Practical 4: Elementary Sorting

What am I doing today?

Today's practical focuses on 3 things:

1. Writing several elementary sorting algorithms
2. Developing a testing framework to assess the performance of your algorithms
3. Summarizing the results

Instructions

Try all the questions. Ask for help from the demonstrators if you get stuck.

***Grading: Remember** if you complete the practical, add the code to your GitHub repo which needs to be submitted at the end of the course **for an extra 5%**

Quick Questions

1. **How many compares does insertion sort make on an input array that is *already sorted*?**

Constant	
Logarithmic	
Linear	x
Quadratic	

2. **What is a stable sorting algorithm?**

   A stable sorting algorithm is when two objects with equal keys appear in the same order in the sorted array as they appeared in the unsorted array.

3. **What is an external sorting algorithm?**

A. Algorithm that uses tape or disk during the sort

B. Algorithm that uses main memory during the sort

C. Algorithm that involves swapping

D. Algorithm that are considered ‘in place’

4. **Identify 6 ways of classifying sorting algorithms?**

1.	Internal or external
2.	Time complexity
3.	Recursive or non-recursive
4.	Memory efficiency
5.	Stable or unstable
6.	Comparison or non-comparison

Algorithmic Development

Today your mission is to develop a Java class that implements several elementary (and silly) sorting algorithms. The problem we want our algorithms to solve is sort an input array of integers into ascending order and output the resulting array.

Possible steps to follow

1. Create a new java class
2. Implement the following sorting algorithms as public static functions within your class that take an array of integers and sorts the array, outputting a sorted array of integers:
    a. Selection sort
    b. Insertion Sort
    c. A silly sort (either from the list below or of your own making)
3. Create a simple framework for generating input arrays of various sizes (e.g., 10, 1000, 100,000) and then testing the performance over several runs
4. Print the resulting sorted array: Implement a function to print out all elements in the array
5. Time the performance of the previous step on your 3 algorithms and output the execution times for various input sizes (e.g. 10,100,1000) on a graph
6. Justify the results of your experiments for the algorithms by proposing the algorithm complexity in big-O notation
7. BONUS: adjust your insertion sort algorithm to be an unstable sort

Java Timer Code Options

Use System.currentTimeMillis()from the previous lab or implement nanoTime if you would like more precision.

public static long nanoTime()

Returns the current value of the most precise available system timer, in nanoseconds.

This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary *origin* time (perhaps in the future, so values may be negative). This method provides nanosecond precision, but not necessarily nanosecond accuracy. No guarantees are made about how frequently values change. Differences in successive calls that span greater than approximately 292 years (2^{63} nanoseconds) will not accurately compute elapsed time due to numerical overflow.

For example, to measure how long some code takes to execute:

long startTime = System.nanoTime(); // ... the code being measured ... long estimatedTime = System.nanoTime() - startTime;

Sorting Algorithms PseudoCode

Selection Sort
Steps

1. Find the smallest card. Swap it with the first card.
2. Find the second-smallest card. Swap it with the second card.
3. Find the third-smallest card. Swap it with the third card.
4. Repeat finding the next-smallest card, and swapping it into the correct position until the array is sorted.

PseudoCode

```
function sort (int arr[]){
    int temp;
    int mind_index;

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

InsertionSort
Steps

1. The first step involves the comparison of the element in question with its adjacent element.
2. And if at every comparison reveals that the element in question can be inserted at a particular position, then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position.
3. The above procedure is repeated until all the elements in the array are in their correct position.

PseudoCode

```
    Function insertSort (int arr[]){

       for (int i = 1; i < n.length; ++i) {
           key = arr[i];
           j = i -1;

           while j >= 0 and a[j] > key{
           a[j+1] = a[j]
           j = j-1
           a[j+1] = key
       }
     }
   }
```

Some Silly Algorithms to pick from

For fun or Computer Science comedic fun, implement one of these obscure sorting algorithms and run it through the sequence of steps above:

- **BogoSort: https://en.wikipedia.org/wiki/Bogosort**
  The stupidest sorting algorithm ever created?
- **Stalin Sort: https://www.quora.com/What-is-Stalin-sort**
  The Stalin sort is a joke sort in which elements that are out of order get removed from a list.
- **Slow Sort**: https://en.wikipedia.org/wiki/Slowsort
  "Slow sort is a sorting algorithm. It is of humorous nature and not useful"

***Alternatively, develop your own stupid sorting algorithm**