Course: CS 2302

Author: Kimberly Morales

Assignment: Lab 2

Instructor: Olac Fuentes

TA(s): Anindita Nath, Maliheh Zargaran

Date: 2/25/2019

# **Introduction**

The task for this lab was to find a median within a single linked list by first sorting the list and then returning the element in the middle. Since sorting can be done with many algorithms, four main algorithms were implemented to compare their running times. These algorithms were bubble sort, merge sort, quick sort, and a modified version of quicksort. To calculate the runtime for each algorithm required the counting of the number of comparisons done for each sorting algorithm.

# **Proposed Solution Design and Implementation**

Since all the sorting algorithms are not dependent on languages I decided to research into each one and analyze it visually. Sorting algorithms are also not dependent on arrays, so I followed each one with a linked list in mind. Then, I thought of general utility functions that I could reuse in the code so that I would not repeat myself within each sorting method. Next, I implemented the sorting algorithms with the lists and created more specific functions that could not be reused. Finally, I created separate global variable counts to count the number of comparisons, timed each execution, and returned the final median from each sorting method.

**Utility**

Gen\_rand\_list:

This method builds a list of size n integers with distinct integers. To do so, I first created an empty list and then generated a Python native list of a range of 1500. The native list was used because an initial problem that I was running into was that the random module of Python has a pseudorandom range that often made duplicates in a list which often got worse the bigger n got so I I used random sample instead that acts similar to a bingo machine that shuffles these numbers with a specified size n, making the integers more random. Once this random native list is built, I used a for loop to append each element in the native list to the linked list. Once the loop stops, it returns a new list.

Swap

Swap is created generally for the sorting algorithms since a common pattern I saw for them was that it often required swapping elements. I decided that I just wanted to create one function that swaps only the items of two nodes since I wanted to keep my code modular. Swap works by having two nodes (initial node and final node) and a temporary node to hold at least of the node’s reference. First, this temporary node holds a reference to the initial node then the initial node is swapped with the final node. Finally, the final node swaps with the temp node since the initial node no longer holds the same item. Nothing is returned since it is swapping and messing with the references.

Copy\_list

Sorting changes the original list a lot by using swapping or appending depending on the element so for each sorting algorithm, it is necessary to create a copy of the list so that the results are not skewed for each run. While it may be simple to just assign the original list to the new list, this merely only assigns the reference and would still modify the original list. So, my approach was to have a temp node assigned with the original list’s head and iterate through the original list while appending each element until temp is none. Then. a new list is returned.

Median

Median functions as the working heart since it uses each sorting algorithm, prints the sorted list, the runtime in seconds, number of comparisons, and the median. I did not want to duplicate a lot of the same code in the main area since it would have looked messy and not organized specifically for each sorting method. Each sorting algorithm has a number assigned to it, like a menu, and will only run that method once and use the copy list. Once the copied list is sorted, it is inputted into the element\_at method to find the median which is half of the list. Med is assigned this value and returns the item of the node.

**Linked List**

Majority of my linked list class was used from the class website except for get\_length, and element\_at.

Get\_length

This method is straightforward and merely counts the number of items in a list. Like the copy\_list method, a temp node holds reference to the list and a count integer variable counts each time in a while loop until the temp node is none. It just returns the count. This method is important for the median function and used for the end pointers for the quicksort algorithm.

Element\_at

Returns an element based on the index while iterating in the list. A major difference between an element and node is that the element contains the actual value (item attribute) rather than the reference. First, a temp node is assigned L’s reference and n, the length of the list, is used for a for loop to iterate through the list until it hits the specified index number. Once the loops count is equal to the index number then it returns the temp’s value at that instance. This method is used for the median and quicksort algorithms for easier comparisons.

**Sorting Algorithms**

Bubble Sort

Bubble sort has the parameter of the list and returns a sorted list. With bubble sort, it was the easiest to implement iteratively and similar with arrays, I had two pointers i and j to compare two elements at a time. I was assigned the head of L and J was the node after the head. I used two while loops since I assume that I do not know the size of the list. The outer loop focuses on i and will go onto the next node after j is done iterating and i is none. The inner loop is for j and actively compares each node after i to i. Since linked lists are just references, I used the item attribute to compare the actual values. To make an ascending list, if i is greater than j then the two nodes are swapped with the swap function. Then, j moves onto the next node until it hits the end of the list. After the inner loop exits then i moves onto the next node and j is assigned the node of i since it doesn’t need to sort to the left of the loop since i moves. After all loop exits, then the sorted list is returned.

