Joshua Greene
Dr. Baas
Data Structures
10/26/20

# String Sorting Lab Report

This lab had us implement 7 sorting algorithms (Bubble, Selection, Insertion, Shell, Heap, Merge, and Quick) to sort a randomized list of words in ascending order. Instead of reordering the words themselves, my code uses the indices of the words for the actual sorting. Computationally, this is way faster than switching or swapping the positions of Strings since Strings are immutable. Thus, my code used an array for the words and an array for the words' indices. Whenever a sorting algorithm required a comparison, I still needed to compare the words themselves. So, I simply found the word by inputting the index array value in for the index value of the words array like so: wordsArr[indexArr[i]]. It is also important to note that before each sort was performed, I made sure to reset the index values, so I didn't sort an already sorted list. I also made sure to verify my sorted lists to make sure they were in fact sorted. Below are the empirical results I found with my program:

**Bubble Sort**:

| Number of Words Sorted: | Time to Sort (seconds): |
|---|---|
| 5,000 | 0.258 |
| 10,000 | 1.157 |
| 15,000 | 2.108 |
| 20,000 | 4.106 |
| 25,000 | 6.928 |
| 30,000 | 10.491 |
| 35,000 | 16.454 |
| 40,000 | 22.813 |
| 45,403 | 34.180 |

**Selection Sort**:

| Number of Words Sorted: | Time to Sort (seconds): |
|---|---|
| 5,000 | 0.113 |
| 10,000 | 0.435 |
| 15,000 | 0.719 |
| 20,000 | 1.444 |
| 25,000 | 2.512 |
| 30,000 | 4.370 |
| 35,000 | 7.636 |
| 40,000 | 11.912 |
| 45,403 | 18.398 |

**Insertion Sort**:

| Number of Words Sorted: | Time to Sort (seconds): |
|---|---|
| 5,000 | 0.04 |
| 10,000 | 0.152 |
| 15,000 | 0.259 |
| 20,000 | 0.520 |
| 25,000 | 0.853 |
| 30,000 | 1.340 |
| 35,000 | 2.091 |
| 40,000 | 3.240 |
| 45,403 | 4.540 |

**Shell Sort**:

| Number of Words Sorted: | Time to Sort (seconds): |
|---|---|
| 5,000 | 0.0064453 |
| 10,000 | 0.0141357 |
| 15,000 | 0.0132345 |
| 20,000 | 0.0279611 |
| 25,000 | 0.0244581 |
| 30,000 | 0.0330374 |
| 35,000 | 0.0447400 |
| 40,000 | 0.0501434 |
| 45,403 | 0.0631557 |

**Heap Sort**:

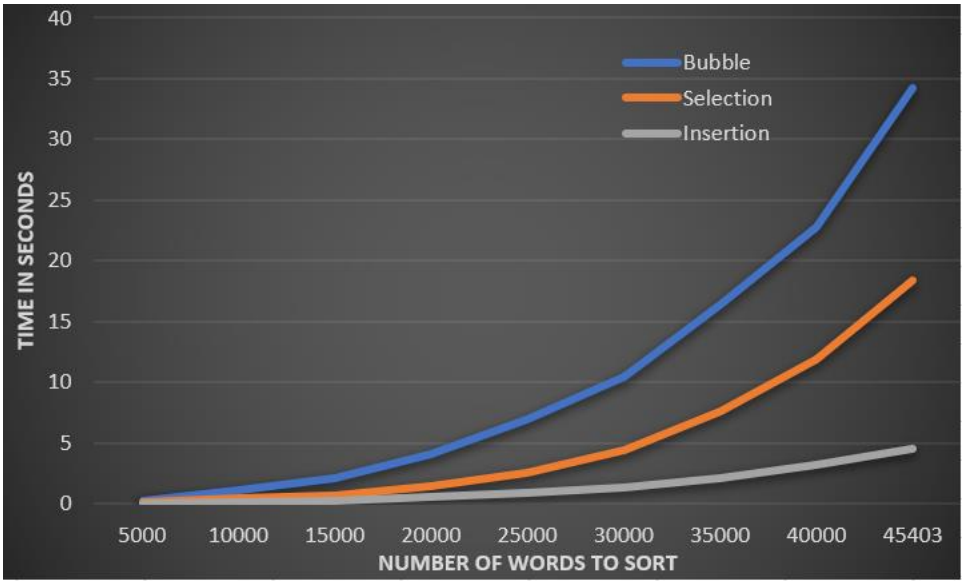
| Number of Words Sorted: | Time to Sort (seconds): |
|---|---|
| 5,000 | 0.0034329 |
| 10,000 | 0.0064005 |
| 15,000 | 0.0086094 |
| 20,000 | 0.0123870 |
| 25,000 | 0.0159041 |
| 30,000 | 0.0203229 |
| 35,000 | 0.0247300 |
| 40,000 | 0.0277186 |
| 45,403 | 0.0368453 |

