Universidad Nacional de Colombia

Algoritmos

Taller 02

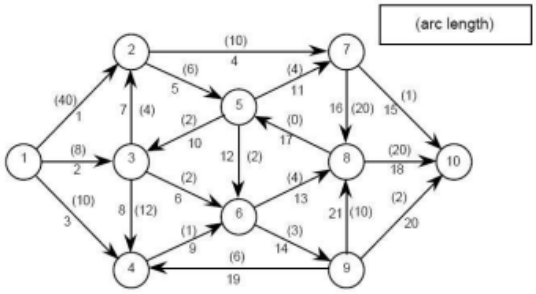
Autor:
Edgar David Garay Forero

Profesor:
German Hernandez

Miercoles, 06 de Noviembre de 2018

1. Considere el grafo mostrado en la figura

Figura 1: Grafo para el punto 1



a) ejecute el algoritmo de dijkstra y muestre los pasos ejecutados

1) Inicializamos todas las distancias en D con un valor infinito debido a que son desconocidas al principio, la de el nodo start se debe colocar en 0 debido a que la distancia de start a start sería 0.
2) Sea a = start (tomamos a como nodo actual).
3) Visitamos todos los nodos adyacentes de a, excepto los nodos marcados, llamaremos a estos nodos no marcados vi.
4) Para el nodo actual, calculamos la distancia desde dicho nodo a sus vecinos con la siguiente fórmula: dt(vi) = Da + d(a,vi). Es decir, la distancia del nodo 'vi' es la distancia que actualmente tiene el nodo en el vector D más la distancia desde dicho el nodo 'a' (el actual) al nodo vi. Si la distancia es menor que la distancia almacenada en el vector, actualizamos el vector con esta distancia tentativa. Es decir:

   newDuv =
   D[u] + G[u][v]
   if newDuv ¡ D[v]:
   P[v] = u
   D[v] = newDuv
   updateheap(Q,D[v],v)

5) Marcamos como completo el nodo a.
6) Tomamos como próximo nodo actual el de menor valor en D (lo hacemos almacenando los valores en una cola de prioridad) y volvemos al paso 3 mientras existan nodos no marcados.
   1:0,2:12,3:8,4:10,5:14,6:10,7:18,8:14,9:13,10:15
   1:inf,2:0,3:8,4:17,5:6,6:8,7:10,8:12,9:11,10:11
   1:inf,2:4,3:0,4:11,5:6,6:2,7:10,8:6,9:5,10:7
   1:inf,2:11,3:7,4:0,5:5,6:1,7:9,8:5,9:4,10:6

1:inf,2:6,3:2,4:11,5:0,6:2,7:4,8:6,9:5,10:5
1:inf,2:10,3:6,4:9,5:4,6:0,7:8,8:4,9:3,10:5
1:inf,2:26,3:22,4:31,5:20,6:22,7:0,8:20,9:25,10:1
1:inf,2:6,3:2,4:11,5:0,6:2,7:4,8:0,9:5,10:5
1:inf,2:16,3:12,4:6,5:10,6:7,7:14,8:10,9:0,10:2
1:inf,2:inf,3:inf,4:inf,5:inf,6:inf,7:inf,8:inf,9:inf,10:0

b) ejecute el algoritmo de bellman-ford y muestre los pasos ejecutados

1) Inicializamos el grafo. Ponemos distancias a INFINITO menos el nodo origen que tiene distancia 0.
2) Tenemos un diccionario de distancias finales y un diccionario de padres.
3) Visitamos cada arista del grafo tantas veces como número de nodos -1 haya en el grafo
4) comprobamos si hay ciclos negativo.
5) La salida es una lista de los vértices en orden de la ruta más corta

```
{1: 0, 2: 12, 3: 8, 4: 10, 5: 14, 6: 10, 7: 18, 8: 14, 9: 13, 10: 15}
{1: inf, 2: 0, 3: 8, 4: 17, 5: 6, 6: 8, 7: 10, 8: 12, 9: 11, 10: 11}
{1: inf, 2: 4, 3: 0, 4: 11, 5: 6, 6: 2, 7: 10, 8: 6, 9: 5, 10: 7}
{1: inf, 2: 11, 3: 7, 4: 0, 5: 5, 6: 1, 7: 9, 8: 5, 9: 4, 10: 6}
{1: inf, 2: 6, 3: 2, 4: 11, 5: 0, 6: 2, 7: 4, 8: 6, 9: 5, 10: 5}
{1: inf, 2: 10, 3: 6, 4: 9, 5: 4, 6: 0, 7: 8, 8: 4, 9: 3, 10: 5}
{1: inf, 2: 26, 3: 22, 4: 31, 5: 20, 6: 22, 7: 0, 8: 20, 9: 25, 10: 1}
{1: inf, 2: 6, 3: 2, 4: 11, 5: 0, 6: 2, 7: 4, 8: 0, 9: 5, 10: 5}
{1: inf, 2: 16, 3: 12, 4: 6, 5: 10, 6: 7, 7: 14, 8: 10, 9: 0, 10: 2}
{1: inf, 2: inf, 3: inf, 4: inf, 5: inf, 6: inf, 7: inf, 8: inf, 9: inf, 10: 0}
```

c) ejecute el algoritmo de floyd-warshal y muestre los pasos ejecutados

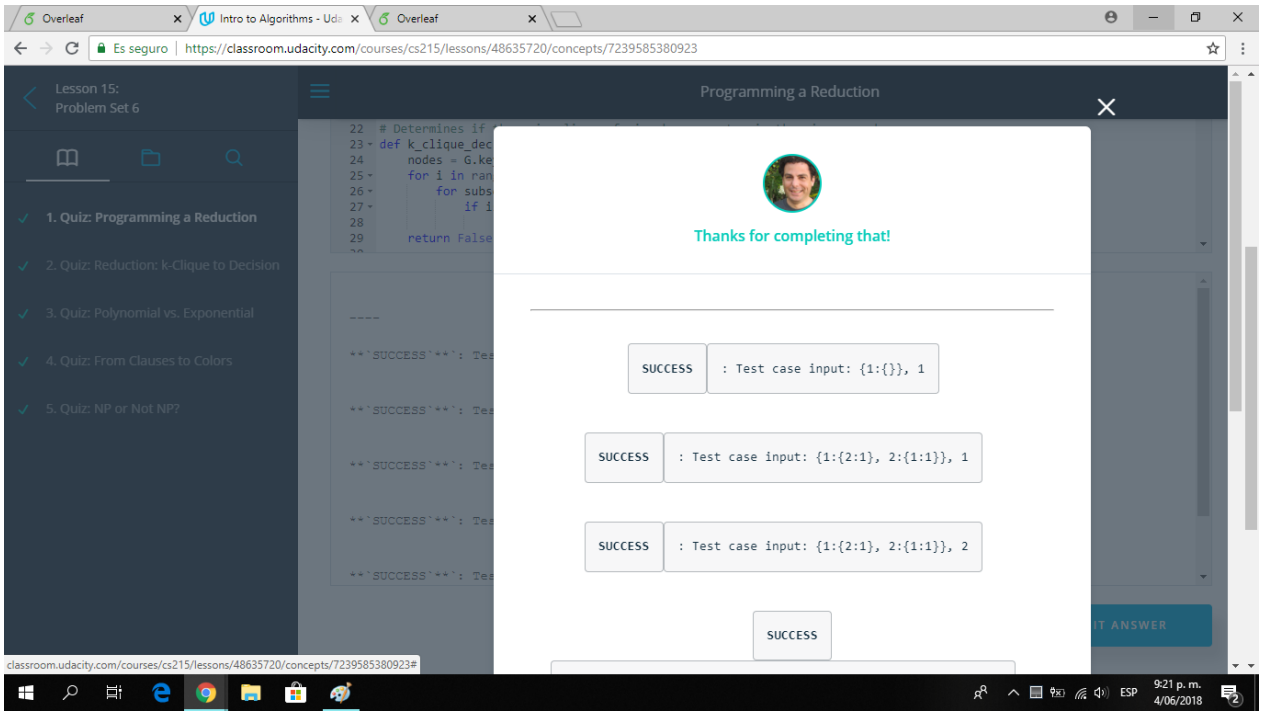
1) Formar las matrices iniciales C y D.
2) Se toma k=1.
3) Se selecciona la fila y la columna k de la matriz C y entonces, para i y j, con i $\neq$ k, j $\neq$ k e i$\neq$j, hacemos:
4) Si (Cik + Ckj) $<$ Cij , Dij = Dkj y Cij = Cik + Ckj
5) En caso contrario, dejamos las matrices como están.
6) Si k $<=$ n, aumentamos k en una unidad y repetimos el paso anterior, en caso contrario paramos las iteraciones.
7) La matriz final C contiene los costes óptimos para ir de un vértice a otro, mientras que la matriz D contiene los penúltimos vértices de los caminos óptimos que unen dos vértices, lo

cual permite reconstruir cualquier camino óptimo para ir de un vértice a otro.

```
   1   10   3   2   5   4   7   6   9   8
('1', [0, 15, 8, 12, 14, 10, 18, 10, 13, 14])
('10', [inf, 0, inf, inf, inf, inf, inf, inf, inf, inf])
('3', [inf, 7, 0, 4, 6, 11, 10, 2, 5, 6])
('2', [inf, 11, 8, 0, 6, 17, 10, 8, 11, 12])
('5', [inf, 5, 2, 6, 0, 11, 4, 2, 5, 6])
('4', [inf, 6, 7, 11, 5, 0, 9, 1, 4, 5])
('7', [inf, 1, 22, 26, 20, 31, 0, 22, 25, 20])
('6', [inf, 5, 6, 10, 4, 9, 8, 0, 3, 4])
('9', [inf, 2, 12, 16, 10, 6, 14, 7, 0, 10])
('8', [inf, 5, 2, 6, 0, 11, 4, 2, 5, 0])
```

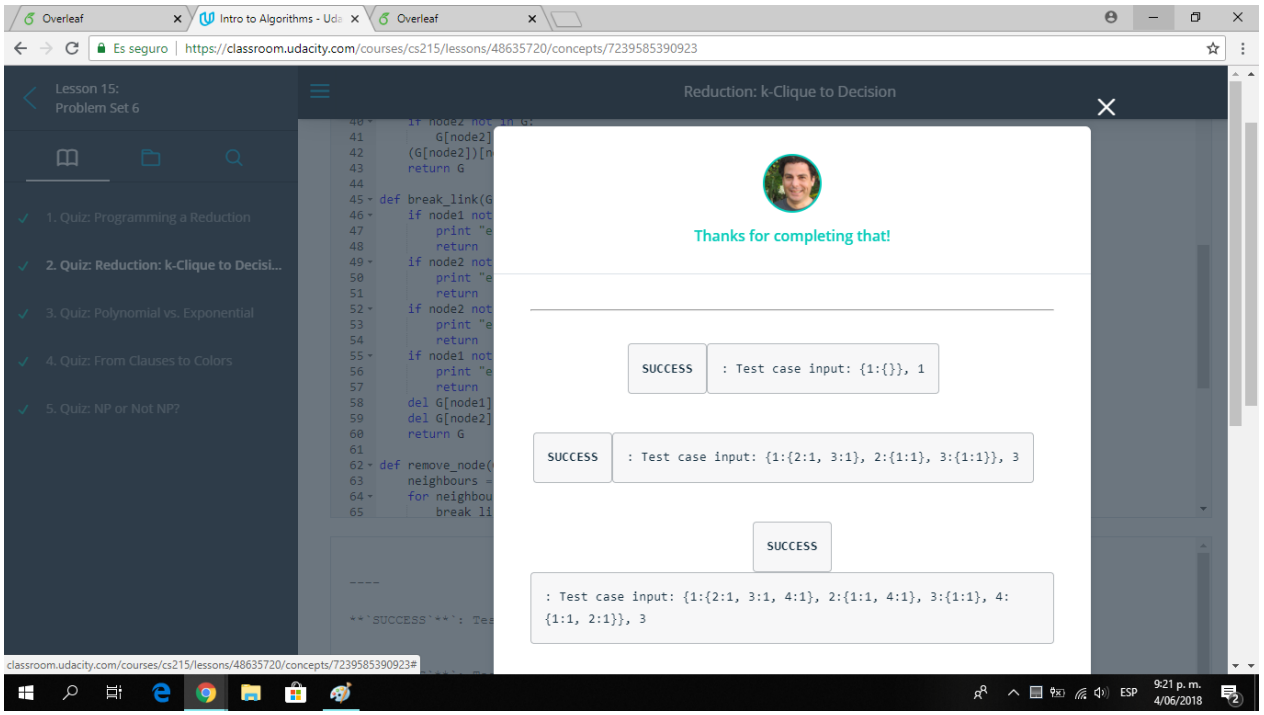
2. resuelva los ejercicios del problem set 6 de udacity

