Python programming
Lab. Work n° 3 : Basic sorting

Preliminaries : Please remind that teachers can be called to help you on any problem you get. Don't get stuck on an issue for too long.

Exercise n°1 : Bubbles

1. Create a script named `bubble.py` and define a function named `random_array(n)` that when called creates a list of n random integers in the range $[0, 3*n[$. Call that function and print the results :

```
Size? 10
[28, 29, 20, 11, 1, 21, 8, 27, 26, 18]
```

2. Modify `bubble.py` to add a function named `bubble_sort(a)` that when called sorts (by ascending order) the elements of the array a with the bubble sorting algorithm.

A **pass** of the bubble sort consists in the scanning of all adjacent elements and their swapping in the case they are not correctly ordered. The **bubble sort** then consists in runing as many passes as necessary (what could be the stopping condition ?). Test it :

```
Size? 10
Before sorting: [9, 19, 15, 6, 20, 18, 5, 11, 26, 28]
After sorting:  [5, 6, 9, 11, 15, 18, 19, 20, 26, 28]
```

3. Add some measure into the function `bubble_sort` so that in return you will get : the total number of swaps and the total number of comparisons :

```
Size? 10
Before sorting:  [25, 2, 20, 4, 6, 1, 28, 30, 29, 2]
After sorting:  [1, 2, 2, 4, 6, 20, 25, 28, 29, 30]
72 comparisons and 19 swaps
```

4. Modify the script so that for a given size, e sorting experiences are generated and the mean of the results is computed :

```
Size? 10
Number of experiments? 1000
66.726 comparisons and 21.642 swaps in the mean for array of size 10.
```

5. One can remark that at pass i the i-th last element is at its final location. So, we can optimise the algorithm such that the i ending comparisons can be eliminated. Implement it in a function `bubble_sort_optimised(a)` and compare the results :

```
Size? 10
Number of experiments? 1000
Basic bubble sorting:     66.4812 comps and 21.7221 swaps for arrays of size 10.
Optimised bubble sorting: 41.6934 comps and 21.7221 swaps for arrays of size 10.
```

6. Another optimisation is possible if one can remark that more than the i-th last elements can be a their right location at pass i... At a given pass i, we know that all elements after the last swap are at their final location... Implement it in a function named `bubble_sort_super_optimised(a)` and compare the results :

```
Size? 15
Number of experiments? 20000
Basic bubble sorting:     161.4655 comps and 51.28295 swaps for arrays of size ▷
   ▷ 15.
Optimised bubble sorting: 98.71885 comps and 51.28295 swaps for arrays of size ▷
   ▷ 15.
Super opt bubble sorting: 92.47475 comps and 51.28295 swaps for arrays of size ▷
   ▷ 15.
```

7. **(Hard)** Draw functions for comparisons and swaps from different sizes (10, 20, 50, 100, 200, 500, 1000, 2000, 5000)... *Warning :* don't use too many experiences for long arrays, it may take too much time and may convert your computer to a radiator.. Hint : use `gnuplot` tool or `mathplotlib` Python module.

Exercise n°2 : Cocktail

1. Leave all the three bubble algorithms in the module `bubble.py` and create a `bubble_main.py` that contains the code that makes the experiments `bubble.py`
2. Bubble sort has a major drawback, if highest values moved quickly to their final location, it is the converse for low values (why ?). The cocktail sort is just alternation of left-to-right then right-to-left bubble passes. Implement the basic cocktail sort (in a module `cocktail.py`) and compare results with bubble sort :

```
Size? 10
Number of experiments? 10000
Basic bubble sorting:     66.4632 comps and 21.7236 swaps for arrays of size 10.
Optimised bubble sorting: 41.6588 comps and 21.7236 swaps for arrays of size 10.
Super opt bubble sorting: 38.6336 comps and 21.7236 swaps for arrays of size 10.
Cocktail sorting:         49.9713 comps and 21.7234 swaps for arrays of size 10.
```