Merge Sort

Merge sort has the parameter of the head node of the list and returns the sorted list’s node which is different. I did not implement the main merge sort method iteratively and instead created two functions, split\_list and merge\_list. The base case is that if the head is empty which means that the list is empty then return the head node since I assume an empty list is a sorted list. First, mergesort creates two lists by splitting the list at the head node.

Within split\_list, it takes in the list and holds the left and right pointers, l and r. Think of l as being the slow pointer that lags the fast r pointer. These pointers go at different speeds since l is making the sub list of l from the head to the midpoint and r is the sub list that starts from the point after the midpoint until the end of the list. (While I could have called the split\_list method twice for the two subsists in the main method, I wanted to utilize the benefits of Python and be able to just return two lists, making it so I could save more memory in the calls in the main method). Back to the split list method, if the right pointer is not none then it will move to the next pointer so that it is not redundant with the l pointer. Then within the while loop, it will iterate until r is none since r would have reached the end of the list. To keep r the faster pointer, r is moved again and if r is not none then r is moved again, and l is moved to make the distance equal to the two points. Then, once the loop exits, mid is assigned l since it will be the start of the right sub list and the original list since it has been modified due to not being copied. Split\_list returns the left sub list and right sub list.

With these two new lists, L1 and L2, then merge\_sort is called recursively to the two lists. This means that these two lists will keep being sliced in half with split\_list until they are lists of size 1. Once these two lists are small then they are merged with the merge\_list function.

Merge\_list oversees comparing each list and sorting it to then make a larger list. This is like breaking LEGOS until they are singular blocks and rebuilding it again. For basic error checking, if L1 is empty it will return the second list since it is already combined and vice versa with L2. Then, the heads of L1 and L2 are compared and if L2 is greater than temp holds the value of L1 and recursively called merge\_list and will keep check if the elements in L1 is less than L2. If at some point L2 is less than L1 then this time temp holds the value of L2 and instead recursive called merge\_list from L1 to the next L2 node. Once temp hits none then it will return the modified list. Basically, merge\_list modifies the list by manipulating the references within the linked list.

The new head node is assigned the new sorted list from merge\_list and is returned.

Quick Sort

When implementing quicksort, I had to realize some important aspects of this algorithm. There are many ways to implement it with various runtimes and it depends on the pivot and how the partitioning is done. To make the typical runtime and easier implementation of quicksort, I made my pivot the beginning of the list and a basic partitioning method that will throw elements less than the pivot to the right and elements more than the pivot to the right. I had a main quicksort method that would call quick\_sort\_pivot, a method in charge of the actual sorting, and partition to move around the elements in the list. I had a third function for readability and to make it a top-down design. Starting from the main quick\_sort method, it calls quick\_sort\_pivot with the list, the start index of 0, the end index from the last index of the list.

Quick\_sort\_pivot is done and will keep going until the start reaches the end index, which means it is sorted. First, a pivot is created from the partition function with the start and end parameters.

Partition starts with the l pointer that is 1 since it would be redundant for it to be 0 and r which is the last element in the list. The pivot is the first element in the list for the typical and easier algorithm. Fin is a Boolean flag that indicates that the pivot has been put in the appropriate place. In the outer while loop, it will keep going until fin is true since it means that all nodes have been shifted for the pivot. The first inner loop oversees moving elements to the left of the list if they are less than the pivot. This means that the left index increases by one since that means that element is already less than the pivot. The second inner loop is also similar but instead with the right index. If the right index is less than the left index then that means they have crossed, meaning that the elements have all shifted appropriately, this makes fin true and swaps the start index and r because r should be in the left due to the new shift and then r is returned as the pivot. If r is still more than l than there is still working to do which means swapping the left and right node to shift because they have been marked to be in the wrong place, so they need to swap.

Once the pivot has been returned, then recursively call from the left and the right from the pivot. This process is repeated for each smaller sub list and for each smaller list, a pivot is chosen. Once the start is greater than end then it means it has already reached the end of the list, which means everything has been sorted. Nothing is returned because of the manipulation of references within the original list.

