**Nombres:**

* Pedro Alejandro Valderrama Tapias **Código:** 2879195
* Brayan Alexander Riascos Ruiz **Código:** 2879762
* Edgar Rafael Cruz Rodriguez **Código:** 2879720

**TALLER 2**

**Grafos, Complejidad Computacional, Programación Dinámica**

1. Considere el grafo de la Figura 1 (solo tenga en cuenta los pesos en paréntesis):
   1. Ejecute el algoritmo de Dijkstra detallando claramente los pasos ejecutados.
   2. Ejecute el algoritmo de Bellman-Ford detallando claramente los pasos ejecutados.
   3. Ejecute el algoritmo de Floyd-Warshall detallando claramente los pasos ejecutados.
2. Resuelva los puntos del Problem Set 6 del curso Algorithms de Udacity. Incluya el código correspondiente con un screenshot de aceptación para cada problema.
3. Resuelva los puntos del Final Exam del curso Algorithms de Udacity. Incluya el código correspondiente con un screenshot de aceptación para cada problema.
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 y 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.
   1. Muestre que el problema cumple con la propiedad de subestructura óptima
   2. Plantee una ecuación recursiva para resolver el problema
   3. Escriba un programa en Python que resuelva el problema de manera e eficiente de cubrir(, , n)
   4. Llene la siguiente tabla para el caso , y n = 10:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| cubrir(,,n) |  |  |  |  |  |  |  |  |  |  |  |

1. Problema de cubrimiento de un tablero 3 xn con chas de domino:
   1. Obtenga y estudie la presentación en [CS97SI-DP]
   2. Revise el problema de cubrir un tablero de 3 n con fichas de dominó (tamaño 2 1 o 1 2).
   3. Plantee las recurrencias para , , y
   4. Por qué siempre es 0?
   5. Escriba un programa en Python para calcular
   6. Calcule para n = 10; 50; 100

**SOLUCIÓN**

1. **a.** Se inicializan las distancias que hay desde el nodo 1, a los demás nodos. La distancia desde el nodo 1 hasta sí mismo es cero y se marca con distancia infinita (inf) los demás nodos.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | inf | inf | inf | inf | inf | inf | inf | inf | inf |

Luego los valores van a una cola de prioridad. Esto con el fin de verificar que nodos se encuentran aún pendientes por ser visitados y así conocer el siguiente nodo para ser evaluado.

# **Paso de relajación**

Costo:

40 de 1 → 2

8 de 1 → 3

10 de 1 → 4

Nodo 1:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 40 | 8 | 10 | inf | inf | inf | inf | inf | inf |

Se retira al nodo 1 de la cola de prioridad, se agrega a una lista de *visitados*. Se evalúa al nodo 3 que es el más cercano al 1.

# **Paso de relajación**

Por cada vecino cercano al nodo actual (*Nodo 3*), se compara si la distancia de ir desde el 3 hasta 6 es la más económica y se actualiza la distancia. Para este caso, el 6 se encuentra con una distancia 40, pero es más eficiente pasar antes por el nodo 3, con distancia de 12.

Nodo 3:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | inf | 10 | inf | inf | inf | inf |

seguimos realizando la misma operación con los siguientes nodos:

nodo 4:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | inf | 4 | inf | inf | inf | inf |

nodo 2:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | 16 | 4 | 22 | inf | inf | inf |

nodo 6:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | 16 | 4 | 22 | 8 | 7 | inf |

nodo 10:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | 16 | 4 | 22 | 8 | 7 | 9 |

nodo 8:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | 8 | 4 | 22 | 8 | 7 | 9 |

nodo 5:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | 8 | 4 | 12 | 8 | 7 | 9 |

nodo 7:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | 8 | 4 | 12 | 8 | 7 | 9 |

Por tal motivo esta es la distancia más corta con la cual se puede llegar a cualquier nodo desde el nodo 1.

**b.** El algoritmo de Bellman-Ford tiene una complejidad de O(V \* E) y permite encontrar el camino más corto para grafos dirigidos con distancias o pesos negativos en sus vertices, por lo que debe ejecutarse el procedimiento de actualización a lo sumo V-1 veces.

Funciona de forma similar que Dijkstra pero aca no escoge el nodo con el menor peso si no que se mueve por todos los vértices del grafo

Se realizan todos los pasos que se realizaron en el algoritmo de Dijkstra, sin embargo como en el grafo no hay aristas negativas el resultado es el mismo.

nodo 1:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | 8 | 4 | 12 | 8 | 7 | 9 |

nodo 3:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| nodo | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| D[i] | 0 | 12 | 8 | 10 | 8 | 4 | 12 | 8 | 7 | 9 |

**c)** Se usa programación dinámica, pues se usan las distancias ya conocidas hasta cierto nodo para averiguar los caminos derivados, sin necesidad de recalcular todo desde el principio.

**Pseudocodigo:**

*for i = 1 to N*

*for j = 1 to N  
if there is an edge from i to j  
dist[0][i][j] = the length of the edge from i to j else  
dist[0][i][j] = INFINITY*

Inicialmente se evalúa cada una de las conexiones entre los nodos y se marca la correspondiente casilla en la matriz N \* N, siendo N el número de nodos del grafo, en este caso N = 10, entonces se procede a llenar una matriz 10 \* 10 con las distancias dadas por el grafo de manera inicial. Si no hay puente directo entre los nodos, la distancia se marca con infinito *(*inf*)*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **nodo** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **1** | 0 | 40 | 8 | 10 | inf | inf | inf | inf | inf | inf |
| **2** | inf | 0 | inf | inf | 6 | inf | 10 | inf | inf | inf |
| **3** | inf | 4 | 0 | 12 | inf | 2 | inf | inf | inf | inf |
| **4** | inf | inf | inf | 0 | inf | 1 | inf | inf | inf | inf |
| **5** | inf | inf | 2 | inf | 0 | 2 | 4 | inf | inf | inf |
| **6** | inf | inf | inf | inf | inf | 0 | inf | 4 | 3 | inf |
| **7** | inf | inf | inf | inf | inf | inf | 0 | 20 | inf | 1 |
| **8** | inf | inf | inf | inf | 0 | inf | inf | 0 | inf | 20 |
| **9** | inf | inf | inf | 19 | inf | inf | inf | 10 | 0 | 20 |
| **10** | inf | inf | inf | inf | inf | inf | inf | inf | inf | 0 |

Luego se debe actualizar cada una de las casillas de la matriz para encontrar las distancias mínimas con la fórmula establecida. Si la distancia almacenada en *[k•1][i][k] +* la distancia almacenada en *[k•1][k][j]* resulta ser menor que la distancia en *[k•1] [i][j]*

Para tal fin resulta útil pensar en una matriz tridimensional pues los valores de k al ir desde 1 hasta 10 siempre k depende de la matriz parcial en k • 1 y se toma como referencia para actualizar los valores.

*k = 1, i = 1, j = 1: dist[1][1][1] = min( dist[0][1][1], dist[0][1][1] + dist[0][1][1]) = min(0, 0 + 0 ) = 0*

*k = 1, i =1, j = 2: dist[1][1][2] = min( dist[0][1][2], dist[0][1][1] + dist[0][1][2])) = min( 40 , 0 + 40)= 40*

*k = 1, i = 1, j = 3: dist[1][1][3] = min( dist[0][1][3], dist[0][1][1] + dist[0][1][3])) = min(8, 0 + 8) = 8*

*k = 1; i = 1, j = 4: dist[1][1][4] = min (min[0][1][4] , dist[0][1][1] + dist[0][1][4]) = min (10, 0 + 10) = 10. k = 1; i = 1; j = 5;*

*dist[1][1][5] = min (min[0][1][5] , dist[0][1][1] + dist[0][1][5]) = min (inf, 0 + inf) = inf.*

Se repite este proceso hasta llegar a j = 10 y procedemos a aumentar el valor de i*.*

*k = 1; i= 2; j =1: Dist[1][2][1] = min(Dist[0][2][1], Dist[0][2][1] + Dist[0][][])*

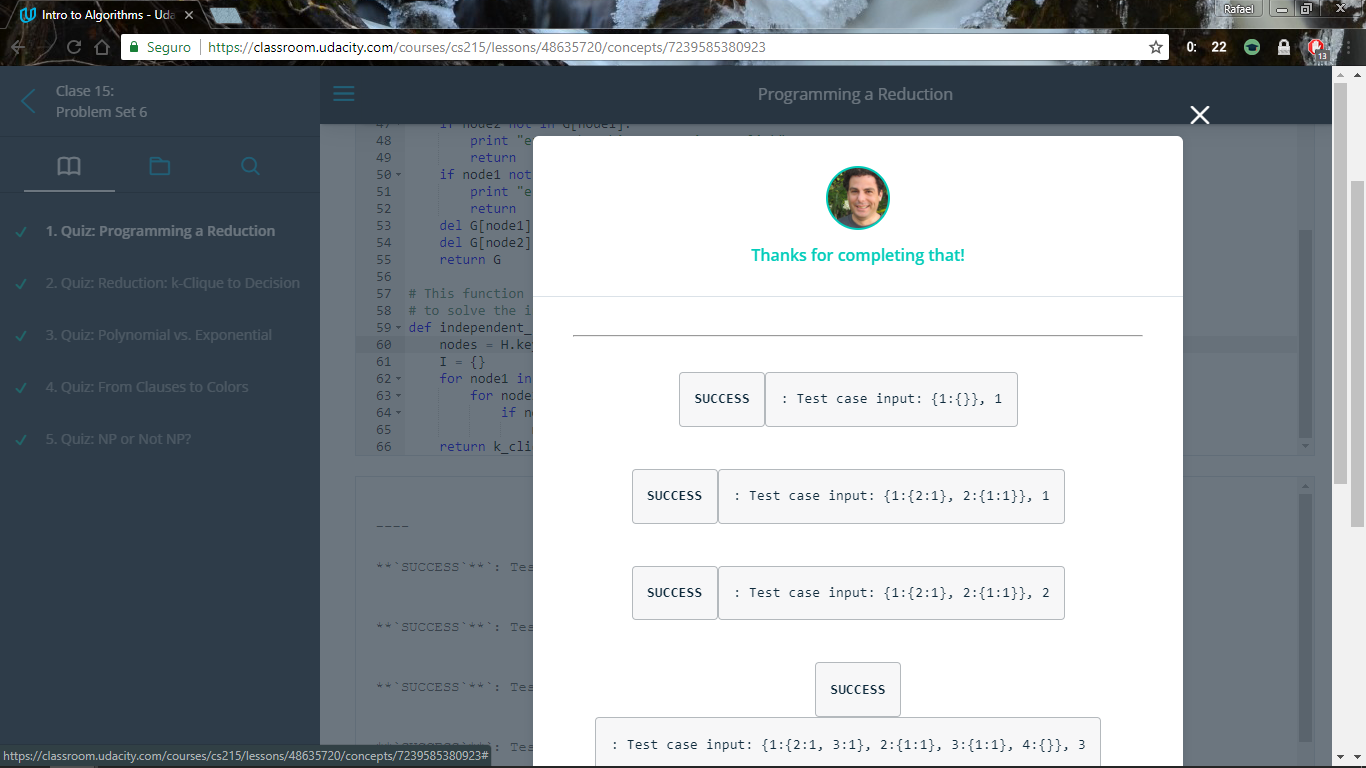
*K = 1, i = 1; j = 4: dist[1][1][4] = min(dist[0][1][4], dist[0][1][1] + dist[0][1][4]) = min (10, 0 + 10) = 10*

De manera análoga la nueva matriz subsecuente mantendrá la distancia más corta encontrada desde el nodo 1 a los demás nodos hasta que realiza 10 revisiones en total de las 10 filas y las 10 columnas.

**2.** SoluciónProblem Set 6

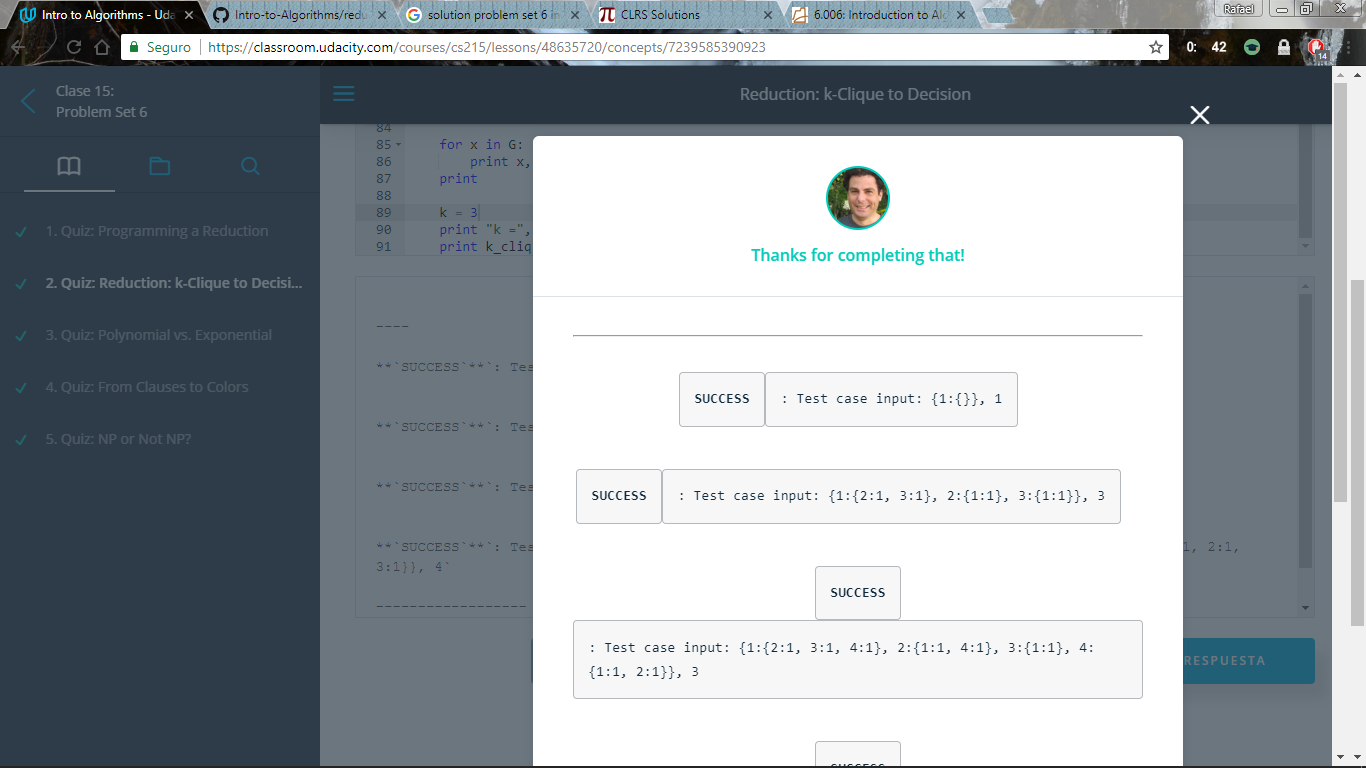
1. Programming a Reduction

def independent\_set\_decision(H, s):  
 nodes = H.keys()  
 I = {}  
 for node1 in H:  
 for node2 in H:  
 if node2 not in H[node1]:  
 make\_link(I, node1, node2)  
 return k\_clique\_decision(I, s)

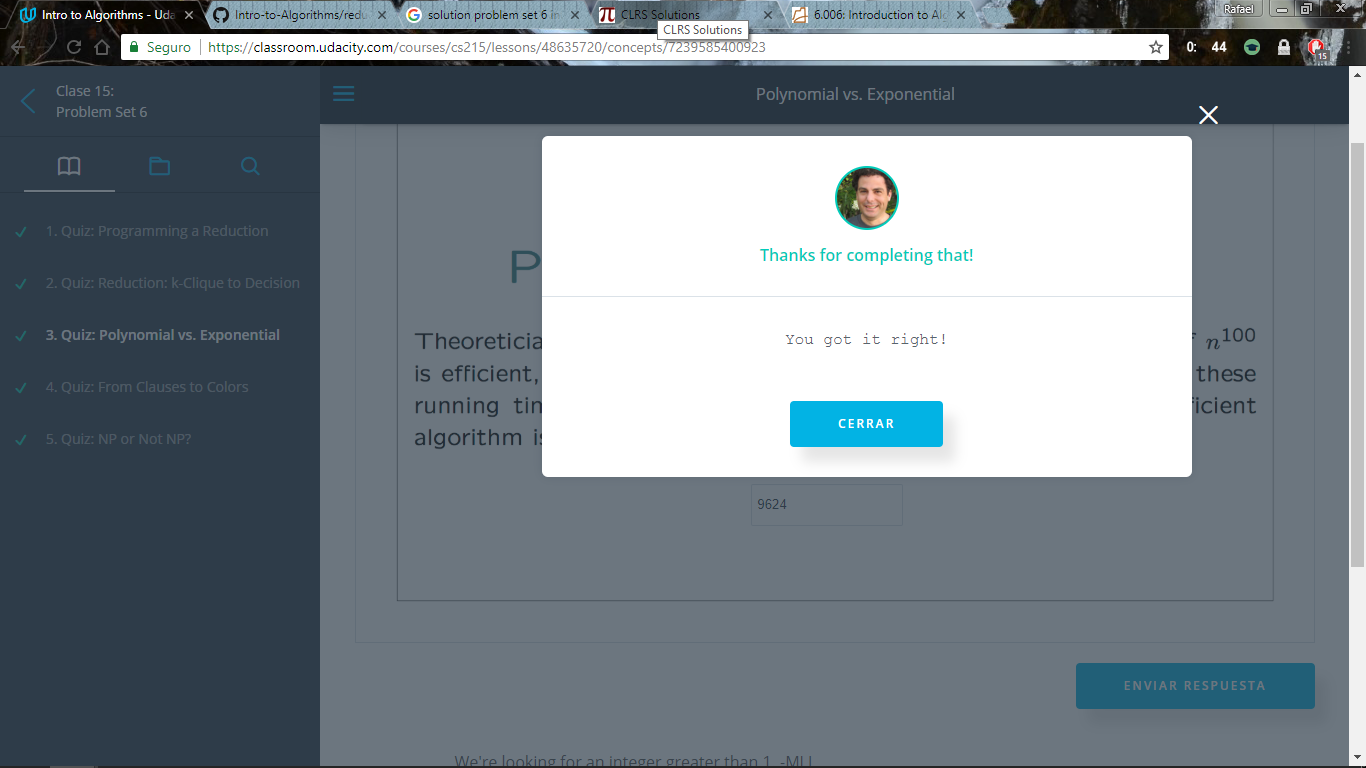


1. Reduction: k-Clique to Decision

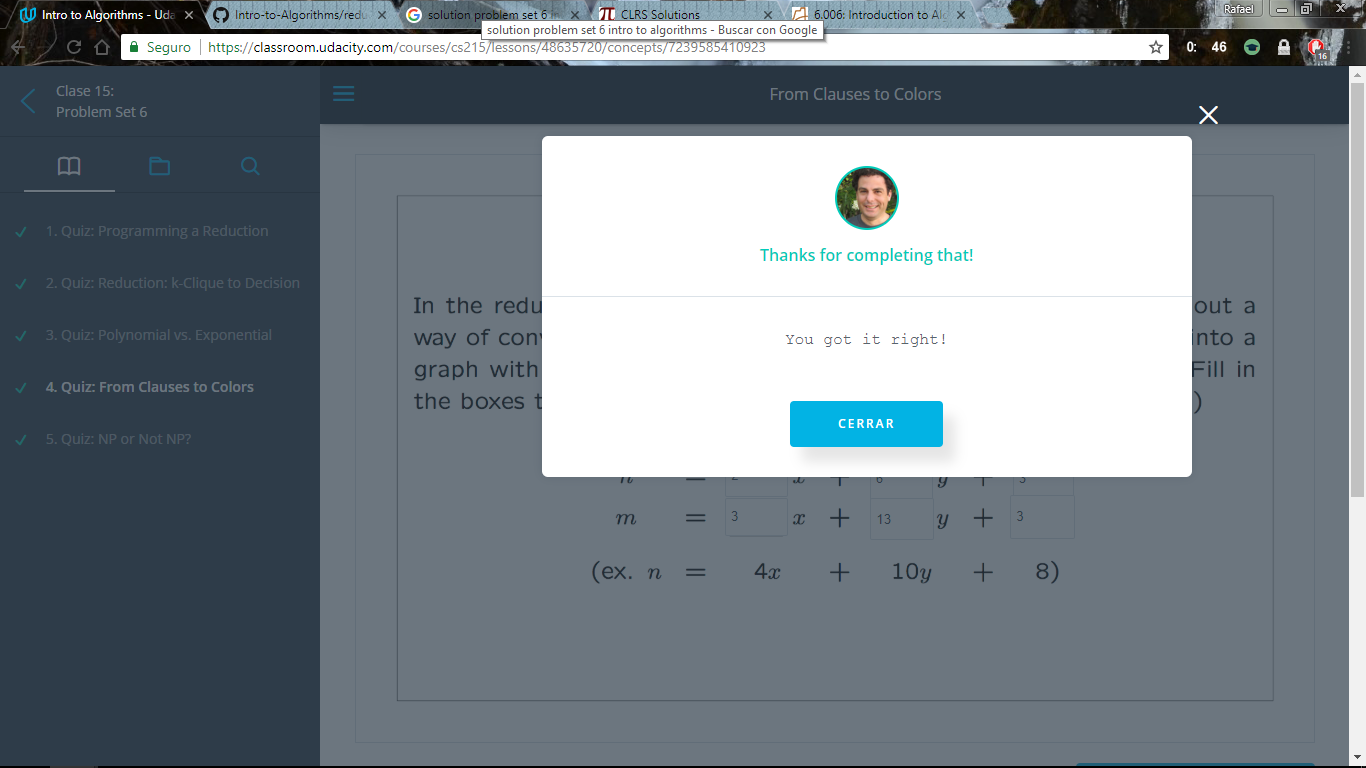
def get\_all\_edges(G):  
 edges = {}  
 for x in itertools.combinations(G.keys(), 2):  
 if x[0] in G[x[1]]:  
 edges[x] = True  
 return edges.keys()  
  
def k\_clique(G, k):  
 if not k\_clique\_decision(G, k):  
 return False  
 if len(G) is k:  
 return G.keys()  
  
 for edge in get\_all\_edges(G):  
 break\_link(G, edge[0], edge[1])  
 if k\_clique\_decision(G, k):  
 return k\_clique(G, k)  
 make\_link(G, edge[0], edge[1])  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 edges = [(1,2),(1,3),(1,4),(2,4)]  
 G = {}  
 for (x,y) in edges:  
 make\_link(G,x,y)  
  
 for x in G:  
 print x, G[x].keys()  
 print  
  
 k = 3  
 print "k =", k  
 print k\_clique(G, k)



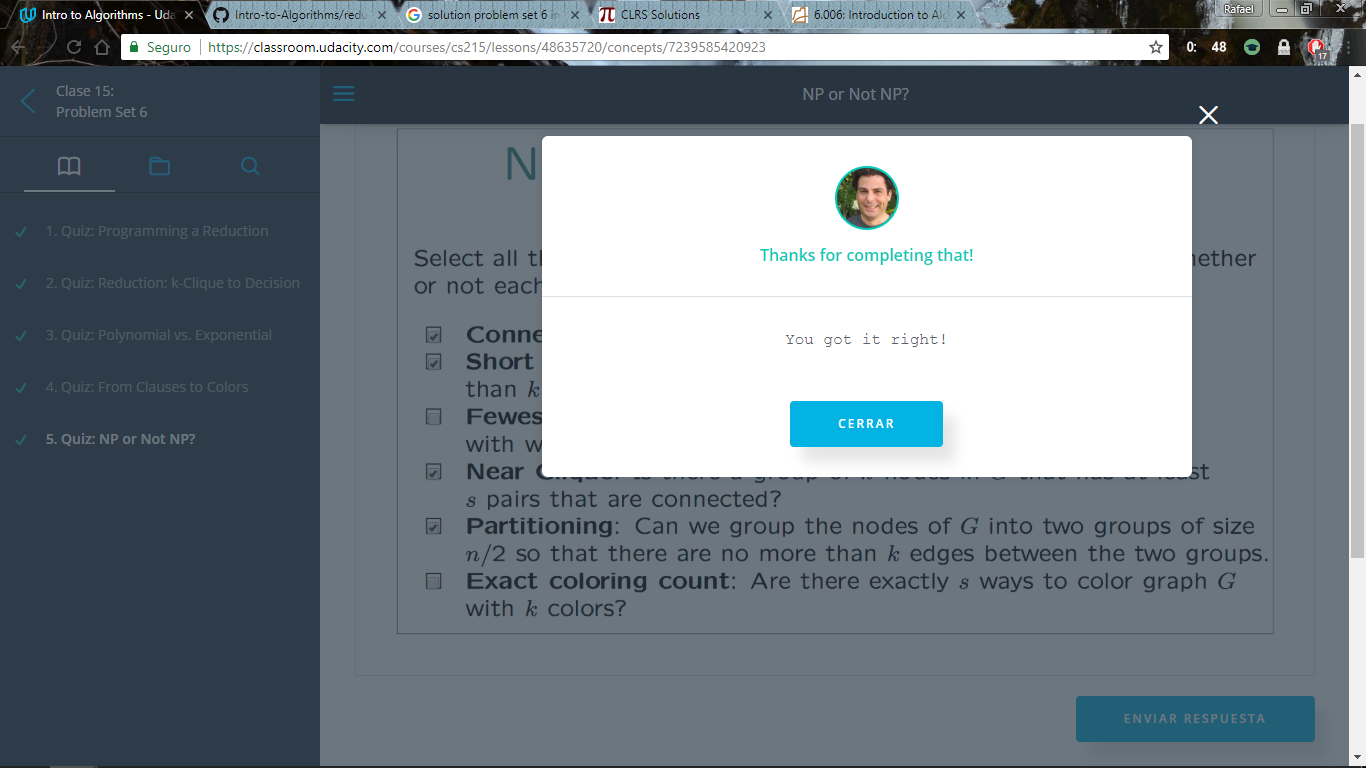
1. Polynomial vs. Exponential



1. From Clauses to Colors



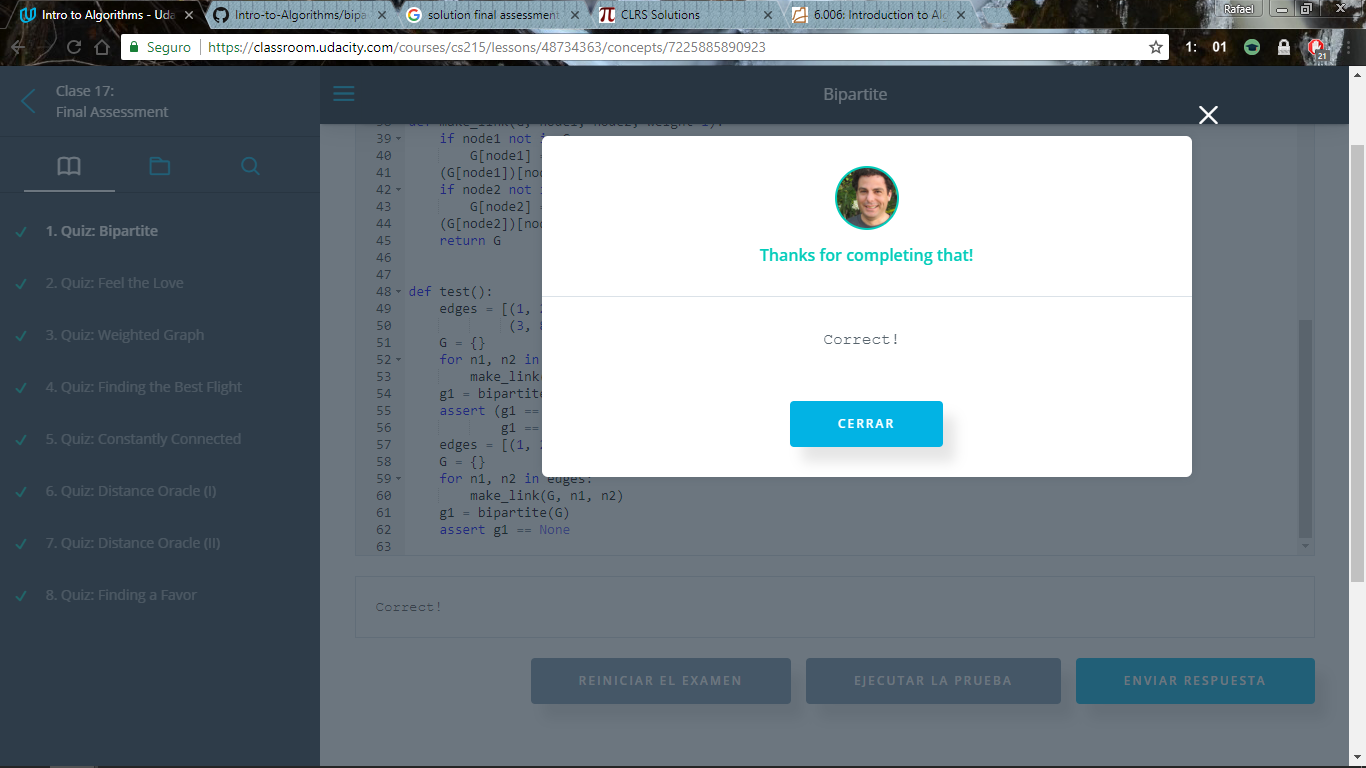
1. NP or Not NP?



**3.** Solución Final Exam

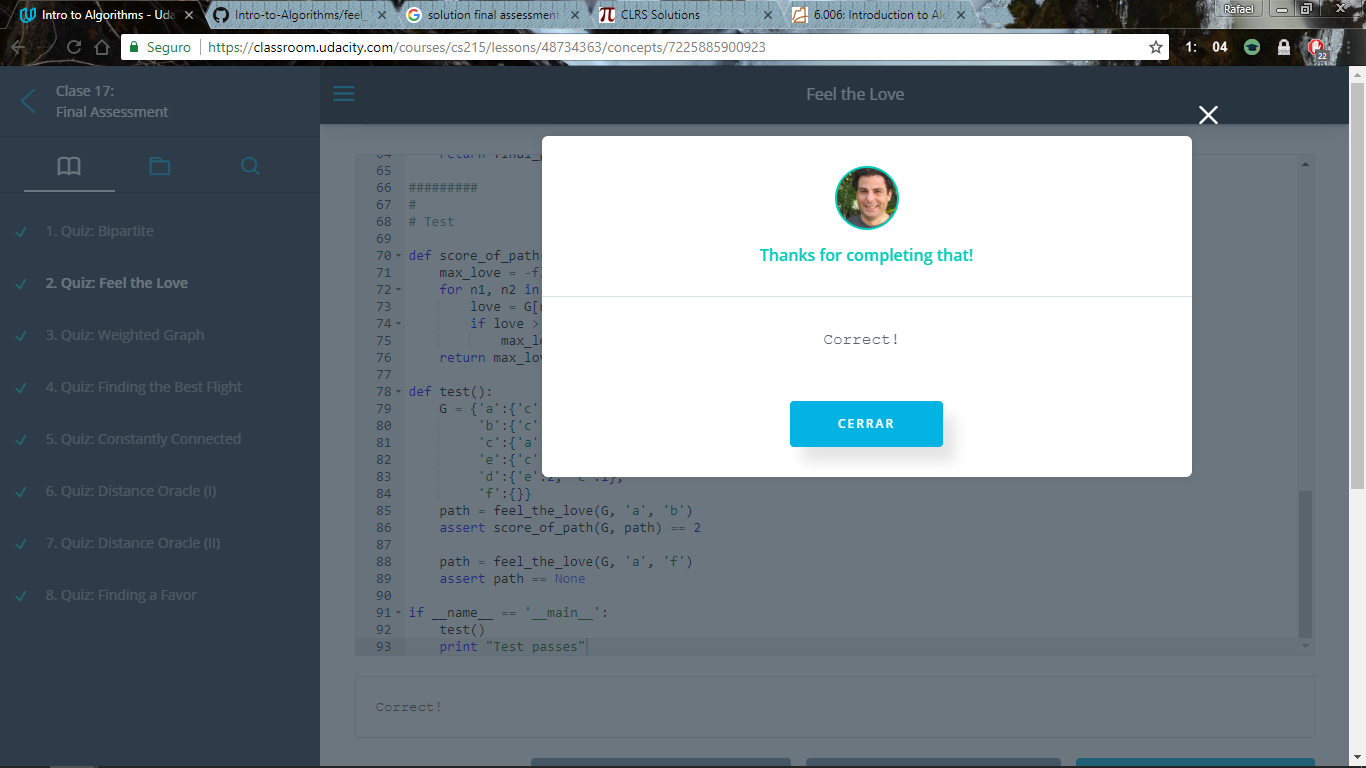
1. Bipartite

def bipartite(G):  
 checked = {}  
  
 def \_iter\_check(node, side):  
 if node in checked:  
 return  
 checked[node] = side  
 for neighbor in G[node]:  
 \_iter\_check(neighbor, not side)  
  
 for node in G:  
 \_iter\_check(node, True)  
  
 def \_valid(subset):  
 for node in subset:  
 for neighbor in G[node]:  
 if neighbor in subset:  
 return False  
 return True  
  
 left\_set = set(filter(lambda x: checked[x], checked))  
 right\_set = set(G.keys()) - left\_set  
  
 if \_valid(left\_set) and \_valid(right\_set):  
 return left\_set



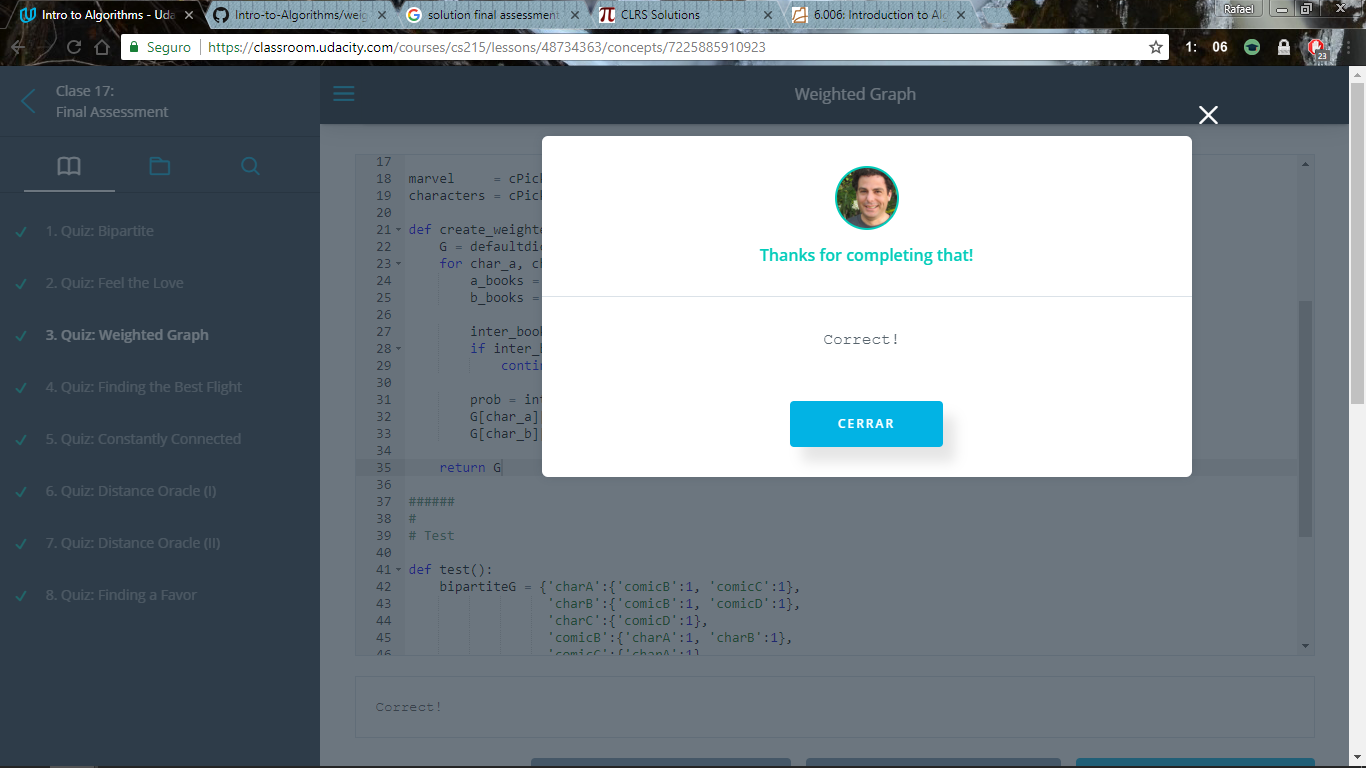
1. Feel the love

import heapq  
from collections import defaultdict  
  
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  
 path = dijkstra\_path(G, i)  
 if not j in path:  
 return None  
  
 node\_a, node\_b = max\_weight\_edge(G, i)  
 path\_a = path[node\_a]  
 path\_b = (dijkstra\_path(G, node\_b))[j]  
  
 return path\_a + path\_b  
  
def max\_weight\_edge(G, i):  
 max\_so\_far = -float('inf')  
 edge = None  
 reachable = dijkstra\_path(G, i)  
 for node in G:  
 for neighbor in G[node]:  
 if (G[node])[neighbor] > max\_so\_far and node in reachable:  
 max\_so\_far = (G[node])[neighbor]  
 edge = node, neighbor  
  
 return edge  
  
def dijkstra\_path(HG, v):  
 dist\_so\_far = {v: 0}  
 final\_dist = {}  
 final\_path = defaultdict(list)  
 heap = [(0, v)]  
 while dist\_so\_far:  
 (w, k) = heapq.heappop(heap)  
 if k in final\_dist or (k in dist\_so\_far and w > dist\_so\_far[k]):  
 continue  
 else:  
 del dist\_so\_far[k]  
 final\_dist[k] = w  
 for neighbor in [nb for nb in HG[k] if nb not in final\_dist]:  
 nw = final\_dist[k]+ HG[k][neighbor]  
 final\_path[neighbor] = final\_path[k] + [k]  
  
 if neighbor not in dist\_so\_far or nw < dist\_so\_far[neighbor]:  
 dist\_so\_far[neighbor] = nw  
 heapq.heappush(heap, (nw, neighbor))  
  
 for node in final\_path:  
 final\_path[node] += [node]  
 return final\_path



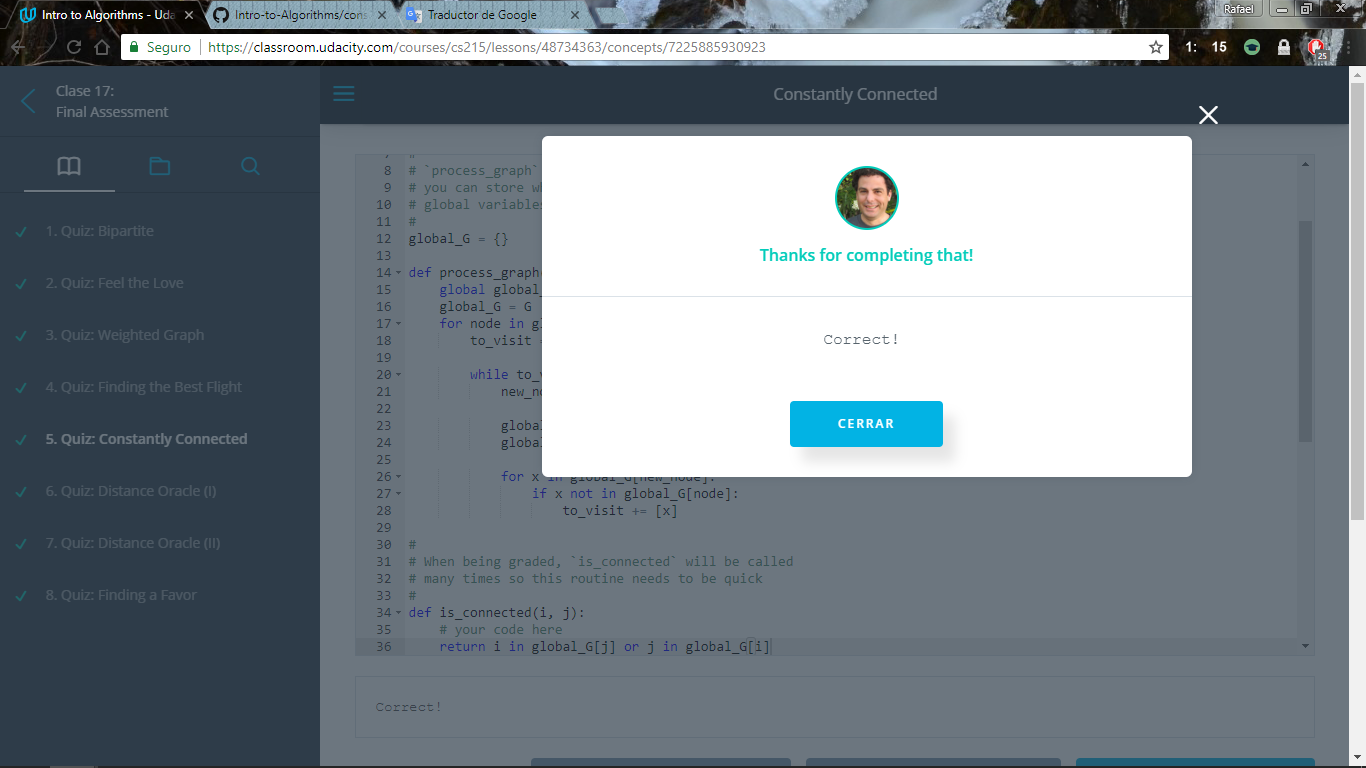
1. Weighted Graph

import itertools  
from collections import defaultdict  
import cPickle  
  
marvel = cPickle.load(open("smallG.pkl"))  
characters = cPickle.load(open("smallChr.pkl"))  
  
def create\_weighted\_graph(bipartiteG, characters):  
 G = defaultdict(dict)  
 for char\_a, char\_b in itertools.combinations(characters, 2):  
 a\_books = set(bipartiteG[char\_a])  
 b\_books = set(bipartiteG[char\_b])  
  
 inter\_book\_num = float(len(a\_books.intersection(b\_books)))  
 if inter\_book\_num == 0:  
 continue  
  
 prob = inter\_book\_num / (len(a\_books)+len(b\_books)-inter\_book\_num)  
 G[char\_a][char\_b] = prob  
 G[char\_b][char\_a] = prob  
  
 return G



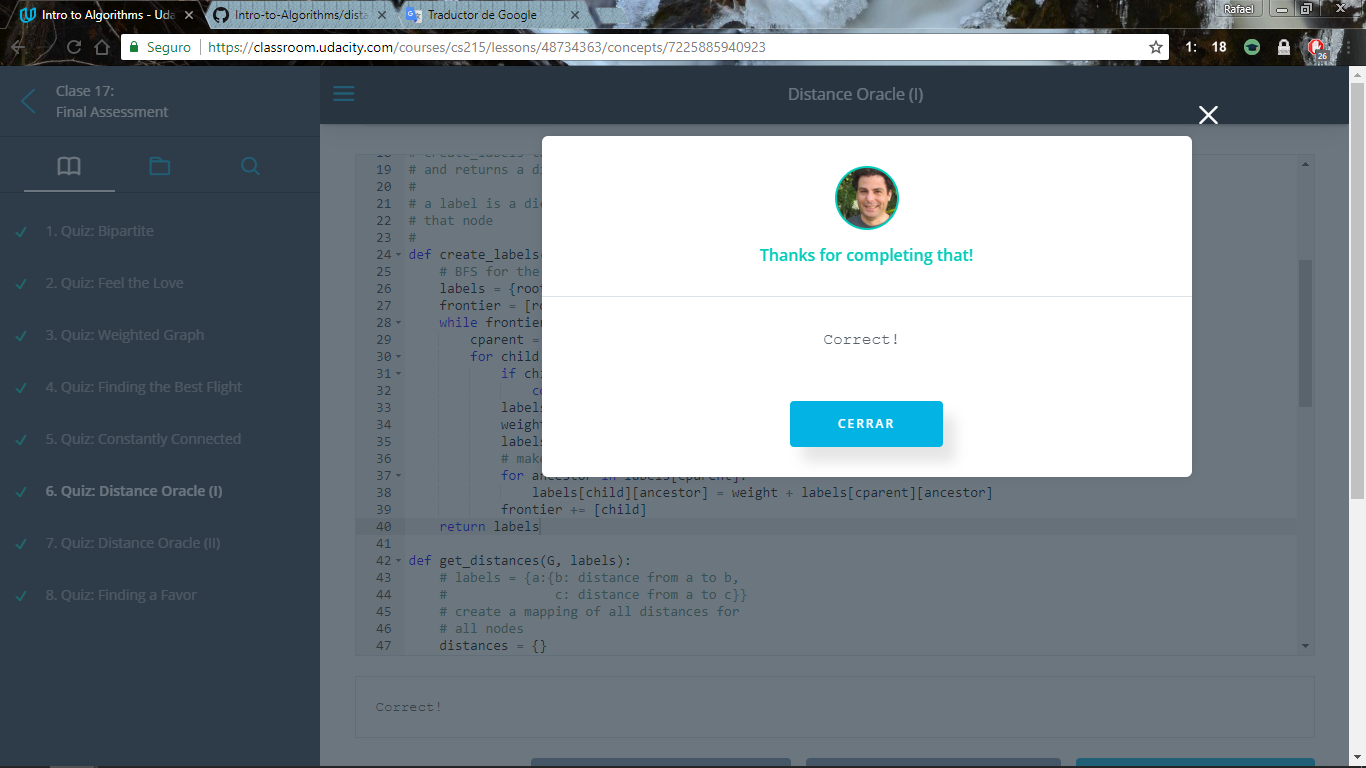
1. Constantly Connected

global\_G = {}  
  
def process\_graph(G):  
 global global\_G  
 global\_G = G  
 for node in global\_G:  
 to\_visit = global\_G[node].keys()  
  
 while to\_visit:  
 new\_node = to\_visit.pop()  
  
 global\_G[node][new\_node] = 1  
 global\_G[new\_node][node] = 1  
  
 for x in global\_G[new\_node]:  
 if x not in global\_G[node]:  
 to\_visit += [x]  
  
def is\_connected(i, j):  
 # your code here  
 return i in global\_G[j] or j in global\_G[i]



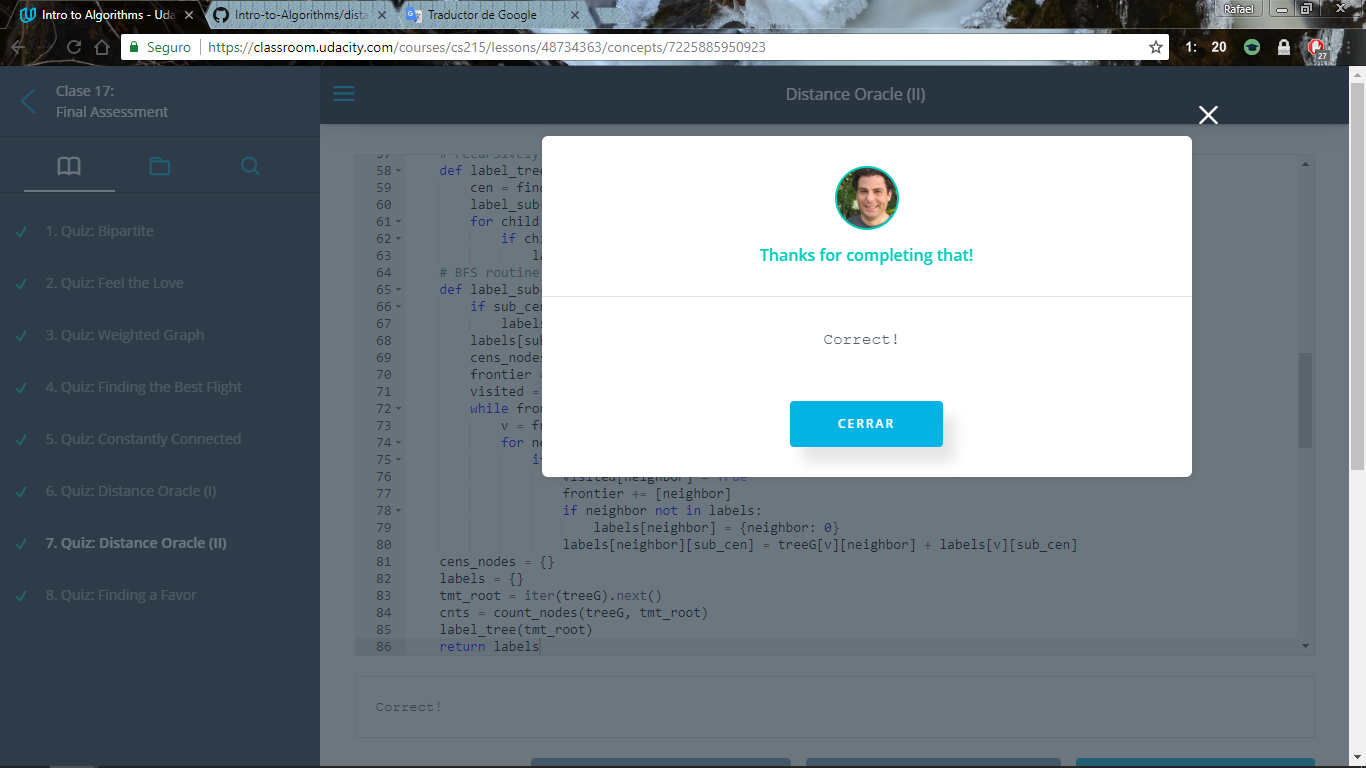
1. Distance Oracle (I)

def create\_labels(binarytreeG, root):  
 # BFS for the binary tree, meanwhile labeling each node in each level  
 labels = {root: {root: 0}}  
 frontier = [root]  
 while frontier:  
 cparent = frontier.pop(0)  
 for child in binarytreeG[cparent]:  
 if child in labels:  
 continue  
 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



