This third lab gets you to work with big-Θ classes and practically check the efficiency of sorting algorithms by testing them on randomly generated arrays.

Marks (max 5): Question 1: 1 | Questions 2-3: 2 | Questions 4-7: 0.5 each

Question 1 (does not require coding)

For each of the following expressions, find if they are $\Theta(1)$, $\Theta(\log n)$, $\Theta(n)$, $\Theta(n^{50})$ or $\Theta(2^n)$:

1. $500 + 5\log n$
2. 5000
3. $500 + n + 5\log n + 50n$
4. $5n\log n + 2^n + 300n^{50}$

Find the complexity, in terms of big-Θ, of the following expression:

$$5(\log n)^{13} + 300n^6 + 30n^5 \log n + 100$$

Question 2

For this question you may use Python's built-in function for producing random numbers. If you import Python's built-in module random by calling:

```
import random
```

then `random.randint(low,high)` will return a random integer in the range `low` to `high` inclusive (i.e. an integer with an equal chance of it being any of the numbers in that range). Use this to write a Python function:

```
def randomIntArray(s,n)
```

which returns an array of length `s` that in each position has a random integer in the range 0 to `n`.

Question 3

Python's built-in function `time()` in the module `time` returns the current time in the form of the number of seconds since 0.00am on 1st January 1970. So, code of the form:

```
t = time.time()
<operation>
t = time.time()-t
```

will set `t` to the time it takes to perform `<operation>`. As `time.time()` returns a floating point number rather than an integer, this could be a fraction of second.
Use this to write a Python function:

```
def sortTime(A)
```

which sorts an array `A` and returns the time taken to do the sorting.
You can use code given in the lecture slides to do the sorting.
Test your code in at least 3 arrays, including an empty array and one that is already sorted!

Question 4

Use the method `randomIntArray` from Question 2 to provide arrays to be sorted by `sortTime`. This will enable you to test how long it takes to sort an array much longer than one you could type in yourself. Then, fill in the following table (but see Note).

array length	10	100	1000	10000	100000	10^6	10^7	10^8
sorting time (sec)								

Note: sorting arrays of length greater than 10000 may make your computer run out of memory and hang. This is why we have greyed out the last three columns, which you do not need to fill in. If you do want to fill them in, make sure you save everything before and be ready to hard-restart your computer!
It would also make sense to stop a test if it runs over a few minutes and fill in “timeout” in the respective column.

Question 5

Write a Python method:

```
def sortTimeUsing(sortf,A)
```

which returns the time taken to sort the array `A`, but does the sorting using a function passed as the argument `sortf`. This uses the concept of passing a function as an argument that was introduced in Question 6 of Lab 1.

Use this and the table of Question 4 to compare the time taken to sort using selection sort with the time taken to sort using insertion sort (the code of which you can find on the lecture slides).

Question 6

Write a version of insertion sort that sorts the array passed to it, and also returns the number of comparisons used in the sorting.

For example, insertion sort of [2,4,6,8,10,1,3,5,7,9] should take 23 comparisons. Insertion sort works fastest when sorting an array that is already sorted, so insertion sort of [1,2,3,4,5,6,7,8,9,10] takes 9 comparisons. It works slowest when sorting an array which is sorted in reverse order, so sorting [10,9,8,7,6,5,4,3,2,1] takes 45 comparisons.

Question 7

Write a version of insertion sort that works by taking each element of an array and inserting it into a new array in a way that the new array always remains sorted. When all elements of the original array have been inserted, the new array is returned. This will give a version of insertion sort that works *constructively*, leaving the original array unchanged.