Data Structures and Algorithms
Lab 10: Sorting

I. Objective

After completing this tutorial, you can:

- Implement some basic sort algorithms, *i.e.*, selection sort, bubble sort, insertion sort, merge sort, quicksort;
- Use Java method to perform sorting.

II. Sorting Algorithms

1. Selection sort

If you were handed a list of names and asked to put them in alphabetical order, you might use this general approach:

- Find the name that comes first in the alphabet, and write it on a second sheet of paper,
- Cross the name out on the original list,
- Continue this cycle until all names on the original list have been crossed out and written onto the second list, at which point the second list is sorted.

Pseudo code of *selection sort* algorithm:

```
Selection Sort
Set current to the index of first item in the array
while more items in unsorted part of array
        Find the index of the smallest unsorted item
        Swap the current item with the smallest unsorted one
        Shrink the unsorted part of the array by incrementing current
```

Implementation of *selection sort* algorithm in Java:

```
public static void selectionSort(int[] arr) {
    int n = arr.length;

    // One by one move boundary of unsorted sub-array
    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 (arr[j] < arr[min_idx])
                min_idx = j;
        }

        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

2. Bubble sort

Like selection sort, bubble sort is also a brute-force approach to the sorting problem. The idea of bubble sort is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n-1$ passes the list is sorted. Pass i $(0 \le i \le n-2)$ of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

Pseudo code of *bubble sort* algorithm:

```
Bubble Sort
Set current to the index of first item in the array
while more items in unsorted part of array
        "Bubble up" the smallest item in the unsorted part,
                causing intermediate swaps as needed
        Shrink the unsorted part of the array by incrementing current
```

Implementation of *bubble sort* algorithm in Java:

```java
public static void bubbleSort(int[] arr)
{
   int n = arr.length;

   for (int i = 0; i < n - 1; i++)
   {
      for (int j = 0; j < n - i - 1; j++)
      {
         if (arr[j] > arr[j + 1])
         {
            // swap temp and arr[i]
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
         }
      }
   }
}
```

3. Insertion sort

The principle of the insertion sort is quite simple: Each successive element in the array to be sorted is inserted into its proper place with respect to the other, already-sorted elements. As with the previous sorts, we divide our array into a sorted part and an unsorted part.

Initially, the sorted portion contains only one element: the first element in the array. Now we take the second element in the array and put it into its correct place in the sorted part; that is, **values[0]** and **values[1]** are in order with respect to each other. Now the value in **values[2]** is put into its proper place, so **values[0] . . values[2]** are in order with respect to each other. This process continues until all the elements have been sorted. The following figure illustrates this process, which we describe in the following algorithm.

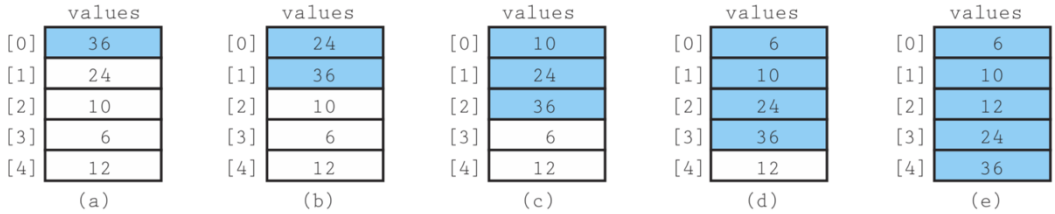

Figure 1 Example of the insertion sort algorithm

Pseudo code of *insertion sort* algorithm:

```
Insertion Sort
for count going from 0 through numValues – 1
        InsertItem(values, 0, count)

InsertItem(values, startIndex, endIndex)
Set finished to false
Set current to endIndex
Set moreToSearch to (current does not equal startIndex)
while moreToSearch AND NOT finished
    if values[current] < values[current - 1]
        Swap(values[current], values[current - 1])
        Decrement current
        Set moreToSearch to (current does not equal startIndex)
    else
        Set finished to true
```

Implementation of *insertion sort* algorithm in Java:

```
public static void insertionSort(int[] arr)
{
   int n = arr.length;

   for (int i = 1; i < n; i++)
   {
      int key = arr[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 && arr[j] > key)
      {
         arr[j+1] = arr[j];
         j = j-1;
      }
      arr[j+1] = key;
   }
}
```

4. Merge sort

The merge sort algorithm involves three steps:

- If the number of items to sort is 0 or 1, return.
- Recursively sort the first and second halves separately.
- Merge the two sorted halves into a sorted group.

The following figure illustrates a recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down).

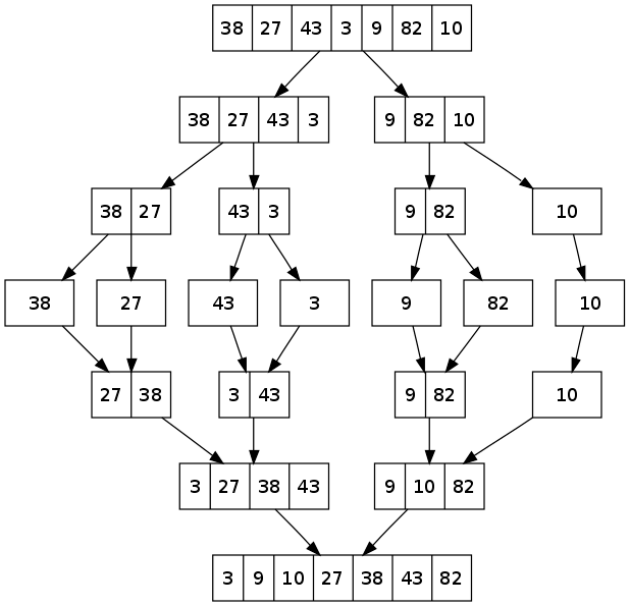

Figure 2 Merge sort - step by step guide

Implementation of *merge sort* algorithm in Java:

```
private static void merge(int arr[], int l, int m, int r)
{
   // Find sizes of two sub-arrays 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] = arr[l + i];
   for (int j = 0; j < n2; j++)
      R[j] = arr[m + 1+ j];


   /* Merge the temp arrays */

   // Initial indexes of first and second sub-arrays
   int i = 0, j = 0;

   // Initial index of merged sub-array
   int k = l;
   while (i < n1 && j < n2)
   {
      if (L[i] <= R[j])
      {
         arr[k] = L[i];
         i++;
      }
```

```
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

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

// Main function that sorts arr[first..last] using merge() method
public static void mergeSort(int[] arr, int first, int last)
{
    if (first < last)
    {
        // Find the middle point
        int middle = (first + last)/2;

        // Sort first and second halves
        mergeSort(arr, first, middle);
        mergeSort(arr, middle + 1, last);

        // Merge the sorted halves
        merge(arr, first, middle, last);
    }
}
```

5. Quicksort

Quicksort is a divide-and-conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

- Pick an element, called a pivot, from the array.
- *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

The following figure illustrates the quicksort algorithm. The shaded element is the *pivot*. It is always chosen as the last element of the partition. However, always choosing the last element in the partition as the pivot in this way results in poor performance, $O(n^2)$, on already sorted arrays, or arrays of identical elements.

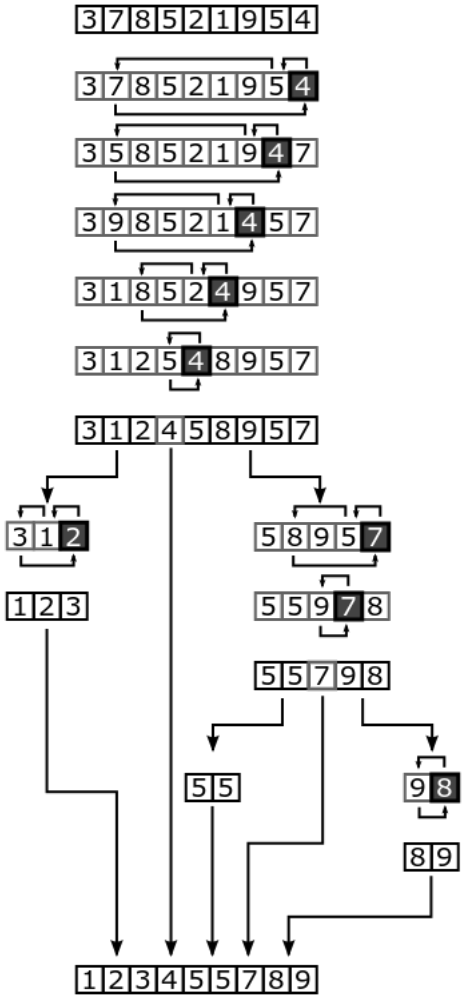

Figure 3 Quicksort - step by step guide

Implementation of *quicksort* algorithm in Java:

