Lab 4
Sorting

Quang D. C.
dungcamquang@tdtu.edu.vn

October 02, 2023

Note

In this tutorial, we will approach some basic sorting algorithms, *i.e.*, Selection sort, Bubble sort, Insertion sort, Merge sort, Quick sort; and use Java method to perform sorting.

Part I
Classwork

In this part, lecturer will:

- Summarize the theory related to this lab.
- Instruct the lesson in this lab to the students.
- Explain the sample implementations.

Responsibility of the students in this part:

- Students practice sample exercises with solutions.
- During these part, students may ask any question that they don't understand or make mistakes. Lecturers can guide students, or do general guidance for the whole class if the errors are common.

1. Sorting Algorithms

We will focus on implementing Selection sort, Bubble sort, Insertion sort and understand the main ideas of Merge sort and Quick sort.

1.1. Selection sort

The algorithm involves finding the minimum or maximum element in the unsorted portion of the array and then placing it in the correct position of the array. In this tutorial, we implement Selection sort by finding the minimum element in the unsorted portion of the array.

Given an array of n items

1. Find the index of the smallest unsorted item.
2. Swap the current item with the smallest unsorted one.
3. Exclude the smallest item which we found in step 2 and go to step 1.

This is Selection Sort pseudocode:

```
SelectionSort(int a[])
Input: array a[0], ..., a[n − 1]
Output: Sorted array
1 n ← a.length
2 for i ← 0, i < n − 1 do
      // Find the minimum element in unsorted array
3     min_idx = i
4     for j ← i + 1, j < n do
5         if a[j] < a[min_idx] then
6             min_idx ← j
      // Swap the found minimum element with the first element
7     temp ← a[min_idx]
8     a[min_idx] ← a[i]
9     a[i] ← temp
```

1.2. 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 $(0 \leq i \leq n-2)$ of bubble sort can be represented by the following diagram:

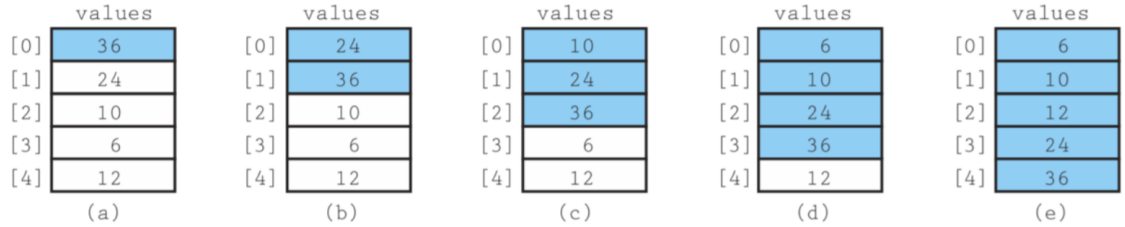
$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \leq \cdots \leq A_{n-1}}_{\text{in their final positions}}$$

This is Bubble Sort pseudocode:

```
BubbleSort(int a[])
Input: array a[0], ..., a[n − 1]
Output: Sorted array
1 n ← a.length
2 for i ← 0, i < n − 1 do
3     for j ← 0, j < n − i − 1 do
4         if a[j] > a[j + 1] then
              // Swap a[j] and a[j + 1]
5             temp ← a[j]
6             a[j] ← a[j + 1]
7             a[j + 1] ← temp
```

1.3. 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.



This is Insertion Sort pseudocode:

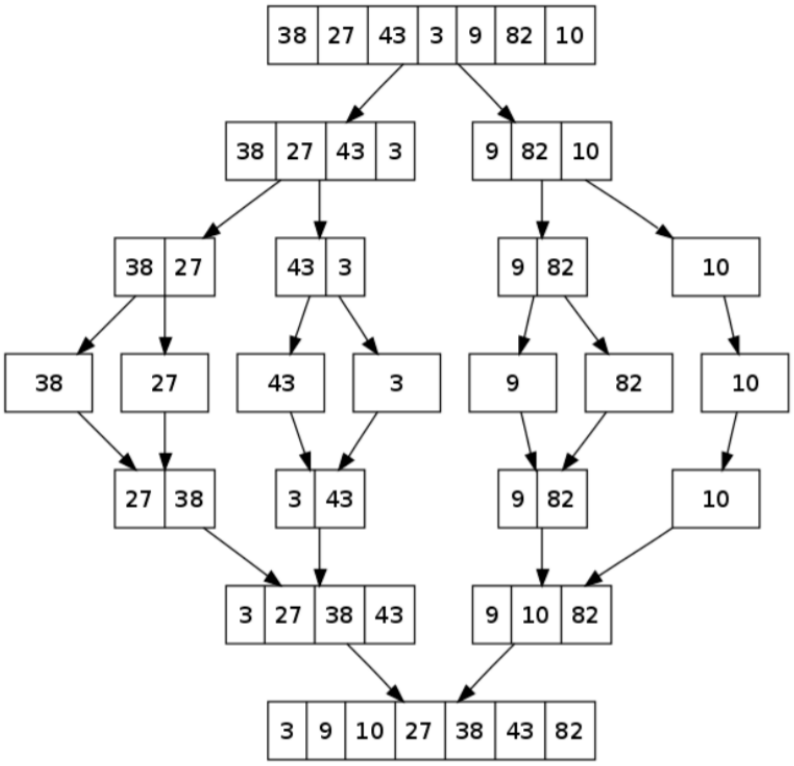
```
InsertionSort(int a[])
Input: array a[0], ..., a[n − 1]
Output: Sorted array
1 n ← a.length
2 for i ← 1, i < n do
3     key ← a[i]
4     j ← i − 1
5     while j ≥ 0 and a[j] > key do
6         a[j + 1] ← a[j]
7         j ← j − 1
8     a[j + 1] ← key
```

1.4. 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).



This is the code which implement 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];
```

```
29              j++;
30          }
31          k++;
32      }
33      // Copy remaining elements of L[] if any
34      while (i < n1)
35      {
36          arr[k] = L[i];
37          i++;
38          k++;
39      }
40      // Copy remaining elements of R[] if any
41      while (j < n2)
42      {
43          arr[k] = R[j];
44          j++;
45          k++;
46      }
47  }
48
49
50  // Main function of merge sort that sorts arr[first..last] using
      merge() method
51  public static void mergeSort(int[] arr, int first, int last)
52  {
53      if (first < last)
54      {
55          // Find the middle point
56          int middle = (first + last)/2;
57          // Sort first and second halves
58          mergeSort(arr, first, middle);
59          mergeSort(arr, middle + 1, last);
60          // Merge the sorted halves
61          merge(arr, first, middle, last);
62      }
63  }
64
```

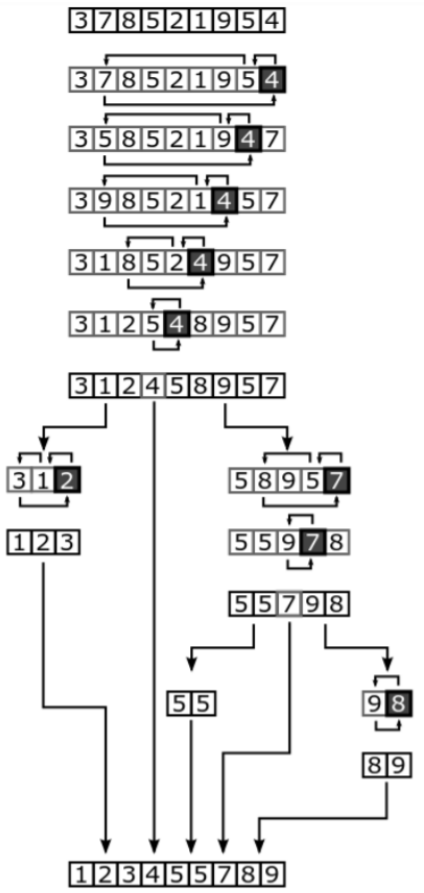
1.5. Quick sort

Quicksort is a divide-and-conquer algorithm. Quicksort first divides a large array into two smaller subarrays: 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.



Implementation of quick sort algorithm in Java:
Notice: there are many ways to pick the pick the pivot. In this example code, we will choice the low position element to make the pivot. This solution will following the algorithm in the theory class.

```
/* This function takes first 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[low];
    int i = low;
    for (int j = low + 1; j <= high; j++)
    {
        // If current element is smaller than 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] and arr[low] (or pivot)
    int temp = arr[i];
    arr[i] = arr[low];
    arr[low] = temp;
    return i;
}

// 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);
    }
}
```

2. 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.

2.1. 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;
    }
}
```

2.2. FractionComparator class

```
import java.util.Comparator;
public class FractionComparator implements Comparator<Fraction> {
    @Override
    public int compare(Fraction f1, Fraction f2)
    {
        // for ascending order
        double ratio = f1.getRatio() - f2.getRatio();

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

2.3. 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);
```

```
15          fractions[3] = new Fraction(3, 5);
16          fractions[4] = new Fraction(2, 3);
17
18          print(fractions);
19
20          Arrays.sort(fractions, new FractionComparator());
21
22          print(fractions);
23      }
24  }
```

Part II
Excercise

Responsibility of the students in this part:

- Complete all the exercises with the knowledge from **Part I**.
- Ask your lecturer if you have any question.
- Submit your solutions according to your lecturer requirement.

Exercise 1

Implement code of Selection sort, Bubble sort, Insertion sort from their pseudocode and re-implement Merge sort and Quick sort by your self.

Exercise 2

Applying the *java.util.Comparator* to sort a list of **Student** by the average grade in *ascending* and *descending* order. The attributes of **Student**:

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