Lesson:

Sorting in an array

Pre Requisites:

- Basic Java syntax

List of concepts involved :

- Sorting in an array
- Bubble Sort
- Insertion Sort
- Selection Sort

What is Sorting in an array?

Sorting in an array means to arrange the elements of the array in a certain order. The order can be ascending order or descending order. There are various sorting algorithms used in Java.

Sorting can be both Internal/Inplace and External/Outplace Sorting.

Internal Sorting can be defined as sorting which takes in place. It means all the data which needs to be sorted is stored in the same place while sorting is in progress.

External sorting: If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device, then we need to do sorting outside the memory. This means for external sorting we require additional memory or space to store the data which is sorted.

There are various ways to judge a sorting algorithm. We would consider 2 important things
1. Time complexity
2. Space Complexity
3. Stability of sorting.

An algorithm is said to be stable if for two equal keys/elements, the order before and after sorting remains the same. In simple terms, if 2 elements are equal they should be in the same order before and after sort.

Now we will study various sorting algorithms.

BUBBLE SORT

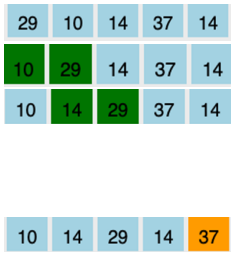
Bubble sort is the simplest amongst all sorting algorithms. The bubble sort works on swapping adjacent elements if they are in the wrong order and keeps on repeating this process till the list is sorted.

We would consider a simple array which becomes sorted using bubble sort.

Working of Bubble Sort
Suppose we are trying to sort the elements in ascending order.
eg . array:

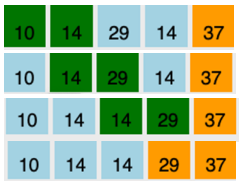
First Iteration (Compare and Swap)

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.



Elements swapped
Elements swapped
Nothing happened.
Elements swapped
we found our largest element of the array at the end.

2. Remaining iterations, we will keep on repeating this procedure until the array is sorted.



Nothing happened
Nothing happened
Elements swapped
29 is now at its correct position

3. We will keep repeating this procedure.

Nothing swapped
Nothing Swapped
Array is sorted.

Program to write bubble sort.

[LP_CODE1.java]

Note: Optimisation Technique.
If there is no swapping taking place in the current turn, then we can break the for loop as the array is already sorted. We don't need to go for indexes ahead. We can use a flag for this as mentioned in code.

Output

```
Enter the size of array
5
Enter the elements
4 2 3 1 5
Array after sorting
1 2 3 4 5
```

Time Complexity:
The time complexity of the following algorithm will be $O(n^2)$ as at each pass we are traversing the array and swapping adjacent elements whenever needed. Each traversal is $O(n)$ and there are n traversals in total so $O(n * n) = O(n^2)$ in worst case time complexity.

Space Complexity:
It does not require any extra space so space complexity is $O(1)$. Thus it will be an in-place sorting algorithm.

Is Bubble Sort Stable?
Yes, Bubble Sort is a stable sorting algorithm. We swap elements only when A is less than B. If A is equal to B, we do not swap them, hence relative order between equal elements will be maintained.

Insertion Sort
Let's suppose you have an array of integer values and you want to sort these values in non-decreasing order. Then the logic of Insertion sort states that you need to insert the element into the correct position by looping over the array, thereby forming a sorted array on the left.
Then increase the pointer and similarly for the next element find its correct position in the left sorted array and insert it. Do this till the pointer reaches the end of the array.

Steps involved in Insertion Sort
- Traverse the array from left to right
- Compare the current element to its previous element and keep swapping it until the previous element is smaller than the current element.
- This way each element reaches it correct position

Note: For inserting the element in the left sorted array, we can also use binary search also to find the best place. Though overall complexity will remain the same for the code. For this we need to track the min and max element in the left portion and then we can use binary search to find the correct index.

