# Lab 2: Simple sorting

## 2.1. Objectives

i. Know how, in reality, three simple sorting methods work.
ii. Know how to use analysis tool to compare performance of sorting algorithms

## 2.2. Problem 1: BubbleSortApp.java

- Trace the algorithm (display the array inside after inner or outer loop)
- Display the number of swaps after the inner loop
- Display the number of comparisons after the inner loop and the total number of comparisons, and estimate the algorithms' complexity (n*(n-1)/2, O(n^2))

## 2.3. Problem 2: SelectSortApp.java

- Trace the algorithm (display the array after the inner loop)
- Print the items that are swapped. Are swaps always needed?
- Display the number of comparisons after the inner loop and the total number of comparisons, and estimate the algorithms' complexity (n*(n-1)/2, O(n^2))

## 2.4. Problem 3: InsertSortApp.java

- Trace the algorithm (display the array after each pass of the outer loop)
- Display the number of passes of the inner loop and total number of passes, and estimate the algorithms' complexity (n*(n-1)/4, O(n^2))

## 2.5. Problem 4

Create an array of integer numbers, fill the array with random data and print the number of **comparisons, copies, and swaps** made for sorting 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000 and 50000 items and fill in the table below. Analyze the trend for the three different algorithms.

| **COPIES/ COMPARISONS/ SWAPS** | | | |
|---|---|---|---|
| | **Bubble Sort** | **Selection Sort** | **Insertion Sort** |
| **10000** | | | |
| **15000** | | | |
| **20000** | | | |
| **25000** | | | |
| **30000** | | | |
| **35000** | | | |
| **40000** | | | |
| **45000** | | | |
| **50000** | | | |

## 2.6. Problem 5: ObjectSortApp.java (sort the array by first name or by age)

**(Option 1) Given the class Student.java** that has variables of first name, last name, grade

- Add a main() method and add create an array of 10 students

- Add methods to sort the array by first name, last name, and by grade.

## 2.7. Problem 6: Airport Runway Scheduling

You are working as a software engineer at an airport, where you need to manage incoming and outgoing flight schedules. Each flight is assigned a runway for landing or takeoff, but no two flights can use the same runway at the same time. Your task is to write a program that efficiently schedules the flights on multiple runways based on their priority and time. Explain ***data structures, algorithms, and time complexity*** in your program.

### Problem Description:

Given a list of flights, where each flight has:

- A unique flight ID.
- A scheduled time (for either takeoff or landing) in 24-hour format.
- A priority (higher priority flights should be scheduled first in case of a time conflict).

You are given `R` runways and need to sort the flights based on the following criteria:

1. Higher-priority flights take precedence over lower-priority flights.
2. If two flights have the same priority, schedule them based on their time (earlier flights go first).
3. You need to assign flights to the available runways in such a way that no two flights are scheduled for the same runway at the same time.

### Input:

- A list of flights, where each flight is a tuple of the form `(flight_id, time, priority)`.
- The number of available runways `R`.

### Output:

- A list of assigned flights where each flight is assigned to a specific runway, sorted by priority and time.

### Constraints:

- If a flight cannot be assigned to a runway because all are occupied at that time, output an error or indicate that the flight cannot be scheduled

**Example:**

Input:
Flights:
`[("F1", "10:00", 2), ("F2", "09:30", 1), ("F3", "09:30", 2), ("F4", "11:00", 1)]`

Number of runways: `2`

Output:
Runway 1: `[("F2", "09:30"), ("F1", "10:00"), ("F4", "11:00")]`
Runway 2: `[("F3", "09:30")]`