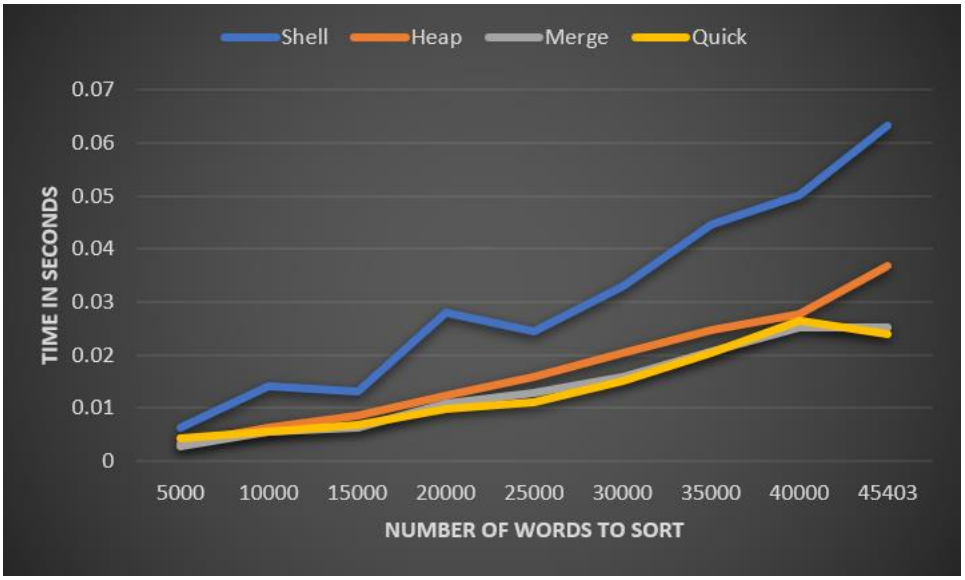
**Merge Sort**:

| Number of Words Sorted: | Time to Sort (seconds): |
|---|---|
| 5,000 | 0.0029143 |
| 10,000 | 0.0055229 |
| 15,000 | 0.0063050 |
| 20,000 | 0.0108851 |
| 25,000 | 0.0128205 |
| 30,000 | 0.0160250 |
| 35,000 | 0.0206688 |
| 40,000 | 0.0251280 |
| 45,403 | 0.0251728 |

**Quick Sort**:

| Number of Words Sorted: | Time to Sort (seconds): |
|---|---|
| 5,000 | 0.0042873 |
| 10,000 | 0.0056830 |
| 15,000 | 0.0068886 |
| 20,000 | 0.0099483 |
| 25,000 | 0.0111224 |
| 30,000 | 0.0150701 |
| 35,000 | 0.0204686 |
| 40,000 | 0.0265997 |
| 45,403 | 0.0238933 |






