NATIONAL UNIVERSITY OF SINGAPORE

Semester 1, 2016/2017

Solutions for CS1010S — PROGRAMMING METHODOLOGY

Time Allowed: 2 Hours

INSTRUCTIONS TO STUDENTS

1. Please write your Student Number only. Do not write your name.
2. The assessment paper contains **FIVE (5) questions** and comprises **TWENTY (20) pages** including this cover page.
3. Weightage of questions is given in square brackets. The maximum attainable score is 100.
4. This is a **CLOSED** book assessment, but you are allowed to bring **TWO** double-sided A4 sheets of notes for this assessment.
5. Write all your answers in the space provided in this booklet.
6. You are allowed to write with pencils, as long as it is legible.
7. **Please write your student number below.**

   STUDENT NO: _____

(this portion is for the examiner's use only)

Question	Marks	Remark
Q1	/ 30	
Q2	/ 24	
Q3	/ 30	
Q4	/ 12	
Q5	/ 4	
Total	/ 100	

Question 1: Python Expressions [30 marks]

There are several parts to this problem. Answer each part **independently and separately**. In each part, one or more Python expressions are entered into the interpreter (Python shell). Determine the response printed by the interpreter for the final expression entered and **write the exact output in the answer box**. If the interpreter produces an error message, or enters an infinite loop, explain why. You may show your workings **outside the answer box** in the space beside the code. Partial marks may be awarded for workings if the final answer is wrong.

A. [5 marks]

```
a = ['Pine', 'pen']
b = ['apple', 'PEN']
a[0] += b[0]
print(sorted(a+b, key=lambda x:x[::-1], reverse=True))
```

```
['pen', 'Pineapple', 'apple', 'PEN']
```

This question tests the understanding of the += operator and how Python sort function works. Note: Capital letters comes before non-capital letters.

B. [5 marks]

```
s = "How cool is that?"
d = {}
for i in range(len(s)):
    d[i%5] = s[i]
print(d)
```

```
{0: 't', 1: '?', 2: 't', 3: 'h', 4: 'a'}
```

This question tests the understanding of Python dictionaries and how integers can be used as keys.

C. [5 marks]

```
def g(x):
    return lambda y: x(x(y))
def f(x):
    return lambda y: y(x)
print(f(lambda x: x*2)(g)(5))
```

```
20
```

This question tests the understanding of lambda functions.

D. [5 marks]

```
j = 1
for i in range(1, 10):
    if j % i == 0:
        i += j
    else:
        j += i
    if i > 10:
        break
print(i, j)
```

```
18 12
```

This question tests the understanding on that the iterating variable is not affected by the body.

E. [5 marks]

```
def yikes(*args):
    try:
        return args[0] / args[-1]
    except IndexError:
        print("Aiyoh")
    except TypeError:
        print("Cannot")
    except ZeroDivisionError:
        print("Die!")
    finally:
        return args[1] + args
print(yikes(4, (0,)))
```

```
Cannot
(0, 4, (0,))
```

This question tests the understanding of exception handling and *args.

F. [5 marks]

```
a = [1, 2]
a += [a]
b = a.copy()
b[2][2] = 0
print(a)
print(b)
```

```
[1, 2, 0]
[1, 2, [1, 2, 0]]
```

This question tests the understanding of shallow copy of lists and *loopback of lists*. This becomes clear when you draw the box-pointer diagrams for the lists.

Question 2: Stock Prices [24 marks]

You are given a sequence of integers representing the daily stock prices on consecutive days. For example, the tuple `(5, 10, 4, 6, 10, 12, 8, 8, 3, 5, 6)` means that the price of the stock is 5 on the first day, 10 on the second day and so on.

A. The function `cost_to_diff` takes as input a sequence of daily stock prices, and outputs a list which shows the daily difference of the stock price compared to the day before.

Example:

```
>>> cost_to_diff([5, 10, 4, 6, 10, 12, 8, 8, 3, 5, 6])
[0, 5, -6, 2, 4, 2, -4, 0, -5, 2, 1]
```

The first day will always be 0. The second day is the difference in price from the previous day, which is $10 - 5 = 5$. The third day is $4 - 10 = -6$, and so on.

