Kevin Savill

10-1-2022

Project 1

CS3345.502

**Part 1. Approach and Challenges for Project**

For Project 1, I decided that to best present my information and have quick reiterations of test I would utilize the JavaFX library. Not only would this allow me to have a visual reminder of what is selected before I go to run my tests, but I can quickly change my data type or algorithm type without having to start the program over again. Along with that, if there is an exception during testing, the entire program does not crash and I can do incremental tests with my gui (changing input values) to see where the breaks are occurring.

I used two sets of radio buttons; one group for the algorithm type selection (insertion sort, selection sort, quick sort, merge sort, heap sort, and radix sort) and the other for list properties (InOrder, ReverseOrder, AlmostOrder and Random). Next, I had an input text field for getting the list size. And a start button to call my main functions. As I was using a input text field, I needed to be mindful of the inputs that are given.

Upon calling the Start button, there is an inner class method (lambda expression) that detects the event action and begins the beginProcss function. One of the first things that is done is input validation for the list size. I first try to parse the input into an integer, and if there is an issue, the program stop and the error is displayed to console. If the input parses to integer successfully and is less than 1, then this issue is also displayed to console and the rest of the program does not continue.

Once input validation for the text field is complete, I pull the values of text values of both radio groups and run this through two sets of if else statements to determine what datatype generator function to call and return the testArray[] to use for algorithm sorting, and the algorithm method for the test within the runSorting function. The runSorting function also initializes a new array labeled “resultsArray.” The length of this array is 6 and is used for returning all result information to my main inner class function which is then displayed to my Gui with each results text field setText method.

Within each sorting algorithm, my initial approach was to use the similar method described above where I would simply create an array in each class and return that single variable as java does not support returning multiple variables at a time (I’m familiar with Python, which does support this feature), however, after successfully implementing this into my Insertion and Selection sort methods, I ran into a roadblock getting this done in the quicksort method. I determined that as the rest of the functions is likely going to call other functions, it would be sloppy to not just initialize a comparison and movement variable in the public class itself and write two methods that can return each variable. So I scrapped my initial implementation and initialized comparison and movement counters after class “sorting type” {. Now, for most of the functions, when I would call the sorting method itself, I would then declare the counter variables to zero, however, some of these sorting methods call itself over and over again (merge sort for example), so this would skew my data to being much smaller than it should be. To amend this problem, I simply made a new method called resetCounters() which declared the comparison and movement variables to zero.

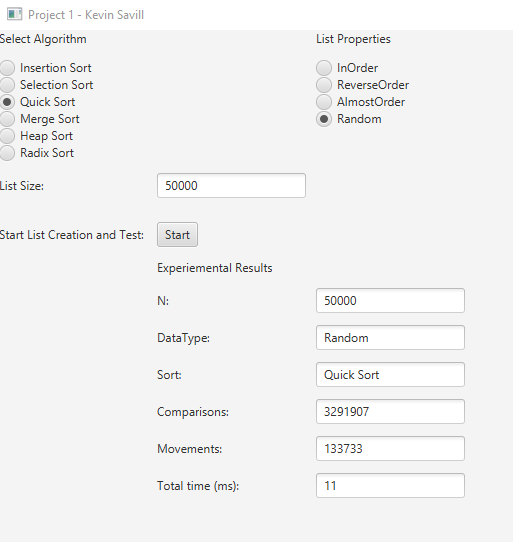
This approach worked for every algorithm, but the heapsort method needed to expand upon this concept a little more. As the heapsort class also extends heap, I would need to initialize the comparison and movements variables within the heap class and add my methods for returning those metrics. The order in which I returned those is as follows:

* 1. Initialize comparisons and movements in heapsort
  2. Initialize comparisons and movements in heap
  3. Run through the classes to count up the sorting metrics
  4. Return the comparison and movement metrics from the heap class into heapsort class
  5. Add those metrics from heap into the comparison and movement metrics
  6. Return the total sum of comparison and movement metrics respectively

In the project description, there were a few caveats in the quick sort that was suggested. I found that quick sort was effectively causing infinite recursion, so I corrected this with student 4’s approach of setting pivot as (front+back)/2.

Another requirement of this project was measuring the total time of the algorithm. I utilized java’s System.nanoTime(); functionality and before I called the sorting algorithm, I marked the current time and then immediately after and calculated the total time in miliseconds by (endTime – startTime) / 1000000.