Modified Quick Sort (MQ)

This quick sort will only sort half of a list since its main purpose is to find the median as fast as possible. I simply repurposed some parts of my original code with slight modifications. For partition, an early issue I ran into by trying to reuse the original partition was that the pivot would not change. I realized that I needed my original pivot to partition around it since I would only be sorting half of the list which means adding a new parameter to the new partition\_r method and by changing some of the original behavior.

Partition r oversees finding an element n that is the index in the position of the sorted list. If trying to find the median, then n has to be half the size of the list. So instead of start and end, they are now l and r because of looking for a certain item rather than the whole list. This means only one loop and like the original partition behavior except for the direct comparison to the pivot. Once the list has been sorted, then it returns the pivot.

In the main method, mod\_quick\_sort, if the left and right index are equal then it is already sorted and is only on the element so just return the list. If not, then shuffle the pivot between the left and right indexes. I implemented this snippet for fun and to see if the algorithm would work with a different pivot each time. Then pivot is assigned the new pivot based on the random pivot by calling partition\_r. If the element, n, which is half of the length of the list is equal to the new pivot then return the element at the position n since it has been sorted. If n is less than pivot, then recursively call to the left list because it must be less. Then if n is greater than recursively call to the right of the pivot because it must be a bigger element. Nothing is returned in this method since of modifying the references and the pivot will be the median since we stated earlier that we were looking for the median.

**User Interface/Main**

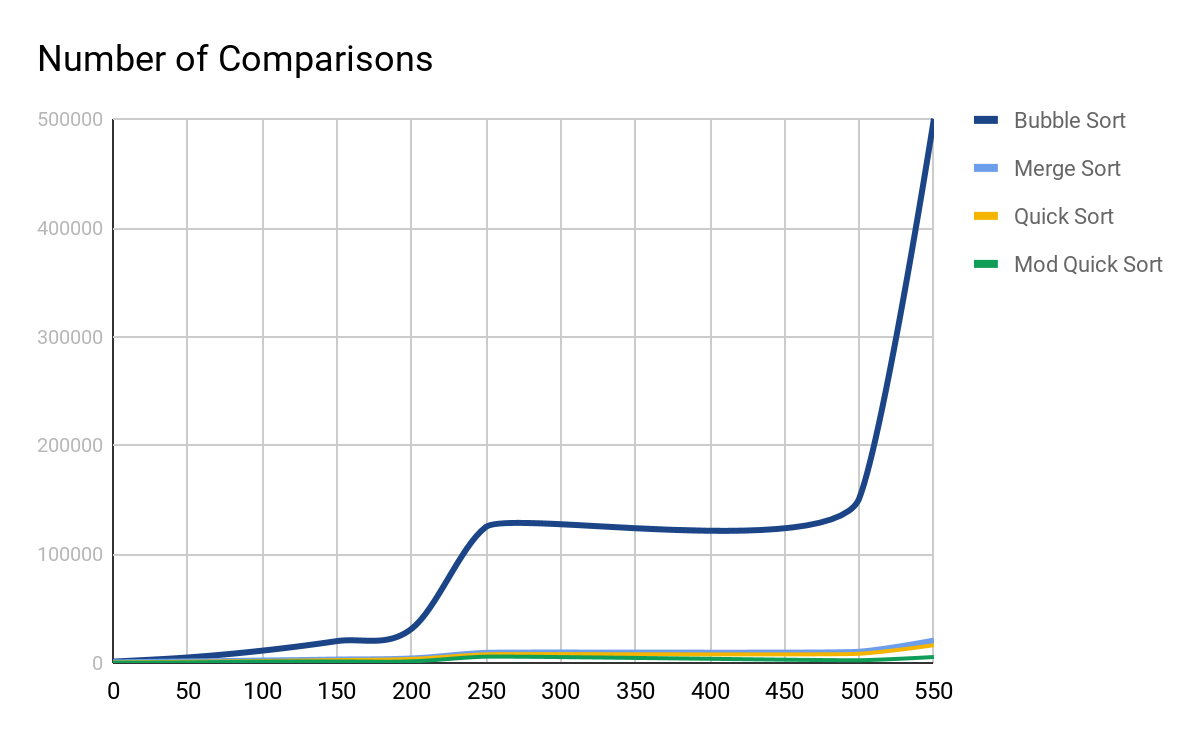
First, a linked list of size n of distinct integers is generated. Then c\_list is made to copy the elements L and is passed to the median function. The original list is also printed for reference and to see if a list was sorted. A menu assigned with the number of each sorting algorithm is printed to the console for readability and access. While I originally had a menu, I wanted to already compare all of them running at the same time, so it was abandoned for just calling the same median function for each algorithm. There is a simple try and catch for erroneous input but can be abandoned easily for more intensive operations.