Provide an implementation of the function `cost_to_diff` [4 marks]

```
def cost_to_diff(stocks):
    diff = [0]
    for i in range(1, len(stocks)):
        diff.append(stocks[i] - stocks[i-1])
    return diff
```

B. Implement a function `longest_rise` that takes as input a sequence of daily stock prices, and outputs the longest number of consecutive days where the stock prices increased from the day before. For example, using the prices `[5, 10, 4, 6, 10, 12, 8, 8, 3, 5, 6]`, the output will be 3 (days 4 to 6 inclusive). [4 marks]

```
def longest_rise(stocks):
    longest = 0
    current = 0
    for i in cost_to_diff(stocks):
        if i > 0:
            current += 1
            if current > longest:
                longest = current
        else:
            current = 0
    return longest
```

C. Some stock analysts want to know how many days did the stock price rise over the previous day. For example, using the prices `[5, 10, 4, 6, 10, 12, 8, 8, 3, 5, 6]`, there are 6 days (days 2, 4, 5, 6, 10 and 11) where the stock price was higher than the day before.

Implement the function `days_increase` which takes as input a sequence of daily stock prices, and outputs the number of days where the stock price rose over the previous day. [4 marks]

```
def days_increase(stocks):
    return len([s for s in cost_to_diff(stocks) if s > 0])

# Using filter
def days_increase(stocks):
    return len(list(filter(lambda x: x > 0, cost_to_diff(stocks))))

# Using iteration
def days_increase(stocks):
    result = 0
    for i in cost_to_diff(stocks):
        if i > 0:
            result += 1
    return result
```

D. Some analysts however, want to know when the stock prices drop. Some want to know when it remains the same. Some want to know when the prices increase by twice the previous day! In all cases, they wish to compare a day's price with its previous day's price.

To handle all such requests, the function `days_of(stocks, cond)` takes as inputs a sequence of daily stock prices, and `cond`, which allows the analysts to specify the condition that they want.

i) Describe the specification of the input `cond` and, ii) show how the analyst should call the function if he is interested in days where the stock price is at least double of the previous day. [2 marks]

i) `cond` will be a function that takes two inputs: the price of the stock on the previous day and the price of the stock on the current day. It returns `True` if the price matches the condition that the analyst is interested in, and `False` otherwise.

ii)
```
days_of(stocks, lambda x,y: y >= 2*x)
```

E. Provide an implementation of the function `days_of`. [4 marks]

```
def days_of(stocks, cond):
    count = 0
    for i in range(1, len(stocks)):
        if cond(stocks[i-1], stocks[i]):
            count += 1
    return count
```

F. Suppose you can buy the stock exactly once and sell it exactly once, how can we find the maximum profit that can be made? This problem is known as the Maximum Single-sell Profit.

For example, given the prices `[5, 10, 4, 6, 10, 12, 8, 8, 3, 5, 6]`, the maximum profit is to buy it on day 3 (at a cost of 4) and sell it on day 6 (at a cost of 12) for a profit of 8. While on day 9 the cost of the stock is the lowest (at a cost of 3), it cannot be sold at 12 because the stock first needs to be bought before selling.

The naive way of solving this is to simply brute force all combinations of buy-sell days, but we can do better with divide-and-conquer strategy which we have learned. Consider splitting the price sequence into two halves. There are 3 possible options where the best possible buy and sell days are:

1. Both the best buy and sell days are in the left half.
2. Both the best buy and sell days are in the right half.
3. The best buy day is the in left half and the best sell day is in the right half.

Thus, the maximum single-sell profit is the greatest profit among these three options. (Note that if the stock prices are always non-increasing, the maximum profit is 0, where we simply buy and sell on the same day.)

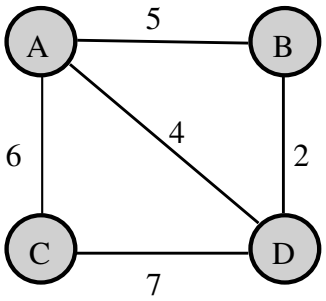
Implement the function `max_profit` which takes as input a sequence of daily stock prices, and outputs the maximum single-sell profit using the divide-and-conquer strategy.

