FIT1008 – Intro to Computer Science
Tutorial 1
Solution
Semester 1, 2019

*Exercise 1 **

Solution

Part i & iv

This algorithm has two nested loops, i.e., one of the loops (referred to as the inner loop) is place inside the other (referred to as the outer loop), and, thus, it is executed for every iteration of the outer loop. For an list of length 7 the algorithm will print "Outside" 7 times, since it prints it once for each element in the list. Given this reasoning, it is clear we can easily generalise this result: for an list of length N, the algorithm will print "Outside" N times.
It will also print " Inside" 49 times, since the inner loop is executed 7 times (once per element in the list) for each iteration of the outer loop. Since there are 7 iterations of the outer loop (one per element in the list), we have 7 * 7 = 49. If you run the algorithm you will see how "Inside" is printed 7 times for each print of "Outside". Again, we can easily generalise this result: for an list of length N, the algorithm will print "Inside" N * N = N^2 times.

Part ii & iv

The value returned in count by the algorithm is 56, since it is incremented by 1 every time we print "Outside" and every time we print " Inside", which gives 49+7=56. Generalising as before, we could say that for an list of length N, the algorithm will return the value $N^2 + N$.

Part iii & iv

For the list 1, 2, 3, 4, 5, 6, 7 the value printed for sum is 224, since both inner and outer loop add all the elements in the list once (1+2+3+4+5+6+7=28) but the inner loop is called 7 times. This results in 28+28*7 = 224. For list 1, 2, 3, . . . , N, the sum of its elements is $(N^2 + N)/2$, thus, it would be $(N^2 + N)/2 + N * (N^2 + N)/2 = (N^2 + N + N^3 + N^2)/2 = (N^3 + 2N^2 + N)/2$. You can check that for N = 7 that results in 224.
There is a clear relationship between the value of count (N^2+ N) and BigO time complexity, since this value counts the number of times

each loop is performed. The algorithm is then $O(N^2)$.

*Exercise 2 **

Solution

One of the simplest ways to solve it is as follows:

```
def is_palindrome(the_string):
    string_reversed = ""
    for item in the_string:
        string_reversed = item + string_reversed
    return the_string == string_reversed
```

which first reverses the string and then checks whether it is identical to the original one. This is extremely clear (great) but not very efficient due to two reasons, as we need to traverse the entire string, and do so before doing any checking. The following two solutions are alternatives to this.
The first one is as follows:

```
def is_palindrome(the_string):
    n = len(the_string)
    for k in range(n//2):
        if the_string[k]!=the_string[n-1-k]:
            return False
    return True
```

which traverses the string from its two ends (`k` and `n-1-k`) checking whether the two elements are different. If they are, it immediately returns False. If they are not, it keeps on checking, until the two halves of the string have been checked. If so, it returns true, as all pairs of elements were equal. Several alternatives for a `for` loop exist, where the two indices (`k` and `n-1-k`) are computed in different ways, all good.
The following alternative solution implements a very similar algorithm with a `while` loop:

```
def is_palindrome(the_string)
    i=0
    j=len(the_string)-1
    while i< j:
        if the_string[i] != the_string[j]:
            return False
        i+= 1
        j-= 1
    return True
```

where the first line provides the initial value of the index (`k` in the previous solution), the condition provides the ending (since `i` and `j` are incremented and decremented by one, respectively, in each

iteration, then `i<j` implies `i` is less than `len//2`, so `len//2-1`), and the `i=1+` line provides the increment for the `for` loop. I usually don't like `while` loops that really are `for` loops in disguise (does not help clarity), but this is relatively clear. A good thing of this solution is that the computation of `j` is more efficient than that of `n-1-k`. A combination of both solutions would be good (do it yourself!)
Another possibility is to use the slicing library function we talked about in the class (but do this only once you know how to do it using the basic iterations):

```
def is_palindrome(the_string):
    return the_string == the_string[::-1]
```

The algorithm implemented by this solution is the same as the first one provided: it first reverses the string (achieved by `the_string[::-1]`) and then simply checks whether the original string is the same as its reversed. Note that the slice function accepts 3 arguments: `[start:stop:increment]`, where `start` is inclusive, and `stop` is exclusive. If `start` is not provided, it assumes 0. If `stop` is not provided, it assumes the length. It is very clear for Python natives, and quite unclear for everyone else :-) It is also very efficient, thanks to the way slicing is implemented.

*Exercise 3 **

Solution

For the intersection, one could define:

```
@pre list1 and list2 do not have duplicate elements
def print_intersection(list1, list2):
    for i in range(len(list1)):
        for j in range(len(list2)):
            if list1[i] == list2[j]:
                print(list1[i])
```

which traverses `list1` and for each element in it (`list1[i]`), traverses `list2` checking whether `list1[i]` appears in it. As soon as it finds it, prints it, otherwise it goes to the nest element in `list1`.
Python allows the above algorithm to be implemented even more clearly as follows:

```
@pre list1 and list2 do not have duplicate elements
def print_intersection(list1, list2):
    for item1 in list1:
        for item2 in list2:
            if item1 == item2:
                print(item1)
```

since we do not need to iterate over the indices (`i` and `j`), we can iterate directly over the elements (`item1` and `item2`).

The two versions above assume the input lists do not have duplicates. If this is not the case, they might work in an unexpected way. For example, `print_intersection1([1,2,2],[1,1,2,3])` would print 1 twice and also 2 twice. A possible solution for not printing 1 twice would be to not iterate over `list2` and instead check whether it appears in it as follows:

```
def print_intersection(list1, list2):
    for item1 in list1:
        if item1 in list2:
            print(item1)
```

To avoid printing the 2 twice, we could build a list to store the intersection and check for duplicates before printing. Something like:

```
def print_intersection(list1, list2):
    intersection = []
    for item1 in list1:
        if item1 in list2 and item1 not in intersection:
            print(item1)
            intersection.append(item1)
```

Exercise 4

Solution

- Heaps vs BSTs
  - heaps have the properties:
    - that all levels are full (filled left to right) except (possibly) the bottom level
    - that a parent is no smaller than either child (in a max heap or no larger in a min heap)
  - binary search trees have the properties:
    - all elements in the right subtree of a node have keys larger than that node
    - all elements in the left subtree of a node have keys smaller than that node
- stacks vs queues
  - for stacks, access is only given to the top of the stack, additions and removals both occur there leading to the elements entering first will be those first to leave
  - for queues, we remove from the front and add to the read, both ends are accessible but for different actions; hence the first to enter is the first to leave the queue.

- iteration vs recursion
  - iteration involves using a loop to repeat code; the block of code inside the loop will be repeated until the condition of the loop is no longer met
  - recursion involves using function calls to repeat code; in this case an if statement is used to represent the termination of the equivalent loop and successive iterations of the equivalent loop are handled by function calls to the same function with a change in argument.