Tutorial Objectives

- Practice working with nested lists
- Practice writing code to search and sort, and evaluate run-time complexity, as well as the other control structures and data types we have used previously.

Part 1a – Generating a grid

Write a function called `generateRandomGrid(n)` that takes an integer as argument and returns an n x n grid filled with random numbers in the range [1,100]. Note the term grid here refers to a 2D (nested) list with equal width and height. For example,

```python
grid = generateRandomGrid(4)
print(grid)
# ➔ [[76,11,85,33],[65,64,12,3],[52,55,55,89],[17,47,68,5]]
```

Part 1b – Searching a grid

Write a function called `searchGrid(g)` that takes a nested list as argument and returns the smallest element in that list. For example,

```python
searchGrid(grid)
# ➔ 3
```

Part 2a – Searching & Sorting

In this part of the tutorial you will be writing searching and sorting functions, and evaluating the amount of ‘clock-time’ that they require to run. You are free to use any code provided in lecture in your answers, however, you should take this opportunity to understand any code that you use.

In a single python file, implement the following searching and sorting functions as taught in class.

- A linear search that finds the smallest element in a given input list.
- A binary search that returns the index of the key in the list if its found.
- A bubble sort that takes a list and sorts it in place.
You will also need to make a function that takes in an integer, \( n \), and generates a list of \( n \) random numbers. Write a main() function to test your algorithms and make sure they work.

**Part 2b – Clock time**

To evaluate the actual run time (so-called ‘clock-time’) of a piece of code you’ll need to import the timeit module. Inside the timeit module is a function called default_timer() that returns the number of seconds that have elapsed since "the epoch" (12:00am Jan 1st 1970). While the epoch itself is an arbitrary number, we can use this function to determine how long our code runs by taking two samples: one before our code, and one after, and then calculating the difference.

```python
before = timeit.default_timer()
... #some code to run ...
after = timeit.default_timer()
elapsedTime = after - before
```

Use this pattern to time the execution of each of the algorithms you wrote in Part 1 to answer the following questions:

- If the size of the input list doubles how much longer does Linear Search take?
- If the size of the input list increased by 10x, how much longer does Linear Search take?
- If the size of the input list doubles how much longer does Bubble Sort take?
- If the size of the input list increased by 10x, how much longer does Bubble Sort take?
- Do these changes in values match the \( O(n) \) and \( O(n^2) \) expected costs of these algorithms?

Note: the amount of clock-time it takes to run a given algorithm will differ from computer-to-computer and even from run-to-run. So your answers to the questions above will be estimates of the time change. **For example:**

If you run algorithm X() on a list of size 1000 and it takes 0.12 seconds. Then you run the algorithm on a list of size 10000 and it takes 1.3 seconds. The input size changed by a factor of 10x. The run-time changed by a factor of: \( 1.3/0.12 = 10.833333 \), so roughly (also) 10x longer.

The time it takes to run your algorithm is affected by other things running on your computer, including background processes. So, run your code a few times to make sure your answers seem consistent. (Also, be sure not to time the function that generates your random lists, as that may skew your results)

**Sample program output:**

```
Linear search on a list of size 1000: _______________ seconds
Linear search on a list of size 2000: _______________ seconds
Linear search on a list of size 10000: _______________ seconds
Bubble Sort on a list of size 1000: _______________ seconds
Bubble Sort on a list of size 2000: _______________ seconds
Bubble Sort on a list of size 10000: _______________ seconds
```
Additional challenge: what do you think is the BigO class of your `searchGrid()` function? Make several random grids of different sizes and time them to verify your prediction.

Additional challenge: time several runs of the `binarySearch()` algorithm, and explain why there is only a small increase in clock-time each time the input size is doubled.

(More practice) Part 3 – To sort or to search unsorted?

Write a function called `searchOrSort()` that runs and times the following two experiments:

1. Create a random list of 1000 elements, and do 10000 linear searches on it.
2. Create a random list of 1000 elements, sort it once, and do 10000 binary searches on it.

Which of these two experiments takes longer?
What happens if you only do 100 searches on a list of 1000 elements?
What if you do 1000 searches on a list of 1000 elements?

Sample output:
Searching an unsorted list of 1000 items 10000 times took: __________ seconds
Sorting then searching a list of 1000 items 10000 times took: __________ seconds

(More practice) Part 4 – Gnome Sort

There are many different algorithms available for sorting a list of values. Here's the pseudocode of a sorting algorithm called GnomeSort:

```python
gnomeSort(lis):
    index = 0
    while index is less than len(lis)
        if index==0 or the item to the left is less than the item at index:
            increment index
        else:
            swap lis[index] with lis[index-1]
            decrement index
```

Write a function called `gnomeSort()` that performs this algorithm in python. Test that it correctly sorts any given list of random elements.

Once you're sure that it works, apply the same timing exercises to `gnomeSort` that you applied to `bubbleSort` and `linearSearch` in Part 2. Can you guess the complexity (in bigO notation) of the gnomesort algorithm from the results of the timing runs?