# **Experimental Results**

To test my sorting algorithms, I printed the original list first then the new c\_list which was sorted. The median was also returned for each function to see if they returned the same value and if each list had distinct integers. The number of comparisons in the table below is based on the number of each assigned global variable for each function. This means that the number of swaps is not counted and almost every comparison, including helper functions, were counted. At 0, no swaps are done due to median automatically returning 0 if there are no elements in a list. The size of the list is incremented by 50 for a consistent graph and for hardware limitations since my machine can only reach up to 1000 to run all algorithms consistently. In my code, I have left the actual amount of time in seconds it took for each function to see if I could implement this for future labs.

For utility functions, they were all tested first with multiple inputs before any implementation of the sorting algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size of List (n) | Number of Comparisons for Algorithm | | | |
|  | Bubble Sort | Merge Sort | Quick Sort | Mod Quick Sort |
| 0 | 0 | 0 | 0 | 0 |
| 50 | 1274 | 573 | 503 | 259 |
| 100 | 5049 | 1376 | 1103 | 533 |
| 150 | 11324 | 2225 | 1824 | 1245 |
| 200 | 20099 | 3141 | 2637 | 1351 |
| 250 | 31374 | 4139 | 3732 | 1505 |
| 500 | 125249 | 9289 | 7942 | 5795 |
| 550 | 151524 | 10301 | 8449 | 2497 |
| 1000 | 500499 | 20595 | 16362 | 5491 |



# **General Input/output**

|  |  |
| --- | --- |
| **Input (n)** | **Output** |
| **0** | **0**  **0**  **0**  **0** |
| **50**  **Original List:**  **1201 1161 1040 818 18 621 1009 643 150 48 398 869 580 689 194 1255 575 638 240 1010 323 521 1055 949 811 840 40 628 701 1135 109 607 993 1294 461 704 622 1251 1466 1309 292 1006 1047 367 145 989 940 33 1254 1123** | **Bubble Sort:**  **18 33 40 48 109 145 150 194 240 292 323 367 398 461 521 575 580 607 621 622 628 638 643 689 701 704 811 818 840 869 940 949 989 993 1006 1009 1010 1040 1047 1055 1123 1135 1161 1201 1251 1254 1255 1294 1309 1466**  **Time: 0.0011558390136840418**  **COUNT: 1274**  **704**  **Merge Sort:**  **18 33 40 48 109 145 150 194 240 292 323 367 398 461 521 575 580 607 621 622 628 638 643 689 701 704 811 818 840 869 940 949 989 993 1006 1009 1010 1040 1047 1055 1123 1135 1161 1201 1251 1254 1255 1294 1309 1466**  **Time: 0.0003699196843352029**  **COUNT: 581**  **704**  **Quick Sort:**  **18 33 40 48 109 145 150 194 240 292 323 367 398 461 521 575 580 607 621 622 628 638 643 689 701 704 811 818 840 869 940 949 989 993 1006 1009 1010 1040 1047 1055 1123 1135 1161 1201 1251 1254 1255 1294 1309 1466**  **Time: 0.0038143967450481103**  **COUNT: 487**  **704**  **Mod Quick Sort:**  **18 150 48 398 194 240 323 521 40 109 461 292 367 145 33 575 638 621 643 628 607 622 580 689 701 704 811 949 993 818 840 869 1006 989 940 1009 1123 1251 1466 1309 1010 1161 1047 1055 1255 1294 1135 1201 1254 1040**  **Time: 0.0019336516832838962**  **COUNT: 255**  **704** |
| **100**  **Original List:**  **1310 1103 1024 902 1077 188 724 1031 1084 177 511 344 1440 253 849 661 101 981 600 530 1060 1228 741 869 1380 88 738 882 1030 766 885 1478 288 486 1188 82 1009 182 1255 165 977 1144 17 789 733 1462 485 770 1299 950 541 1001 807 1038 437 984 1045 251 138 1252 23 1081 848 460 118 1493 1461 147 405 1139 227 72 952 736 124 270 1291 1344 688 429 187 396 1289 341 1192 145 853 1237 465 368 921 995 276 825 1405 287 420 475 878 1158** | **Bubble Sort:**  **17 23 72 82 88 101 118 124 138 145 147 165 177 182 187 188 227 251 253 270 276 287 288 341 344 368 396 405 420 429 437 460 465 475 485 486 511 530 541 600 661 688 724 733 736 738 741 766 770 789 807 825 848 849 853 869 878 882 885 902 921 950 952 977 981 984 995 1001 1009 1024 1030 1031 1038 1045 1060 1077 1081 1084 1103 1139 1144 1158 1188 1192 1228 1237 1252 1255 1289 1291 1299 1310 1344 1380 1405 1440 1461 1462 1478 1493**  **Time: 0.003173117292273244**  **COUNT: 5049**  **807**  **Merge Sort:**  **17 23 72 82 88 101 118 124 138 145 147 165 177 182 187 188 227 251 253 270 276 287 288 341 344 368 396 405 420 429 437 460 465 475 485 486 511 530 541 600 661 688 724 733 736 738 741 766 770 789 807 825 848 849 853 869 878 882 885 902 921 950 952 977 981 984 995 1001 1009 1024 1030 1031 1038 1045 1060 1077 1081 1084 1103 1139 1144 1158 1188 1192 1228 1237 1252 1255 1289 1291 1299 1310 1344 1380 1405 1440 1461 1462 1478 1493**  **Time: 0.0008093859759906331**  **COUNT: 1384**  **807**  **Quick Sort:**  **17 23 72 82 88 101 118 124 138 145 147 165 177 182 187 188 227 251 253 270 276 287 288 341 344 368 396 405 420 429 437 460 465 475 485 486 511 530 541 600 661 688 724 733 736 738 741 766 770 789 807 825 848 849 853 869 878 882 885 902 921 950 952 977 981 984 995 1001 1009 1024 1030 1031 1038 1045 1060 1077 1081 1084 1103 1139 1144 1158 1188 1192 1228 1237 1252 1255 1289 1291 1299 1310 1344 1380 1405 1440 1461 1462 1478 1493**  **Time: 0.017589318323781696**  **COUNT: 1157**  **807**  **Mod Quick Sort:**  **17 177 344 253 101 88 288 188 82 182 165 475 485 437 251 138 23 460 118 147 405 227 72 124 270 429 187 396 341 145 465 368 276 287 420 486 733 736 511 688 541 661 724 600 530 738 741 766 770 789 807 825 848 853 849 869 1045 1030 1103 1252 885 1081 1478 1031 1084 1493 1461 1188 981 1139 1009 1024 952 1255 902 977 1291 1344 1144 1060 1228 1077 1289 1462 1192 1158 1380 1237 1299 950 921 995 1440 1001 1405 1310 1038 882 878 984**  **Time: 0.006217381361167902**  **COUNT: 505**  **807** |

# 

# 

# **Conclusions**

I learned the importance of understanding algorithms and then applying it to a variety of situations and proposed. I also learned a lot about the manipulation of linked lists and how fluid it is when navigating and sorting it. Compared to the rigidness of an array and weirdness of a python native list, I enjoyed navigating a class that I created. By meshing these two concepts, it really is important to know the true mechanisms. While it is easy to know how to sort in one language, it is better to do so with a general concept in mind and then translate it to code.