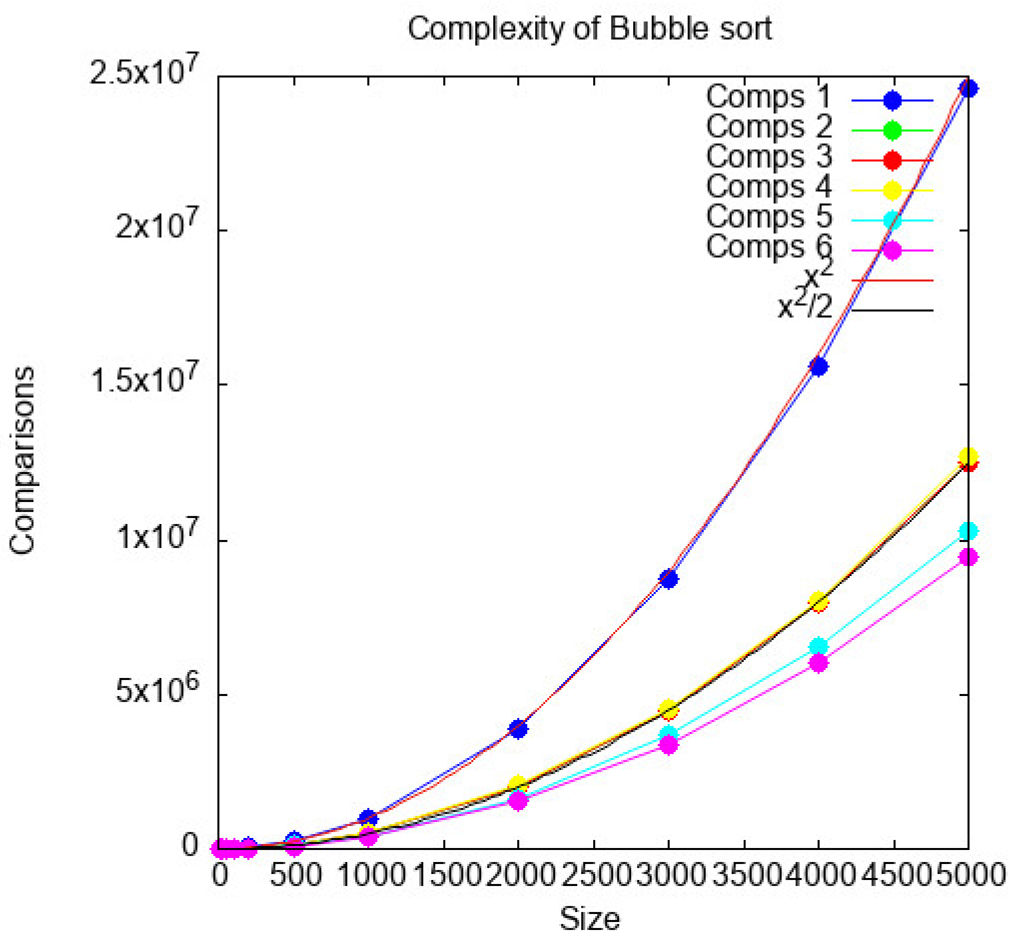
3. Implements the `cocktail_sort_optimised` and `cocktail_sort_super_optimised`...