Hint: you might want to make use of the Python built-in `max` and `min` functions. [6 marks]

```
def max_profit(stock):
    if len(stock) <= 1:
        return 0
    else:
        mid = len(stock) // 2
        left = stock[:mid]
        right = stock[mid:]
        return max(max_profit(left),
                   max_profit(right),
                   max(right) - min(left))
```

Question 3: Travel Woes [30 marks]

Suppose we are given a bunch of cities, and the cost to travel between certain pairs of cities. We can model the routes pictorially, for example:



The figure above shows 4 cities connected by links of different costs. Note that there is no direct link between cities C and B, and we can assume that the cost for each link is the same going both ways.

We can represent the routes between cities in Python using a nested list of pairs. For example, the above routes can be stored as the list:

```
routes = [('A', [('B', 5), ('C', 6), ('D', 4)]),
          ('B', [('A', 5), ('D', 2)]),
          ('C', [('D', 7), ('A', 6)]),
          ('D', [('C', 7), ('B', 2), ('A', 4)])]
```

Each element of the list is a pair `(city, links)`, where `city` is the name of the city, and `links` is another list of pairs of a city with a direct link and the cost of the link. Note the lists are unsorted. We shall denote this representation as "**list-of-lists**".

Another way of representing the routes is using a "**dictionary-of-dictionaries**". For example, the same routes can be stored as the dictionary:

```
routes = {"A": {"B": 5, "C": 6, "D": 4},
          "B": {"A": 5, "D": 2},
          "C": {"A": 6, "D": 7},
          "D": {"A": 4, "B": 2, "C": 7}}
```

The keys are the cities and the values are dictionaries which keys are the destination city and the value the cost of the direct link to the city.

The function `cost_between` which takes as inputs the **routes** (in either one of the above representations), a **source** city (which is a string) and a **destination** city (which is a string). The function outputs the cost of the direct link from the source to the destination city. If no direct link exists, a `NoRouteException` is raised.

Example:

```
>>> cost_between(routes, 'A', 'B')
5
>>> cost_between(routes, 'C', 'B')
NoRouteException
```

A. Provide an implementation of the function `cost_between` which takes in the **list-of-lists representation** of routes. and state the order of growth in terms of time for the function. You may assume `NoRouteException` has been defined. [5 marks]

```
def cost_between(routes, src, dest):
    for s, costs in routes:
        if s == src:
            for d, cost in costs:
                if d == dest:
                    return cost
    raise NoRouteException

# using filter or comprehension
def cost_between(routes, src, dest):
    cities = [s[1] for s in routes if s[0] == src] # note the indexing
    costs = [c[1] for c in s[0] if c[0] == dest]
    return costs[0]
```

If using filter, you must be careful to note the result is a list in a list, because there is only one result.

Order of growth in Time: $O(n)$ where n is the number of cities.

At first glace it might appear to be $O(n^2)$ because of nested loops, but the inner loop only runs once. So it is actually $n+n$.

B. Provide an implementation of the function `cost_between` which takes in the **dictionary-of-dictionaries representation** of routes, and state the order of growth in terms of time for the function. [5 marks]

```
def cost_between(routes, src, dest):
    if dest in routes[src]:
        return routes[src][dest]
    else:
        raise NoRouteException
```

Order of growth in Time: $O(1)$

C. Between the list-of-lists representation and the dictionary-of-dictionaries representation, which do you think is more efficient? [2 marks]

The dictionary-of-dictionaries is more efficient because dictionary look up is constant time, as compared to having to linearly search through a list.

For the rest of this question, assume a dictionary-of-dictionaries representation is used for the routes.

D. The function `add_city` takes as inputs the routes, and a pair `(city, links)` where `city` is a new city and `links` is a sequence of pairs of an existing city and cost. The function updates routes with the new city and the respective cost to the existing cities found in `links`.

