Lab 02 – CSCI 112

Due Date: Wednesday, Sep 9 at 5:00pm EST

Information

- This lab is intended to be completed **individually.**
- The files must be submitted with the exact file name provided in this file. If the file names do not match you will receive **zero** points for that file.
- Before you submit, make sure that your code runs. Any code which does not run without errors will receive **zero** points.
- Do not share your work with anyone other than Professor Khan. You may discuss algorithms, approaches, ideas, but **NOT** exact code.
- No late work is accepted. Make sure you are aware of the due date. A second past the due date **WILL** be locked out from submission.

Review

Lambda: To pass functions in python as arguments is very easy. If no modifications are necessary, the identifier of the function can be passed in the function just like any other variable. If modifications are necessary, lambda can be of help. For example, to pass a function which, given one argument, creates a list of the numbers **0** to **n-1** would be:

```
lambda x: list(range(x))
```

Which then can be passed to a function or stored in a variable:

```
func1(10, 20, lambda x: list(range(x)))
func2 = lambda x: list(range(x))
```

Nested Functions: In python, you can define a function within another function. This is called a **nested function**, just like nested loops or if-statements. This nested function has visibility to the parameters passed in to the outer function. For example:

```
def f1(var1):
    var2 = 2
    def f2(var3):
        return var1 + var2 + var3
```

Importing: To import modules that are inside the same directory, simply import the filename (missing the .py) or use from (filename) import (specific identifier(s)). For example:

```
import time
from sorts import quickSort
```

Assignment

Task 1 – sorts.py [8 points]

Examine the functions **selectionSort**, **bubbleSort**, **quickSort** and **mergeSort**, discussed in lecture and in the book. Using the function headers provided in **sorts.py**, convert the algorithms from the book into functions which use nested functions (you should be intuitive in your design of what you decide to create a separate function of). **You should use the built-in python lists and their functionality for now instead of the Array class.** Add a `return numbers` statement to the end of each function for the functions to work with `show` from **tools.py**. Test to verify that they work correctly with `show`. You should modify the bottom of **sorts.py** to use the `show` function instead of `main`. Hint: Get the functions working as they are in the book first, then work to make them with nested functions

Task 2 - Counters [3 points]

It is important to know how much **work** is being performed. Work can be defined as a few different aspects: how many **comparisons**, how many variable **increments** or **changes**, how many **function calls**, etc. This part of the lab requires you to add counter parameters to everything created above. These parameters are initially set to `None`. If the parameter is **not None**, then it will be a dictionary of string to int pairs, with each string being a descriptor of what that counter will measure. Add `counter=None` to the parameter list of all your sorting functions.

For sorting, the important runtime analysis depends on how many **comparisons** and **swapping** are performed. A possible counter argument to be passed to **compare** could be:

```
compare(["Selection", "Bubble"],
            [selection, bubble],
            [10,20,30],
            dataSet=getRandomList
            counter={"swaps" : 0, "comparisons" : 0})
```

Now use the counter argument in your sort functions. Increment the appropriate string-integer value in the dictionary anywhere a **swap** of data occurs or a **comparison** occurs. The best location for incrementation of the counter is the line directly **before** the line of code which is being counted. An example of how to increment either:

```
if counter:
      counter["comparisons"] += 1
```

```
if a < b:
      if counter:
            counter["swaps"] += 1
      a,b = b,a
```

The **`if counter`** statement is necessary because it is possible that the counter parameter is set to **`None`**. Note that while the counter is called "swaps", it should record any change of data. Finally, run the **`main()`** function in **`sorts.py`** for the analysis step.

Task 3 - Analysis **[4 points]**

Examine the runtimes, comparisons, and swaps that each sorting method does. In **analysis.txt**, describe the best, worst, and average cases for all sorting algorithms, noting differences in swap and comparisons. Note any unique situations or setbacks for each if the list is already sorted. Be precise, using big-O notation to characterize the run-time behavior.

Finally, for each of the following situations, which sorting method would you recommend and why?

1. The items are likely to be in sorted order when sort is called
2. The system needs to conserve memory while sorting
3. Sorting will be frequently called on the data set

What To Turn In

Create a zip file named **lab1_<your W&L ID>.zip**. Inside this zip archive should be **`sorts.py`**, **`tools.py`** and **analysis.txt**. No other files are needed.