TALLER 2

Grafos, Complejidad Computacional, Programación Dinámica

Laura Valentina Castaño Morales- 1010229518

Ing. Germán Hernández

(Docente)

UNIVERSIDAD NACIONAL DE COLOMBIA

SEDE BOGOTÁ

FACULTAD DE INGENIERÍA

DEPARTAMENTO DE SISTEMAS E INDUSTRIAL

ALGORITMOS (2016696 - 1)

BOGOTÁ D.C

2018

Desarrollo

1. Considerando el grafo de la Figura 1

a. Algoritmo de Dijkstra

1. Inicializamos todas las distancias en D con un valor infinito debido a que son desconocidas al principio, la del 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 v_i.

4. Para el nodo actual, calculamos la distancia desde dicho nodo a sus vecinos con la siguiente fórmula: $dt(v_i) = D_a + d(a,v_i)$. Es decir, la distancia del nodo 'v_i' es la distancia que actualmente tiene el nodo en el vector D más la distancia desde dicho el nodo 'a' (el actual) al nodo v_i. 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.

Una vez terminado al algoritmo, D estará completamente lleno.

```
{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. Algoritmo de Bellman-Ford

1. Inicializamos el grafo. Ponemos distancias a INFINITO en todos los nodos menos en 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 negativos.

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. Algoritmo de Floyd-Warshal
El algoritmo de Floyd-Warshall compara todos los posibles caminos a través del grafo entre cada par de vértices.

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 $(C_{ik} + C_{kj}) < C_{ij} \rightarrow D_{ij} = D_{kj}$ y $C_{ij} = C_{ik} + C_{kj}$

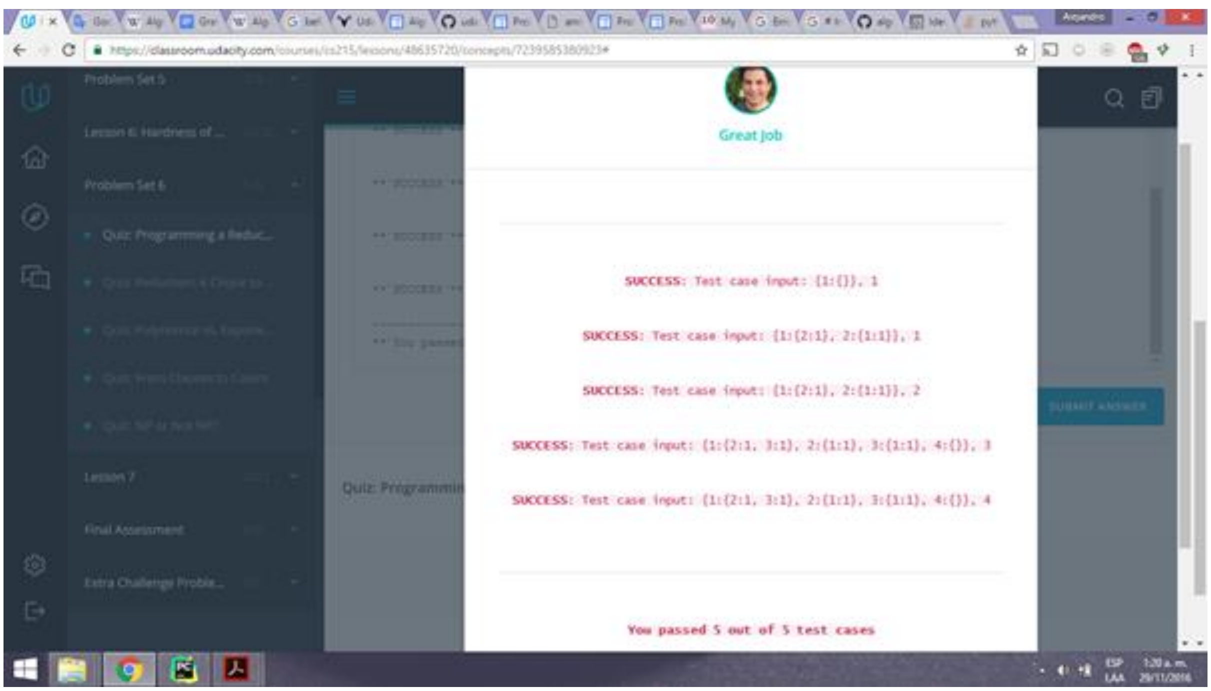
5. En caso contrario, dejamos las matrices como están.

6. Si $k \leq n$, aumentamos k en una unidad y repetimos el paso anterior, en caso contrario para 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])
```

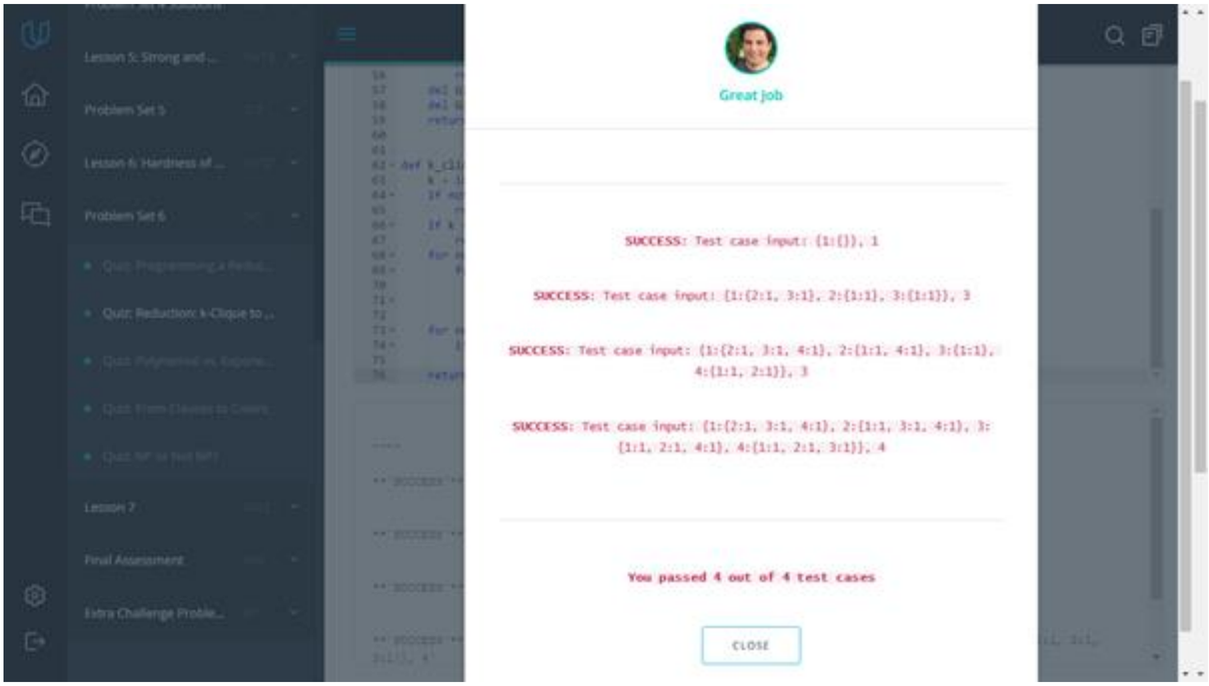
2.




```python
# This function should use the k_clique_decision function
# to solve the independent set decision problem
def independent_set_decision(H, s):
    # your code here
    G = {}
```

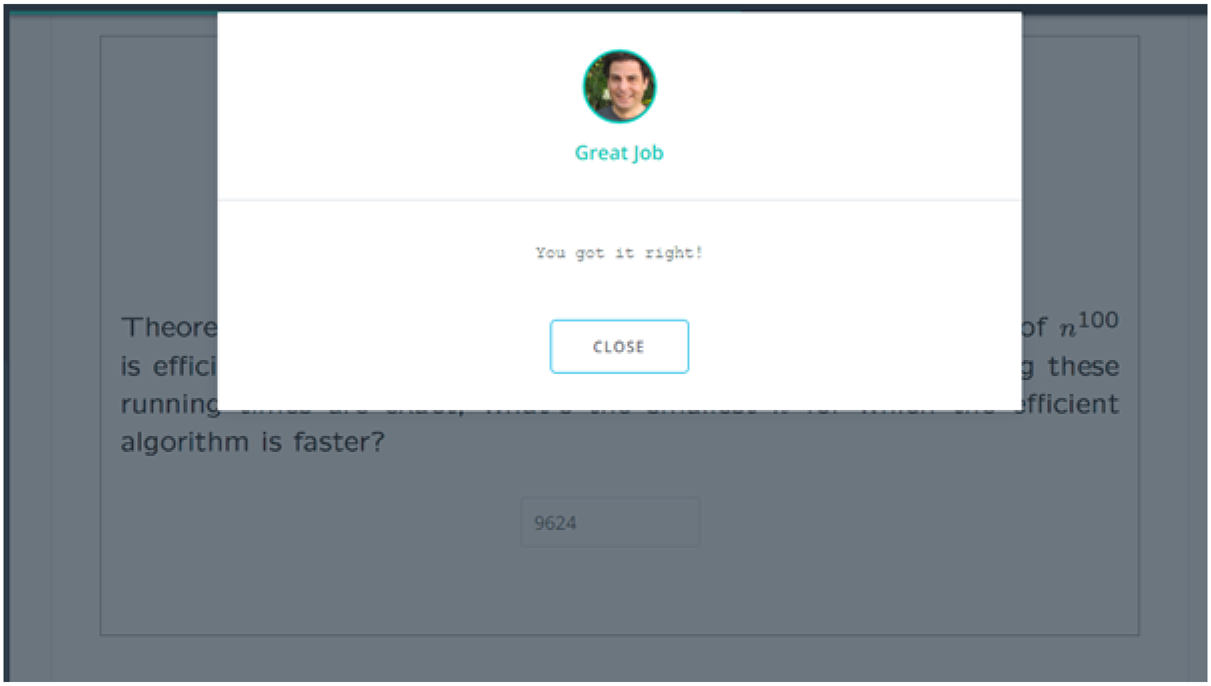
```
    all_nodes = H.keys()
    for v in H.keys():
        G[v] = {}
        for other in list(set(all_nodes) - set(H[v].keys()) - set([v])):
                G[v][other] = 1
    print G
    return k_clique_decision(G, s)

def test():
    H={}
    edges = [(1,2), (1,4), (1,7), (2,3), (2,5), (3,5), (3,6), (5,6), (6,7)]
    for u,v in edges:
        make_link(H,u,v)
    for i in range(1,8):
        print(i, independent_set_decision(H, i))

test()
```