# **Appendix**

|  |  |
| --- | --- |
| """ |  |
|  | Course: CS 2302 [MW 1:30-2:50] |
|  | Author: Kimberly Morales |
|  | Assignment: Lab 2 |
|  | Instructor: Olac Fuentes |
|  | TA(s): Anindita Nath , Maliheh Zargaran |
|  | Date: 2/22/2019 |
|  | Date of last modification: 2/22/2019 |
|  |  |
|  | Purpose of program: |
|  | To implement the sorting algorithms to sort a singly linked list: quick, merge, bubble, and a modified version of quick sort. |
|  | Then, after the list is sorted, a method called median will find the middle element in the sorted list. |
|  | The program will attempt to count the number of comparisions in each algorithm. |
|  |  |
|  | """ |
|  | import timeit |
|  | import random |
|  |  |
|  | ################################################################################################# |
|  | #NODE CLASS |
|  | ################################################################################################# |
|  | class Node(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self, item, next=None): |
|  | self.item = item |
|  | self.next = next |
|  |  |
|  | ################################################################################################# |
|  | #LINKED LIST CLASS |
|  | ################################################################################################# |
|  | class List(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self): |
|  | self.head = None |
|  | self.tail = None |
|  |  |
|  | def IsEmpty(L): |
|  | return L.head == None |
|  |  |
|  | def append\_list(L,x): |
|  | #Appends an item to the end of the list |
|  | if IsEmpty(L): |
|  | L.head = Node(x) |
|  | L.tail = L.head |
|  |  |
|  | else: |
|  | L.tail.next = Node(x) |
|  | L.tail = L.tail.next |
|  |  |
|  | def print\_list(title,L): |
|  | # Prints list L's items in order using a loop |
|  | temp = L.head |
|  | print(title) |
|  |  |
|  | while temp is not None: |
|  | print(str(temp.item), end=' ') |
|  | temp = temp.next |
|  | print()# New line |
|  |  |
|  | #METHOD get\_length: Gets the length of the linked list by counting the number of nodes within a while loop |
|  | def get\_length(L): |
|  | temp = L.head |
|  | size\_list = 0 |
|  |  |
|  | while temp is not None: |
|  | size\_list += 1 |
|  | temp = temp.next |
|  |  |
|  | return size\_list |
|  |  |
|  | #METHOD element\_at: Finds the node reference with an index |
|  | def element\_at(L,e): |
|  | temp = L.head |
|  | n = get\_length(L) |
|  |  |
|  | #If the index is out of bounds then return the head |
|  | if e >= n : |
|  | return L.head |
|  |  |
|  | for i in range(n): |
|  | if i == e: |
|  | return temp |
|  | temp = temp.next |
|  |  |
|  | ################################################################################################# |
|  | #SORTING METHODS |
|  | ################################################################################################# |
|  |  |
|  | #Num of comparisions for each algorithm |
|  | BCOUNT = 0 #Bubble Sort |
|  | MCOUNT = 0 #Merge Sort |
|  | QCOUNT = 0 #Quick Sort |
|  | QMCOUNT = 0 #Modified Quick Sort |
|  |  |
|  | ############################################Bubble Sort########################################### |
|  | def bubble\_sort(L): |
|  | global BCOUNT |
|  |  |
|  | if L.head is None: |
|  | return L |
|  |  |
|  | i = L.head |
|  | j = L.head.next |
|  |  |
|  | #Stay in i while comparing with j until it hits none |
|  | while i is not None: |
|  |  |
|  | while j is not None: |
|  | BCOUNT+=1 |
|  |  |
|  | if i.item > j.item: |
|  | swap(i,j) |
|  |  |
|  | j = j.next |
|  | #move the i node to the next one |
|  | i = i.next |
|  |  |
|  | #j is assigned the ref to i |
|  | j = i |
|  |  |
|  | return L |
|  |  |
|  | ############################################Merge Sort########################################### |
|  | def merge\_sort(h): |
|  | global MCOUNT |
|  |  |
|  | if h is None or h.next is None: |
|  | return h |
|  | MCOUNT+=1 |
|  | L1, L2 = split\_list(h) #split the list into two different lists |
|  | L1 = merge\_sort(L1) |
|  | L2 = merge\_sort(L2) |
|  | h = merge\_list(L1,L2) |
|  |  |
|  | return h |
|  |  |
