Java Basics

Content By: Bairon J. Vasquez

Week 8 - Day 1

Agenda

- Students should be able to understand the following Sorting Algorithm
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

Bubble Sort

- Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.
- How does Bubble Sort works?
  - Assume that you would like to sort the following elements which are store into a collection
    - [5 1 4 2 8]

Bubble Sort

- **First Pass:**
  - (**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  - (1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
  - (1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
  - (1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
  - Since we are comparing for which element is larger, the first pass guarantees that the largest element would be in the right position
- **Second Pass:**
  - (**1 4** 2 5 8) –> (**1 4** 2 5 8)
  - (1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
  - (1 2 **4 5** 8) –> (1 2 **4 5** 8)
  - (1 2 4 **5 8**) –> (1 2 4 **5 8**)
  - Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Bubble Sort

- **Third Pass:**
  - (**1 2** 4 5 8) –> (**1 2** 4 5 8)
  - (1 **2 4** 5 8) –> (1 **2 4** 5 8)
  - (1 2 **4 5** 8) –> (1 2 **4 5** 8)
  - (1 2 4 **5 8**) –> (1 2 4 **5 8**)
- Let's Code it

Bubble Sort

- **Optimized Implementation:**
  - The above function always runs O(n^2) time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn't cause any swap.
- Let's Modified code

Selection Sort

- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.
- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.
- In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.
- How does Selection Sort works:
  - Assume that you have the following elements which are stored into a collection:
  - [64, 25, 12, 22, 11]

Selection Sort

- **First Pass**
  - [64, 25, 12, 22, 11] find the minimum number between arr[0...length - 1] and swap it with the first element
- **Second Pass**
  - [**11**, 25, 12, 22, 64] find the minimum number between arr[1...length - 1] and swap it with the first element
- **Third Pass**
  - [**11**, **12**, 25, 22, 64] find the minimum number between arr[2...length - 1] and swap it with the first element
- **Fourth Pass**
  - [**11**, **12**, **22**, **25**, 64] find the minimum number between arr[3...length - 1] and swap it with the first element
- **Fifth Pass**
  - [**11**, **12**, **22**, **25**, **64**] find the minimum number between arr[4...length - 1] and swap it with the first element
- Let's code it:

Insertion Sort

- The insertion sort, although still O(n2), works in a slightly different way. It always maintains a sorted sublist in the lower positions of the list. Each new item is then "inserted" back into the previous sublist such that the sorted sublist is one item larger.
- How does Insertion Sort works:
  - Assume that you have the following elements which are stored into a collection:
    - [3 1 4 8 2 6]

Insertion Sort

- **First Pass**
  - **3** 1<- 4 8 2 6 - Loop thru index 1 up to length - 1. Then, check that while you are greater or equal to 0 and the current index - 1 is greater than than current index swap elements
- **Second Pass**
  - **1 3** 4<- 8 2 6 - Loop thru index 2 up to length - 1. Then, check that while you are greater or equal to 0 and the current index - 1 is greater than than current index swap elements
- **Third Pass**
  - **1 3 4** 8<- 2 6 - Loop thru index 3 up to length - 1. Then, check that while you are greater or equal to 0 and the current index - 1 is greater than than current index swap elements
- **Four Pass**
  - **1 3 4 8** 2<- 6 - Loop thru index 4 up to length - 1. Then, check that while you are greater or equal to 0 and the current index - 1 is greater than than current index swap elements

Insertion Sort

- **Fifth Pass**
  - **1 2 3 4 8** 6<- - Loop thru index 5 up to length - 1. Then, check that while you are greater or equal to 0 and the current index - 1 is greater than than current index swap elements
  - **1 2 3 4 6 8**
- Let's code it

Questions?

End of Section