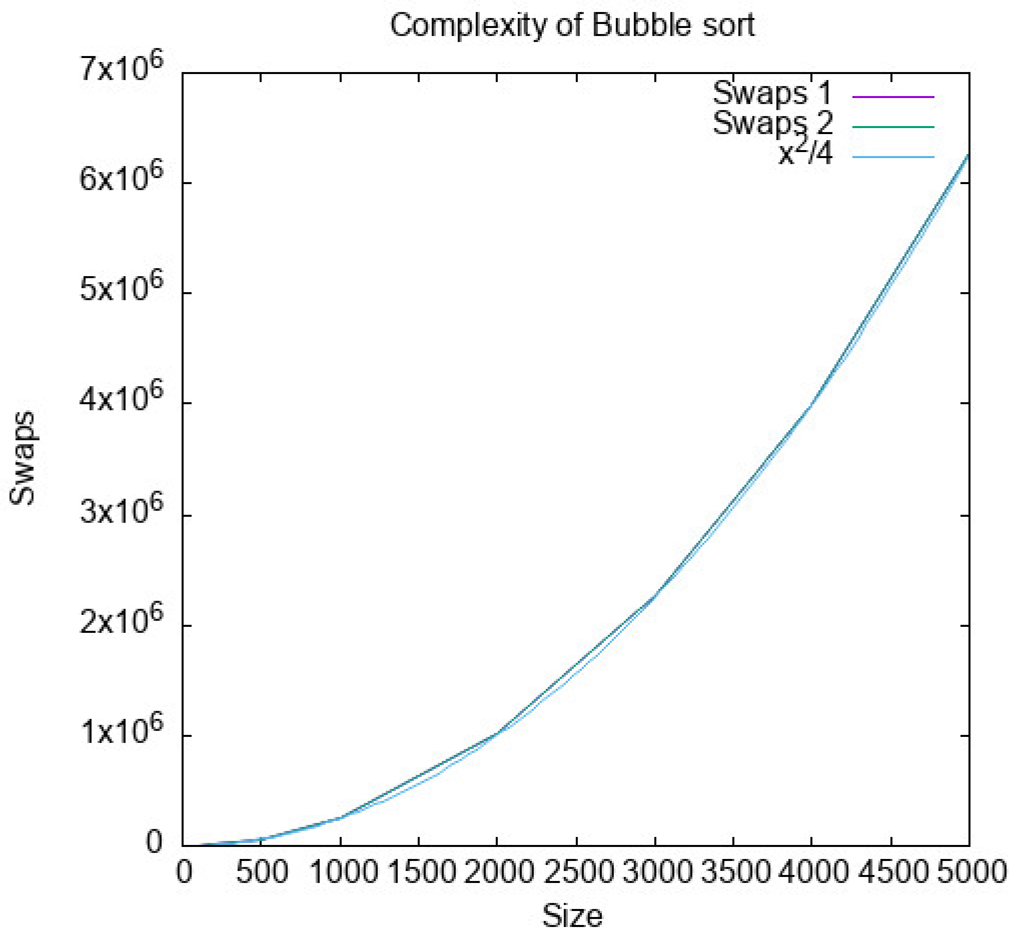
```
Size? 15
Number of experiments? 10000
Basic bubble sorting:     161.476 comps and 51.4261 swaps for arrays of size 15.
Optimised bubble sorting: 98.7368 comps and 51.4261 swaps for arrays of size 15.
Super opt bubble sorting: 92.4789 comps and 51.4261 swaps for arrays of size 15.
Cocktail sorting:         125.8572 comps and 51.4261 swaps for arrays of size 15.
Opt cocktail sorting:     97.9426 comps and 51.4261 swaps for arrays of size 15.
Super opt cocktail sort:  89.6867 comps and 51.4261 swaps for arrays of size 15.
```

4. (Hard) Draw all the functions and compare them to n^2.
   Hint : use `gnuplot` tool or `mathplotlib` Python module.

```
Number of experiments? 10
10, 15, 20, 50, 100, 200, 500, 1000, 2000, 5000,
Bubble: 62.1 comps and 19.1 swaps (size 10)
Bubble: 163.8 comps and 51.3 swaps (size 15)
```

```
Bubble: 304.0 comps and 97.1 swaps (size 20)
Bubble: 2077.6 comps and 590.0 swaps (size 50)
Bubble: 8910.0 comps and 2548.1 swaps (size 100)
Bubble: 37392.1 comps and 9775.5 swaps (size 200)
Bubble: 239320.4 comps and 62012.2 swaps (size 500)
Bubble: 957241.8 comps and 251157.7 swaps (size 1000)
Bubble: 3877460.3 comps and 1003253.9 swaps (size 2000)
Bubble: 24556087.8 comps and 6245843.2 swaps (size 5000)
Opt bubble: 40.7 comps and 19.1 swaps (size 10)
Opt bubble: 100.4 comps and 51.3 swaps (size 15)
...
Opt cocktail: 414824.4 comps and 251157.7 swaps (size 1000)
Opt cocktail: 1640822.7 comps and 1003253.9 swaps (size 2000)
Opt cocktail: 10175999.6 comps and 6245843.2 swaps (size 5000)
Super cocktail: 37.0 comps and 19.1 swaps (size 10)
Super cocktail: 88.4 comps and 51.3 swaps (size 15)
Super cocktail: 162.8 comps and 97.1 swaps (size 20)
Super cocktail: 967.0 comps and 590.0 swaps (size 50)
Super cocktail: 4034.6 comps and 2548.1 swaps (size 100)
Super cocktail: 15062.3 comps and 9775.5 swaps (size 200)
Super cocktail: 94434.9 comps and 62012.2 swaps (size 500)
Super cocktail: 380100.9 comps and 251157.7 swaps (size 1000)
Super cocktail: 1510832.3 comps and 1003253.9 swaps (size 2000)
Super cocktail: 9387841.7 comps and 6245843.2 swaps (size 5000)
```





5. How are you convinced that your sorting algorithms are correctly implemented ? Add some verification code at appropriate steps.