|  | #Merges the left and right lists into a single list |
|  | def merge\_list(L1,L2): |
|  | temp = None |
|  |  |
|  | #If the left list is empty then return the right list only |
|  | if L1 is None: |
|  | return L2 |
|  |  |
|  | #If the right list is empty then return the left list only |
|  | if L2 is None: |
|  | return L1 |
|  |  |
|  | #If the left list node value is right list node value then append the |
|  | if L1.item <= L2.item: |
|  | temp = L1 |
|  | temp.next = merge\_list(L1.next,L2) |
|  | else: |
|  | temp = L2 |
|  | temp.next = merge\_list(L1, L2.next) |
|  |  |
|  | return temp |
|  |  |
|  | #Splits list into two halves based on the middle index |
|  | def split\_list(L): |
|  | l = L #Left/ slow pointer |
|  | r = L #right/ fast pointer |
|  |  |
|  | #If r is not none then shift r to the next node |
|  | if r: |
|  | r = r.next |
|  |  |
|  | #While r is not none then shift r to the next node and if r is not none still then shift r again and l to the next node |
|  | while r: |
|  | r = r.next |
|  |  |
|  | if r: |
|  | r = r.next |
|  | l = l.next |
|  |  |
|  | #mid is the reference to the start of the right l |
|  | mid = l.next |
|  | l.next = None |
|  | return L, mid |
|  |  |
|  | ############################################Quick Sort########################################### |
|  |  |
|  | #Method quick\_sort: Handles partition and pivot functions and is seperated for legibility |
|  | def quick\_sort(L): |
|  | quick\_sort\_pivot(L,0,get\_length(L)-1) |
|  |  |
|  |  |
|  | #Method quick\_sort\_pivot: Based on the pivots position, will sort from the left and right |
|  | def quick\_sort\_pivot(L, start, end): |
|  | if start < end: |
|  | global QCOUNT |
|  | QCOUNT+=1 |
|  |  |
|  | pivot = partition(L,start,end) |
|  | quick\_sort\_pivot(L,start,pivot-1) |
|  | quick\_sort\_pivot(L,pivot+1,end) |
|  |  |
|  | #Method partition: moves elements to the left or right sublist based on if its less than or greather than the pivot |
|  | def partition(L,start,end): |
|  | global QCOUNT |
|  | l = start + 1 #Should initially start after pivot |
|  | r = end |
|  | pivot = element\_at(L,start).item #pivot: is the start of the list |
|  | fin = False #fin: boolean flag that indicates that all elements in a list was sorted |
|  |  |
|  | while not fin: |
|  | #While the l pointer is behind then shift element to the left |
|  |  |
|  | while l <= r and element\_at(L,l).item <= pivot: |
|  | l = l+1 |
|  | #QCOUNT+=1 |
|  |  |
|  | #While the r pointer is ahead then shift element to the right by one less |
|  | while element\_at(L,r).item >= pivot and r >= l: |
|  | #QCOUNT+=1 |
|  | r = r-1 |
|  |  |
|  | #If r list less than pivot and l list more than pivot then partition is done |
|  | if r < l: |
|  | #QCOUNT+=1 |
|  | fin = True |
|  |  |
|  | #Swap the l and r pointers in the list |
|  | else: |
|  | swap(element\_at(L,l), element\_at(L,r)) |
|  |  |
|  | swap(element\_at(L,start),element\_at(L,r)) |
|  |  |
|  | return r |
|  |  |
|  | ########################################Modified Quick Sort##################################### |
|  | def mod\_quick\_sort(L,l,r,n): |
|  | #Return the left point if the array is one element |
|  | global QMCOUNT |
|  |  |
|  | QMCOUNT+=1 |
|  | if l == r: |
|  | return element\_at(L,l) |
|  |  |
|  | pivot = random.randint(l,r) #Randomize the pivot between left and right point |
|  | pivot = partition\_r(L,l,r,pivot) #Move pivot based on elements |
|  |  |
|  | #If n is equal to pivot then return the element at n |
|  | QMCOUNT+=1 |
|  | if n == pivot: |
|  | return element\_at(L,n) |
|  |  |
|  | #If n is less than or greater than pivot then recurse one sublist |
|  | QMCOUNT+=1 |
|  | if n < pivot: |
|  | return mod\_quick\_sort(L,l,pivot-1,n) |
|  |  |
|  | else: |
|  | return mod\_quick\_sort(L,pivot+1,r,n) |
|  |  |
|  | #Modified partition that also swaps with a specified pivot index |
|  | def partition\_r(L,l,r, pivot): |
|  | pivot\_val = element\_at(L,pivot).item #Value of the pivot |
|  | swap(element\_at(L,pivot),element\_at(L,r)) #move pivot to the end |
|  | temp = l #Stores l temporarily |
|  | fin = False #flag that indicates that partition is done |