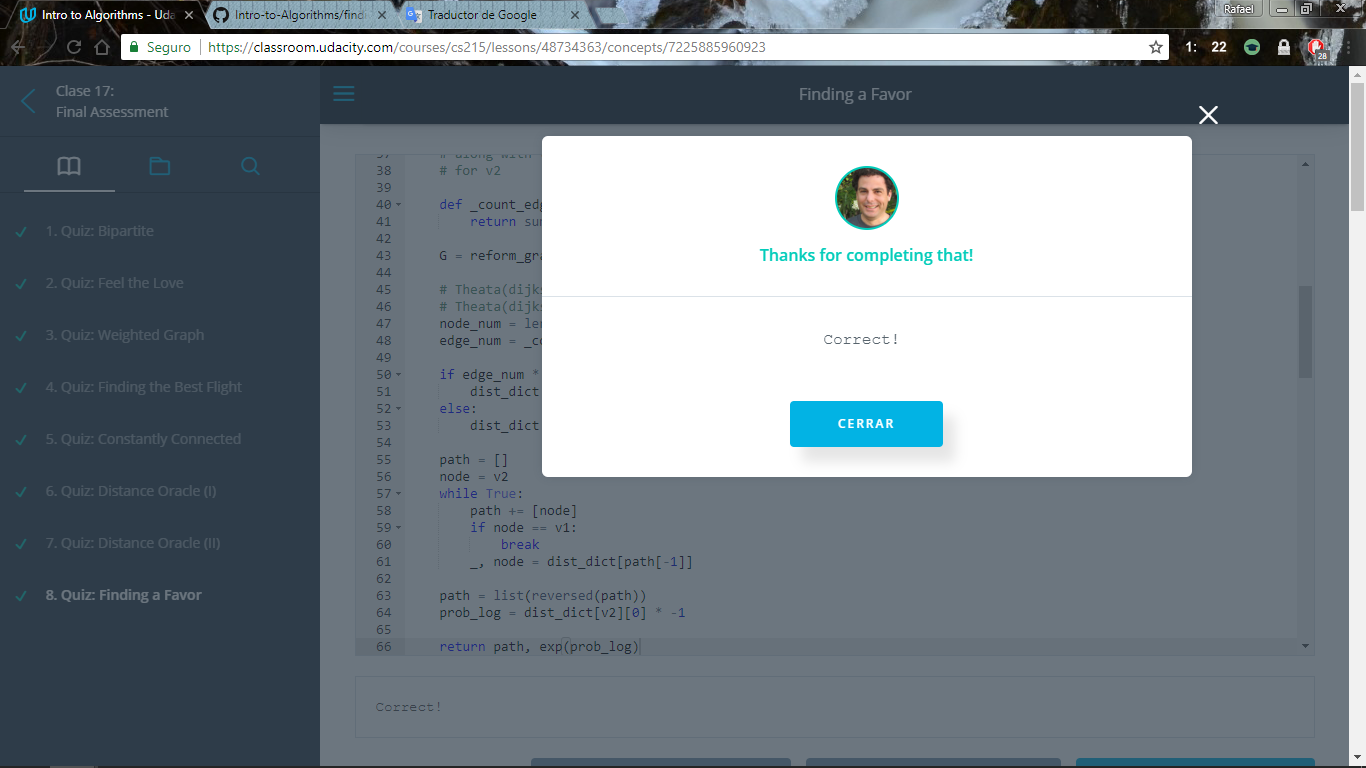
1. Distance Oracle (II)

def count\_nodes(treeG, node):  
 # count all sub-nodes including itself  
 cnts = {}  
 visited = {}  
 cnts[node] = count\_nodes\_rec(treeG, node, cnts, visited)  
 return cnts  
  
def count\_nodes\_rec(treeG, node, cnts, visited):  
 visited[node] = True  
 frontier = [node]  
 cnts[node] = 1  
 for v in treeG[node]:  
 if v not in visited:  
 cnts[node] += count\_nodes\_rec(treeG, v, cnts, visited)  
 return cnts[node]  
  
def create\_labels(treeG):  
 # find center node via rotation  
 def find\_cen(treeG, tmt\_root, cnts):  
 if cnts[tmt\_root] == 1:  
 return tmt\_root  
 mcc, mc = max((cnts[v], v) for v in treeG[tmt\_root] if v not in cens\_nodes)  
 # center node found!  
 if cnts[tmt\_root] - mcc >= mcc:  
 return tmt\_root  
 # rotate 'tmt\_root' to mc  
 cnts[mc] += cnts[tmt\_root] - mcc  
 cnts[tmt\_root] -= mcc  
 return find\_cen(treeG, mc, cnts)  
 # recursively finding center node for each 'sub-tree'  
 def label\_tree(tmt\_root):  
 cen = find\_cen(treeG, tmt\_root, cnts)  
 label\_sub(cen)  
 for child in treeG[cen]:  
 if child not in cens\_nodes:  
 label\_tree(child)  
 # BFS routine for tagging each descendant node with its sub-center node  
 def label\_sub(sub\_cen):  
 if sub\_cen not in labels:  
 labels[sub\_cen] = {}  
 labels[sub\_cen][sub\_cen] = 0  
 cens\_nodes[sub\_cen] = True  
 frontier = [sub\_cen]  
 visited = {}  
 while frontier:  
 v = frontier.pop(0)  
 for neighbor in treeG[v]:  
 if neighbor not in visited and neighbor not in cens\_nodes:  
 visited[neighbor] = True  
 frontier += [neighbor]  
 if neighbor not in labels:  
 labels[neighbor] = {neighbor: 0}  
 labels[neighbor][sub\_cen] = treeG[v][neighbor] + labels[v][sub\_cen]  
 cens\_nodes = {}  
 labels = {}  
 tmt\_root = iter(treeG).next()  
 cnts = count\_nodes(treeG, tmt\_root)  
 label\_tree(tmt\_root)  
 return labels



1. Finding a Favor

from heap import \*  
from operator import itemgetter  
from collections import defaultdict  
from math import log, exp  
  
def reform\_graph(G):  
 new\_graph = defaultdict(dict)  
 for node in G:  
 for neighbor in G[node]:  
 new\_graph[node][neighbor] = log(G[node][neighbor]) \* -1  
 return new\_graph  
  
def maximize\_probability\_of\_favor(G, v1, v2):  
 # your code here  
 # call either the heap or list version of dijkstra  
 # and return the path from `v1` to `v2`  
 # along with the probability that v1 will do a favor  
 # for v2  
  
 def \_count\_edges():  
 return sum([len(G[v]) for v in G])  
  
 G = reform\_graph(G)  
  
 # Theata(dijkstra\_list) = Theata(n^2 + m) = Theata(n^2)  
 # Theata(dijkstra\_heap) = Theata(n \* log(n) + m \* log(n)) = Theata(m \* log(n))  
 node\_num = len(G.keys())  
 edge\_num = \_count\_edges()  
  
 if edge\_num \* log(node\_num) <= node\_num \*\* 2:  
 dist\_dict = dijkstra\_heap(G, v1)  
 else:  
 dist\_dict = dijkstra\_list(G, v1)  
  
 path = []  
 node = v2  
 while True:  
 path += [node]  
 if node == v1:  
 break  
 \_, node = dist\_dict[path[-1]]  
  
 path = list(reversed(path))  
 prob\_log = dist\_dict[v2][0] \* -1  
  
 return path, exp(prob\_log)



**4.**

1. Si decimos que f(n) es el costo mínimo de cubrir la cinta hasta n, se puede encontrar a partir de las soluciones para los largos n-2 y n-3

Esto se logra cuando añadimos la ficha con el costo mínimo para que cubra lo que falta

1. def cubrir(c2, c3, n):

dp = [0 for i in range(n+1)]

dp[1] = dp[2] = min(c2,c3)

for i in range(3, n+1):

dp[1] = min(dp[i-1] + c2, dp[i-1] + c3, dp[i-2] + c2, dp[i-1] + c3, dp[i-3] + c3)

return dp

d)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| cubrir(5,7,10 | 0 | 5 | 5 | 7 | 10 | 12 | 14 | 17 | 19 | 21 | 24 |

**5.**

c) Se tienen las recurrencias:

d) Siempre es cero porque En es complemento de Bn, pues existe solo una forma de organizar una cinta de tamaño n.  
En = 0 ya que no hay más formas de organizar las fichas

e)

def tiling(n):

A = [0 for i in range(n+1)]

C = [0 for i in range(n+1)]

D = [0 for i in range(n+1)]

D[0] = A[1] = 1

for i in range(2, n+1):

C[i] = A[i-1]

A[i] = D[i-1] + C[i-1]

D[i] = D[i-1] + 2\*A[i-1]

return D[n]

f)

|  |  |  |  |
| --- | --- | --- | --- |
| N | 10 | 50 | 100 |
| Dn | 571 | 156886956080403 | 31208688988045323113527764971 |