FIT1045/FIT1053 Algorithmic Problem Solving – Tutorial 5.
Solutions

Objectives

After this tutorial, you should be able to:

- Give the definition of important graph terms.
- Reason about copying nested objects in Python.
- Identify simple loop invariants.
- Use decomposition to simplify problems.

Prepared Question

1. This can be treated as a *graph* problem.
2. We can use an *edge list* to efficiently represent a spanning tree:
   $(m, a), (m, d), (a, i), (m, r), (r, c), (c, s), (s, l), (s, f), (f, t)$

Magic Squares

i) An example of a 3×3 magic square.

8	1	6
3	5	7
4	9	2

ii) An example of a 4×4 magic square.

1	15	14	4
12	6	7	9
8	10	11	5
13	3	2	16

An algorithm to check if a given $n \times n$ table is a valid magic square:

1. Sum up each entry in the first row, or each entry in the first column, or each entry in one of the two diagonals.
2. Set the sum to a variable, say *magic sum*.
3. Calculate the sum of each row, each column and each diagonal.
4. If the sum of the integers in one of the rows, or sum of the integers in one of the columns, or sum of the integers in one of the two diagonals, is not equal to *magic sum*, the table is not a magic square.
5. Otherwise, the table is a magic square.

Deep Copy

Copying lists with depth exactly 2

An algorithmic solution might look like this:

1. create a new empty list, `new`
2. for every list in the original list, append a new empty list to `new`
3. for every item in the nested lists in the original list, add item to the corrosponding list in `new`
4. return `new`

Implemented in Python might look like this:

```
def deepcopy_exactly_2(lst):
    new = []
    for _ in range(len(lst)):
        new.append([])
    for i in range(len(lst)):
        for j in range(len(lst[i])):
            new[i].append(lst[i][j])
    return new
```

Copying lists with depth at-most 2

An algorithmic solution might look like this:

1. create a new empty list, `new`
2. for every item in the original list, append a `None` object to `new`
3. for every item in the original list, check whether the item is type `list`
    - if item is a list, insert a new list at the correct index in `new` and populate with items from original list
    - if item is not a list, insert a copy of the item at the correct index in `new`
4. return `new`

Implemented in Python might look like this:

```
def deepcopy_exactly_2(lst):
    new = []
    for _ in range(len(lst)):
        new.append(None)
    for i in range(len(lst)):
        if isinstance(lst[i], list):
            new[i] = []
            for j in range(len(lst[i])):
                new[i].append(lst[i][j])
        else:
            new[i] = lst[i]
    return new
```

Decomposition

1. This is one possible solution, not necessarily the only or even the best;

```
def StringsToInts(aList):
    for index in range(aList):
        aList[index] = int(aList[index])
    return aList

def squareList(aList):
    for index in range(len(aList)):
        aList[index] = aList[index]**2
    return aList

def diffs(aList,number):
    diffList = []
    for item in aList:
        diffList.append(item - number)
    return diffList

def sum(aList):
    the_sum=0
    for item in aList:
        the_sum = the_sum + item
    return the_sum

def average(aList):
    return sum(aList)/len(aList)

def variance(aList):
    aList = stringsToInts(aList)
    av = average(aList)
    diffList = diffs(aList,av)
    sqList = squareList(diffList)
    return sum(sqList)
```

Invariants (compulsory for FIT1053)

Let B be the number of black balls and W be the number of white balls in the urn. The following table shows the changes of black and white balls out of three possibilities:

		Before		**After**	
1.	2 white balls are taken out and 1 black ball is returned back	B	W	$B+1$	$W-2$
2.	2 black balls are taken out and 1 black ball is returned back	B	W	$B-1$	W
3.	2 different colour balls are taken out and 1 white ball is returned back	B	W	$B-1$	W

Note that the total number of balls in the urn decreases by 1 after each draw, and the number of white balls can only be decreased by 0 or 2. Hence, the parity of the white balls stays the same.

Based on the invariant, there are two possibilities for the final ball depending on the initial number of white balls:

1. If we start with an even number of white balls, then the final ball is a black ball.
2. If we start with an odd number of white balls, then the final ball is a white ball.

Checkpoint

1. What is the definition of a *tree*?
2. What is the value of `x` and `y` after running the following code block?

```
import copy

table = [[1, 2, 3], [4, 5, 6]]
table_copy = copy.deepcopy(table)
row1 = copy.copy(table[0])
row1[0] = 9
table_copy[1] = row1

x = table
y = table_copy
```

3. What is returned by the following function call? Describe the output, and the purpose of the function.

```
def mystery(adj_graph, v):
    vertices = []
    for i in range(len(adj_graph)):
        if adj_graph[i][v]:
            vertices += [i]
    return vertices

my_graph = \
  [[0, 0, 1, 1],
   [0, 0, 1, 1]
   [1, 1, 0, 0]
   [1, 1, 0, 0]]

mystery(my_graph, 2)
```

4. Write a Python function, `is_adjacent`, that takes an adjacency matrix representing an undirected graph, and two vertices, and returns `True` if the vertices are adjacent, otherwise returns `False`.

Solutions

1. "A *tree* is a graph that is connected and does not contain a cycle." Note: wording may be different, but key underlined concepts should be present.
2. `x == [[1,2,3],[4,5,6]]` and `y == [[1,2,3],[9,2,3]]`
3. There are multiple correct solutions. If we use the additional rule of always choosing the lowest available vertex as our next extension, we would get: `[(0,1), (1,2), (2,3), (3,4)]`.
4. `[0,1]`. The function returns a list of all adjacent vertices.
5.
```
def is_adjacent(adj_graph, v1, v2):
    return bool(adj_graph[v1][v2])
```