FIT1008 – Intro to Computer Science
Solutions for Tutorial 5

Semester 1, 2018

Exercise 1

- This algorithm reverses a list.
- There are 2 assignments inside the loop which we can assume take k units of time. The loop itself will run exactly $n/2$ times. There is one assignment outside the loop, thus in total we have $f(n) = 6(\frac{n-1}{2}) + 3$. Because Big O ignores constants we have $O(n)$ complexity.
- There is no difference between best and worst case. Running time is completely determined by the size of the list and does not depend on the content of the list.

Exercise 2

```
def swap(a_list, i, j):
    a_list[i], a_list[j] = a_list[j], a_list[i]

def shaker_sort(a_list):
    no_swaps = True

    n = len(a_list)
    moving_right_start = 0
    moving_right_end = n-1
    moving_left_start = n-1
    moving_left_end = 0

    for j in range(n-1):
        if (j % 2 == 0):
            for i in range(moving_right_start, moving_right_end):
                if a_list[i] > a_list[i+1]:
                    no_swaps = False
                    swap(a_list, i, i+1)
            moving_left_start -= 1
            moving_right_end -= 1

        else:
            for i in range(moving_left_start, moving_left_end, -1):
                if a_list[i-1] > a_list[i]:
                    no_swaps = False
                    swap(a_list, i, i-1)
            moving_left_end += 1
```

```
28              moving_right_start += 1
29
30          if no_swaps:
31              break
32          no_swaps = True
```

- The best time complexity is $O(N)$, where N is the length of the list, occurs when the list is sorted. In this case the algorithm stops after one pass through the list.
- The worst time complexity is $O(N^2)$, where N is the length of the list, occurs when the list is sorted in reverse order. In this case we have nested loops both which depend linearly upon N.
- The algorithm is stable since the relative order of equal valued items does not change.
- The algorithm is incremental because if you put the item at the start of the list and bubble sort left-right, the list will be sorted after one pass.

Exercise 3

The code shown doesn't print anything. Attempting to index past the length of a list throws an IndexError, which doesn't get caught by the `except ValueError` statement. Thus, the exception is not handled, and the program exits with a stack trace. In order to catch the error, we would have had to write the code thus:

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

This code prints "no it didn't!" and exits cleanly, because the exception was caught and not re-raised.

Exercise 4

Catching all exceptions with a blanket `except` statement forces you to handle *everything*. This is usually a bad idea, first because exception handling is supposed to be about the errors that you are expecting and, secondly, because not all errors are supposed to be handled by you. *Exception handling is about expected errors:* If you are opening a file, you brace for the file not being available, or the possibility of a hardware error. If you are sanitising user input, you know that users

may enter values outside the scope of your function. Those are the errors you should catch. But if you implement a catch-all, you'll also supress the expression of many errors that you don't expect, and force the bugs to appear in other parts of your code.
Not all exceptions are supposed to be handled by you: if you are writing library code, functions that other programmers will use in their own programs, there are three things you can do with exceptions that you receive: some of them you'll want to handle yourself. Some exceptions you'll want to catch, do something with them, and then raise them again so the calling code can handle them. And some of them you'll just let pass to the calling code. The blanket `except` makes more difficult for you to decide among these three choices. Knowing this, and excusing the fact that the example is horribly contrived, how should we have written the code in the exercise? Here's a better option:

```
a = [0, 1]
try:
    b = a[2]
    c = int('foo')
    d = e
    f = 1/0
    print(1 + '1')
except IndexError:
    # handle the IndexError from a[2]
except ValueError:
    # handle the ValueError from int('foo')
except ZeroDivisionError:
    # handle the division by zero
except TypeError:
    # handle the Typeerror from "1 + '1'"
except Exception as e:
    # log exception
    raise e
```

Here the catch-all `except` is written in a modified form (`except Exception as e:`) that allows you to actually capture the exception and re-raise it. This is one of the few cases where capturing all exceptions is acceptable: you have already dealt with all errors you excepted, so you want to log anything you didn't expect before raising it in case it can be dealt with in the calling code.

Exercise 5

```
def divide(a, b):
    return a / b

def test_divide():
```

```
5       try:
6           divide(1, 0)
7           raise Exception('ZeroDivisionError not raised.')
8       except ZeroDivisionError:
9           pass
10
11  if __name__ == '__main__':
12      test_divide()
13      print('All test cases passed.')
```