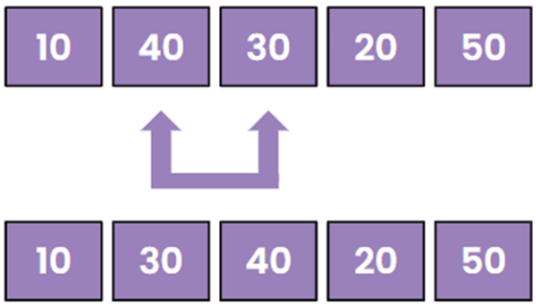
Let's understand insertion sort with an example.

Example (with all the steps):
Array = [10, 40, 30, 20, 50]

In the above example, as we iterate through the array from left to right, we keep inserting elements into their corresponding correct positions thereby forming a sorted array on the left.
First Pass: 10 is to be compared with its predecessor, here there is no previous element so simply move to the next element.
The sorted part of the array on the left is currently 10.
[10, 40, 30, 20, 50]

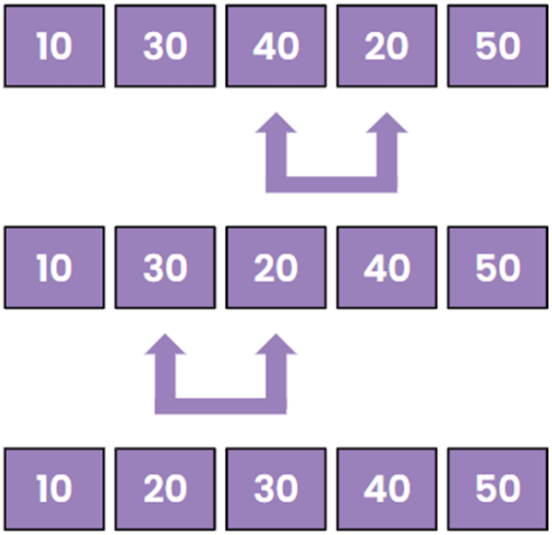
Second Pass: 40 is compared with its predecessor and the previous element is already smaller than the current element (10 < 40) so no need to swap anything and move to the next element.
The sorted part of the array on the left is currently [10,40].
[10, 40, 30, 20, 50]

Third Pass: 30 is now compared with its predecessor and here it is 40. As 40 < 30, we need to keep swapping the current element with its predecessors until the previous element is smaller than the current element. This way 30 will reach its correct position.
[10, 40, 30, 20, 50] (arrow between 40 and 30)
[10, 30, 40, 20, 50]



The sorted part of the array on the left is currently [10,30,40].

Fourth Pass: 20 is now compared with its predecessor that is 40 so as it is smaller than 40. We keep swapping it, it now reaches and array looks after swapping will be [10,30,20,40]. Again 20 is checked with its predecessor and as it is smaller than 30 also, it will be swapped again and the array will look like [10,20,30,40]. Now 20 has reached its correct position and no more swaps are needed.
[10, 30, 40, 20, 50] (arrow between 20 and 40)
[10, 30, 20, 40, 50] (arrow between 30 and 20)
[10, 20, 30, 40, 50]



Fifth Pass: Here the current element is 50 and is compared with its predecessor and no swaps are needed here. So the final array looks like this - [10,20,30,40,50]
[10,20,30,40,50]

Program for Insertion sort is

LP_CODE2.java

```
Enter the size of array
5
Enter the elements
10 40 30 20 50
Array after sorting
10 20 30 40 50
```

Time Complexity:
The time complexity of the following algorithm will be $O(n^2)$ as for every particular element we find its correct position and insert it at that particular position. This requires $O(n)$ time complexity for one element. We are doing this for every element so $O(n^2)$ time complexity in total

Space Complexity:
It does not require any extra space so space complexity is $O(1)$.

Is insertion sort stable?
Here we just pick an element and place it in its correct place and in the logic we are only swapping the elements if the element is larger than the key, i.e. we are not swapping the element with the key when it holds equality condition, so insertion sort is stable sort.