|  | i = l #Index to swap |
|  |  |
|  | while i < r: |
|  | if element\_at(L,i).item < pivot\_val: #If i val is less than pivot then store moves |
|  | swap(element\_at(L,i), element\_at(L,temp)) |
|  | temp+=1 |
|  | i+=1 |
|  |  |
|  | swap(element\_at(L,r), element\_at(L,temp)) #Swap store and r to place pivot appropriately |
|  | return temp |
|  |  |
|  | ################################################################################################# |
|  | #UTILITY METHODS |
|  | ################################################################################################# |
|  | #Generate random ints and appeand to the list |
|  | def gen\_rand\_list(n): |
|  | L = List() |
|  | #Creates a python native list of size n with distinct elements |
|  | r = random.sample(range(1000), n) |
|  |  |
|  | for i in range(n): |
|  | #Append the r list to the list |
|  | append\_list(L,r[i]) |
|  | return L |
|  |  |
|  | #Swap two nodes with each other |
|  | def swap(i,j): |
|  | k = Node(i.item,None) |
|  | i.item = j.item |
|  | j.item = k.item |
|  |  |
|  | #Copys a list into a new one |
|  | def copy\_list(L): |
|  | temp = L.head |
|  | c\_list = List() |
|  |  |
|  | while temp is not None: |
|  | append\_list(c\_list,temp.item) |
|  | temp = temp.next |
|  |  |
|  | return c\_list |
|  |  |
|  | #Method Median: Finds the median in each sorted list, prints time when running, and the number of comparisions |
|  | def median(L,a\_num): |
|  | c\_list = copy\_list(L) |
|  | if L is None: |
|  | return None |
|  |  |
|  | #Error message |
|  | if a\_num > 3 or a\_num < 0: |
|  | return "ERROR: Menu choice is over 3 or less than 0" |
|  |  |
|  | #Bubble Sort |
|  | if a\_num == 0: |
|  | start = timeit.default\_timer() |
|  | bubble\_sort(c\_list) |
|  | stop = timeit.default\_timer() |
|  |  |
|  | print\_list("Bubble Sort: ", c\_list) |
|  | print('Time: ', stop - start) |
|  | print("COUNT: " + str(BCOUNT)) |
|  |  |
|  | #Merge Sort |
|  | if a\_num == 1: |
|  | start = timeit.default\_timer() |
|  | c\_list.head = merge\_sort(c\_list.head) |
|  | stop = timeit.default\_timer() |
|  |  |
|  | print\_list("Merge Sort: ", c\_list) |
|  | print('Time: ', stop - start) |
|  | print("COUNT: " + str(MCOUNT)) |
|  |  |
|  | #Quick Sort |
|  | if a\_num == 2: |
|  | start = timeit.default\_timer() |
|  | quick\_sort(c\_list) |
|  | stop = timeit.default\_timer() |
|  |  |
|  | print\_list("Quick Sort: ", c\_list) |
|  | print('Time: ', stop - start) |
|  | print("COUNT: " + str(QCOUNT)) |
|  |  |
|  | #Mod Quick Sort |
|  | if a\_num == 3: |
|  | start = timeit.default\_timer() |
|  | mod\_quick\_sort(c\_list,0,get\_length(c\_list)-1,get\_length(c\_list)//2) |
|  | stop = timeit.default\_timer() |
|  |  |
|  | print\_list("Mod Quick Sort: ", c\_list) |
|  | print('Time: ', stop - start) |
|  | print("COUNT: " + str(QMCOUNT)) |
|  |  |
|  | #Get median from the list |
|  | med = element\_at(c\_list, get\_length(c\_list) //2) |
|  | return med |
|  |  |
|  | ################################################################################################# |
|  | #MAIN |
|  | ################################################################################################# |
|  | try: |
|  | size\_list = int(input('Enter the size of the list: \n')) |
|  |  |
|  | #Generate list of random numbers and copy it |
|  | L = gen\_rand\_list(size\_list) |
|  | c\_list = copy\_list(L) |
|  |  |
|  | print\_list("Original List: ", L) |
|  | print() |
|  |  |
|  | print("Enter the sorting algorithm\n" |
|  | + "\n0: Bubble Sort " |
|  | + "\n1: Merge Sort" |
|  | + "\n2: Quick Sort" |
|  | + "\n3: Mod Quick Sort" ) |
|  | #choice = int(input()) |
|  |  |
|  | #Print the medians of the methods |
|  | print(str(median(L,0).item) + "\n") #Bubble sort |
|  | print(str(median(L,1).item) + "\n") #Merge Sort |
|  | print(str(median(L,2).item) + "\n") #Quick Sort |
|  | print(str(median(L,3).item) + "\n") #M Quick Sort |
|  |  |
|  | except ValueError: |
|  | print("ERROR: Invalid input") |

# **Academic Honesty**

“I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”

Name: Kimberly Morales

Signature: