Ryan Schulman
Design and Analysis of Algorithms
HW 2

Problem 1)

a) Proof:

We know the sum of all numbers in a magic square is

$$Sn = \sum_{i=1}^{n^2} i = \frac{n^2(n^2+1)}{2}$$

where n is the side length of the magic squre.

We know each row, column, and diagonal of a magic square has n elements. We also know that the sum of all elements in the square must be distributed evenly through each row and column.. In other words, the sum of each row Sn must be equivalent to the sum of each row sum. So Sn = sum(row 1) + sum(row 2) + sum(row 3) …. + sum(row n). Since each of these sums is equivalent, we have that Sn = n * sum(row n). So, we see sum(row n) = Sn / n or

$$sum(rown) = \frac{n^2(n^2+1)}{2n} = \frac{n(n^2+1)}{2}$$

b)

This function accepts input n, cur_val, cur_arr, and good_arrs. And outputs an array of arrays M_n where the arrays of M_n have length n^2 and represent a "flattened" magic square.Cur_val can be any integer from 1 to n^2, cur_arr should be an empty array, and good_arrs should be an empty array of arrays.

```
getMagicSquare(n, cur_val, cur_arr, good_arrs)
If cur_val not in cur_arr DO
    If cur_val != 0 DO
        cur_arr.append(cur_val)
    END
    For x from cur_val + 1 to n*n
        If length(cur_arr) == n*n
            If check(cur_arr)
                good_arrs.append(cur_arr)
        Else
            getMagicSquare(n, x, copy of cur_arr, good_arrs)
        END
    END
END
```

c) Analysis

Primary operator => n*n in the if statement

$$T(n) = \sum_{i=1}^{n^2} T(n-1)$$

if n > 2
T(n) = d if n <= 2

$$T(n) = \sum_{i=1}^{n^2} T(n-1) = T(n-1) * \sum_{i=1}^{n^2} 1 = n^2 T(n-1)$$

$T(n-1) = (n-1)^2T(n-2)$

$\Rightarrow T(n) = n^2(n-1)^2T(n-2)$

$T(n-2) = (n-2)^2(n-3)^2T(n-4)$

$\Rightarrow T(n) = n^2(n-1)^2(n-2)^2(n-3)^2T(n-4)$

.
.
.

After k steps

$$T(n) = \frac{n^2!}{(n-k)^2!}T(n-k)$$

Assume n-k = 1

$\Rightarrow T(n) = n^2!T(1) = n^2! * d$

$\Rightarrow \Theta(n^2!)$

d)

```
#!/usr/bin/python
import timeit
import numpy as np
import math
import  matplotlib.pyplot as plt
import pdb
testArr = [2,7,6,9,5,1,4,3,8]

def check(arr, n):
    magicNumber = n * (n * n + 1) / 2.0
    # print "magicNumber", magicNumber
    # print arr
    # should_pass = False
```

```
    # if testArr == arr:
    #     print "should have passed"
    #     should_pass = True
        # pdb.set_trace()
    for x in range(n):
        # check rows
        summ = np.sum(arr[x*n:x*n + n])
        if summ != magicNumber:
            return False
        # check columns
        tempArr = [arr[y] for y in range(x, n*n, n)]
        summ = np.sum(tempArr)
        if summ  != magicNumber:
            return False
    # check diagonals
    tempArr = [arr[x] for x in range(0, n*n, n+1)]
    summ = np.sum(tempArr)
    if summ != magicNumber:
        return False
    tempArr = [arr[x] for x in range(n-1, n*n - n + 1, n-1)]
    summ = np.sum(tempArr)
    if summ != magicNumber:
        return False
    return True



def getMagicSquares(n, good_arrs, cur_val=0, cur_arr=[]):
    # print cur_arr
    # print cur_val
    if cur_val not in cur_arr:
        if cur_val != 0:
            cur_arr.append(cur_val)
        # print "cur_arr", cur_arr
        # print "depth: ", len(cur_arr)
        # print "cur_val", cur_val
        # print "range", range(cur_val + 1, n, 1)
        rr = range(1, n*n + 1, 1)
        # print rr
        if len(cur_arr) == n*n:
            # print "found right length"
            if check(cur_arr, n):
                print "passed check"
                print cur_arr
                good_arrs.append(cur_arr)
```

```
        else:
            for x  in rr:
                getMagicSquares(n, good_arrs, x, list(cur_arr))

if __name__ == "__main__":
    good_arrs = []
    n = 3
        # pdb.set_trace()
    getMagicSquares(n, good_arrs, cur_arr=[])
    print good_arrs
```

e)
3x3 check:
[6, 1, 8,
7, 5, 3,
2, 9, 4]
Do rows = 15? Yes
Do columns = 15? Yes
Do diagonals = 15? Yes

4x4 check:
[16, 5, 4, 9,
3, 10, 15, 6,
13, 8, 1, 12,
2, 11, 14, 17]
Do rows =34? Yes
Do columns = 34? Yes
Do diagonals = 34? Yes

Problem 2)

a)

Pseudo-code:

Input: G = graph of nodes and edges, n = number of nodes, good_path = pointer to output

Output: good_path = data now pointed to by input pointed

```
minimum <- 99999999

getPath(G, n, good_path, curPath <- [], curNode<-None):
    if curNode is None DO
        for node in G.nodes:
            getPath(G, n, good_path, curPath <- [], curNode<-node)
         END
    else:
        if curNode not in curPath DO
            curPath.append(curNode)
        else:
            return
        END
        for edge in G.edges(curNode) DO
            node <- edge[1]
            if lengeth(curPath) == n DO
                tmp <- path_len(curPath, G)
                if tmp < minimum DO
                    minimum <- tmp
                    good_path.append(curPath)
                    while len(good_path) > 1 DO
                        good_path.pop(0)
                    END
                END
            else:
                getPath(G, n, good_path, curPath<-list(curPath), curNode<-node)
            END
         END
```

b)

$$T(n) = \sum_{i=0}^{n-1} T(n-1) = (n-1)T(n-1)$$

… n > 2

T(2) = d … n = 2

T(n-1) = (n-2)T(n-2)

=> T(n) = (n-1)(n-2)T(n-2)

T(n-2) = (n-3)(n-4)T(n-4)

=> T(n) = (n-1)(n-2)(n-3)(n-4)T(n-4)

… After k steps

=> $T(n) = (n-1)!T(n-k)$
Assume n-k = 2
=> $T(n) = (n-1)!T(2) = (n-1)! * d$
=> $T(n) = \theta((n-1)!)$
c)
```
#!/usr/bin/python
import timeit
import numpy as np
import math
import  matplotlib.pyplot as plt
import pdb
import networkx as nx
import random
import pprint

minimum=99999999

def path_len(path, G):
    # pdb.set_trace()
    result = 0
    length = len(path)
    for x in range(0, length - 1, 1):
        cost = G[path[x]][path[x+1]]["weight"]
        result += cost
    cost = G[path[length-1]][path[0]]["weight"]
    result += cost
    return cost
def getPath(G, n, good_path, curPath = [], curNode=None):
    global minimum
    if curNode is None:
        for node in G.nodes():
            getPath(G, n, good_path, curPath = [],
curNode=node)
    else:
        if curNode not in curPath:
            # print "curNode", curNode
            curPath.append(curNode)
        else:
            return
        for edge in G.edges(curNode):
            node = edge[1]
```

```
            if len(curPath) == n:
                tmp = path_len(curPath, G)
                # print curPath, ":",tmp
                if tmp < minimum:
                    minimum = tmp
                    good_path.append(curPath)
                    while len(good_path) > 1:
                        good_path.pop(0)
            else:
                getPath(G, n, good_path,
curPath=list(curPath), curNode=node)

if __name__ == "__main__":
    good_path = []
    G = nx.Graph()
    labels = {}
    n = 10
    numEdges = (n**2)/2.0 - 1
    for x in range(n):
        for y in range(n):
            if y != x:

G.add_edge(x,y,weight=random.randint(1,10))
    getPath(G, n, good_path)
    print "good_path", good_path
    form =  pprint.pformat(dict(G.adj))
    f = open('cities.txt', 'w')
    print form
    f.write(form)
    pos = nx.shell_layout(G, scale=10)
    nx.draw(G, pos, with_labels=True)
    edge_labels = nx.get_edge_attributes(G, 'r')
    # plt.plot(nx.draw_shell(G, with_labels=True))
    nx.draw_networkx_edge_labels(G, pos, labels=edge_labels)
    plt.title("Good path"+str(good_path))
    plt.show()
    # pdb.set_trace()
```

