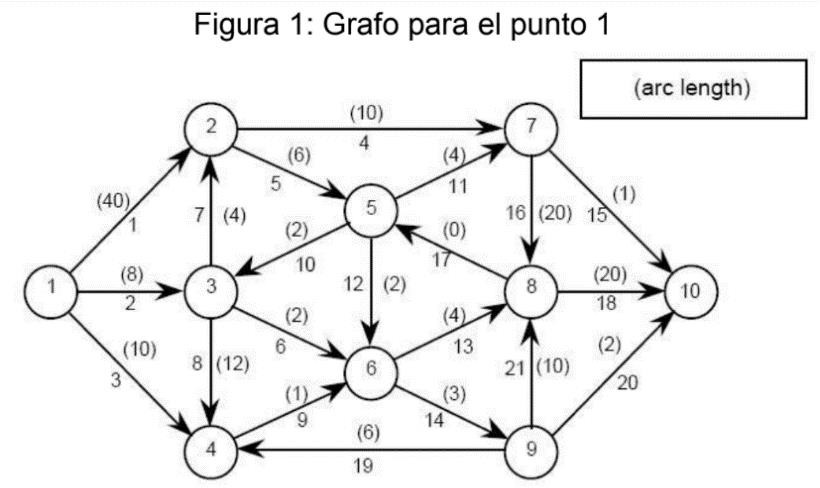
# Grafos, Complejidad Computacional, Programación Dinámica

1.

* 1. Ejecute el algoritmo de Dijkstra detallando claramente los pasos ejecutados.



Para comenzar, se marcan inicialmente las distancias que se conocen 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 estos valores van a una cola de prioridad. Esto se hace con el fin de verificar que nodos se encuentran aún pendientes por ser visitados y conocer el nodo siguiente a ser evaluado. Lo que se sabe es que será siempre el más cercano al nodo actual.

# Paso de relajación

Se evalúa el costo que tiene ir desde el nodo 1 hasta sus vecinos (*40 para ir hasta 2, 8 para ir hasta 3 y 10 para ir hasta 4),* ya que estas distancias son las mínimas conocidas hasta el momento, es el costo total de ir desde 1 hasta 2, 3 y 4 respectivamente.

**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*. Ahora hay que evaluar el 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** |

Se repite este proceso hasta actualizar cada uno de los vecinos del nodo actual y se toma el siguiente nodo en la cola de prioridad para saber, el vecino más cercano al nodo actual.

**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** |

Se repite esta lógica hasta que los vecinos del nodo actual ya hayan sido visitados y el algoritmo se detiene al conocer las distancias más cortas desde uno a los demás nodos.

**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** |

Dado que cada nodo tiene una distancia óptima con la cual se llega desde el nodo 1, se sabe que es el camino más corto hasta estos nodos.

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

Es necesario llevar un registro de las distancias del mismo hacia los demás nodos del grafo como se ilustró en el punto a. El algoritmo Bellman • Ford posee una complejidad de O(V \* E) y permite encontrar el camino más corto para grafos dirigidos con distancias o pesos negativos en sus aristas, por lo que debe ejecutarse el procedimiento de actualización a lo sumo V •1 veces.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **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** |

Se repite este procedimiento un total de V • 1 veces sin realización de cambios en las distancias encontradas. Finalizado el total de 9 iteraciones, se procede a verificar por cada una de las aristas el paso de relajación *(D[v] > D[u] + w(u, v))* un total de E veces. Si esta condición se hace verdadera, se sabe que existe un ciclo negativo dentro del grafo, sin embargo, este no es el caso.

Si un grafo *G = (V, E)* contiene únicamente ciclos positivos, el algoritmo de Bellman • Ford garantiza la correctitud de las distancias *D[v] = d (inicial , v)* para todo v perteneciente a V es la mínima.

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

Se trata de un caso de programación dinámica, pues se hace uso de las distancias ya conocidas hasta cierto nodo para averiguar los caminos derivados a partir de ahí, sin necesidad de recalcular todo desde el nodo inicial.

Lógica empleada:

*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*

Se realiza este paso de manera iterativa hasta que j = 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.

