sort

```
361 // Shuffle before sorting
362 shuffle(magicItemsArray, magicItemsSize);
363
364 int comparisonsSelectionSort = 0;
365 // Sort magicItemsArray using selection sort
366 selectionSort(magicItemsArray, magicItemsSize, comparisonsSelectionSort);
367
368 cout << "\nSorted Magic Items (Selection Sort):" << endl;
369 for (int i = 0; i < magicItemsSize; ++i) {
370     cout << magicItemsArray[i] << endl;
371 }
372 cout << "Comparisons in Selection Sort: " << comparisonsSelectionSort << endl;
```

Line 362: Call the 'shuffle' function to shuffle the elements in 'magicItemsArray'.
Lines 365-366: Initialize an integer variable named 'comparisonsSelectionSort' to zero. Then, Sort the array using the selection sort algorithm. The 'comparisonsSelectionSort' variable is passed into the function, allowing the number of comparisons to be counted.
Lines 368-371: Use a 'for' loop to print each element in the sorted 'magicItemsArray'.
Line 372: Print the number of comparisons made during selection sort.

3.5 Insertion Sort

Listing 16: Insertion Sort

```
373 // Shuffle before sorting again
374 shuffle(magicItemsArray, magicItemsSize);
375
376 int comparisonsInsertionSort = 0;
377 // Sort magicItemsArray using selection sort
378 insertionSort(magicItemsArray, magicItemsSize, comparisonsInsertionSort);
379
380 cout << "\nSorted Magic Items (Insertion Sort):" << endl;
381 for (int i = 0; i < magicItemsSize; ++i) {
382     cout << magicItemsArray[i] << endl;
383 }
384 cout << "Comparisons in Insertion Sort: " << comparisonsInsertionSort << endl;
```

Line 374: Shuffle the elements in 'magicItemsArray'.
Lines 376-378: Initialize an integer variable named 'comparisonsInsertionSort' to zero. Sort the array using the insertion sort algorithm. The 'comparisonsInsertionSort' variable is passed into the function, allowing the number of comparisons to be counted.
Lines 380-383: Use a 'for' loop to print each element in the sorted 'magicItemsArray'.
Line 384: Print the number of comparisons made during insertion sort.

3.6 Merge Sort

Listing 17: Merge Sort

```
385 // Shuffle before sorting again
386 shuffle(magicItemsArray, magicItemsSize);
387
388 int comparisonsMergeSort = 0;
389 // Sort the strings using merge sort in a case-insensitive manner
390 mergeSort(magicItemsArray, 0, magicItemsSize - 1, comparisonsMergeSort);
391
392 cout << "\nSorted Magic Items (Merge Sort):" << endl;
393 for (int i = 0; i < magicItemsSize; ++i) {
394     cout << magicItemsArray[i] << endl;
395 }
396 cout << "Comparisons in Merge Sort: " << comparisonsMergeSort << endl;
```

Line 386: Shuffle the elements in 'magicItemsArray'.
Lines 388-390: Initialize an integer variable named 'comparisonsMergeSort' to zero. Sort the array using the merge sort algorithm. The 'comparisonsMergeSort' variable is passed into the function, allowing the number of comparisons to be counted.
Lines 392-395: Use a 'for' loop to print each element in the sorted 'magicItemsArray'.
Line 396: Print the number of comparisons made during merge sort.

3.7 Quick Sort

Listing 18: Quick Sort

```
397     // Shuffle before sorting again
398     shuffle(magicItemsArray, magicItemsSize);
399
400     // Sort magicItemsArray using quick sort
401     int comparisonsQuickSort = 0; // Initialize comparisons counter
402     quickSort(magicItemsArray, 0, magicItemsSize - 1, comparisonsQuickSort);
403
404     cout << "\nSorted Magic Items (Quick Sort):" << endl;
405     for (int i = 0; i < magicItemsSize; ++i) {
406         cout << magicItemsArray[i] << endl;
407     }
408
409     cout << "Comparisons in Quick Sort: " << comparisonsQuickSort << endl;
410
411
412     // Delete dynamically allocated memory
413     delete[] magicItemsArray;
414
415     return 0;
416 }
```

Line 398: Shuffle the elements in 'magicItemsArray'.
Lines 400-402: Initialize an integer variable named 'comparisonsQuickSort' to zero. Sort the array using the quick sort algorithm. The 'comparisonsQuickSort' variable is passed into the function, allowing the number of comparisons to be counted.

Lines 404-407: Use a 'for' loop to print each element in the sorted 'magicItemsArray'.
Line 409: Print the number of comparisons made during quick sort.
Line 413: Delete dynamically allocated memory.

4 Results

Table 4.1: Number of Comparisons

	1	2	3	4	5	Avg
Selection Sort	221445	221445	221445	221445	221445	221445
Insertion Sort	111238	112774	107069	111565	109727	110474.6
Merge Sort	5435	5417	5440	5405	5423	5424
Quick Sort	7052	7407	6809	7234	6815	7063.4

4.1 Selection Sort

Selection sort has a time complexity of $O(n^2)$ in the worst, average, and best cases. In this case, we always get 221,445 comparisons because the size of the array is fixed at 666, and selection sort performs the same number of comparisons regardless of the initial order of the array. In the code, the outer loop runs for '(size - 1)' iterations, and the inner loop runs for '(size)' iterations. This results in a total of $comparisons = (size - 1) * (size)/2$ comparisons, plugging in 666 for size we get 221,445 comparisons.

4.2 Insertion Sort

Insertion sort has a time complexity of $O(n^2)$ in its worst cases. In the code, the 'for' loop runs for '(size - 1)' iterations. The 'while' loop runs as long as the current element is less than its previous element; in the worst case, this would be the size of the array, 666. This results in a worst case of $comparisons = (size-1)*(size)/2$ comparisons, plugging in 666 for size we get 221,445 comparisons. To get the average we would divide the worst case by 2, making our average 110,722.5.

4.3 Merge Sort

Merge sort has a time complexity of $O(nlogn)$ in its worst cases. In the code, the 'merge' function merges two sorted subarrays into a single sorted array. The time complexity of this function is $O(n)$. The 'mergeSort' function recursively divides the array into smaller subarrays until each subarray contains only one element. Then, it merges these subarrays back together in sorted order. The recursion takes $O(logn)$ time because the array is repeatedly halved. So we get a total time complexity is $O(nlogn)$. Plugging in 666, our worst case is 6246.

4.4 Quick Sort

Quick sort has a time complexity of $O(nlogn)$ in its worst cases. In the code, the 'partitioning' function takes a pivot and rearranges the elements. The average time complexity of this function is $O(n)$. In the 'quicksort' function, each recursive call splits the array into two subarrays. The expected size of these subarrays is supposed to be nearly half. The recursion takes $O(logn)$ time because the array is repeatedly split. So we get an average time complexity of $O(nlogn)$. Plugging in 666, our average case should be 6246.7.