Once everything runs, this is what the GUI application will look like:



**Part 2 Sorting Algorithm Types**

**Insertion Sort**

* How it works: Insertion sort works by looping through the list each iteration by removing one element and inserting it within the array where it is supposed to be. This is best for smaller sets of data. Once there are no more elements to move, the algorithm finishes. This is fastest in cases where the array is already sorted. It is recommended to use this algorithm for very small sets of data as it becomes exponentially more expensive in terms of time.
* Best Case Time: O(N)
* Average Case Time: O(N2)
* Worst Case Time: O(N2)

**Selection Sort**

* How it works: Selection sort works by extracting the smallest element in the array and moving this to the front of the array until the underlying array is sorted. This is a very time consuming algorithm as even if the data is already sorted, you are scanning through each element of the array, finding the smallest element and moving to the front of the array. This is only best to use in cases where your dataset is small.
* Best Case Time: O(N2)
* Average Case Time: O(N2)
* Worst Case Time: O(N2)

**Quick Sort**

* How it works: Quick sort works by breaking the initial array down into smaller components and using a “pivot” to see where the smaller arrays should be moved. Smaller data is shifted left while larger is shifted right and combined. This step is repeated until the underlying array is sorted. The best case for this algorithm would be when the pivot is already in the middle of the data after the first passthrough, and is most inefficient when the pivot is not in the center and has to go through more iterations to shift the data until sorted.
* Best Case Time: O(NlogN)
* Average Case Time: O(NlogN)
* Worst Case Time: O(N2)

**Merge Sort**

* How it works: Merge sort works by a “divide and conquer” approach, breaking down the array into two or more subarrays until they are each one element and then begins merging them together in order. As this is performing the same operation for a set of data in any order, the output times will be around the same no matter what.
* Best Case Time: O(NlogN)
* Average Case Time: O(NlogN)
* Worst Case Time: O(NlogN)

**Heap Sort**

* How it works: Heap sort works by splitting up the data into two sections: unsorted and sorted. The algorithm loops through the data, extracting the largest value until the data is sorted in the sorted section.
* Best Case Time: O(NlogN)
* Average Case Time: O(NlogN)
* Worst Case Time: O(NlogN)

**Radix Sort**

* How it works: Radix sort works by ordering the digits in a base 10 format (sorting in the 1’s place, then 10’s, 100’s ,… ) until ordered from smallest to largest. This is one of the faster algorithms overall, where the total time across every use case is O(N) multiplied by the size of the amount of digits in the largest value.
* Best Case Time: O(Nk)
* Average Case Time: O(Nk)
* Worst Case Time: O(Nk)

**Part 3. Experiment results data**

|  |  |  |  |
| --- | --- | --- | --- |
| Experiment Results | ArraySize | 50k elements |  |
| List Property: In Order | Comparisons | Movements | Total Time |
| Insertion Sort | 49999 | 0 | 1 |
| Selection Sort | 1250024999 | 0 | 373 |
| Quick Sort | 890524 | 23446 | 6 |
| Merge Sort | 1034460 | 884462 | 5 |
| Heap Sort | 4841370 | 5023887 | 6 |
| Radix Sort | 8000070 | 8000049 | 3 |
|  |  |  |  |
| List Property: Reverse Order |  |  |  |
| Insertion Sort | 1250024999 | 1249975000 | 839 |
| Selection Sort | 1250024999 | 25000 | 1258 |
| Quick Sort | 940522 | 48445 | 1 |
| Merge Sort | 1034460 | 884462 | 3 |
| Heap Sort | 3621908 | 3070822 | 8 |
| Radix Sort | 800070 | 750050 | 2 |
|  |  |  |  |
| List Order: Almost Order |  |  |  |
| Insertion Sort | 255224356 | 255174357 | 179 |
| Selection Sort | 1250024999 | 49526 | 944 |
| Quick Sort | 3724078 | 102844 | 4 |
| Merge Sort | 1034460 | 884462 | 4 |
| Heap Sort | 3907901 | 3492224 | 4 |
| Radix Sort | 800070 | 750068 | 2 |
|  |  |  |  |
| List Order: Random Order |  |  |  |
| Insertion Sort | 624357548 | 624307549 | 444 |
| Selection Sort | 1250024999 | 49995 | 948 |
| Quick Sort | 5655694 | 116957 | 3 |
| Merge Sort | 1034460 | 884462 | 5 |
| Heap Sort | 3751861 | 3274319 | 8 |
| Radix Sort | 800070 | 750061 | 2 |

**Part 4. Table results of best and worst cases**

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Best Algorithm** | **Worst Algorithm** |
| In Order | Insertion Sort | Selection Sort |
| Reverse Order | Quick Sort | Selection Sort |
| Almost Order | Radix Sort | Selection Sort |
| Random Order | Radix Sort | Selection sort |