UNIVERSIDAD NACIONAL DE COLOMBIA

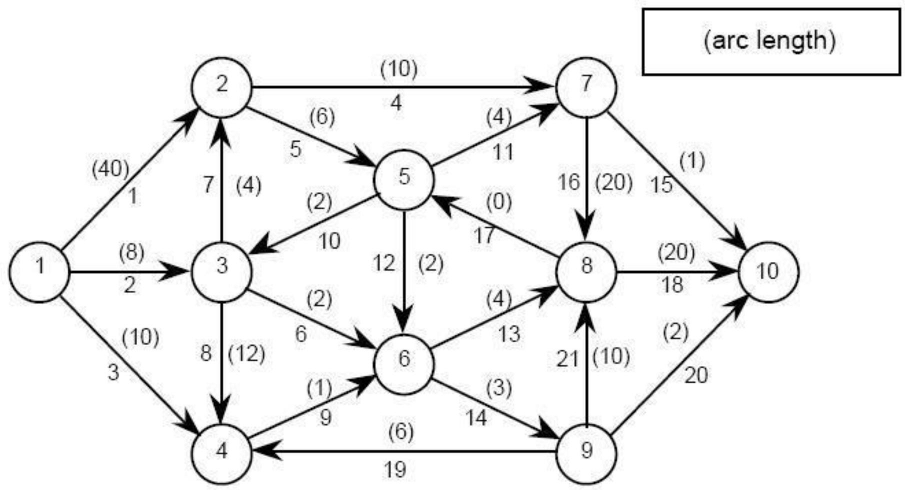
FACULTAD DE INGENIERIA

ALGORITMOS – Taller 2

EDWIN RICARDO MAHECHA PARRA – 1013679888

1. Considere el grafo de la Figura 1 (solo tenga en cuenta los pesos en paréntesis):

   Figura 1: Grafo para el punto 1



   a. Ejecute el algoritmo de Dijkstra detallando claramente los pasos ejecutados.
      1. Se inicializa el arreglo de distancias en infinito, y el nodo de origen con distancia 0.
      2. Se crea una cola de prioridad que recibe la pareja (d,v) en donde d es la distancia para el nodo v. Se agrega a dicha lista el nodo de origen (source).
      3. Se procede a sacar el nodo de la lista y si se cumple que la distancia que tenia almacenada la cola de prioridad es mayor que la distancia almacenada en un arreglo de distancias, se procede a iterar por todos los nodos adyacentes al nodo que se sacó de la lista. Es decir:
         If(d>dist[u]): iterar por los adyacentes.
      4. En el paso de iteración por TODOS los adyacentes se revisa que: if(dist[u] + weight[v.second] < dist[u.first]) entonces se actualiza dist[u.first] = dist[u] + weight[v.second]. Cabe notar que u se refiere a el nodo que se saca de la lista en el paso anterior y v se refiere a uno de los nodos adyacentes. first se refiere al nodo y second al peso.
         Al actualizar dist, se agrega v.first y el nueo peso a la cola de prioridad.
      5. El paso 3 y 4 se repiten hasta que la cola de prioridad este vacía.

   Simulación:

nodo	1	2	3	4	5	6	7	8	9	10
	0	inf	inf	inf	inf	inf	inf	inf	inf	inf
	0	40	8	10	inf	inf	inf	inf	inf	inf
	0	12	8	10	inf	10	inf	inf	inf	inf
	0	12	8	10	inf	4	inf	inf	inf	inf
	0	12	8	10	16	4	22	inf	inf	inf
	0	12	8	10	16	4	22	8	7	inf
	0	12	8	10	16	4	22	8	7	9
	0	12	8	10	8	4	22	8	7	9
	0	12	8	10	8	4	12	8	7	9

b. Ejecute el algoritmo de Bellman-Ford detallando claramente los pasos ejecutados.
   1. Se inicializa el grafo, y todas las distancias se ponen en infinito, menos el nodo inicial al que se le pone distancia 0. Se crea un arreglo que contiene los nodos padres para cada nodo y se inicializa en -1.
   2. Se visitan todas los aristas #nodos-1 veces y se realiza el paso de relajación.
   3. Se verifica si hay ciclos negativos. Si no se posee ningún ciclo negativo se procede a dar la salida.
      NOTA: *En este caso el resultado es similar a la tabla de arriba, puesto que el grafo no posee ciclos negativos.*

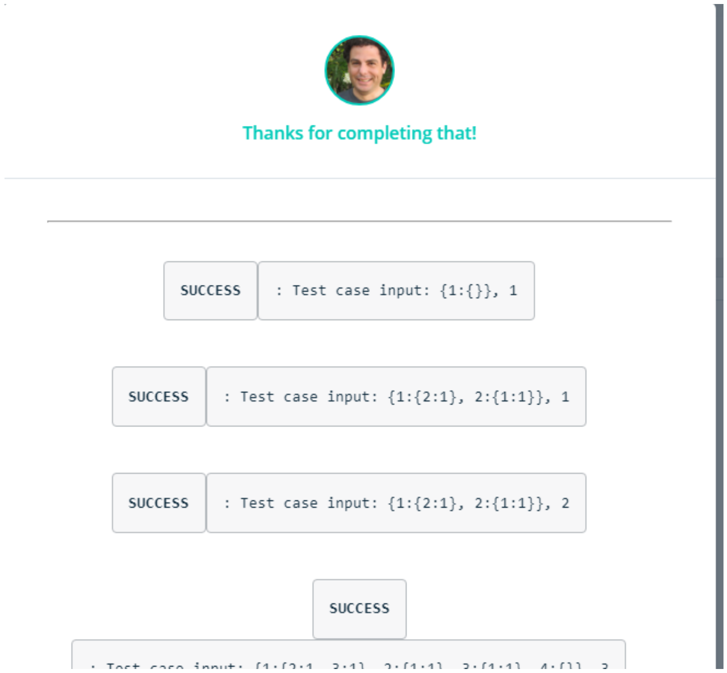
c. Ejecute el algoritmo de Floyd-Warshal detallando claramente los pasos ejecutados.
   1. Se crea una matriz de adyacencia, no un arreglo de aristas. Se inicializa en infinito, menos en las posiciones en las que hay algún peso. Recordemos que Floyd-Warshal calcula las distancias de todos a todos.
   2. Se procede con una verificación para determinar si la distancia que posee una arista actualmente mejora si se pasa por un nodo intermedio o si es mejor dejar la distancia que ya esta.

```
for(int k = 0 ; k < N;k++)
    for(int i = 0 ; i < N; i++)
        for(int j = 0 ; j< N; j++)
            d[i][j] = min(d[i][j],d[i][k] + d[k][j]);
```

   3. La matriz de salida posee las distancias de todos a todos los nodos.

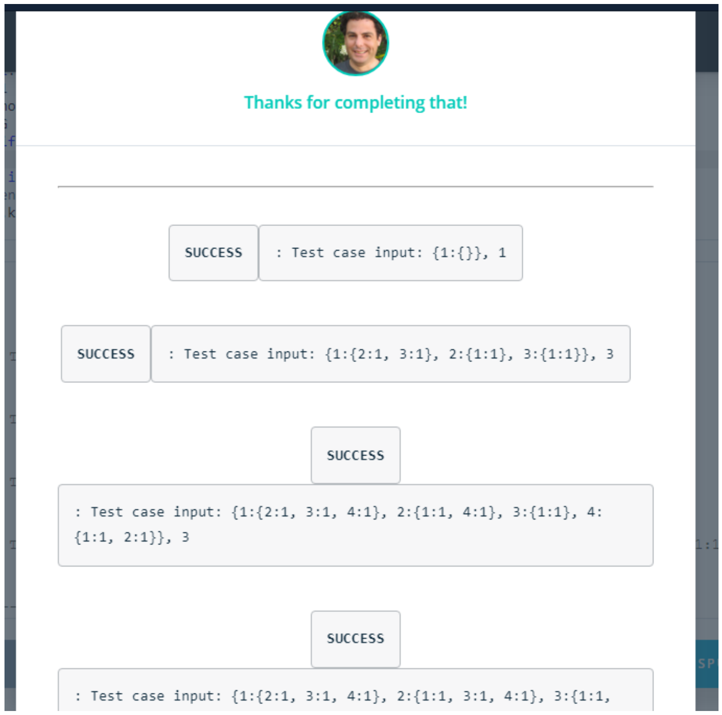
```
I, 15, 8, 10, 14, 10, 18, 14, 13, 15,
I, 15, 8, 17, 6, 8, 10, 12, 11, 11,
I, 7, 8, 11, 6, 2, 10, 6, 5, 7,
I, 14, 7, 10, 5, 1, 9, 5, 4, 6,
I, 9, 2, 11, 6, 2, 4, 6, 5, 5,
I, 13, 6, 9, 4, 6, 8, 4, 3, 5,
I, 29, 22, 31, 20, 22, 24, 20, 25, 1,
I, 9, 2, 11, 0, 2, 4, 6, 5, 5,
I, 19, 12, 6, 10, 7, 14, 10, 10, 2,
I, I, I, I, I, I, I, I, I, I,
```

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.
    a. Programming reduction:



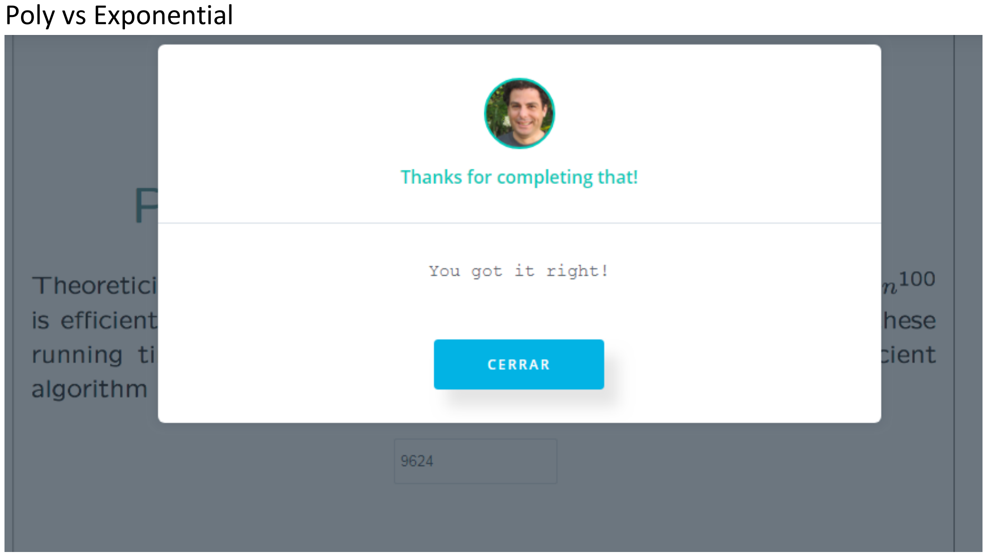

```python
def independent_set_decision(H, s):
    nodes = H.keys()
    I = {}
    for node1 in H:
        for node2 in H:
            if node2 not in H[node1]:
                make_link(I, node1, node2)
    return k_clique_decision(I, s)
```

    b. Reduction k-clique to decision

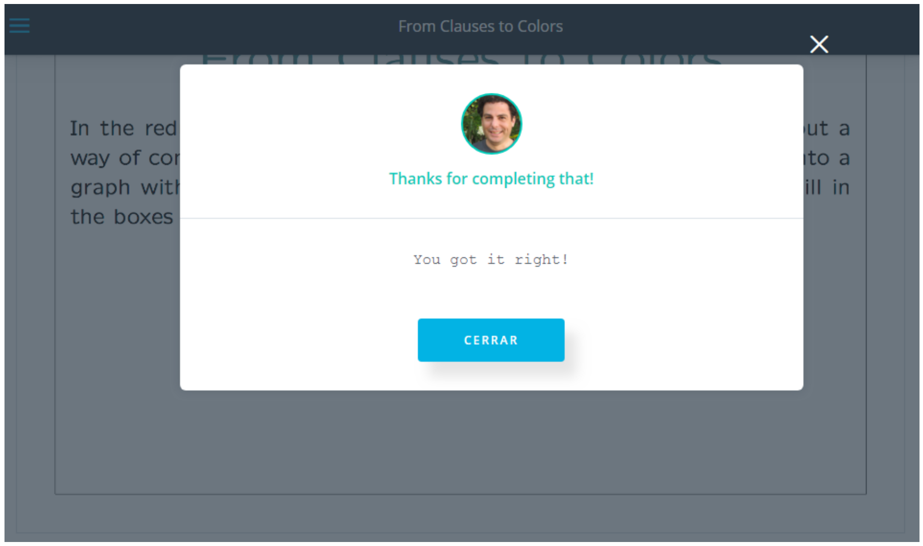



```python
def k_clique(G, k):
    k = int(k)
    if not k_clique_decision(G, k):return False
    if k == 1:return [G.keys()[0]]
    for node1 in G.keys():
        for node2 in G[node1].keys():
            G = break_link(G, node1, node2)
            if not k_clique_decision(G, k):G = make_link(G, node1,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

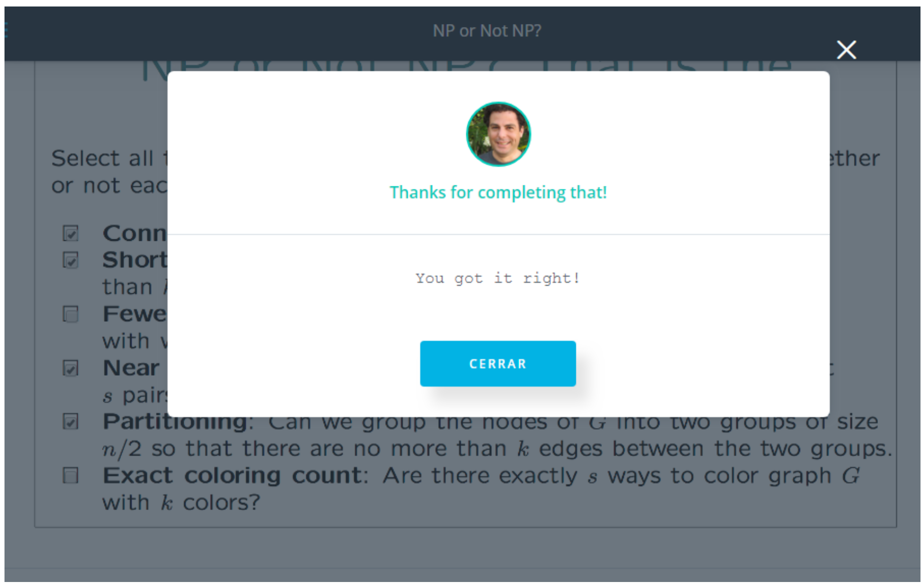
c. Poly vs Exponential



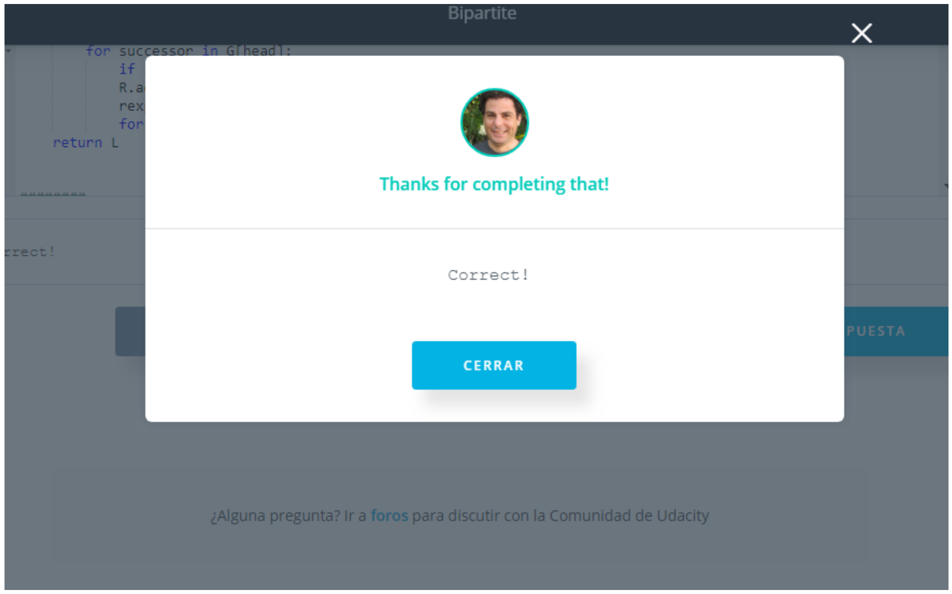
d. From Clauses to Colors



e. NP or not NP

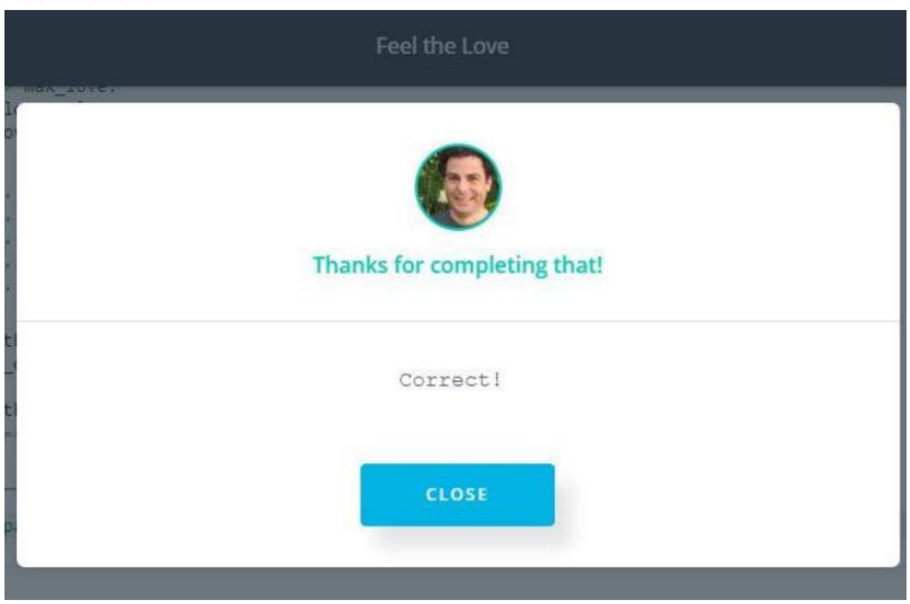


3. Resuelva los puntos del Final Exam del curso Algorithms de Udacity. Incluya el código correspondiente con un screenshot de aceptación para cada problema.
    a. Bipartite




```python
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

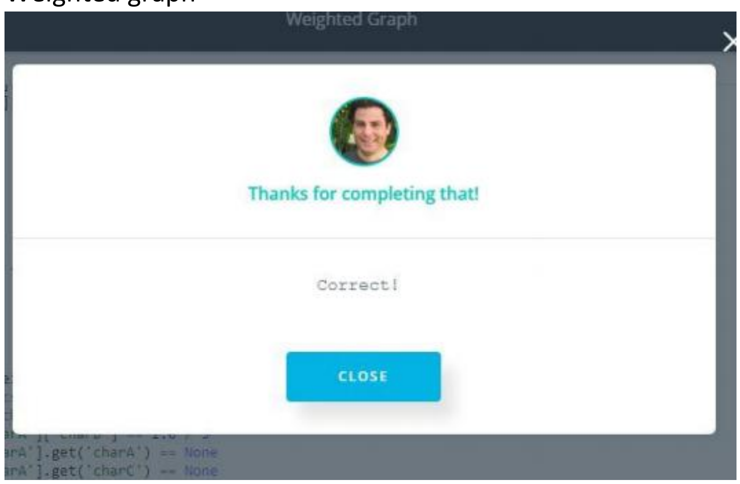


```python
def feel_the_love(G, i, j):
    # return a path (a list of nodes) between `i` and `j`,
    # with `i` as the first node and `j` as the last node,
    # or None if no path exists
    result = dijkstra(G, i)
    if j in result:return result[j][1]
    return None

def dijkstra(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=dijkstra(g,'a')
```

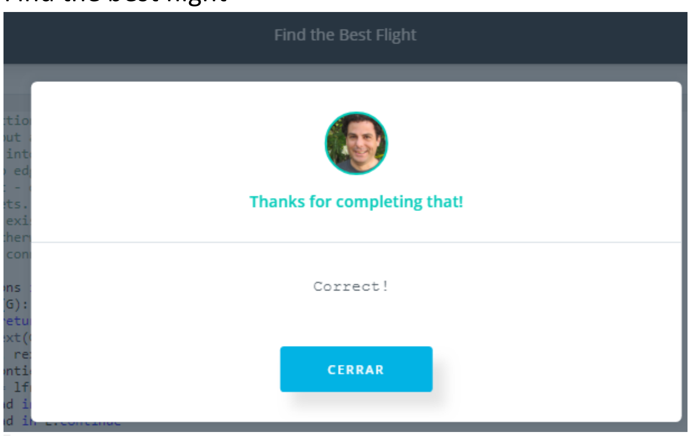
c. Weighted graph



```python
from marvel import marvel, characters

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])
```

d. Find the best flight



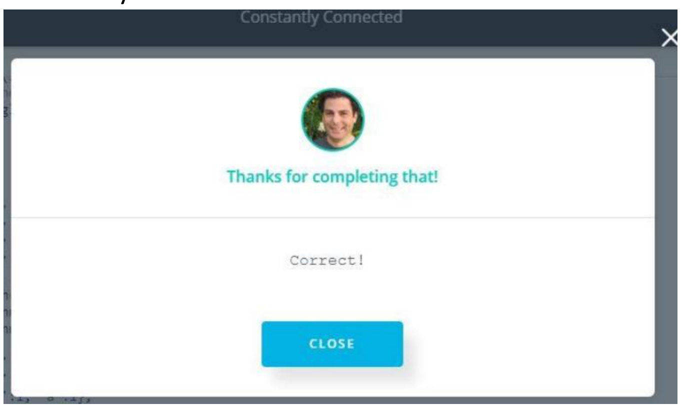

```python
def find_best_flights(flights, origin, destination):
    G = make_graph(flights)
    R = find_route(G, origin, destination)
    return R


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
```

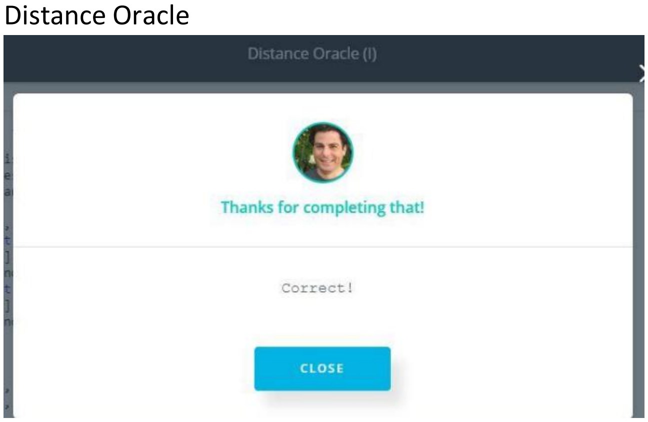
```python
def find_route(G, origin, destination):
    heap = [(0,0,None,[])]
    while heap:
     c_cost, c_away, c_start, c_path = heapq.heappop(heap)
     if not c_path:
            c_town = origin
     else:
            c_town = G[c_path[-1]]['dest']
     if c_town == destination:
         return 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 conected



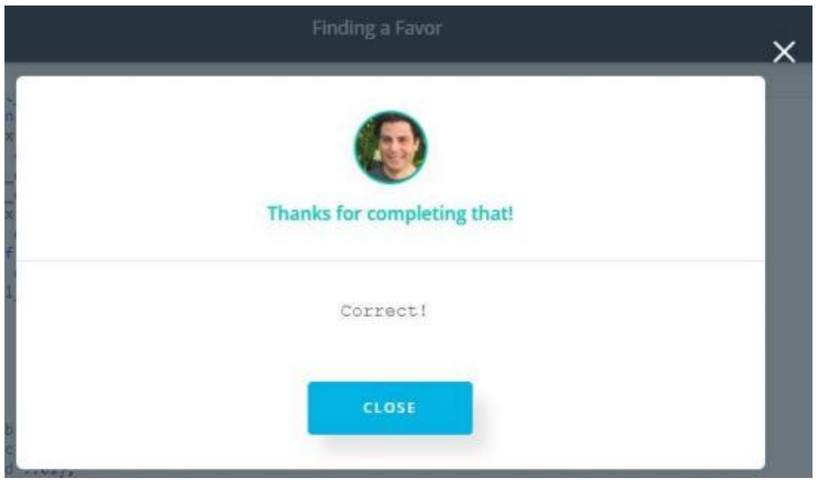

```python
def process_graph(G):
    # your code here
    global conns
    conns = {}
    groupId = 0
    nodes = G.keys()
    while len(conns) < len(G):
     c_node = nodes.pop()
     if c_node not in conns: conns[c_node] = groupId
     open_list = [c_node]
     while open_list:
            reached = 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
```

f. Distance Oracle



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

g. Finding a Favor




```python
def maximize_probability_of_favor(G, v1, v2):
    def _count_edges():
        return sum([len(G[v]) for v in G])
    G = reform_graph(G)
    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. 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 fichas que tenga costo mínimo. La longitud de la secuencia de fichas 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
    Similar al problema de rod cuting. En este caso poseemos dos fichas con determinada longitud y costo asociado, y se busca minimizar el costo total. Debido a esto se puede definir una ecuación recursiva que dependa de las soluciones de n-2 y n-3 que son las longitudes de restar una de las fichas a la longitud total en determinado momento.
    b. Plantee una ecuación recursiva para resolver el problema

$$f(n) = 0 \; si \; n = 0$$
$$f(n) = \min(c2, c3) \; si \; n \leq 2$$

$$f(n) = \min(f(n-1) + c2, f(n-1) + c3, f(n-2) + c2, f(n-2) + c3, f(n-3) + c3) \;\; en\ otro\ caso$$

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

```
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. 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	21

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

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

$$A_n = D_{(N-1)} + C_{(N-1)}$$
$$C_n = A_{N-1}$$
$$D_n = D_{N-2} + 2 * C_{n-1}$$

d. ¿Por qué En siempre es 0?

Ya que si n es impar entonces las líneas inferior y superior también son impares. Como solo se pueden cubrir con dominos de tamaño 2 no es posible cubrirlas totalmente. Si n es par, la fila central tendrá impar espacios para llenar. Por esto no es posible con fichas 2x1 o 1x2 llenar 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

1. n = 10, 203
2. n = 50, 238039524083
3. n = 100, Tarda demasiado.