FIT1008 – Intro to Computer Science
Solutions for Tutorial 9

Semester 1, 2018

Exercise 1

(a) A call to `a_list.mystery()` where `a_list` contains the elements 1,2,3,4,5 with 1 being at the head, would traverse the entire list doing nothing in the way to the last node, while in the way back will modify the content of each node by adding to it the content of the successor node (i.e., the content of the node that used to be 4, will now be 4+5=9, the content of the node that had a 3 will now be 3+9=12, and so on). For this example, the list would be returned as 15, 14, 12, 9, 5, with 15 being at the head.

(b) The best and worst case time complexity is O(N) where N is the number of elements in the list, since the recursion traverses every element (regardless of the actual numbers in the list) and, for each, if performs exactly the same constant time operations (+=).

(c) The method could be defined iteratively by using two stacks as follows:

```
def mystery(self):
    current = self.head
    my_stack = Stack()
    while current is not None:
         my_stack1.push(current.item)
         current = current.next

    my_stack2 = Stack()
    int temp = 0
    while not my_stack1.is_empty():
         temp += my_stack1.pop()
         my_stack2.push(temp)

    current = self.head
    while current is not None:
         current.item = my_stack2.pop()
         current = current.next
   }
```

The first loop puts the elements in reverse order into `my_stack1` (i.e., the elements are pushed into `my_stack1` as 5,4,3,2,1 with 5 being at the top). The second loop empties `my_stack1` creating a new stack `my_stack2` whose elements are the sum of those already poped from `my_stack1` (i.e., the elements are pushed into

`my_stack2` as 15,14,12,9,5 with 15 being at the top). And the third loop empties `my_stack2` transferring the content to the nodes in the list.

Clearly, the recursive method is much simpler and clearer.

Exercise 2

(a) One possible solution is:

```
def sum_of_digits(int x):
    if x < 10:
        return x
    else:
        return x % 10 + sum_of_digits(x//10)
```

(b): For complexity, observe that we start at the least significant digit and, for each position, we do a constant amount of processing, and that we move to the left one position at a time, shortening the number as we go. So the time is $O(N)$, where N is the number of digits of x.

Note that, if the input to an algorithm is a *number*, then the complexity of the algorithm should be given in terms of the *number of digits* of the number, since that measures the size of the number *as input data to the algorithm*. Don't give the complexity in terms of the number itself. Doing so can be misleading. If the input is a number n, and the algorithm requires (say) n steps, then this might sound like linear time. But it is actually exponential time! If a number n has d decimal digits, then we know $10^{d-1} \leq n < 10^d$, so n is *exponential* in d.

This may take a little getting used to. But it's really not so surprising. Big numbers are easy to write down, but very hard to count up to! For example, if you write down 1 followed by say a hundred zeros — a "googol" (correctly spelt) — this might take you about a minute. But this number is many times larger than the number of atoms in the universe, and counting to it would take much longer than the current age of our universe.

By the way, it is not hard to give a precise formula for the number of digits d in a number x:

$$d = \lfloor \log_{10} x \rfloor + 1.$$

(Here, $\lfloor \cdots \rfloor$ means *truncate*.) If you want the number of digits used for representing the number x in some other base b, then just replace the base 10 for the logarithm here by b.

So the number of digits of x is $O(\log x)$.

(c): One possible solution is:

```
def digital_root(x):
    if x < 10:
        return x
    else:
        return digital_root(sum_of_digits(x))
```

(d): We know that, for sufficiently large x, the sum of the digits of x can be computed in time at most $K \cdot N$, where N is the number of digits of x and K is some constant. (This is because the complexity of `sum_of_digits` is $O(N)$, as explained in (b) above.) Now, the value returned by `sum_of_digits(x)` will be $\leq 9N$, since it is biggest when all the N digits of x are 9. How many digits does this value returned have? Certainly $\leq K' \cdot \log_{10}(9N)$, which is $K'(\log_{10} 9 + \log_{10} N)$ which in turn is $\leq K'' \log_{10} N$ (for appropriate constants K' and K''). The important point here is that the number of digits in `sum_of_digits(x)` is at most logarithmic in the number of digits in the original number x. This pattern continues, so we get a sequence of numbers whose numbers of digits are bounded above by constant multiples of N, $\log N$, $\log\log N$, $\log\log\log N$, ..., etc. The time taken for each `sum_of_digits` calculation is linear in the number of digits, but that number decreases very rapidly as we have just seen. So the total time will be at most some constant times $N + \log_N + \log\log N + \log\log\log N + \cdots$, which in turn is at most some bigger constant times N. (To see this, note that, for sufficiently large N, we have $\log N \leq N/2$, and $\log\log N \leq N/4$, etc (in fact these inequalities are very loose), so $N + \log N + \log\log N + \log\log\log N + \cdots \leq N + N/2 + N/4 + N/8 + \cdots \leq 2N$.)

So the total time taken by `digital_root` will be $O(N)$, where N is the number of digits in x. The constant (hidden by the Big-O notation) will be larger for `digital_root` than for `sum_of_digits`, though.

This discussion has been about worst-case complexity. (Remember that, if people just talk about "complexity" or "time complexity" or "computational complexity", without specifying best- or worst-case, then, by default, it is taken to mean *worst-case* time complexity.) Best-case complexity of each algorithm is also $O(N)$. The hidden constant will be the same as for worst-case for `sum_of_digits`, since the procedure is the same whatever the digits are. However the hidden constant for `digital_root` will be a bit smaller, in the best case, than for the worst case, since in a

number with a single non-zero digit followed by zeros, you only need to call `sum_of_digits` once, and then you get straight to the base case of `digital_root` (without having to do a series of recursive calls).

Other questions to consider are: How would you modify the above methods to work for all integers, not just positive ones? And how would you modify them to find digital roots in bases other than 10?

Extra challenges for the advanced student: Digital roots have some interesting properties. For example, an integer is divisible by 3 if and only if its digital root is 3, 6 or 9. An integer is divisible by 9 if and only if its digital root is 9. These observations can be useful for testing divisibility by pencil-and-paper (though they offer no advantage to a computer!). You don't get digital-root-based tests for numbers other than 3 or 9, though. You may care to try to prove that these divisibility tests for 3 and 9 work, and to see what divisibility tests you could do using digital roots for bases other than 10.

Exercise 3

The pivot is the deciding factor in how "balanced" are the two partitions. The best pivot is one that guarantees that both partitions have the same length plus/minus one (i.e., absolute difference value of one). Such pivot will result in a Quick sort method that has a complexity of O(N log N).
The worst pivot is one that forces all elements into a single partition, leaving the other one empty. Such pivot will result in a Quick sort method that has a complexity of $O(N^2)$.
Note that the position of the pivot in the array (i.e., whether you choose the first, last, middle, etc) makes no difference, the difference is in how it partitions the elements in the array.
An approach that achieves reasonable behaviour while being simple and fast to compute, is to read three numbers and take the median one. Also, if the input is relatively sorted, the middle element is quite a good choice. For reasonably large lists, a (truly) random index is also a good choice.

Exercise 4

Merge sort is stable as coded in the lectures because it always takes elements (if any) from the right hand side of the array before it considers elements from the left hand side.
The common Quick sort is not a stable algorithm due to the way the partitioning algorithm is defined (think about the swap with the pivot element and how it can modify the relative order between two elements with the same key).
However, note that any sort algorithm can be made stable by adjusting the compare function. Before sorting, annotate each element with an integer denoting its index. Then, adjust the compare function so that whenever it says that two elements are equal, it uses the extra index data to determine the correct order. This adds a small linear cost (annotating the array) and a small constant cost to every comparison that would return "equal".