2. 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()
```

3. Polynomial vs. Exponential



4. From Clauses to Colors

From Clauses to Colors

In the reduction from 3-SAT to 3-COLORABILITY, we talked about a way of converting a 3-SAT problem with x variables and y clauses into a graph with n nodes and m edges. Give a formula for n and m. (Fill in the boxes to complete the equation. See the example given below.)

$$n = \boxed{2}\,x + \boxed{6}\,y + \boxed{3}$$

$$m = \boxed{3}\,x + \boxed{12}\,y + \boxed{3}$$

(ex. $n = 4x + 10y + 8$)

5. NP or Not NP?

NP or Not NP? That is the Question

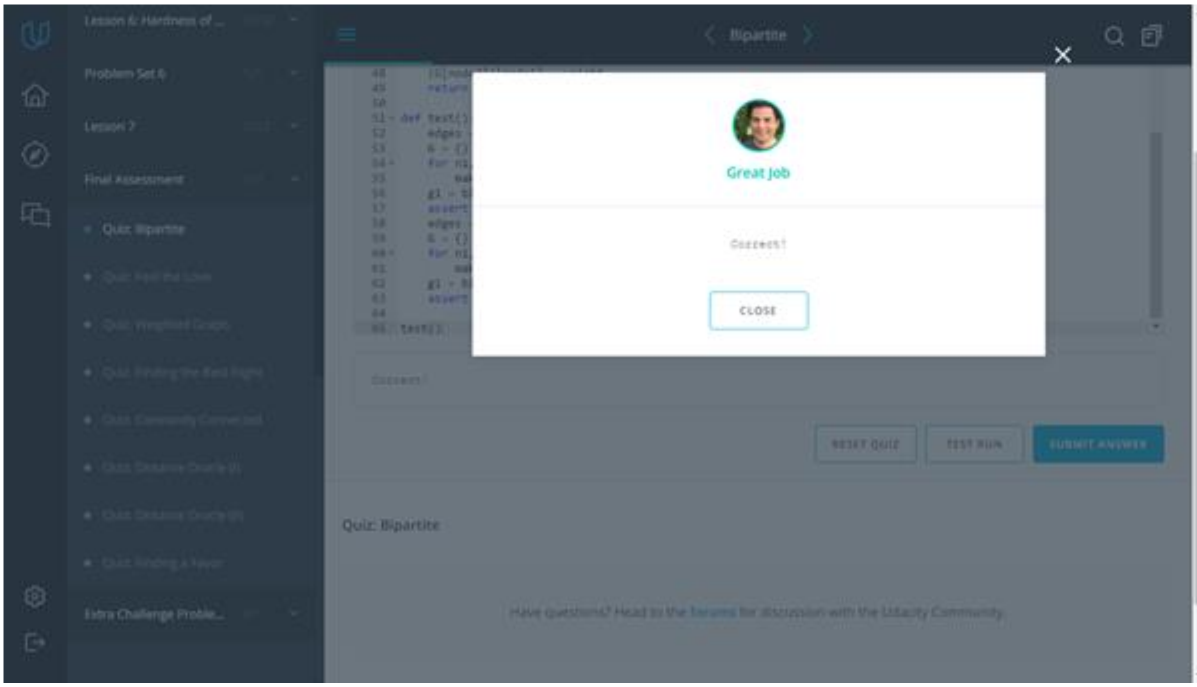
Select all the problems below that are in NP. Hint: Think about whether or not each one has a short accepting certificate.

- [ ] **Connectivity**: Is there a path from x to y in G?
- [ ] **Short path**: Is there a path from x to y in G that is no more than k steps long?
- [x] **Fewest colors**: Is k the absolute minimum number of colors with which G can be colored?
- [x] **Near Clique**: Is there a group of k nodes in G that has at least s pairs that are connected?
- [ ] **Partitioning**: Can we group the nodes of G into two groups of size $n/2$ so that there are no more than k edges between the two groups.
- [x] **Exact coloring count**: Are there exactly s ways to color graph G with k colors?

SUBMIT ANSWER

3. dsa

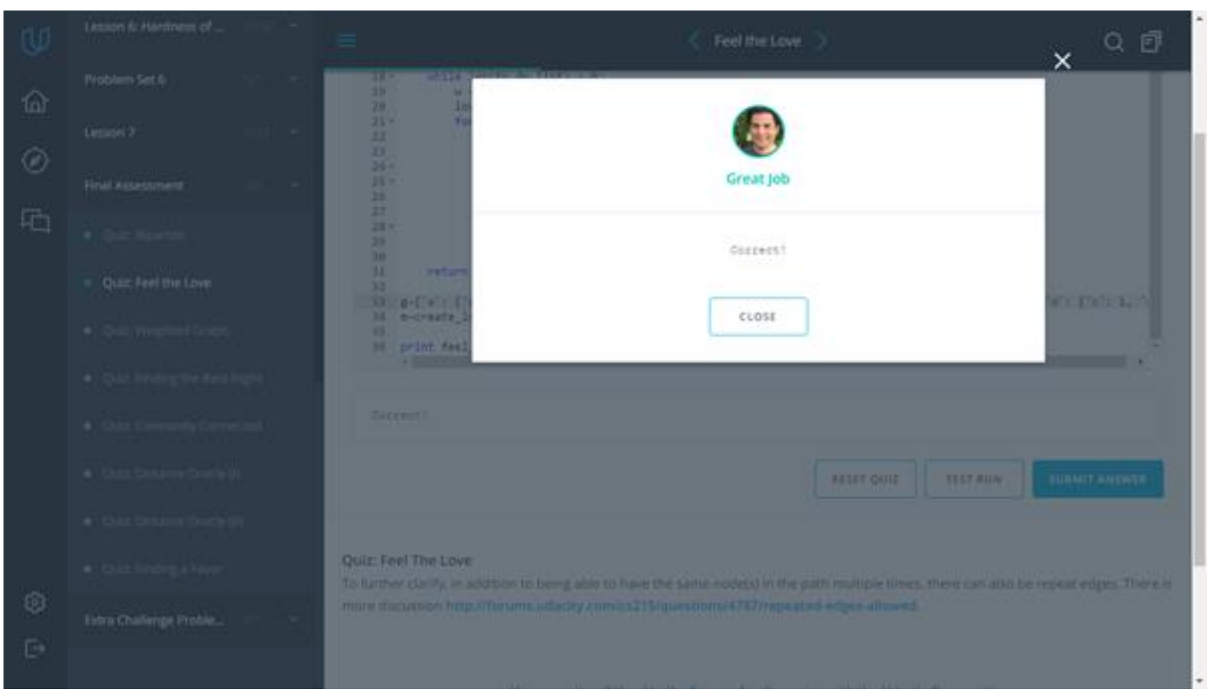
   1. Bipartite




```
from collections import deque
def bipartite(G):
    # your code here
    # return a set
    if not G:
        return None
    start = next(G.iterkeys())
    lfrontier, rexplored, L, R = deque([start]), set(), set(), set()
    while lfrontier:
        head = lfrontier.popleft()
        if head in rexplored:
            return None
```

```
            if head in L:
                continue
            L.add(head)
            for successor in G[head]:
                if successor in rexplored:
                continue
                R.add(successor)
                rexplored.add(successor)
                for nxt in G[successor]:
                    lfrontier.append(nxt)
    return L
```

2. Feel the Love

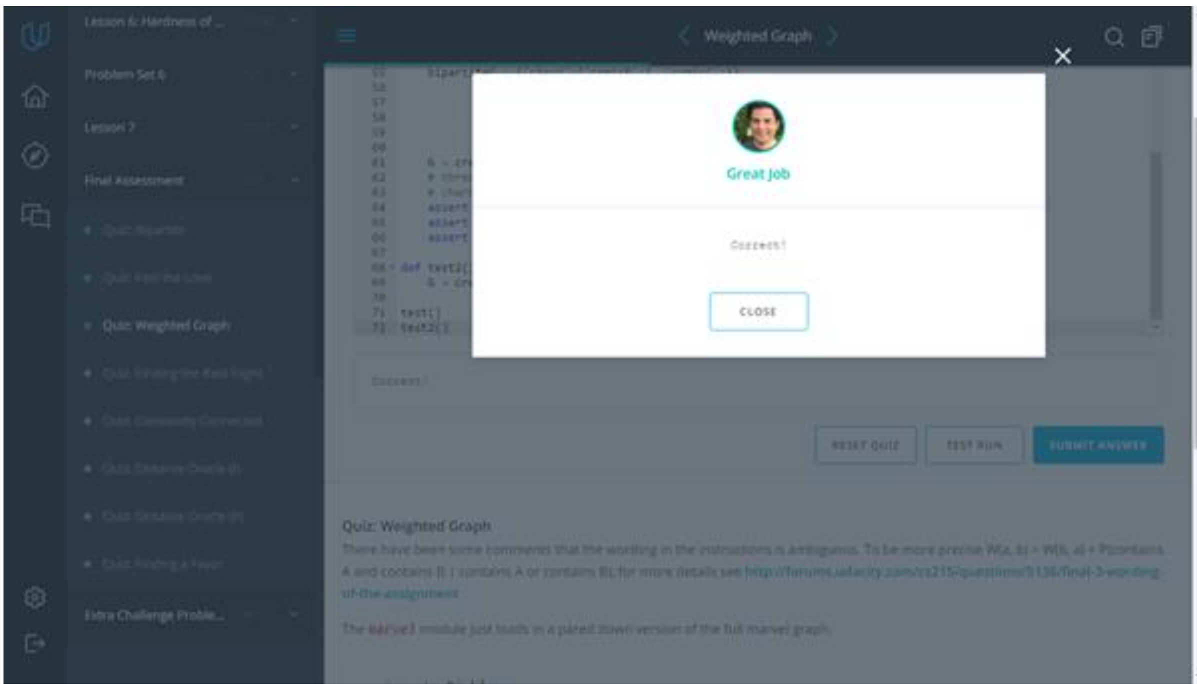



```
def feel_the_love(G, i, j):
```

```
    # 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
    result = create_love_paths(G, i)
    if j in result:
        return result[j][1]
    else:
        return None

def create_love_paths(G, v):
    love_so_far = {}
    love_so_far[v] = (0, [v])
    to_do_list = [v]
    while len(to_do_list) > 0:
        w = to_do_list.pop(0)
        love, path = love_so_far[w]
        for x in G[w]:
            new_path = path + [x]
            new_love = max([love, G[w][x]])
            if x in love_so_far:
                if new_love > love_so_far[x][0]:
                    love_so_far[x] = (new_love, new_path)
                    if x not in to_do_list: to_do_list.append(x)
            else:
                love_so_far[x] = (new_love, new_path)
                if x not in to_do_list: to_do_list.append(x)
    return love_so_far
```

```
g={'a': {'c': 1}, 'c': {'a': 1, 'b': 1, 'e': 1, 'd': 1}, 'b': {'c': 1},
'e': {'c': 1, 'd': 2}, 'd': {'c': 1, 'e': 2}}

m=create_love_paths(g,'a')


print feel_the_love(g, 'a', 'e')
```

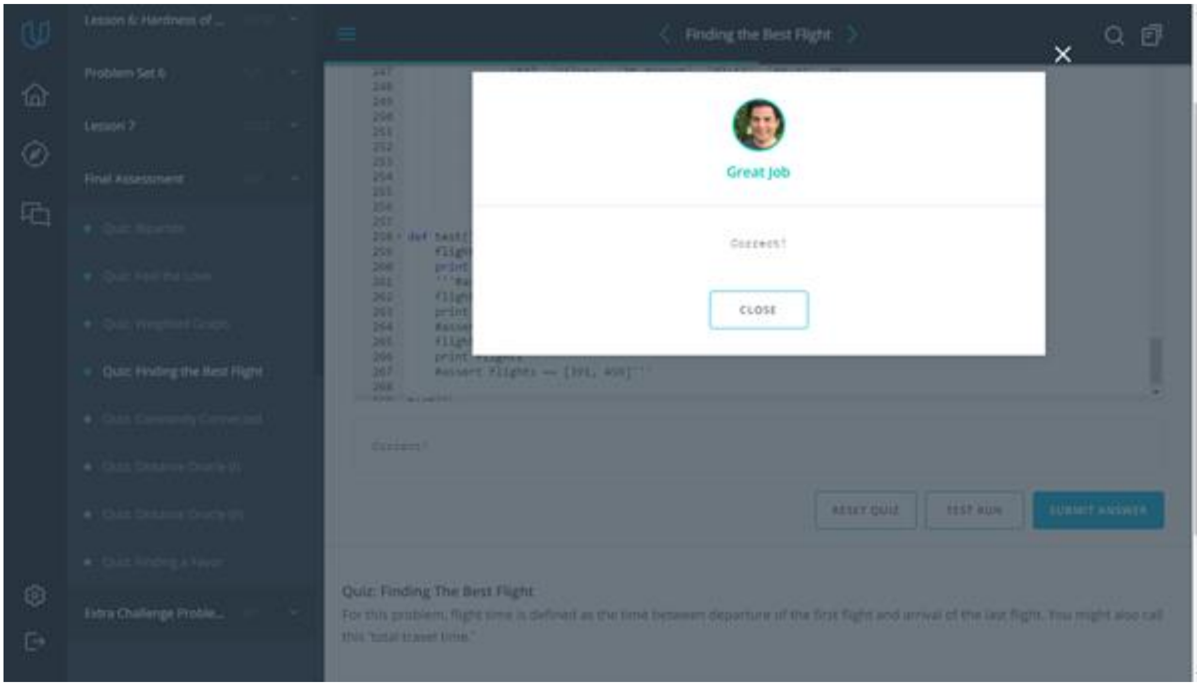
3. Weighted Graph




```
def create_weighted_graph(bipartiteG, characters):
    comic_size = len(set(bipartiteG.keys()) - set(characters))
    # your code here
    AB = {}
    for ch1 in characters:
        if ch1 not in AB:
            AB[ch1] = {}
        for book in bipartiteG[ch1]:
```

```
            for ch2 in bipartiteG[book]:
            if ch1 != ch2:
                    if ch2 not in AB[ch1]:
                        AB[ch1][ch2] = 1
                  else:
                        AB[ch1][ch2] += 1
    contains = {}
    for ch1 in characters:
       if ch1 not in contains:
             contains[ch1] = {}
        contains[ch1] = len(bipartiteG[ch1].keys())
    G = {}
    for ch1 in characters:
       if ch1 not in G:
             G[ch1] = {}
       for book in bipartiteG[ch1]:
             for ch2 in bipartiteG[book]:
             if ch2 != ch1:
                     G[ch1][ch2] = (0.0 + AB[ch1][ch2]) / (contains[ch1] +
contains[ch2] - AB[ch1][ch2])
    return G
```

4. Finding the best Flight

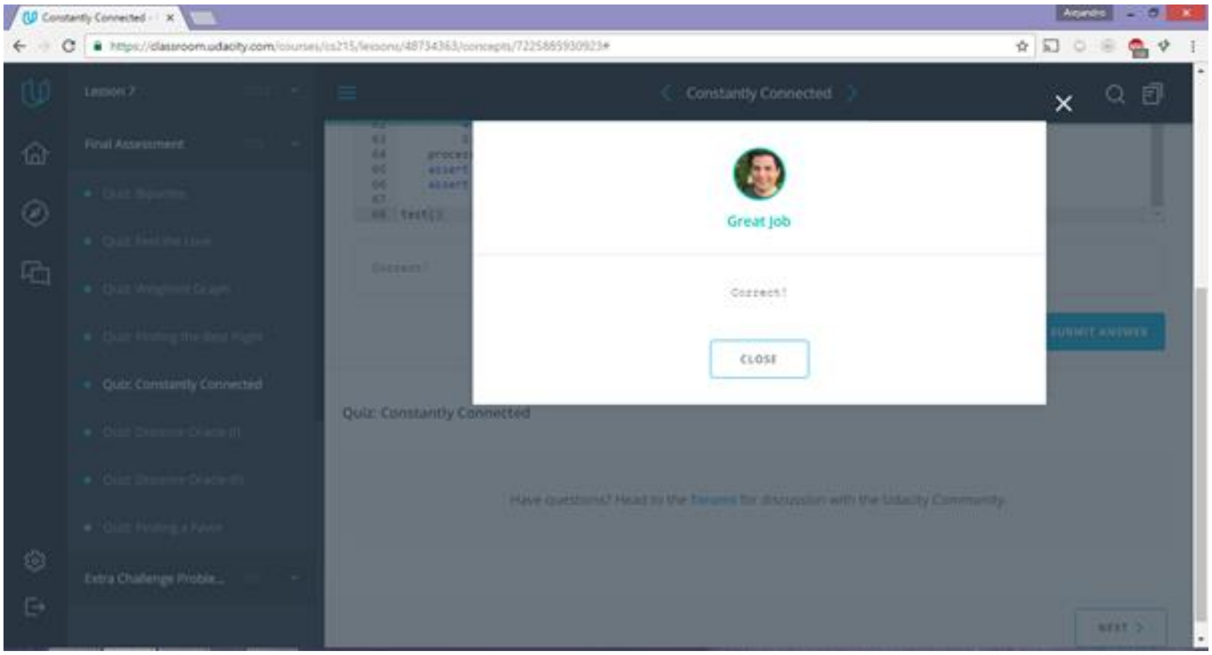



```python
import heapq


def find_best_flights(flights, origin, destination):
    G = make_graph(flights)
    R = find_route(G, origin, destination)
    return R



def make_graph(flights):
    edges = {}
    for (flight_number, origin, dest, take_off, landing, cost) in flights:
        to = make_time(take_off)
        land = make_time(landing)
        edges[flight_number] = {'origin':origin, 'dest':dest, 'take_off':to, 'land':land, 'cost':cost}
        if origin not in edges:
```

```
            edges[origin] = []
        edges[origin] += [flight_number]
    return edges


def make_time(t):
    hour = int(t[:2])
    min = int(t[3:])
    return hour*60+min

def find_route(G, origin, destination):
    heap = [(0,0,None,[])]
    while heap:
        c_cost, c_away, c_start, c_path = heapq.heappop(heap)
        if not c_path:
            c_town = origin
        else:
            c_town = G[c_path[-1]]['dest']
        if c_town == destination:
            return c_path
        for flight in G[c_town]:
            if c_town == origin:
                c_start = G[flight]['take_off']
            if c_start + c_away <= G[flight]['take_off']:
                heapq.heappush(heap, (c_cost + G[flight]['cost'],
                                      G[flight]['land'] -c_start,
                                      c_start,
                                      c_path + [flight]))
```

```
    return None
```

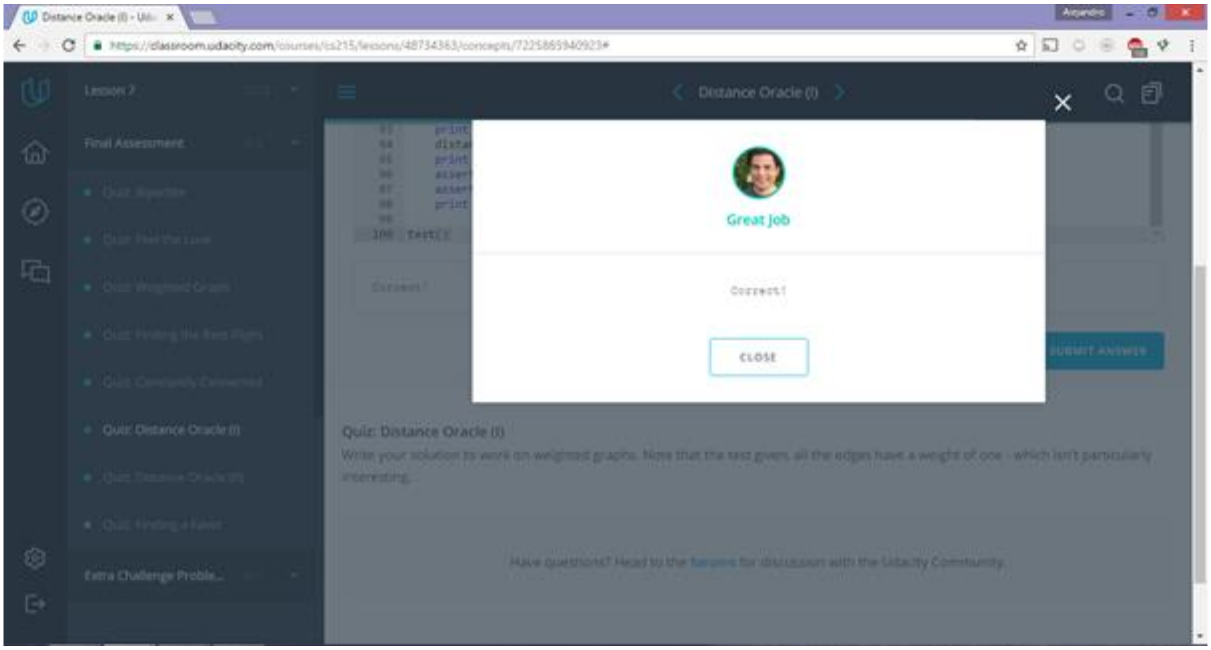
5. Constantly Connected




```
conns = {}


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
```

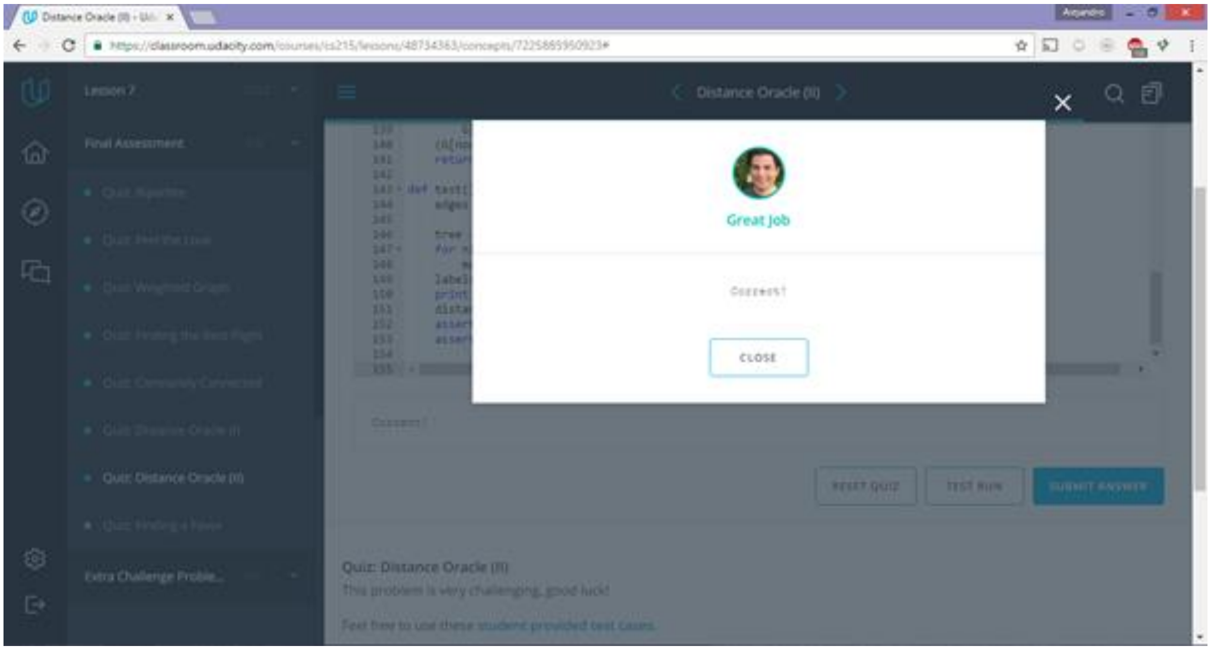
```python
    open_list = [c_node]
    while open_list:
        reached = open_list.pop()
        for neighbor in G[reached]:
            if neighbor not in conns:
                open_list.append(neighbor)
                conns[neighbor] = groupId
                if neighbor in nodes:
                    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]
```

6. Distance Oracle (I)




```python
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
```

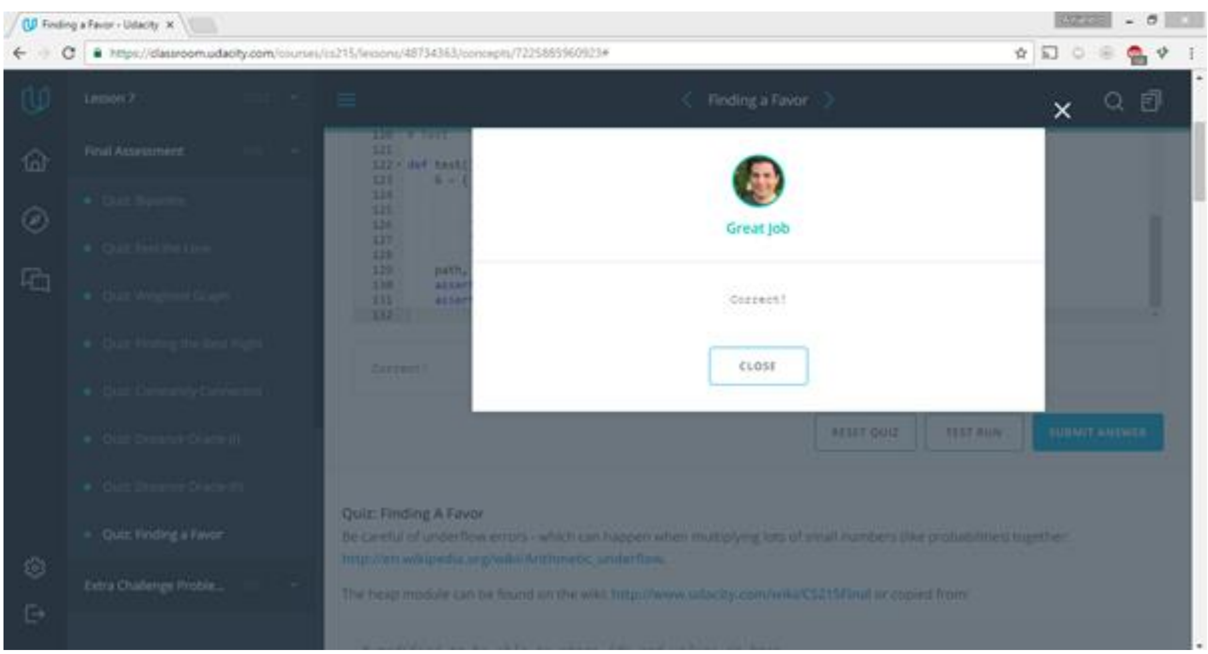
7. Distance Oracle (II)




```
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)
```

```python
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
```

8. Finding a Favor



```python
def maximize_probability_of_favor(G, v1, v2):
    # your code here
    from math import log, exp
    logG = {}
    n = len(G.keys())
    m = 0
    for node in G.keys():
        logG[node] = {}
        m += len(G[node].keys())
        for neighbor in G[node].keys():
            logG[node][neighbor] = -log(G[node][neighbor])
    if n**2 < (n+m)*log(n):
        final_dist = dijkstra_list(logG, v1)
    else:
        final_dist = dijkstra_heap(logG, v1)
    if v2 not in final_dist: return None, 0
    node = v2
    path = [v2]
    while node != v1:
        node = final_dist[node][1]
        path.append(node)
    path = list(reversed(path))
    prob = exp(-final_dist[v2][0])
    return path, prob
```

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) 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) Ecuación recursiva.:

$$P_n = \{min\ (P_2, P_3)\ \ si\ \ n <= 2;\ min(2P_2, P_3)\ \ si\ \ n = 3\ \ min(P_i + P_{n-i})\ 1 <= i < = n - 1\ \ \ si\ n > 3\}$$

c) Programa python:

```
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) Tabla para C2 = 5, C3 = 7, n = 10.

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 chas de domino:

❖ Ecuaciones:

$$A_n = D_{(N-1)} + C_{(N-1)}$$

$$c_n = 2 * A_{(N-2)}$$

$$D_n = D_{(N-2)} + 2 * C_{(N-1)}$$

- Be y En son siempre cero ya que no es posible que resulte la forma del tablero que representan.
- Implementación:

```
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)
```

- Resultados:

10	50	100
203	238039524083	31208688988045323113527764971