   a) programming a reduction



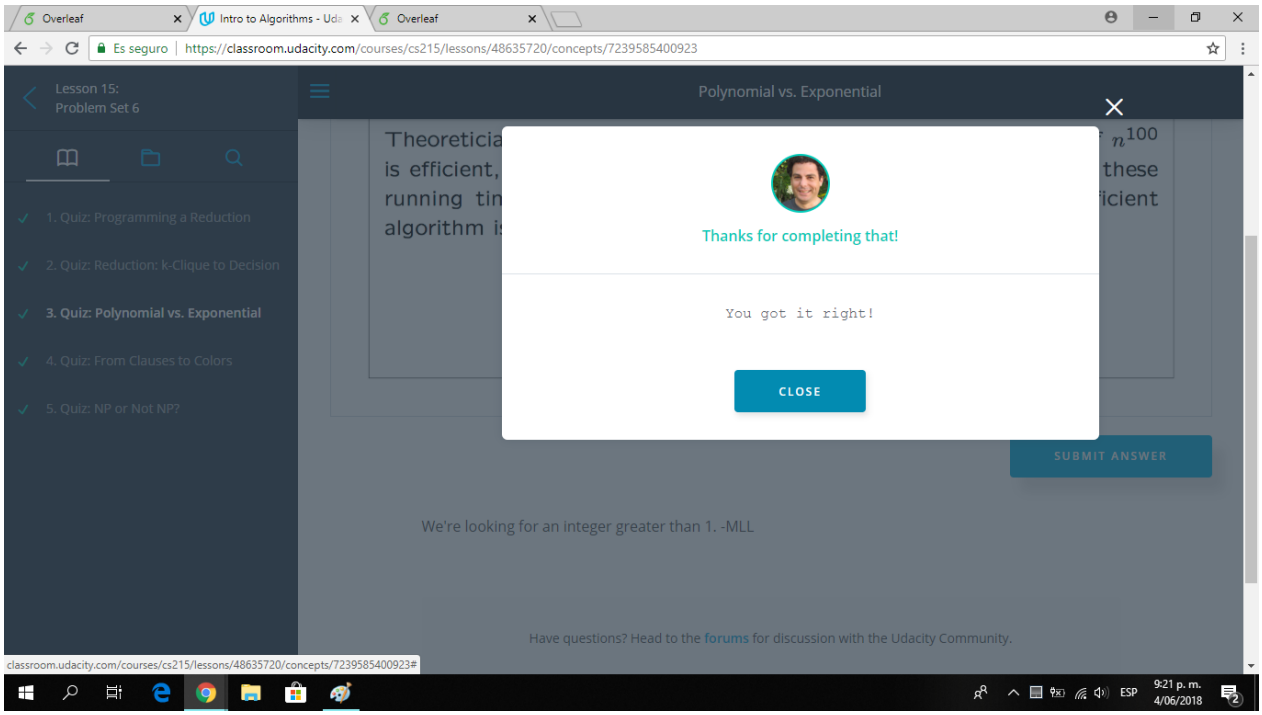

```
def independent_set_decision(H, s):
        # your code here
        G = {}
        all_nodes = H.keys()
        for v in H.keys():
                G[v] = {}
                for other in list(set(all_nodes) - set(H[v].keys()) - se
                        G[v][other] = 1
        print G
        return k_clique_decision(G, s)
def test():
        H={}
        edges = [(1,2), (1,4), (1,7), (2,3), (2,5), (3,5), (3,6), (5,6),
        for u,v in edges:
                make_link(H,u,v)
        for i in range(1,8):
                print(i, independent_set_decision(H, i))
test()
```

b) Reduction: k-Clique to Decision

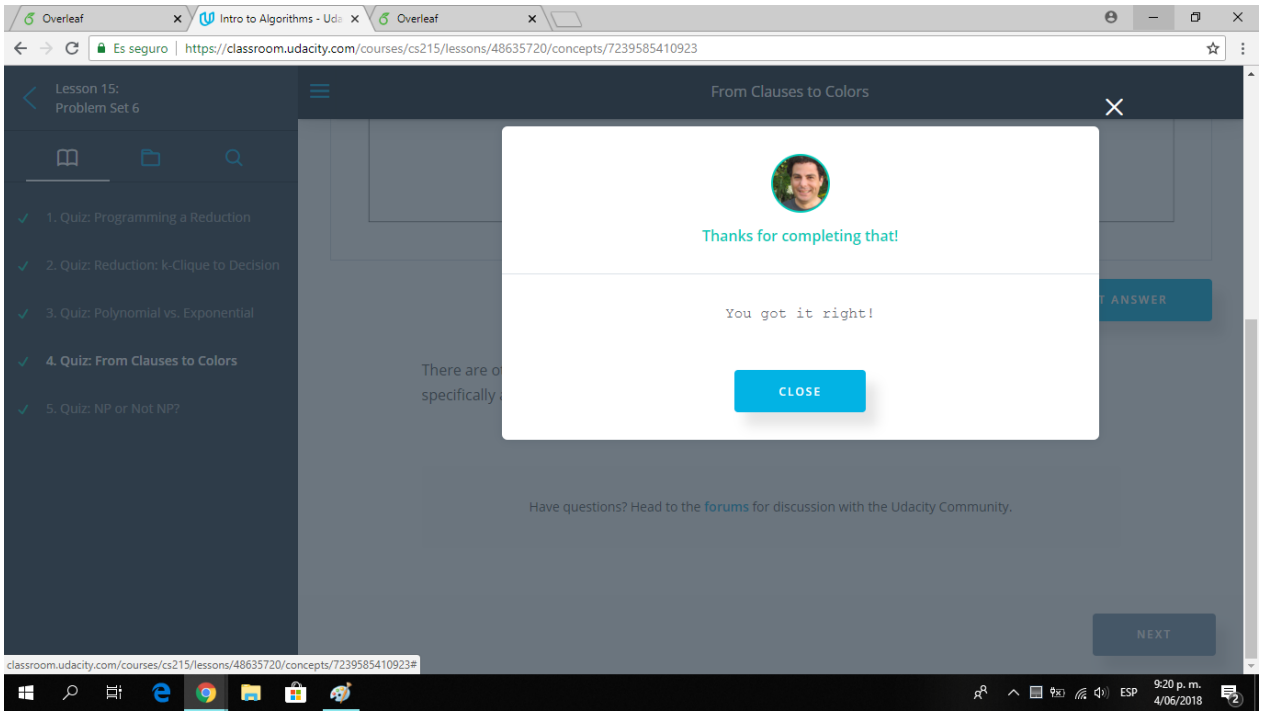



```
def k_clique(G, k):
        k = int(k)
        if not k_clique_decision(G, k):
        #your code here
                return False
        if k == 1:
                return [G.keys()[0]]
        for node1 in G.keys():
                for node2 in G[node1].keys():
                        G = break_link(G, node1, node2)
                        if not k_clique_decision(G, k):
                                G = make_link(G, node1, node2)
        for node in G.keys():
                if len(G[node]) == 0:
                        del G[node]
         return G.keys()
```

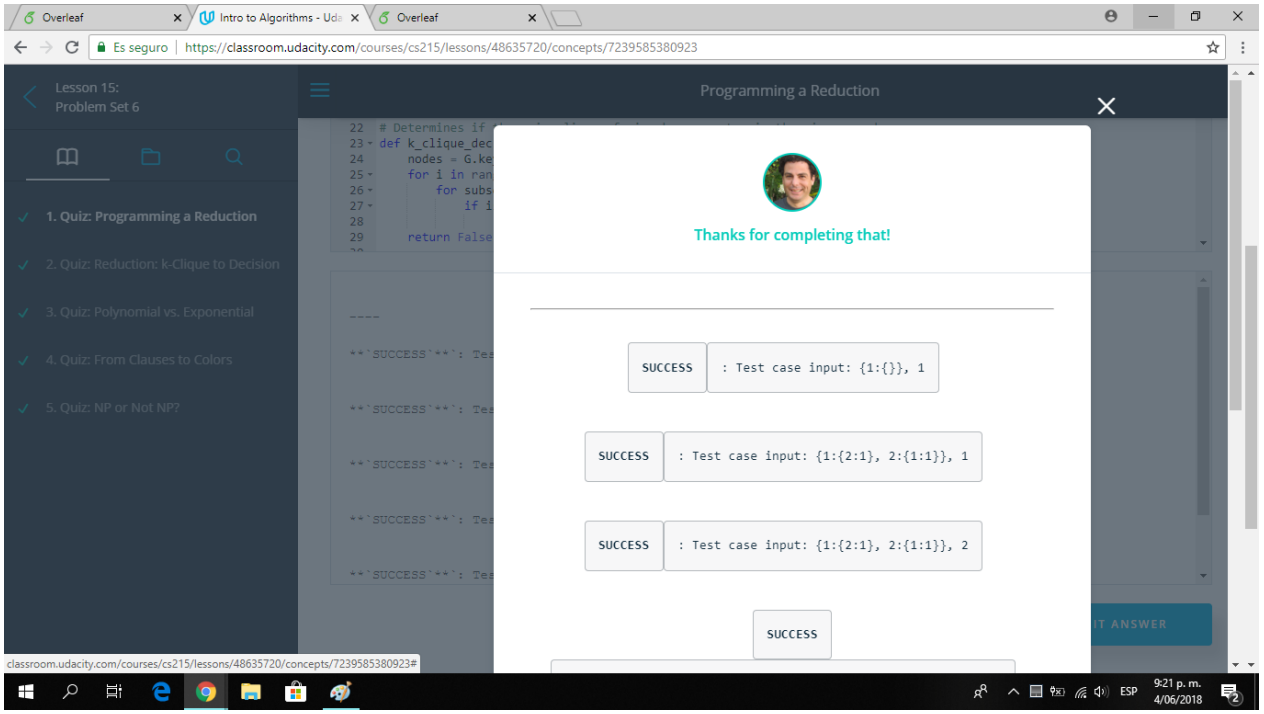
c) Polynomial vs. Exponential



d) from clauses to colors

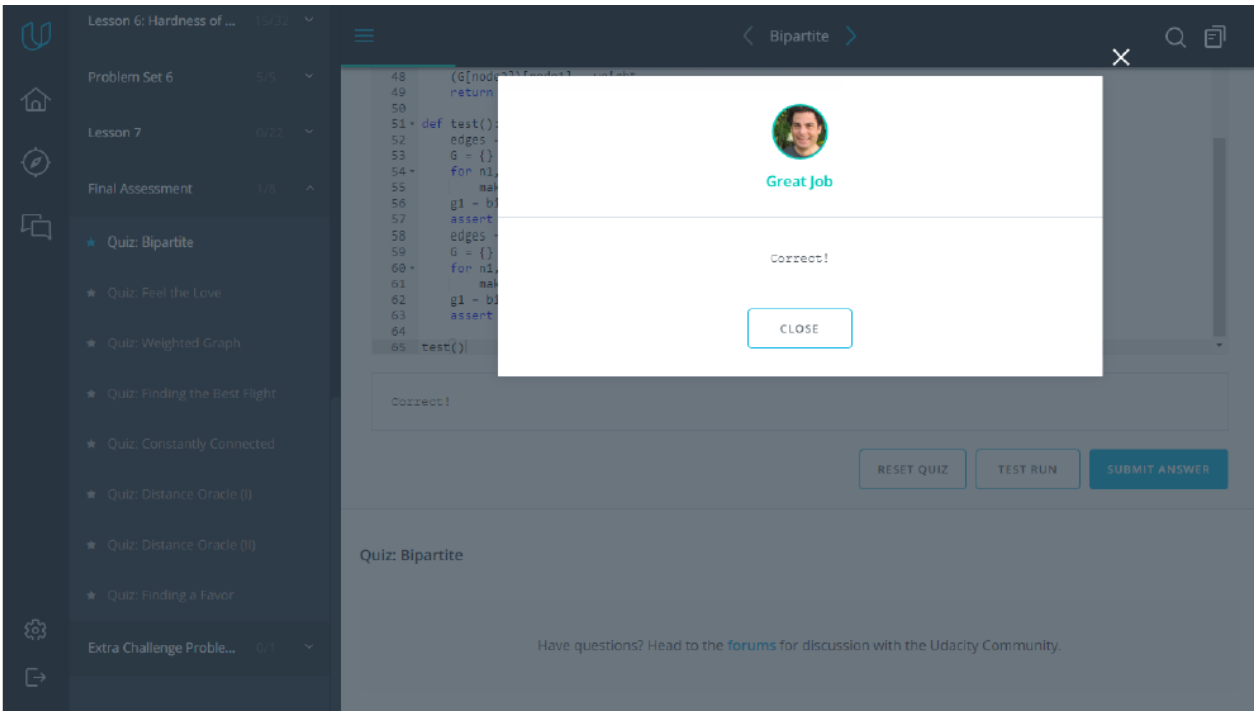


e) NP or not NP



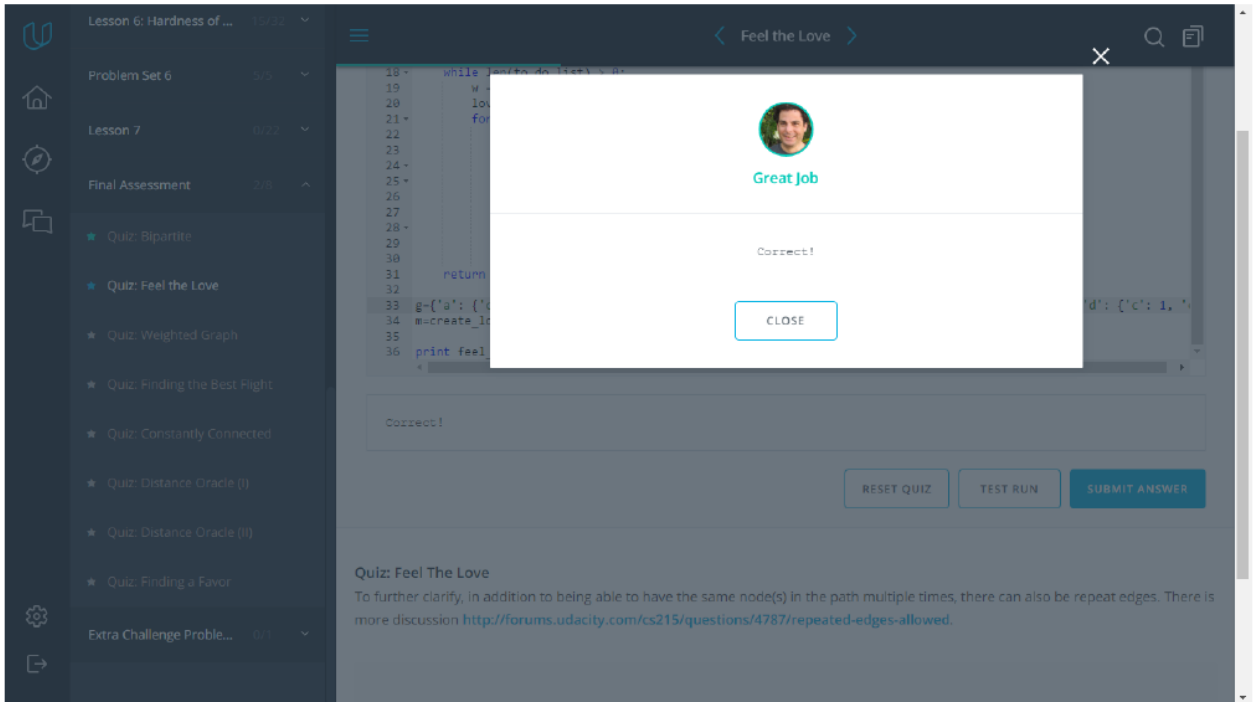
3. Resuelva los puntos del Final Exam del curso Algorithms de Udacity. Incluya el código correspondiente con un screenshot de aceptación para cada problema.

   a) bipartite




```
def bipartite(G):
    # your code here
    # return a set
    if len(G) < 2:
        return None
    if len(G) == 2:
        return set([G.keys()[0]])
    nodes = G.keys()
    a = []
    b = []
    for n in nodes:
        if not any([t for t in a if t in G[n]]):
            a = a + [n]
        else:
            if not any([t for t in b if t in G[n]]):
                b = b + [n]
        if (n not in a) and (n not in b):
            return None
    return set(a)
```

b) Feel the Love

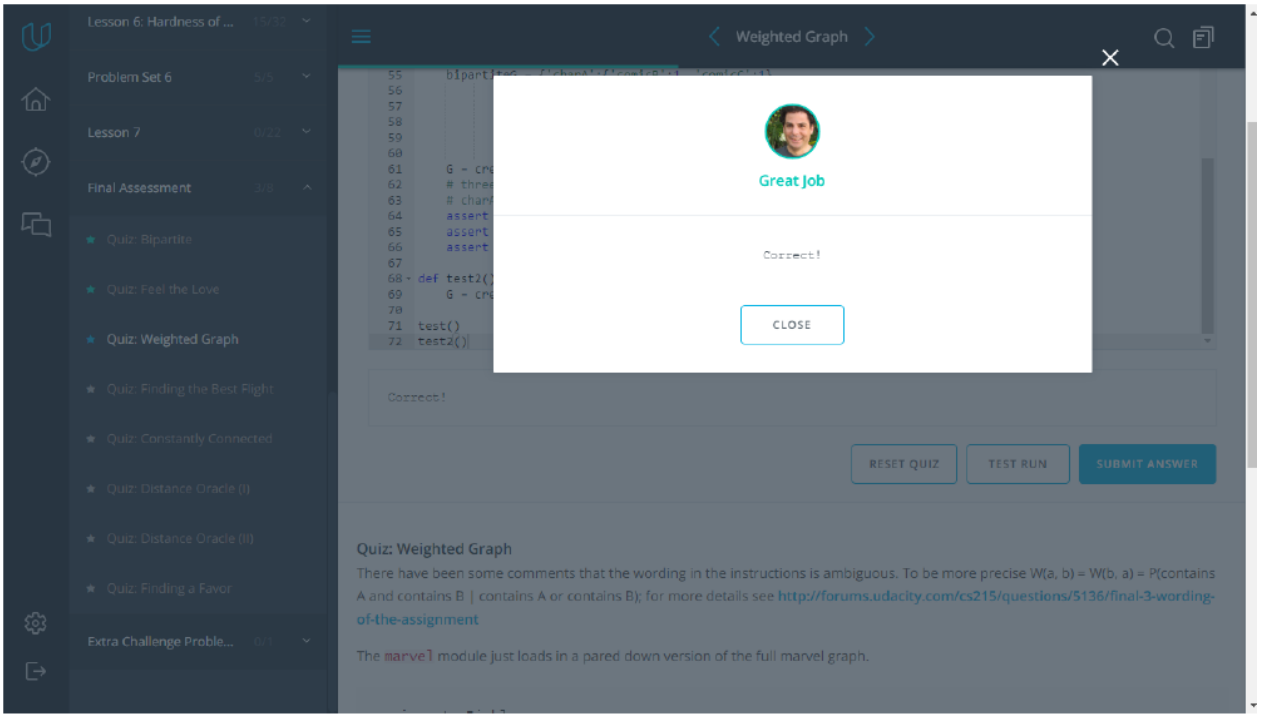



```
# Take a weighted graph representing a social network where the weight
# between two nodes is the "love" between them.  In this "feel the
# love of a path" problem, we want to find the best path from node `i`
# and node `j` where the score for a path is the maximum love of an
# edge on this path. If there is no path from `i` to `j` return
# `None`.  The returned path doesn't need to be simple, ie it can
# contain cycles or repeated vertices.
#
# Devise and implement an algorithm for this problem.
#

def bfs(G, node):

    visited = []
    queue = [node]
    path = { node : [node] }

    while len(queue) > 0:
        node = queue.pop()

        for t in G[node]:
```

```
            if t not in visited:
                path[t] = path[node] + [t]
                queue.append(t)

        if node not in visited:
            visited.append(node)

    return visited, path

def longest_edge(G, nodes):
    e = None
    for n1 in nodes:
        for n2 in G[n1]:
            if n2 in nodes:
                if e == None or G[n1][n2] > G[e[0]][e[1]]:
                    e = (n1, n2)
    return e

def feel_the_love(G, i, j):
    # return a path (a list of nodes) between `i` and `j`,
    # with `i` as the first node and `j` as the last node,
    # or None if no path exists
    nodes, paths = bfs(G, i)
    if j in nodes:
        e = longest_edge(G, nodes)
        tailnodes, tailpaths = bfs(G, e[1])
        path = paths[e[0]] + tailpaths[j]
        return path
    return None

#########
#
# Test

def score_of_path(G, path):
    max_love = -float('inf')
    for n1, n2 in zip(path[:-1], path[1:]):
        love = G[n1][n2]
        if love > max_love:
```

```
                max_love = love
        return max_love

    def test():
        G = {'a':{'c':1},
             'b':{'c':1},
             'c':{'a':1, 'b':1, 'e':1, 'd':1},
             'e':{'c':1, 'd':2},
             'd':{'e':2, 'c':1},
             'f':{}}
        path = feel_the_love(G, 'a', 'b')
        assert score_of_path(G, path) == 2

        path = feel_the_love(G, 'a', 'f')
        assert path == None

    test()
```

c) Weighted Graph

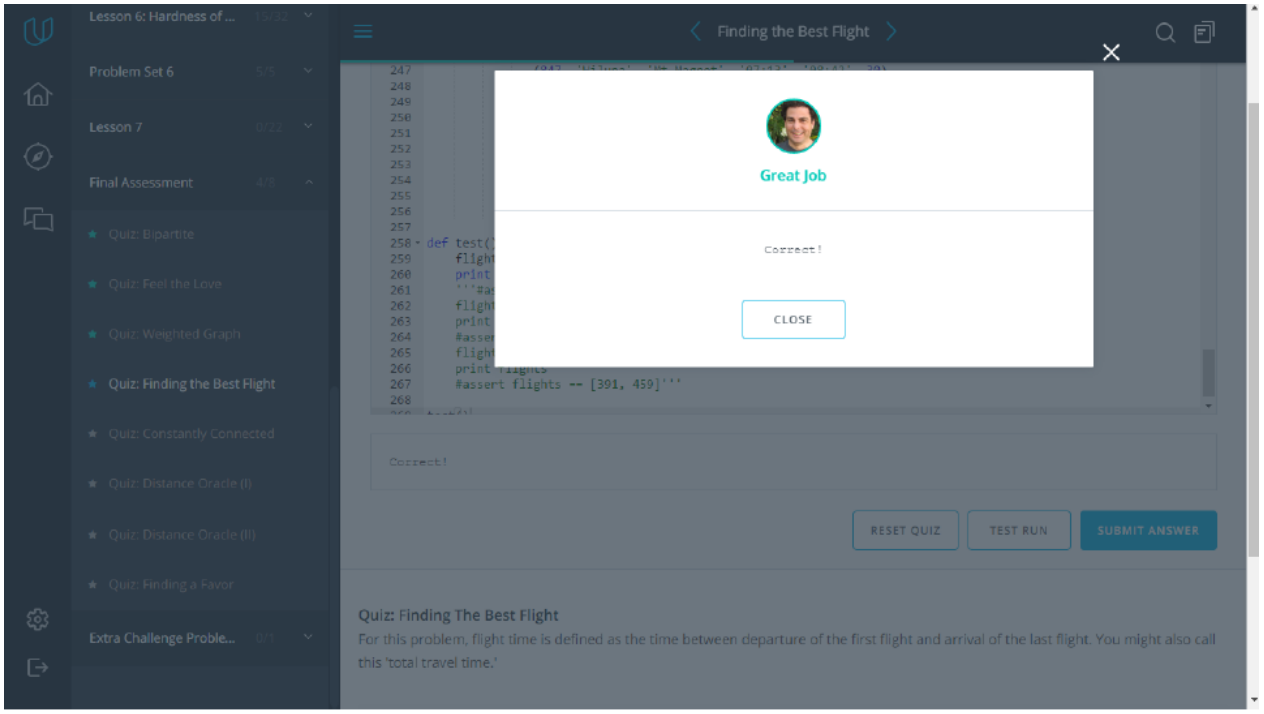



```
#
# In lecture, we took the bipartite Marvel graph,
```

```
# where edges went between characters and the comics
# books they appeared in, and created a weighted graph
# with edges between characters where the weight was the
# number of comic books in which they both appeared.
#
# In this assignment, determine the weights between
# comic book characters by giving the probability
# that a randomly chosen comic book containing one of
# the characters will also contain the other
#

from marvel import marvel, characters
from collections import defaultdict

def create_weighted_graph(graph, characters):
    G = defaultdict(dict)

    for hero in characters:
        for comic in graph[hero]:
            for other in graph[comic]:
                if hero > other:
                    separate_movies = set(graph[hero].keys() + graph[other].keys
                    together_movies = set(graph[hero].keys()) & set(graph[other]
                    weight = 1.0 * len(together_movies) / len(separate_movies)
                    G[hero][other] = weight
                    G[other][hero] = weight
    return G

######
#
# Test

def test():
    bipartiteG = {'charA':{'comicB':1, 'comicC':1},
                  'charB':{'comicB':1, 'comicD':1},
                  'charC':{'comicD':1},
                  'comicB':{'charA':1, 'charB':1},
                  'comicC':{'charA':1},
                  'comicD': {'charC':1, 'charB':1}}
```

```
    G = create_weighted_graph(bipartiteG, ['charA', 'charB', 'charC'])
    # three comics contain charA or charB
    # charA and charB are together in one of them
    assert G['charA']['charB'] == 1.0 / 3
    assert G['charA'].get('charA') == None
    assert G['charA'].get('charC') == None

def test2():
    G = create_weighted_graph(marvel, characters)

test()
test2()
```

d) Finding the best Flight



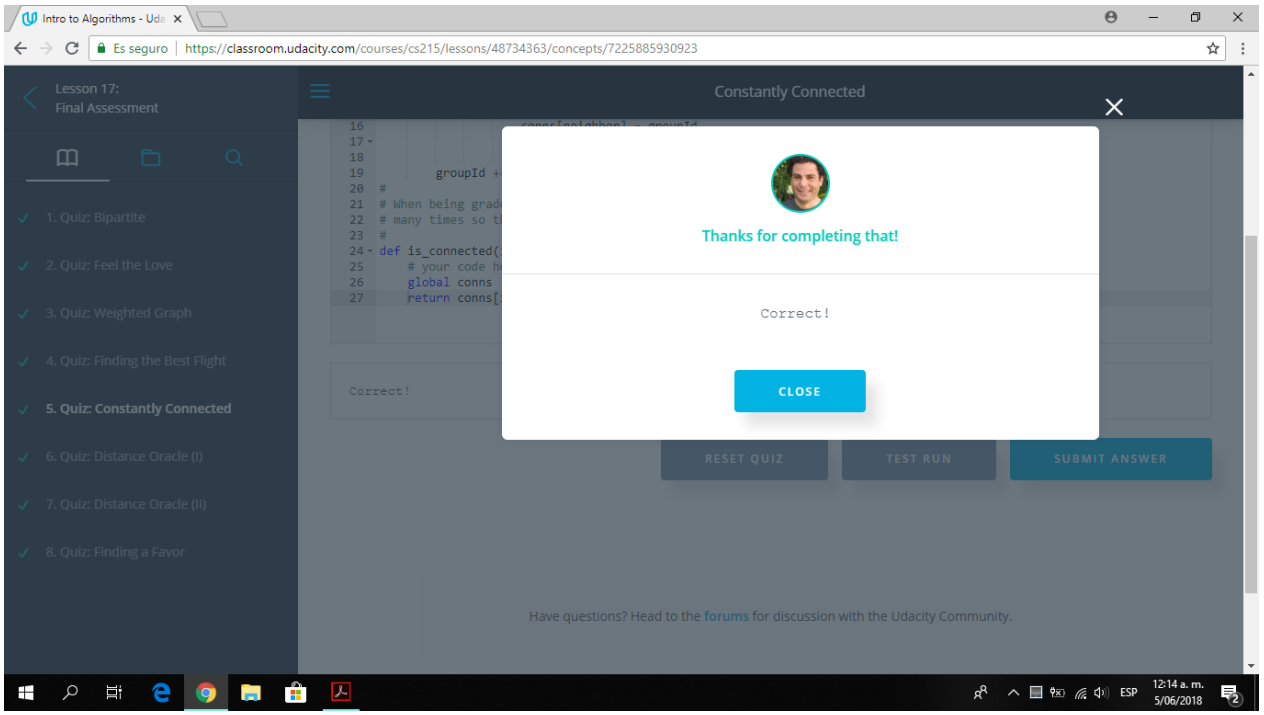

```
from heapq import heappush, heappop
from datetime import datetime
from collections import defaultdict

def timevalue(t):
    return datetime.strptime(t, '%H:%M')
```

```
def timediff(tdept, tarrv):
    tdelta = tarrv - tdept
    return total_minutes(tdelta)

def total_minutes(tdelta):
    totalseconds = tdelta.seconds + tdelta.days * 24 * 3600
    return int(totalseconds / 60)

def total_time_f(tdelta):
    return str(tdelta)[:-3]

def evaluate_path(FINFO, path):
    fi1 = FINFO[path[0]]
    fi2 = FINFO[path[-1:][0]]

    t1 = fi1['dept']
    t2 = fi2['arrv']
    td = timevalue(t2) - timevalue(t1)
    total_time = total_minutes(td)
    total_printable = total_time_f(td)
    total_cost = fi1['cost']
    if fi1['fn'] != fi2['fn']: total_cost += fi2['cost']

    for i in range(1, len(path) - 2):
        total_cost += FINFO[path[i]]['cost']

    return (total_cost, total_time, total_printable)

def assert_no_loops(FINFO, path, dest):
    for fn in path:
        if FINFO[fn]['orig'] == dest or FINFO[fn]['dest'] == dest:
            return False
    return True

def connect_to_path(FINFO, paths, flight):

    if len(paths) == 0:
        return [[flight[2]]]
```

```
    fi2 = FINFO[flight[2]]
    possible = []

    for i in range(len(paths)):
        fi1 = FINFO[paths[i][-1:][0]]
        if timevalue(fi1['arrv']) < timevalue(fi2['dept']) and assert_no_loops(FI
            newpath = [f for f in paths[i]]
            newpath = newpath + [fi2['fn']]
            possible.append(newpath)

    return possible

def dijkstra(G, FINFO, start):
    heap = []

    paths = {}
    distance = {}
    distance_t = {}

    distance_t[start] = (0, 0)
    heappush(heap, ((0, 0), start))
    paths[start] = []

    while len(distance) < len(G) and heap:
        d, nearest = heappop(heap)
        distance[nearest] = d

        for neighbor in G[nearest]:
            for flight in G[nearest][neighbor]:
                fcost, fduration, fn = flight
                possible_paths = connect_to_path(FINFO, paths[nearest], flight)
                if len(possible_paths) > 0:
                    for newpath in possible_paths:
                        cost, duration, dprintable = evaluate_path(FINFO, newpath
                        newcost = (cost, duration)
                        if neighbor not in distance_t:
                            distance_t[neighbor] = newcost
                            heappush(heap, (newcost, neighbor))
                            if neighbor not in paths:
```

```
                    paths[neighbor] = []
                if newpath not in paths[neighbor]:
                    paths[neighbor].append(newpath)
            elif newcost < distance_t[neighbor]:
                distance_t[neighbor] = newcost
                heappush(heap, (newcost, neighbor))
                if neighbor not in paths:
                    paths[neighbor] = []
                if newpath not in paths[neighbor]:
                    paths[neighbor].append(newpath)
            elif newcost >= distance_t[neighbor]:
                if neighbor not in paths:
                    paths[neighbor] = []
                if newpath not in paths[neighbor]:
                    paths[neighbor].append(newpath)

    return distance, paths

def find_best_flights(flights, origin, destination):
    best_flights = []

    G, FINFO = create_weighted_graph(flights)

    distance, paths = dijkstra(G, FINFO, origin)

    if destination in paths:
        return min(paths[destination], key=lambda x: evaluate_path(FINFO, x))

    return None

def print_route(FINFO, path):
    print path, ':'
    for f in path:
        print '%s to %s, %s - %s' %(FINFO[f]['orig'], FINFO[f]['dest'], FINFO[f
    print '---'
    print evaluate_path(FINFO, path)
```

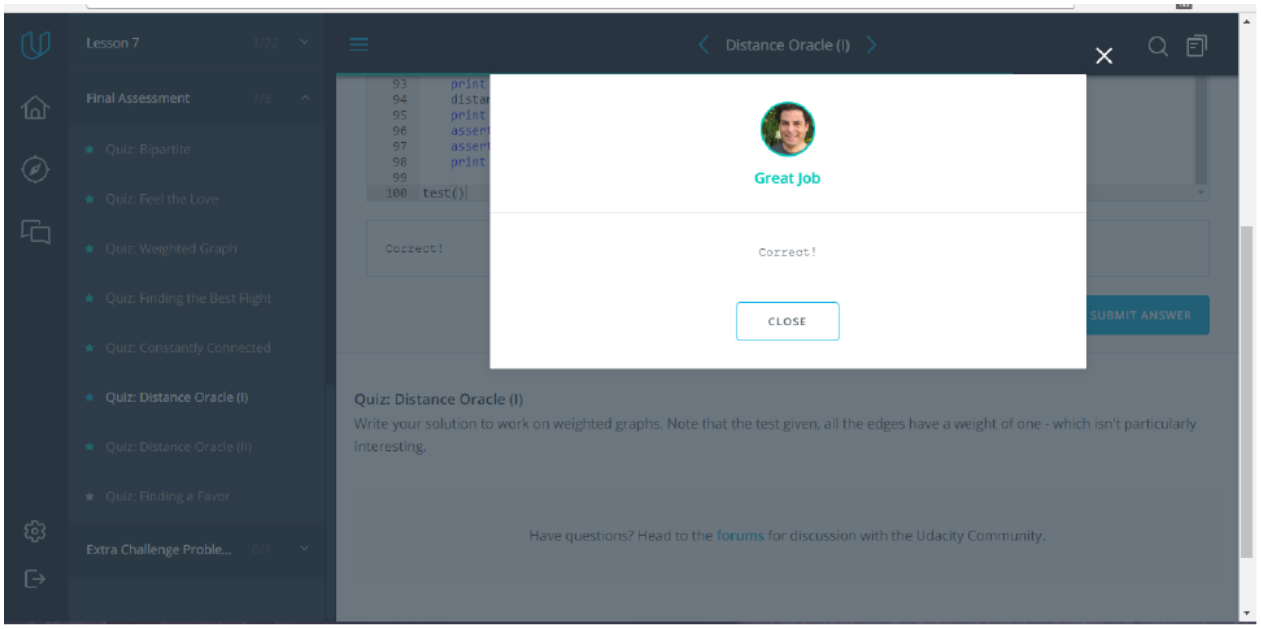
e) Constantly Connected




```
def process_graph(G):
 # your code here
 global conns
 conns = {}
 groupId = 0
 nodes = G.keys()
 while len(conns) < len(G):
 c_node = nodes.pop()
 if c_node not in conns: conns[c_node] = groupId
 open_list = [c_node]
 while open_list:
 reached = open_list.pop()
 for neighbor in G[reached]:
 if neighbor not in conns:
 open_list.append(neighbor)
conns[neighbor] = groupId
if neighbor in nodes:
 del nodes[nodes.index(neighbor)]
 groupId += 1
#
# When being graded, 'is_connected' will be called
```

```
# many times so this routine needs to be quick
#
def is_connected(i, j):
 # your code here
 global conns
 return conns[i] == conns[j]
```

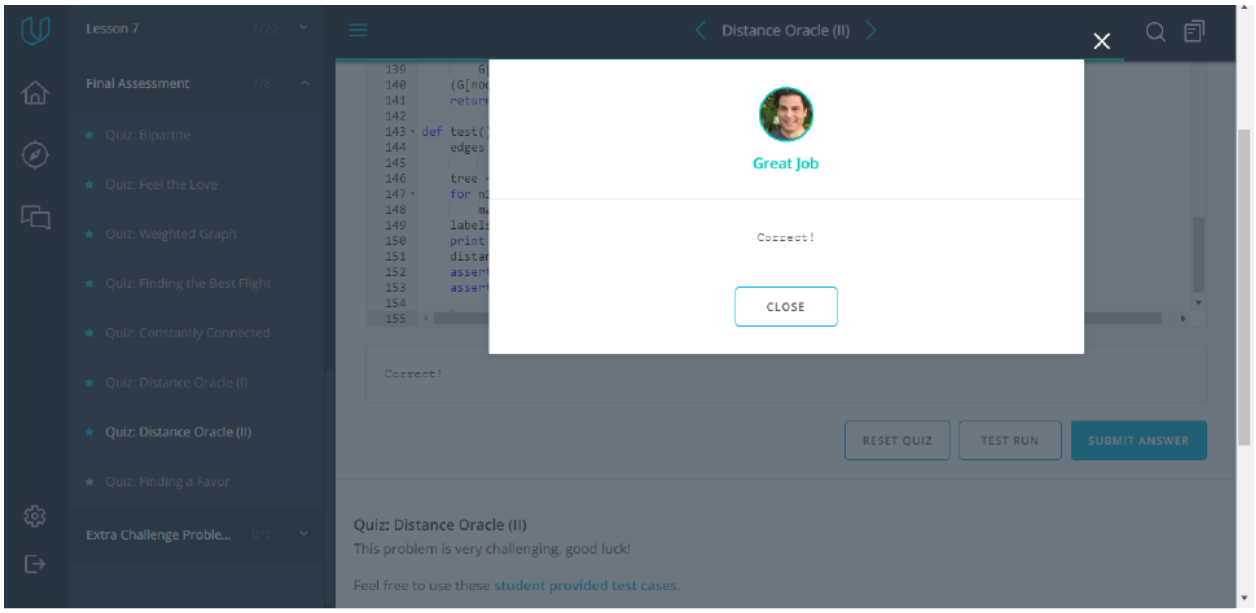
f) Distance Oracle 1




```
def create_labels(binarytreeG, root):
 labels = {root: {root: 0}}
 frontier = [root]
 while frontier:
 cparent = frontier.pop(0)
 for child in binarytreeG[cparent]:
 if child not in labels:
 labels[child] = {child: 0}
 weight = binarytreeG[cparent][child]
 labels[child][cparent] = weight
 # make use of the labels already computed
 for ancestor in labels[cparent]:
 labels[child][ancestor] = weight +
labels[cparent][ancestor]
 frontier += [child]
```

```
    return labels
```

g) Distance Oracle 2




```
def apply_labels(treeG, labels, found_roots, root):
 if root not in labels: labels[root] = {}
 labels[root][root] = 0
 visited = set()
 open_list = [root]
 while open_list:
 c_node = open_list.pop()
 for child in treeG[c_node]:
 if child in visited or child in found_roots: continue
 if child not in labels: labels[child] = {}
 labels[child][root] = labels[c_node][root] +
treeG[child][c_node]
 visited.add(child)
 open_list.append(child)
def update_labels(treeG, labels, found_roots, root):
 best_root = find_best_root(treeG, found_roots, root)
 found_roots.add(best_root)
 apply_labels(treeG, labels, found_roots, best_root)
 for child in treeG[best_root]:
 if child in found_roots: continue
```

```
    update_labels(treeG, labels, found_roots, child)
def create_labels(treeG):
    found_roots = set()
    labels = {}
    # your code here
    update_labels(treeG, labels, found_roots, iter(treeG).next())
    return labels
```

h)

4. Considere el problema de cubrir una tira rectangular de longitud n con 2 tipos de fichas de dominó con longitud 2 y 3 respectivamente. Cada ficha tiene un costo C2 y C3 respectivamente. El objetivo es cubrir totalmente la tira con un conjunto de chas que tenga costo mínimo. La longitud de la secuencia de chas puede ser mayor o igual a n, pero en ningún caso puede ser menor.

   a) Muestre que el problema cumple con la propiedad de subestructura óptima
   Para la resolución de un problema de longitud n, primero se obtiene la solución obtiene la solución para una tira de longitud menor a n, calculando estas soluciones puede dar solución al problema de longitud n.

   b) Plantee una ecuación recursiva para resolver el problema

   c) Escriba un programa en Python que resuelva el problema de manera e eficiente de cubrir(C2, C3, n)

```
def cubrir(C2, C3, n, r):
    r[0] = 0
q = float('inf')
if n == 1 or n == 2:
    q = min(C2, C3)
elif n == 3:
    q = min(2 * C2, C3)
if i in r and (n - i) in r:
    q = min(q, r[i] + r[n - i])
else:
    q = min(q, cubrir(C2, C3, i, r) + cubrir(C2, C3, n - i, r))
r[n] = q
return q
```

   d) llene la siguiente tabla

n	1	2	3	4	5	6	7	8	9	10
cubrir(5,7,n)	5	5	7	10	12	14	17	19	21	24

5. Problema de cubrimiento de un tablero 3 xn con fichas de domino:

   An = $D_{(N-1)} + C_{(N-1)}$
   Cn = $A_{(N-1)}$
   $D_n = D_{N-2} + 2 + C_{n-1}$

   b_n y E_n son 0 ya que no pueden cumplir con la forma del tablero

   Codigo

```
def A(N):
        if N == 0:
        return 0
        if N <= 1:
        return 1
        return D(N - 2) + C(N -1)
def C(N):
        if N == 0:
        return 0
        if N <= 2:
        return 1
        return A(N - 1)
def D(N):
        if N == 0:
        return 0
        if N <= 2:
        return 3
        return D(N - 2) + 2*A(N-1)
```

   para un tablero de 10 la cantidad de fichas es de 203
   para un tablero de 50 la cantidad de fichas no puede cubrirlo de manera eficiente
   para un tablero de 100 la cantidad de fichas no puede cubrirlo de manera eficiente