As expected, when looking at these graphs, one can see just how more efficient the non-quadratic sort algorithms are! I didn't expect the Shell sort to work as well as it did! The textbook seemed to group it with the quadratic sorts but based off this lab, I would say it is far more in-line along with the other sorts! It is interesting to note that the Shell and Quick sorts had points where the time to compute the sort actually decreased as more words were added! I am not sure what could have caused this but it's possible that the partitioning and gaps used in these sorts weren't optimized for some of the word counts. This was a very eye-opening lab and I definitely have a stronger understanding of these sorting algorithms! Below is the source code used for this lab:

```java
/*
File Name : StringSortingLab.java
Program Author(s) : Joshua Greene
Course Number & Title : Data Structures (COSC2203)
Assignment Number & Name : Lab #4: String Sorting Lab
Due Date : 10/21/2020

Description :

This lab had us implement 7 sorting algorithms (Bubble, Selection, Insertion,
Shell, Heap, Merge, and Quick sort) and had us calculate the time to sort a list
of randomized words from a txt file.
 */

package stringsortinglab;

import java.util.*;
import java.io.*;

public class StringSortingLab {

  static String[] words = new String[45403];
  static int[] wordIndex = new int[45403];
  static int wordCount = 0;

  public static void main(String[] args) throws IOException {

    long start = 0;
    long end = 0;
    File file = new File("undictionary.txt");
    Scanner fileIn = new Scanner(file);

    // filling the words[] array with words from file:
    for (int i = 0; i < 45403; i++) {
      words[wordCount] = fileIn.nextLine();
      wordCount++;
    }

    // setting values in the wordIndex[] array:
    reInit();

    // creating Sorts object:
    SortingMethods sorts = new SortingMethods(wordIndex, words);

    start = System.currentTimeMillis();
```

```
sorts.bubbleSort();
end = System.currentTimeMillis();

System.out.println("Took " + (end - start) + " milliseconds to sort!");

// checking
if (sortVerifier()) {
  System.out.println("GOOD!");
}

System.out.println("END OF BUBBLE SORT.");
////////////////////////////////////////////////////////////////////////
// resetting values in the wordIndex[] array:
reInit();

start = System.currentTimeMillis();
sorts.selectionSort();
end = System.currentTimeMillis();

System.out.println("Took " + (end - start) + " milliseconds to sort!");

// checking
if (sortVerifier()) {
  System.out.println("GOOD!");
}

System.out.println("END OF SELECTION SORT.");
////////////////////////////////////////////////////////////////////////
// resetting values in the wordIndex[] array:
reInit();

start = System.currentTimeMillis();
sorts.insertionSort();
end = System.currentTimeMillis();

System.out.println("Took " + (end - start) + " milliseconds to sort!");

// checking
if (sortVerifier()) {
  System.out.println("GOOD!");
}

System.out.println("END OF INSERTION SORT.");
////////////////////////////////////////////////////////////////////////
// resetting values in the wordIndex[] array:
```

```
reInit();

start = System.nanoTime();
sorts.shellSort();
end = System.nanoTime();

System.out.println("Took " + (end - start) + " nanoseconds to sort!");

// checking
if (sortVerifier()) {
  System.out.println("GOOD!");
}

System.out.println("END OF SHELL SORT.");
////////////////////////////////////////////////////////////////////////
// resetting values in the wordIndex[] array:
reInit();

start = System.nanoTime();
sorts.heapSort();
end = System.nanoTime();

System.out.println("Took " + (end - start) + " nanoseconds to sort!");

// checking
if (sortVerifier()) {
  System.out.println("GOOD!");
}

System.out.println("END OF HEAP SORT.");

////////////////////////////////////////////////////////////////////////
// resetting values in the wordIndex[] array:
reInit();

start = System.nanoTime();
sorts.mergeSort(0, wordCount - 1);
end = System.nanoTime();

System.out.println("Took " + (end - start) + " nanoseconds to sort!");

// checking
if (sortVerifier()) {
  System.out.println("GOOD!");
}
```

```
    System.out.println("END OF MERGE SORT.");

    ////////////////////////////////////////////////////////////////////////
    // resetting values in the wordIndex[] array:
    reInit();

    start = System.nanoTime();
    sorts.quickSort(0, wordCount - 1);
    end = System.nanoTime();