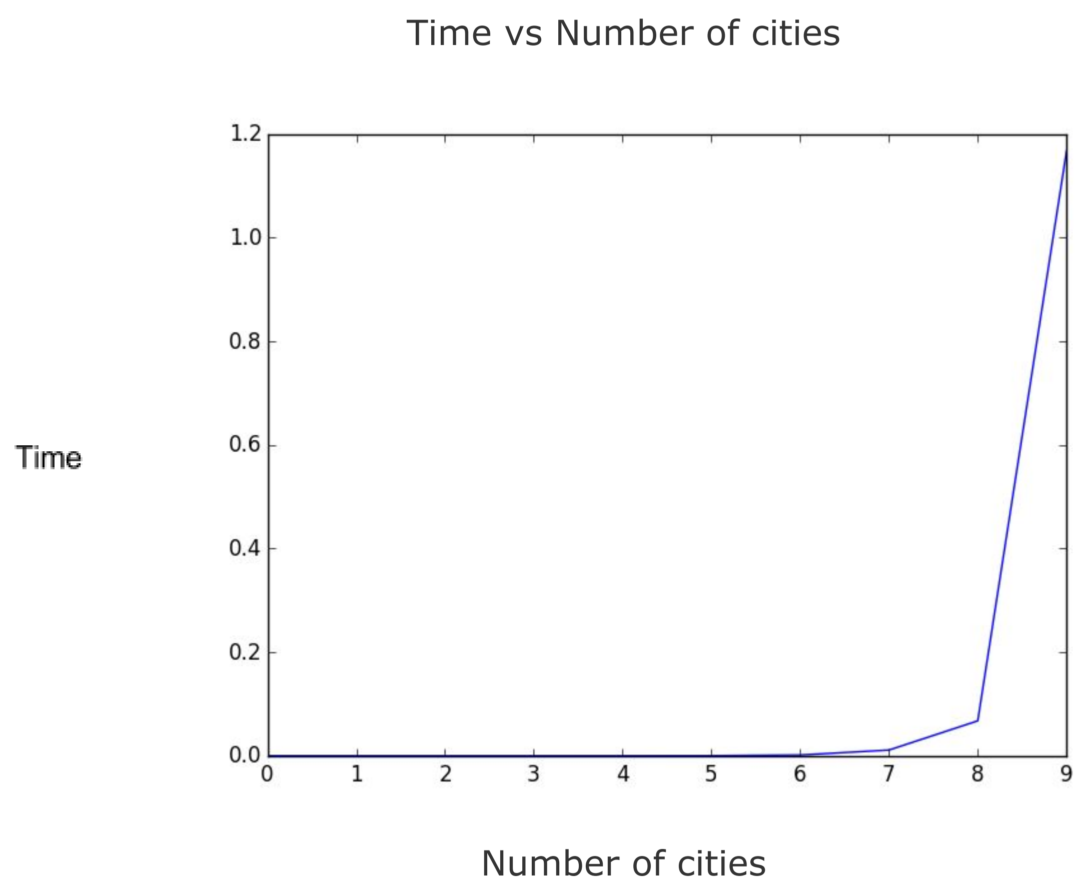
d)

Generated cities:

```
{0: AtlasView({1: {'weight': 8}, 2: {'weight': 7}, 3: {'weight': 1}, 4:
{'weight': 2}, 5: {'weight': 2}, 6: {'weight': 6}, 7: {'weight': 5}, 8:
{'weight': 8}, 9: {'weight': 10}}),
 1: AtlasView({0: {'weight': 8}, 2: {'weight': 10}, 3: {'weight': 4}, 4:
{'weight': 4}, 5: {'weight': 9}, 6: {'weight': 7}, 7: {'weight': 5}, 8:
{'weight': 8}, 9: {'weight': 2}}),
 2: AtlasView({0: {'weight': 7}, 1: {'weight': 10}, 3: {'weight': 9}, 4:
{'weight': 4}, 5: {'weight': 3}, 6: {'weight': 5}, 7: {'weight': 2}, 8:
{'weight': 4}, 9: {'weight': 8}}),
 3: AtlasView({0: {'weight': 1}, 1: {'weight': 4}, 2: {'weight': 9}, 4:
{'weight': 7}, 5: {'weight': 5}, 6: {'weight': 2}, 7: {'weight': 1}, 8:
{'weight': 1}, 9: {'weight': 1}}),
 4: AtlasView({0: {'weight': 2}, 1: {'weight': 4}, 2: {'weight': 4}, 3:
{'weight': 7}, 5: {'weight': 1}, 6: {'weight': 2}, 7: {'weight': 3}, 8:
{'weight': 4}, 9: {'weight': 9}}),
 5: AtlasView({0: {'weight': 2}, 1: {'weight': 9}, 2: {'weight': 3}, 3:
{'weight': 5}, 4: {'weight': 1}, 6: {'weight': 8}, 7: {'weight': 7}, 8:
{'weight': 9}, 9: {'weight': 2}}),
 6: AtlasView({0: {'weight': 6}, 1: {'weight': 7}, 2: {'weight': 5}, 3:
{'weight': 2}, 4: {'weight': 2}, 5: {'weight': 8}, 7: {'weight': 8}, 8:
{'weight': 5}, 9: {'weight': 8}}),
 7: AtlasView({0: {'weight': 5}, 1: {'weight': 5}, 2: {'weight': 2}, 3:
{'weight': 1}, 4: {'weight': 3}, 5: {'weight': 7}, 6: {'weight': 8}, 8:
{'weight': 2}, 9: {'weight': 6}}),
 8: AtlasView({0: {'weight': 8}, 1: {'weight': 8}, 2: {'weight': 4}, 3:
{'weight': 1}, 4: {'weight': 4}, 5: {'weight': 9}, 6: {'weight': 5}, 7:
{'weight': 2}, 9: {'weight': 6}}),
 9: AtlasView({0: {'weight': 10}, 1: {'weight': 2}, 2: {'weight': 8}, 3:
{'weight': 1}, 4: {'weight': 9}, 5: {'weight': 2}, 6: {'weight': 8}, 7:
{'weight': 6}, 8: {'weight': 6}})}
```

Program output:

```
good_path [[0, 1, 2, 4, 5, 6, 7, 8, 9, 3]]
```


Time vs Number of cities
Time
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
7
8
9
Number of cities

e)
Graph of cities: