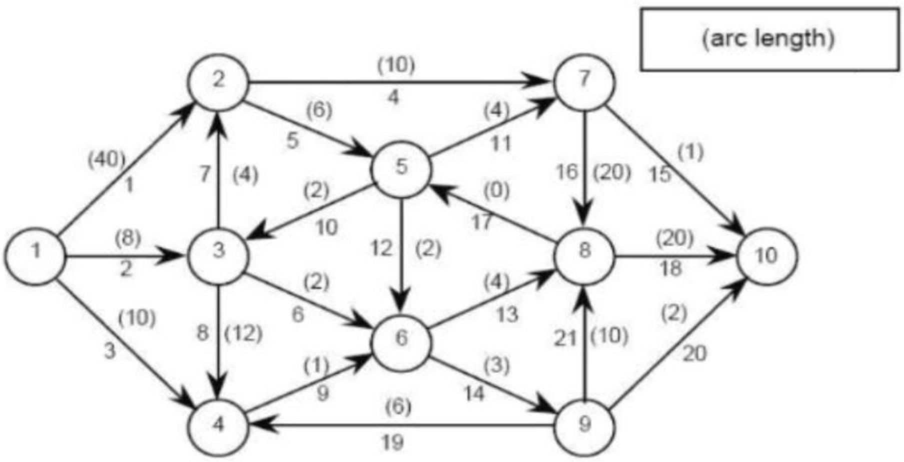
Taller 2: Grafos, Complejidad Computacional, Programación Dinámica

Entrega: David Jose Leon Aroca

- Considere el grafo de la Figura 1(Solo tenga en cuenta los pesos en paréntesis)

Figura 1: Grafo para el punto 1



a.

Pasos del Algoritmo de Dijkstra:

1. Se inicializan las distancias en infinito ya que no se sabe cuál es la menor se asume que todas son muy altas, excepto la primera que tiene distancia 0 dado que se asume es el nodo de origen o nodo inicial.

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	∞	∞	∞	∞	∞	∞	∞	∞	∞

2. Se visitan los nodos adyacentes al nodo actual, exceptuando los marcados, los demás nodos no marcados se toman como Vj.

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	40	8	10	∞	∞	∞	∞	∞	∞

3. Se calcula la distancia para el nodo actual con sus vecinos mediante la fórmula dt(Vj) = Da + d(a, Vj). Es decir que la distancia del nodo ‘Vj’ es la distancia que actualmente tiene el nodo en el vector D más la

distancia desde el nodo actual(a) al nodo Vj. Si la distancia es menor que la distancia almacenada en el vector, se actualiza el vector con esta distancia tentativa.

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	∞	∞	∞	∞	∞	∞

4. Se marca como completo el nodo actual y se toma como próximo nodo el de menor valor en D almacenando los valores en una cola de prioridad.

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	inf	inf	inf	inf	inf	inf

5. Se repite el procedimiento desde el paso 3 mientras sigan existiendo nodos no marcados.

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	inf	10	inf	inf	inf	inf

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	inf	10	inf	inf	inf	inf

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	18	10	inf	inf	inf	inf

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	18	10	inf	14	13	inf

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	18	10	inf	14	13	15

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	14	10	inf	14	13	15

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	14	10	18	14	13	15

Dado que ya no existe ningún nodo no visitado o en infinito, se toman estas como las distancias más cortas desde el nodo 1.

b. Ejecute el algoritmo de Bellman-Ford detallando claramente los pasos ejecutados

Algoritmo de Bellman-Ford:

1. Se inicializa el grafo colocando distancias en infinito excepto el nodo inicial que se toma con distancia 0.

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	∞	∞	∞	∞	∞	∞	∞	∞	∞

2. Se toma un diccionario de padres y uno de distancias finales

Nodo	P	P1	P1	P1	P2	P2	P2	P3	P3	P3
Dist.	0	40	8	10	46	10	50	14	13	51

3. Se visita cada arista un total de número de nodos -1 veces, buscando y reemplazando por la distancia más corta hasta ese nodo, después se comprueba si hay ciclos negativos y el resultado es una suma de la lista de los vértices en orden de la ruta más corta para cada nodo.

Nodo	1	2	3	4	5	6	7	8	9	10
Dist.	0	12	8	10	14	10	18	14	13	15

c. Ejecute el algoritmo de Floyd-Warshal detallando claramente los pasos ejecutados

Algoritmo de Floyd-Warshal:

1. Se crea una matriz de nxn siendo n el número de nodos que componen el grafo

Nodos	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

2. Se recorre la matriz llenando en cada casilla correspondiente la distancia que hay entre los nodos.

Nodos	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	40	8	10	∞	∞	∞	∞	∞	∞
2	∞	0	∞	∞	6	∞	10	∞	∞	∞
3	∞	4	0	12	∞	2	∞	∞	∞	∞
4	∞	∞	∞	0	∞	1	∞	∞	∞	∞
5	∞	∞	2	∞	0	2	4	∞	∞	∞
6	∞	∞	∞	∞	∞	0	∞	4	3	∞

7	∞	∞	∞	∞	∞	∞	0	20	∞	1
8	∞	∞	∞	∞	0	∞	∞	0	∞	20
9	∞	∞	∞	19	∞	∞	∞	10	0	20
10	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

3. Tomando K= 1, se empieza a iterar sobre la matriz con ayuda de la funcion camino mínimo, con esta se comparan las distancias que hay entre los nodos adyacentes, verificando cual es la más corta y se va sumando hasta obtener la distancia más corta entre nodos.

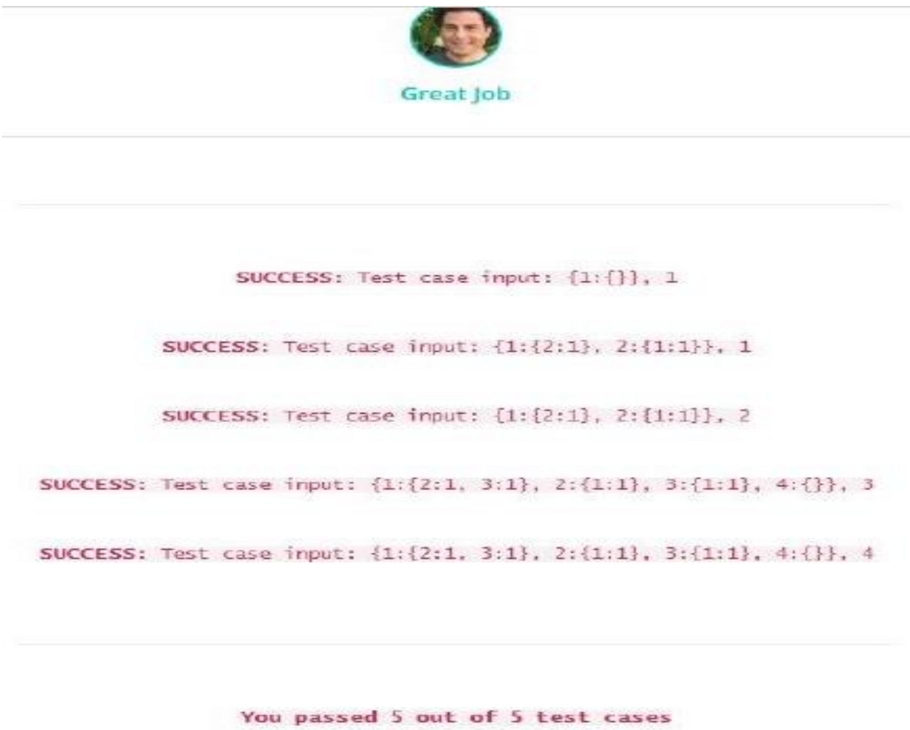
Nodos	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	12	8	10	14	10	18	14	13	15
2	∞	0	8	17	6	8	10	12	11	11
3	∞	4	0	11	6	2	10f	6	5	7
4	∞	11	7	0	5	1	9	5	4	6
5	∞	6	2	11	0	2	4	6	5	5
6	∞	10	6	9	4	0	8	4	3	5
7	∞	26	22	31	20	22	0	20	25	1
8	∞	6	2	11	0	2	4	0	5	5
9	∞	16	12	6	10	7	14	10	0	2
10	∞	∞	∞	∞	∞	∞	∞	∞	∞	0

- Resuelva los puntos del problem Set 6 del curso Algoritmos de Udacity. Incluya el código correspondiente con un screenshot de aceptación para cada problema

a. Programming a Reduction

```
def independent_set_decision(H, s):  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()
```



b. Reduction: k-Clique to Decision

```
def k_clique(G, k):
k = int(k)  if not
k_clique_decision(G, k):
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()
```

Great Job

SUCCESS: Test case input: {1:{}}, 1

SUCCESS: Test case input: {1:{2:1, 3:1}, 2:{1:1}, 3:{1:1}}, 3

SUCCESS: Test case input: {1:{2:1, 3:1, 4:1}, 2:{1:1, 4:1}, 3:{1:1}, 4:{1:1, 2:1}}, 3

SUCCESS: Test case input: {1:{2:1, 3:1, 4:1}, 2:{1:1, 3:1, 4:1}, 3:{1:1, 2:1, 4:1}, 4:{1:1, 2:1, 3:1}}, 4

You passed 4 out of 4 test cases

c. Polynomial vs. Exponential

Great Job

You got it right!

CLOSE

d. 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. $n = 4x + 10y + 8$)

e. 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.

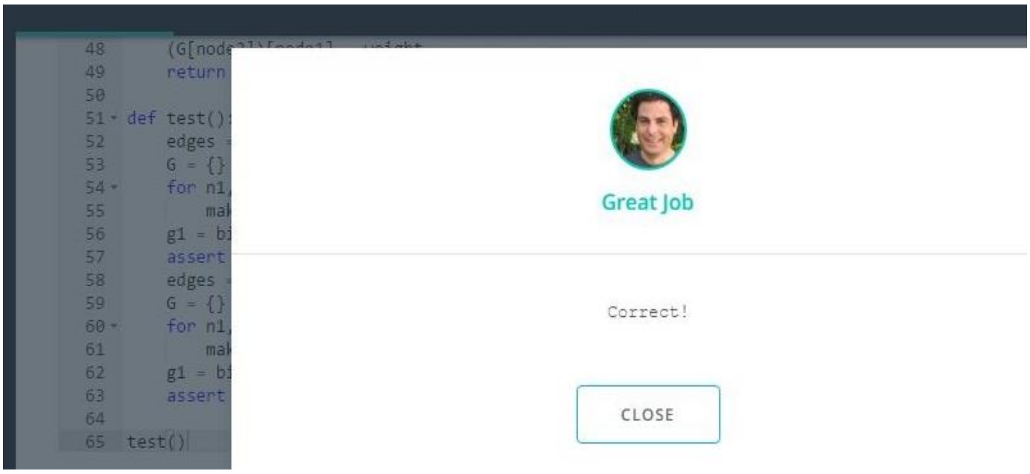
- [ ] **Connectivity**: Is there a path from x to y in G?
- [ ] **Short path**: Is there a path from x to y in G that is no more than k steps long?
- [x] **Fewest colors**: Is k the absolute minimum number of colors with which G can be colored?
- [x] **Near Clique**: Is there a group of k nodes in G 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?

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

a. Bipartite

```
from collections import deque def
bipartite(G):
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

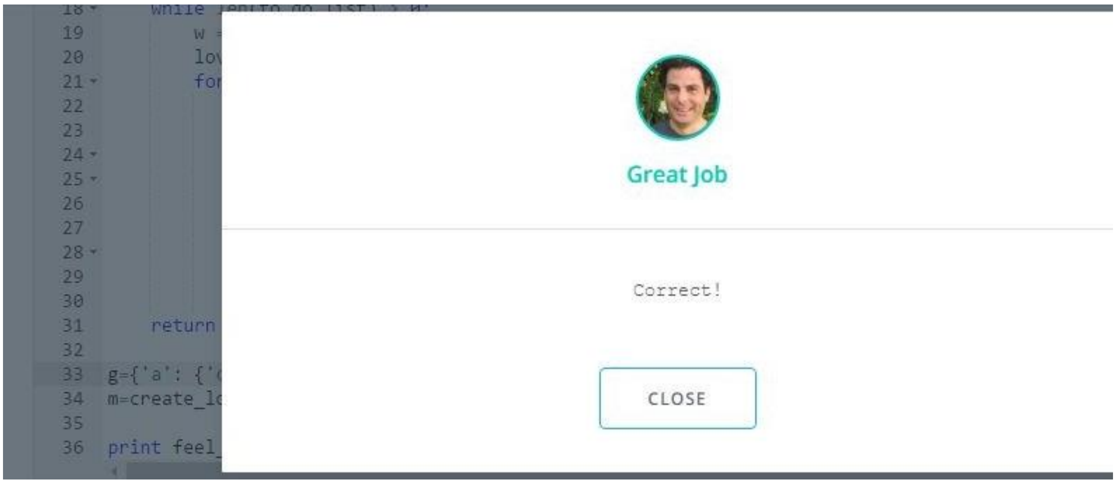


b. Feel the love

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

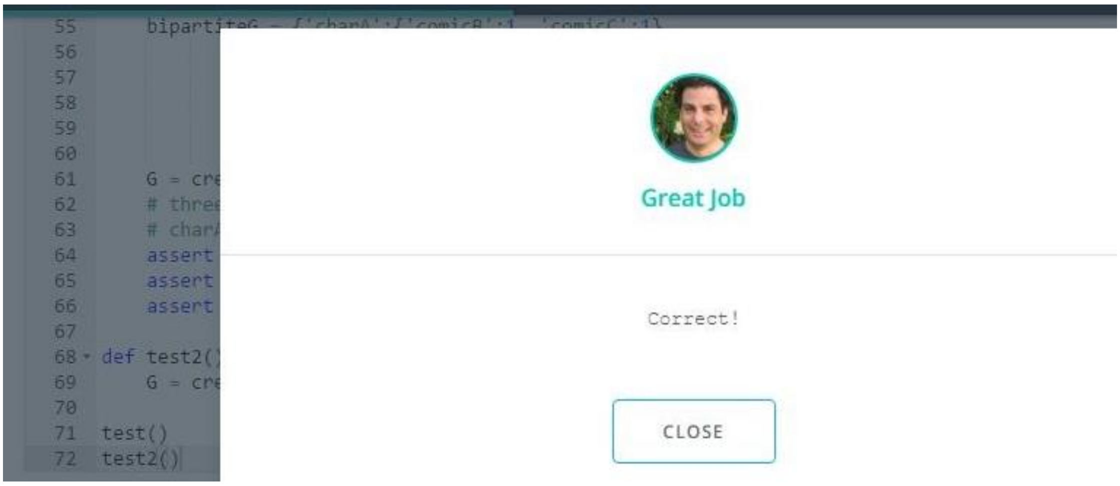
def create_love_paths(G, v):
love_so_far = {} love_so_far[v] = (0, [v]) to_do_list
= [v] while len(to_do_list) > 0: w = to_do_list.pop(0)
love, path = love_so_far[w] for x in G[w]: new_path
= path + [x] new_love = max([love, G[w][x]])
if x in love_so_far: if new_love >
love_so_far[x][0]:
love_so_far[x] = (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')



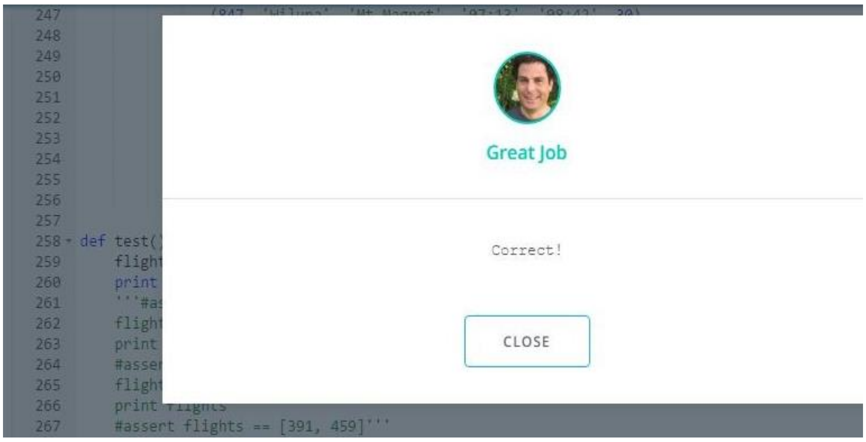
c. Weighted Graph

```
def create_weighted_graph(bipartiteG, characters):
comic_size = len(set(bipartiteG.keys()) -
set(characters)) 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
```



d. Finding the best Flight

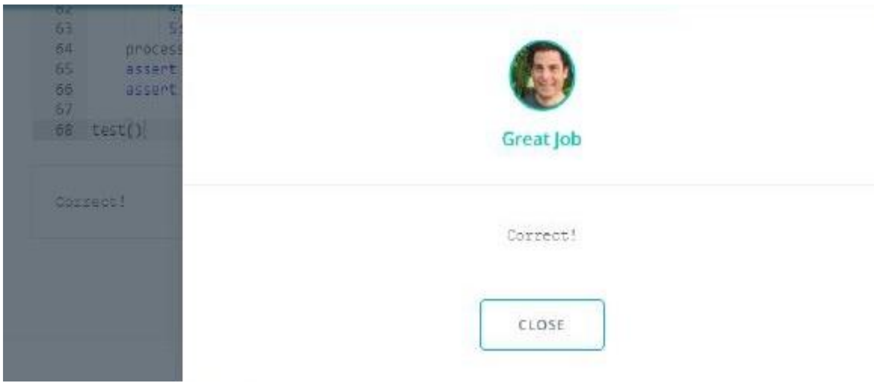
```
import heapq def find_best_flights(flights,
origin, destination):
G = make_graph(flights)
R = find_route(G, origin, destination)
return R

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
```



e. Constantly Connected

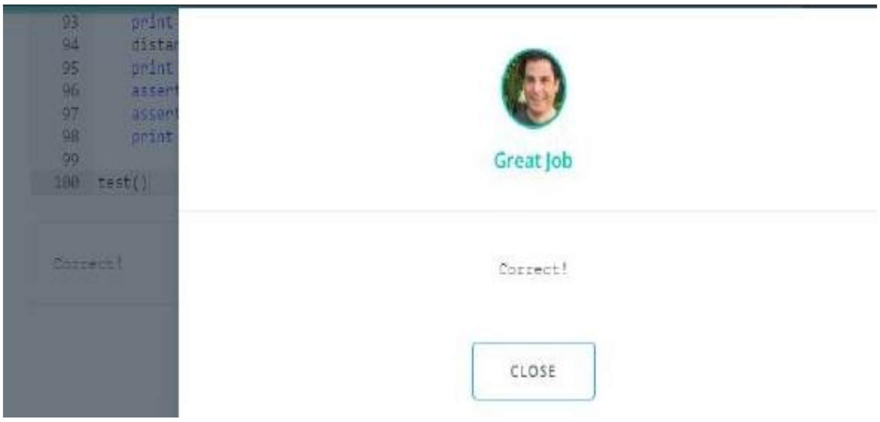
def process_graph(G):
global conns conns = {} groupId = 0 nodes = G.keys() while len(conns) < len(G): c_node = nodes.pop() if c_node not in conns: conns[c_node] = groupId open_list = [c_node] while open_list: reached = open_list.pop() for neighbor in G[reached]: if neighbor not in conns: open_list.append(neighbor) conns[neighbor] = groupId if neighbor in nodes: del nodes[nodes.index(neighbor)]
groupId += 1

def is_connected(i, j):
global conns
return conns[i] == conns[j]



f. Distance Oracle (I)

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 for ancestor in labels[cparent]: labels[child][ancestor] = weight + labels[cparent][ancestor]
frontier += [child]
return labels

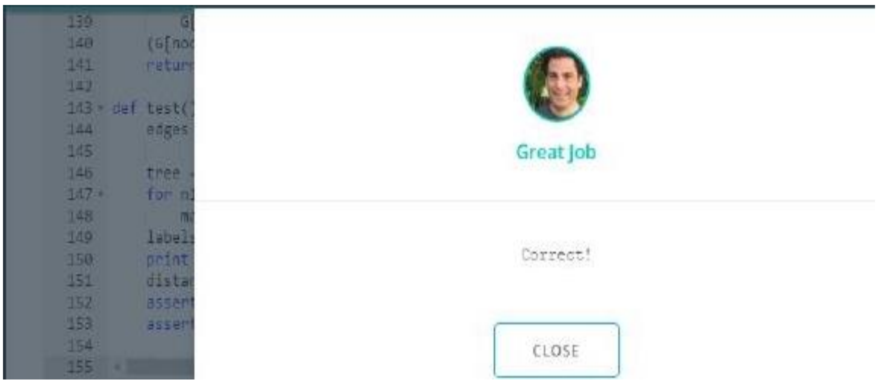


g. Distance Oracle (II)

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)

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)

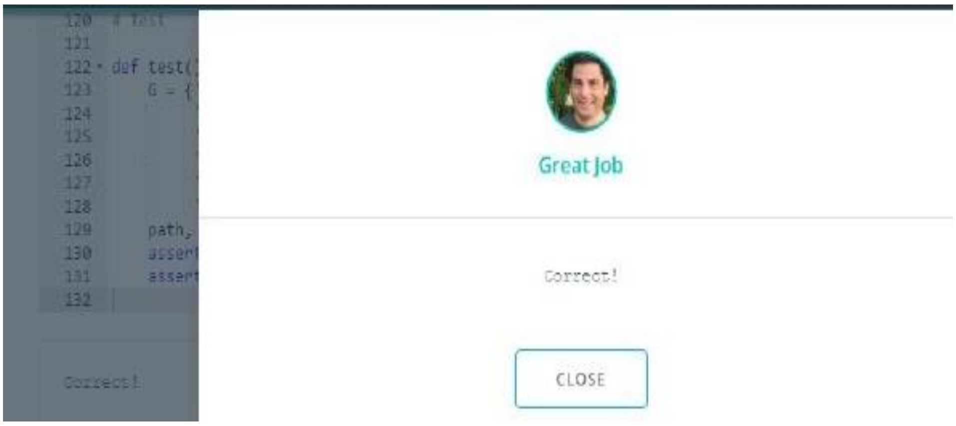
def create_labels(treeG):
found_roots = set()
labels = {} update_ labels(treeG, labels, found_roots,
ter(treeG).next()) return labels



h. Finding a Favor

def maximize_probability_of_favor(G, v1, v2):
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
```



- 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 que un problema de longitud n sea resuelto, es necesario resolver con anterioridad el problema de una longitud menor a n, al calcular las soluciones de menores longitudes se puede dar solución al problema de longitud n. Por lo tanto si este problema puede ser resuelto para una subestructura de menor longitud, podrá ser resuelto para n.

    b. Plantee una ecuación recursiva para resolver el problema

$P_n = \{ \min (P_2, P_3)$ si $n <= 2$ $\min(2P_2, P_3)$ si $n = 3$

$\min (P_i + P_{n-i})$ con $1 <= i <= n$ si $n > 3$

    c. Escriba un programa en Python que resuelva el problema de manera 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 para el caso C2 = 5, C3 = 7 y n = 10:

n	0	1	2	3	4	5	6	7	8	9	10
Cubrir(5; 7; n)	0	5	5	7	10	12	14	17	19	21	24

Problema de cubrimiento de un tablero 3 xn con chas de dominó:

a. Obtenga y estudie la presentación en [CS97SI-DP]

b. Revise el problema de cubrir un tablero de 3 n con fichas de dominó (tamaño 2 1 o 1 2)

c. Plantee las recurrencias para An, Bn, Cn y Dn

Casos base:

- $B_0 = B_1 = 0$
- $A_0 = 0$
- $A_1 = 1$
- $C_0 = C_1 = 0$
- $D_0 = 1$
- $D_1 = 0$

Recurrencias:

- $A_n = D_{n-1} + C_{n-1}$
- $B_n = 0$
- $C_n = A_{n-1}$
- $D_n = D_{n-2} + 2 * A_{n-1}$

d. ¿Porque En siempre es 0?

N Impar: Si N es impar entonces las líneas superiores e inferiores serán impares y dado que solo se pueden cubrir con dominós de tamaño 2, será imposible cubrirlas por completo.
N Par: Si N es par entonces la fila de la mitad tendría un número impar de espacios lo que haría imposible llenarlos, haciendo imposible cubrir la figura.

e. Escriba un programa en Python para calcular Dn

```
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)
```

f. Calcule Dn para n = 10; 50; 100

10	50	100
203	156886956	3120868898804532 31