FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 5

Semester 2, 2018

Objectives of this tutorial

- To understand Big O notation.
- To be able to find the best and worst case time complexity for simple algorithms.
- To be able to determine whether a sorting method is stable.
- To learn how to use Exceptions

Exercise 1 *

Write a version of bubble sort that alternates left-to-right and right-to-left passes through the list. The left-to-right passes move the largest unserted element to its final place, the right-to-left passes move the smallest unsorted element to its place. This algorithm is called *shaker sort*.

(a) Give the best and worst case time complexity for your algorithm (together with the kind of list that will give that complexity), and explain why.

(b) Is this sorting method stable? Explain.

Exercise 2 *

(a) Write a method for computing the sum of the digits of a number. For example, for number 979853562951413, the sum of its digits is $9+7+9+8+5+3+5+6+2+9+5+1+4+1+3 = 77$. To do this you can use integer division by 10 (//10) which returns an integer with the same digits except the last one, and reminder by 10 (% 10), which returns the last digit. For example, if you have X = 3456, then X//10 gives you 345, while X%10 gives you 6.

(b) Determine its complexity, in Big-O notation.

Exercise 3 *

This question is designed to help you understand big-O complexity, and how it relates to the actual running time of a program.

Suppose you have a problem and you have designed five different algorithms to solve it, which you call Alpha, Bravo, Charlie, Delta and Echo. To study which is best, you implement them all and try them out. You run them on several inputs of each size $N = 10, 20, 30, \ldots, 90, 100$, and note down the longest running time, in milliseconds, for each input size. The data you collect is:

input size	maximum running time (ms)				
N	Alpha	Bravo	Charlie	Delta	Echo
10	9999999	10	20	9876543	1415926535
20	99999	40	200	3010	8979323846
30	220	777777	2000	4771	2643383279
40	330	160	20000	6021	5028841971
50	439	245	200000	6990	6939937510
60	549	360	2000000	7782	5820974944
70	660	489	20000000	8451	5923078164
80	767	640	200000000	9031	628620899
90	880	808	2000000000	9542	8628034825
100	989	1000	20000000000	10000	3421170679

a) Alpha: O(N), Bravo: O(N^2) , Charlie: O(10^N/10) , Delta: O(N log N) , Echo: O(1)
b) Echo is the most scalable algorithm, because it has constant complexity. This is only efficient for large inputs.
For input size of 40, the most efficient algorithm is Bravo, even though it has the second worst time-complexity, but it runs the fastest.
For input size of 10000,
For input size of 10000000000000000000, the best algorithm and most efficient is Echo, because the lowest complexity O(1) will be the most efficient algorithm as it runs constant operations no matter how large the input is.

Now, this data does not, of itself, *prove* anything about the time complexities of these algorithms. (Why not?) But it may still *suggest* to you some likely big-O expressions for the worst-case complexity of each algorithm.

(a) Give a plausible big-O expression for the worst-case time complexity of each of the algorithms Alpha, Bravo, Charlie, Delta and Echo. To do this, it might be useful to consider (as we did in lecture L10) what happens to the time when the input size N doubles or when it increases by any constant amount.

(b) Which algorithm is most scalable, according to big-O complexity? Which algorithm would you choose for an input of size 40? *Guess* which algorithm would you choose for an input of size 10000. (I'm not looking for detailed calculation for this input size, just wanting to get you thinking about it.) What about an input of size 100000000000000000000 (assuming you had a computer big enough to work with so much data!)?

Exercise 4 *

What does the following snippet of code do? Discuss.

```
a = [0, 1]
try:
    b = a[2]
    print('that worked!')
except ValueError:
    print("no it didn't!")
```

It will output IndexError instead of catching the exception for ValueError, because the maximum index in the array a that can be accessed is 1 and not 2.

Exercise 5

You can catch all exceptions by using a bare `except` statement, such as the one shown in the example below. However, this is in general acknowledged as a bad idea. Why do you think this is the case? And what would be the correct way to do it?

```
a = [0, 1]
try:
    b = a[2]
    c = int('foo')
    d = e
    f = 1/0
    print(1 + '1')
except:
    print("what happened!?")
```

A general exception statement does not specify what the error is, and if the code is a huge file, the programmer would have to read through and debug them one by one, which is time consuming.

A better approach would be going through each line of the try statement and create an exception for that particular line of code. For example, possible exceptions could be IndexError, ValueError, NameError, ZeroDivisionError, TypeError and specifying each of the exception individually on separate lines of code.

Exercise 6

Definition: The *digital root* of a decimal integer is obtained by adding up its digits, and then doing the same to *that* number, and so on, until you get a single digit, which is the digital root of the number you started with.

For example, to find the digital root of 979853562951413, we calculate: sum of digits $= 9+7+9+8+5+3+5+6+2+9+5+1+4+1+3 = 77$, then sum of digits $= 7+7 = 14$, then sum of digits $= 1+4 = 5$. Now we have just one digit, 5, so that's the digital root of the number we started with.

(c) Write a method to compute the digital root of a positive integer.

(d) Determine its complexity, in Big-O notation.