```
// 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
private static int partition(int[] arr, int low, int high)
{
   int pivot = arr[high];
```

```
    // index of smaller element
    int i = (low - 1);

    for (int j = low; j < high; j++)
    {
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++;

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

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

    return i+1;
}


// The main function that implements QuickSort algorithm
// arr[] --> Array to be sorted,
// low --> Starting index,
// high --> Ending index
public static void QuickSort(int[] arr, int low, int high)
{
    if (low < high)
    {
        // pi is partitioning index, arr[pi] is now at right place
        int pi = partition(arr, low, high);

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

III. Using Java method to perform sorting

In this section, we'll shows the use of ***java.util.Comparator*** to sort a Java object based on its property value.

The following example program will illustrate how-to use Comparator to sort an array of objects on different attributes. The example is organized as follows:

- Define ***Fraction*** class;
- Then, define ***FractionComparator*** class to implement the ***Comparator*** interface and it will be used to sort an array of ***Fraction***;
- Finally, define *Test.java* to test the program.

1. *Fraction* class

```
public class Fraction {

   private int num;
   private int denom;

   public Fraction()
   {
      this.num = 0;
      this.denom = 1;
   }

   public Fraction(int num, int denom)
   {
      this.num = num;
      this.denom = denom;
   }

   public double getRatio()
   {
      return (double) this.num / this.denom;
   }

   @Override
   public String toString()
   {
      return this.num + "/" + this.denom;
   }

}
```

2. FractionComparator class

```
import java.util.Comparator;

public class FractionComparator implements Comparator {

   @Override
   public int compare(Object o1, Object o2)
   {
      Fraction f1 = (Fraction) o1;
      Fraction f2 = (Fraction) o2;

      // for ascending order
      double ratio = f1.getRatio() - f2.getRatio();

      if(ratio > 0) return 1;
      if(ratio < 0) return -1;
      return 0;
   }
}
```

3. *Test.java* program

```
import java.util.Arrays;

public class Test {
```

```
    public static void print(Fraction[] arr) {
        for(Fraction f : arr) {
            System.out.print(f + "\t");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        Fraction[] fractions = new Fraction[5];
        fractions[0] = new Fraction(5, 6);
        fractions[1] = new Fraction(1, 2);
        fractions[2] = new Fraction(7, 3);
        fractions[3] = new Fraction(3, 5);
        fractions[4] = new Fraction(2, 3);

        print(fractions);

        Arrays.sort(fractions, new FractionComparator());

        print(fractions);
    }
}
```

IV. Exercises

1. Implement all sorting algorithms presented in this tutorial.
2. Applying the `java.util.Comparator` to sort a list of ***Student*** by the average grade in *ascending* and *descending*. The attributes of ***Student***:

- Student's name: **name** (`String`);
- Student's grade: **mathematics**, **programming**, **DSA1** (`double`).
- Student's average grade: $avg = \frac{1}{3}(mathematics + programming + DSA1)$.