Selection Sort
This algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

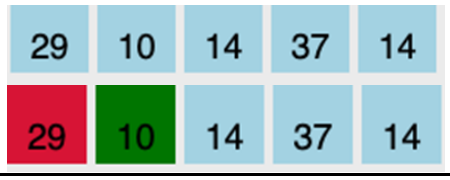
- The subarray which is already sorted.
- The remaining subarray was unsorted

Let's understand the selection sort for sorting the array in non decreasing order.

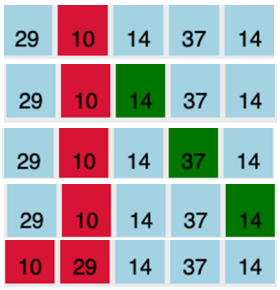
As the name suggests, in the Selection Sort algorithm we select the index having minimum value by traversing through the array and swap it with the current index, then increment the current index.
This way we keep on traversing the rest of the array and finding the minimum index and swap it with the current index. This will result in forming a sorted array on the left side.

- Iterate through the array and consider the current element index.
- Now traverse the rest of the array elements and find the minimum element index.
- Swap it with the current element index and increment the current element index by one.
- Repeat until we reach the end of the array.

Now, let's take an example to understand the working of selection sort with sorting the array in increasing order.



The current index is 0. We start from next index and find minimum element as compared to 29 if any.



Now 10 is the minimum element.
We start moving ahead to find element smaller than 10

No Element is smaller than 10, so swapping current index and index 1 (Index of 10)

Second Pass

Now the current index is 1 and the value is 29. We will find elements smaller than 29 if any.

14 is smaller than 29.

As 14 is the smallest element we swap current index and 2nd index (index of 14)

Third Pass

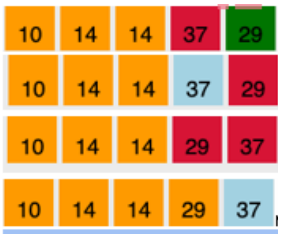
Now the current index is 2 and the value is 29, we will find values smaller than 29 if any

14 is the smallest value and the index is 4th

We swap current index 2 and index 4(Index of 14)

Fourth Pass

Now the current index is 3 and the value is 37, we will find elements smaller than 37 is any.



29 is the smallest element and the index is 4

We swap current index and index 4 (index of 29).

Now the current index is last, so our algorithm stops and we got the sorted array.

The array is now sorted in non decreasing order.

Program for Selection Sort.

LP_CODE3.java

```
Enter the size of array
5
Enter the elements
10 40 30 50 20
Array after sorting
10 20 30 40 50
```

Time Complexity:
The time complexity of the following algorithm will be $O(n^2)$ as at each particular index from left to right as we are iterating in an array, we are finding the minimum value index by traversing through the rest of the array. Thus we require $O(n)$ to find the minimum value index and we are doing this process n time so final time complexity will be $O(n^2)$.

Space Complexity:
It does not require any extra space so space complexity is $O(1)$. Thus it will be an in-place sorting algorithm.

Is selection sort stable?
As we know(Explained above) A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appeared in the input unsorted array.

Selection sort works by finding the minimum element in the array and then inserting it in its correct position by swapping with the element which has minimum value. This is what makes it unstable.

Let's see an example :
arr[] ={2,3,2,1}
In the first iteration of the outer for-loop, the algorithm determines that "1"is the minimal element and exchanges it with the blue "2" :
arr[] ={1,3,2,2}
Then, it finds that the red 2 is the minimal item in the rest of the array and swaps it with "3":
arr[] ={1,2,3,2}
Finally, since 2 < 3 Selection Sort swaps the blue Deer with Monkey:
arr[] ={1,2,2,3}

As you can see, arr doesn't maintain the relative order of the two numbers. Since they are equal, the one that was initially before the other should come first in the output array. But, Selection Sort places the red 2 before the blue one even though their initial relative order was opposite.

So, we can conclude that **Selection Sort isn't stable.**

Next class Teaser

- Number theory
- Bits data structure
- Operators on bits