Example:

```
>>> cost_between(routes, "A", "E")
NoRouteException

>>> add_city(routes, ("E", (("A", 3), ("D", 8))))
>>> cost_between(routes, "A", "E")
3
```

Hint: the `dict` function converts a sequence of pairs into a dictionary. [4 marks]

```
def add_city(routes, new):
    name = new[0]
    for city, cost in new[1]:      # add city as destination
        routes[city][name] = cost
    routes[name] = dict(new[1])    # add city as source

# without using dict() function
def add_city(routes, new):
    routes[new[0]] = {}              # create new entry
    for city, cost in new[1]:
        routes[new[0]][city] = cost  # add new -> existing
        routes[city][new[0]] = cost  # add existing -> new
```

Most students forget to update the dictionary for **both** directions.

E. Professor Siva wants to find a path to walk from one city to another. The best way he can think of is to randomly choose an adjacent city to walk to and repeat until he reaches his destination.

Implement the function `random_walk` which takes as inputs the **routes**, a **source** city and a **destination** city, and outputs a list of cities that were randomly visited until the destination is reached.

Example:

```
>>> random_walk(routes, 'A', 'C')
['A', 'D', 'B', 'D', 'A', 'B', 'A', 'C']

>>> random_walk(routes, 'C', 'A')
['C', 'A']  # lucky
```

Hint: The `random` *package contains a few useful functions. For example, the function* `randint(a, b)` *returns a random integer N such that* $a \leq N \leq b$*, and the function* `choice(seq)` *returns a random element from the non-empty sequence* `seq`*. If* `seq` *is empty, it raises* `IndexError`*.*

[6 marks]

```
from random import *

# Iterative solution
def random_walk(routes, src, dest):
    path = [src]
    while src != dest:  # keep trying until we reach dest
        src = choice(list(routes[src].keys()))  # random neighbour
        path.append(src)
    return path


# Recursive solution
def random_walk(routes, src, dest):
    if src == dest:
        return [src]
    else:
        nxt = choice(list(routes[src].keys()))
        return [src] + random_walk(routes, nxt, dest)
```

Common mistake with recursion is that students choose the next city and have the base case as `[src, dest]`.

F. Yan Rong thinks she has a better strategy. She will choose the next city based on the link with the least cost. However, she realizes that simply doing that might cause her to get stuck in a loop even if she does not go back to the previous city, e.g., going from B to C will result in $B \to D \to A \to B \to D \to A \to B \to ...$

So she introduces an additional rule that she will not visit a city again after visiting it once. Thus, she can now go from B to C as $B \to D \to A \to C$.

Implement the function `greedy_walk` which takes the same set of inputs as `random_walk`, and outputs a list of cities that were visited following Yan Rong's greedy[1] strategy. [6 marks]

```
def greedy_walk(routes, src, dest):
    path = [src]
    while src != dest:
        # obtain a list of next unvisited cities
        left = filter(lambda x: x not in path, routes[src])
        # choose the minimum cost
        src = min(left, key=lambda x:routes[src][x])
        path.append(src)
    return path
```

This is almost identical to the iterative version of the previous part. The only change is instead of choosing the next city at random, the one with the least cost is chosen after filtering out cities that have been visited.

A recursive solution requires a way to reference the history, which is a direct translation from the iterative version. Or, you can start deleting links in `routes` as you go along:

```
# recursive solution
def greedy_walk(routes, src, dest):
    if src == dest:
        return [src]
    else:
        # choose next city
        chosen = min(routes[src], key=lambda x:routes[src][x])
        # remove src from routes
        for city in routes[src]:
            del routes[city][src]  # must burn these bridges
        del routes[src]            # but this can be left. why?
        return [src] + greedy_walk(routes, chosen, dest)
```

Now an error may occur if there are no cities left to visit. In this code `min` will raise a `ValueError` when the sequence is empty. This is fine because you were not told how to handle it. In fact, that is the premise of the next part.

[1] Greedy because it greedily chooses the smallest link.

G. Caryn spots a problem with Yan Rong's strategy (as expected from someone who tops the Leaderboard). Depending on the links and the cities chosen, it is sometimes not able to find a path!

Give an counter-example and show how Yan Rong's greedy walk will fail to find a path. You can define a particular set of cities and links to support your counter-example, which you can illustrate pictorially like what is shown at the start of this question. [2 marks]

A to C using the routes given at the beginning of the question will result in a dead end: $A \rightarrow D \rightarrow B$.

Question 4: Pokémon! [12 marks]

Professor Oak has modeled some Pokémon as Python classes. He has provided a partial implementation of the `Pokemon` class and two subclasses as follows:

```
class Pokemon:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_name(self, new_name):
        name = new_name

class WaterPokemon(Pokemon):
    def __init__(self, name, speed):
        self.speed = speed
        self.distance = 0

    def swim(self, time):
        self.distance += self.speed * time
        print(self.name + " swam " + str(self.distance) + " meters")

class FlyingPokemon(Pokemon):
    def __init__(self, name):
        super().__init__(name)

    def fly(self):
        print(self.name + " flies")
```

A. After taking a look at the code, Ash complains that he will not be able to change the name of a `WaterPokemon` instance because there are no methods to handle that in the `WaterPokemon` class definition. Do you agree with Ash? Briefly explain your reason. [3 marks]

No. `WaterPokemon` is a subclass of `Pokemon`, thus it inherits the methods of `Pokemon`, which includes `get_name` and `set_name`.

In order to get the full marks, you must mention that the methods of the super class are inherited. Just stating that `WaterPokemon` is a subclass of `Pokemon` is not enough.

Some students correctly noted that the `set_name` method will fail to work because of some errors in the code. The thing is the question specifically states that Ash's reason is that "there are no methods to handle" the changing of name. This is not true as the methods are inherited.

But regardless of whether you agree with Ash or not, you will get full marks as long as you mention that the method `set_name` is inherited. Conversely, saying the wrong thing like the methods are not inherited will get you 0 marks.

B. Misty further demonstrates that she cannot set or get the name of a `WaterPokemon` that she created:

```
>>> squirtle = WaterPokemon('Squirtle', 10)

>>> squirtle.get_name()
AttributeError: 'WaterPokemon' object has no attribute 'name'

>>> squirtle.set_name('Squirty')

>>> squirtle.get_name()
AttributeError: 'WaterPokemon' object has no attribute 'name'
```

Professor Oak must have made some mistakes in his code. Highlight the errors in his code and show how it should be fixed. [3 marks]

1. The init method of `WaterPokemon` should call `super().__init__(name)`.
2. `set_name` method of `Pokemon` should be `self.name = new_name`.

C. Misty would like to define a new class `DragonPokemon` that is both a `WaterPokemon` and `FlyingPokemon`. In addition to the existing parameters, a `DragonPokemon` takes in another string called `colour`. It also has a method breathe which allows it to breathe out a coloured flame, like:

```
>>> gyarados = DragonPokemon("Gyarados ", 20, "blue")
>>> gyarados.breathe()
Gyarados breathes a blue flame
```

Also, a `DragonPokemon` extends the `fly` method with a new behaviour — it will breathe after it flies!

```
>>> gyarados.fly()
Gyarados flies
Gyarados breathes a blue flame
```

Provide an implementation of the `DragonPokemon` class. [6 marks]

```
class DragonPokemon(WaterPokemon, FlyingPokemon):
    def __init__(self, name, speed, colour):
        self.colour = colour
        super().__init__(name, speed)

    def breathe(self):
        print(self.name, "breathes a", self.colour, "flame")

    def fly(self):
        super().fly()
        self.breathe()
```

Question 5: 42 and the Meaning of Life [4 marks]

Either: (a) explain how you think some of what you have learnt in CS1010S will be helpful for you for the rest of your life and/or studies at NUS; or (b) tell us an interesting story about your experience with CS1010S this semester. [4 marks]

The student will be awarded points as long as he/she is coherent and doesn't say something obviously wrong.

— E N D O F P A P E R —

Scratch Paper

Scratch Paper

Scratch Paper

– H A P P Y H O L I D A Y S ! –