    System.out.println("Took " + (end - start) + " nanoseconds to sort!");

    // checking
    if (sortVerifier()) {
      System.out.println("GOOD!");
    }

    System.out.println("END OF QUICK SORT.");

  } // end of main

  ////////////////////////////////////////////////////////////////////////
  // method for re-initializing the index array so that it is not sorted!
  public static void reInit() {
    for (int i = 0; i < wordCount; i++) {
      wordIndex[i] = i;
    }
  }

  // method for verifying the sort is in ascending order or not:
  public static boolean sortVerifier() {
    for (int i = 0; i < wordIndex.length - 1; i++) {

      // if the word before is "greater" than the word after:
      if (words[wordIndex[i]].compareTo(words[wordIndex[i + 1]]) > 0) {
        return false;
      }
    }
    return true;
  }

}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////


/*
This class contains methods for all the sorting algorithms.
 */
package stringsortinglab;

public class SortingMethods {

    int[] indexArr;
    String[] wordsArr;

    //constructor:
    public SortingMethods(int[] arrayOfIndexes, String[] arrayOfWords) {
        indexArr = arrayOfIndexes;
        wordsArr = arrayOfWords;
    }

 //sorting methods:
 ////////////////////////////////////////////////////////////////////////////
    //BUBBLE SORT:

    public void bubbleSort() {
        int n = indexArr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (wordsArr[indexArr[j]].compareTo(wordsArr[indexArr[j+1]]) > 0)
                {
                    //swap arr[j+1] and arr[j]
                    int temp = indexArr[j];
                    indexArr[j] = indexArr[j+1];
                    indexArr[j+1] = temp;
                }
    }
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
    //SELECTION SORT:

    public void selectionSort() {

        int n = indexArr.length;
```

```
        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            int min_idx = i;
            for (int j = i+1; j < n; j++) {
                if(wordsArr[indexArr[j]].compareTo(wordsArr[indexArr[min_idx]]) <
0) {
                    min_idx = j;
                }
            }
            // Swap the found minimum element with the first
            // element
            int temp = indexArr[min_idx];
            indexArr[min_idx] = indexArr[i];
            indexArr[i] = temp;
        }
    }
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
    //INSERTION SORT:

    public void insertionSort() {
        int n = indexArr.length;
        for (int i = 1; i < n; ++i) {
            int key = indexArr[i];
            int j = i - 1;

            /* Move elements of arr[0..i-1], that are
               greater than key, to one position ahead
               of their current position */
            while (j >= 0 && wordsArr[indexArr[j]].compareTo(wordsArr[key]) > 0)
{
                indexArr[j + 1] = indexArr[j];
                j = j - 1;
            }
            indexArr[j + 1] = key;
        }
    }
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
    //SHELL SORT:

    public void shellSort() {
```

```
        int n = indexArr.length;

        // Start with a big gap, then reduce the gap
        for (int gap = n/2; gap > 0; gap /= 2)
        {
            // Do a gapped insertion sort for this gap size.
            // The first gap elements a[0..gap-1] are already
            // in gapped order keep adding one more element
            // until the entire array is gap sorted
            for (int i = gap; i < n; i += 1)
            {
                // add a[i] to the elements that have been gap
                // sorted save a[i] in temp and make a hole at
                // position i
                int temp = indexArr[i];

                // shift earlier gap-sorted elements up until
                // the correct location for a[i] is found
                int j;
                for (j = i; j >= gap && wordsArr[indexArr[j -
gap]].compareTo(wordsArr[temp]) > 0; j -= gap) {
                    indexArr[j] = indexArr[j - gap];
                }
                // put temp (the original a[i]) in its correct
                // location
                indexArr[j] = temp;
            }
        }
    }
    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
    //HEAP SORT:

    public void heapSort()
    {
        int n = indexArr.length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(n, i);

        // One by one extract an element from heap
        for (int i=n-1; i>0; i--)
        {
            // Move current root to end
```

```
            int temp = indexArr[0];
            indexArr[0] = indexArr[i];
            indexArr[i] = temp;

            // call max heapify on the reduced heap
            heapify(i, 0);
        }
    }
 ////////////////////////////////////////////////////////////////////////////
    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    void heapify(int n, int i)
    {
        int largest = i; // Initialize largest as root
        int l = 2*i + 1; // left = 2*i + 1
        int r = 2*i + 2; // right = 2*i + 2

        // If left child is larger than root
        if (l < n && wordsArr[indexArr[l]].compareTo(wordsArr[indexArr[largest]])
> 0)
            largest = l;

        // If right child is larger than largest so far
        if (r < n && wordsArr[indexArr[r]].compareTo(wordsArr[indexArr[largest]])
> 0)
            largest = r;

        // If largest is not root
        if (largest != i)
        {
            int swap = indexArr[i];
            indexArr[i] = indexArr[largest];
            indexArr[largest] = swap;

            // Recursively heapify the affected sub-tree
            heapify(n, largest);
        }
    }


    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
    //MERGE SORT:

    public void merge(int l, int m, int r)
```

```
{
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int[n1];
    int R[] = new int[n2];

    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i)
        L[i] = indexArr[l + i];
    for (int j = 0; j < n2; ++j)
        R[j] = indexArr[m + 1 + j];

    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarry array
    int k = l;
    while (i < n1 && j < n2) {
        if (wordsArr[L[i]].compareTo(wordsArr[R[j]]) <= 0) {
            indexArr[k] = L[i];
            i++;
        }
        else {
            indexArr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy remaining elements of L[] if any */
    while (i < n1) {
        indexArr[k] = L[i];
        i++;
        k++;
    }

    /* Copy remaining elements of R[] if any */
    while (j < n2) {
        indexArr[k] = R[j];
        j++;
```

```
            k++;
        }
    }
////////////////////////////////////////////////////////////////////////////////
    // Main function that sorts arr[l..r] using
    // mergeSort()
    void mergeSort(int l, int r)
    {
        if (l < r) {
            // Find the middle point
            int m = (l + r) / 2;

            // Sort first and second halves
            mergeSort(l, m);
            mergeSort(m + 1, r);

            // Merge the sorted halves
            merge(l, m, r);
        }
    }

    ////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////
    //QUICK SORT:

     /* This function takes last element as pivot,
       places the pivot element at its correct
       position in sorted array, and places all
       smaller (smaller than pivot) to left of
       pivot and all greater elements to right
       of pivot */
    int partition(int low, int high)
    {
        int pivot = indexArr[high];
        int i = (low-1); // index of smaller element
        for (int j = low; j < high; j++)
        {
            // If current element is smaller than the pivot
            if (wordsArr[indexArr[j]].compareTo(wordsArr[pivot]) < 0)
            {
                i++;

                // swap arr[i] and arr[j]
                int temp = indexArr[i];
                indexArr[i] = indexArr[j];
```

```
                indexArr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = indexArr[i+1];
        indexArr[i+1] = indexArr[high];
        indexArr[high] = temp;

        return i+1;
    }

  ////////////////////////////////////////////////////////////////////////////
    /* The main function that implements QuickSort()
      low  --> Starting index,
      high  --> Ending index */
    void quickSort(int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is
              now at right place */
            int pi = partition(low, high);

            // Recursively sort elements before
            // partition and after partition
            quickSort(low, pi-1);
            quickSort(pi+1, high);
        }
    }
   ////////////////////////////////////////////////////////////////////////////
   ////////////////////////////////////////////////////////////////////////////


}
```