1. Resuelva los puntos del Problem Set 6 del curso Algorithms de Udacity. Incluya el código correspondiente con un screenshot de aceptación para cada problema.
   1. Programming a Reduction

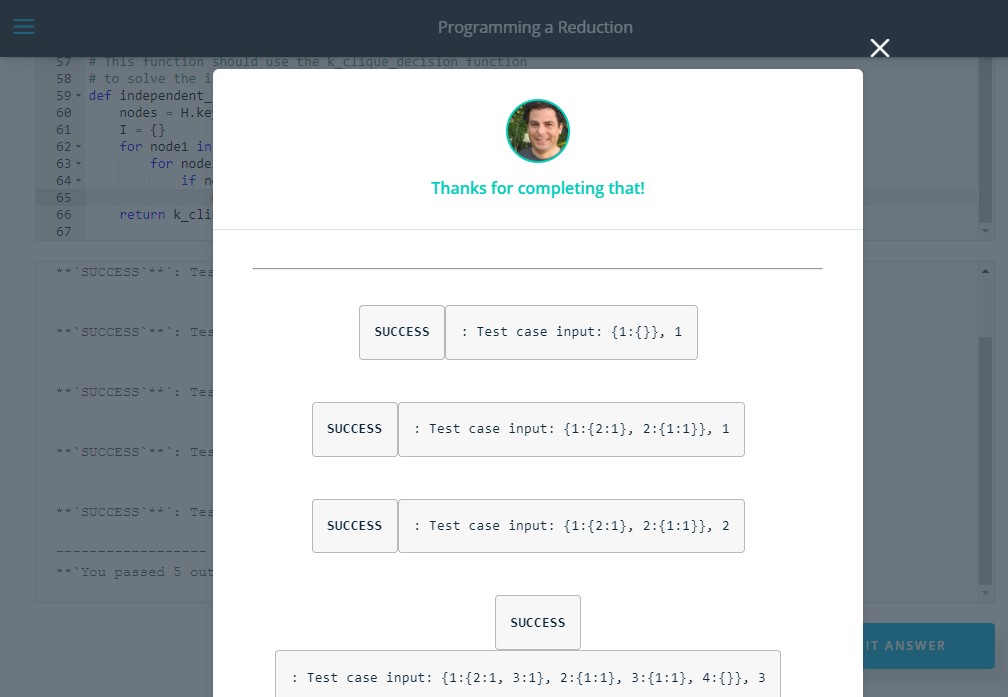
*def independent\_set\_decision(H, s): nodes = H.keys()*

*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

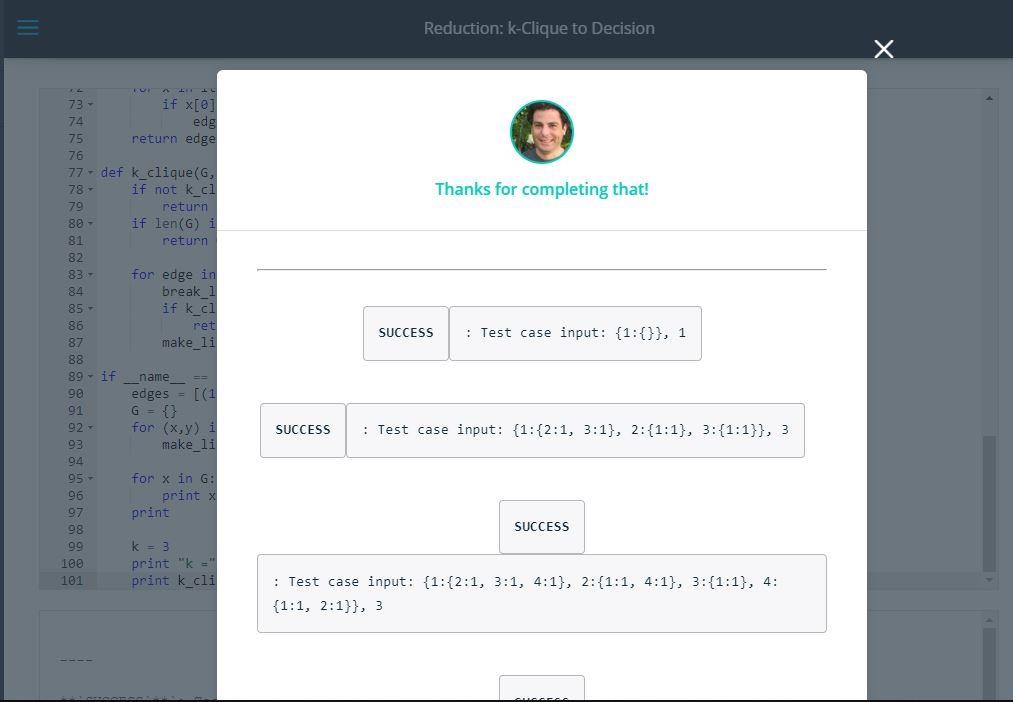
def k\_clique\_decision(G, k): nodes = G.keys()

for i in range(k, len(nodes) + 1):

for subset in k\_subsets(nodes, i): if is\_clique(G, subset):

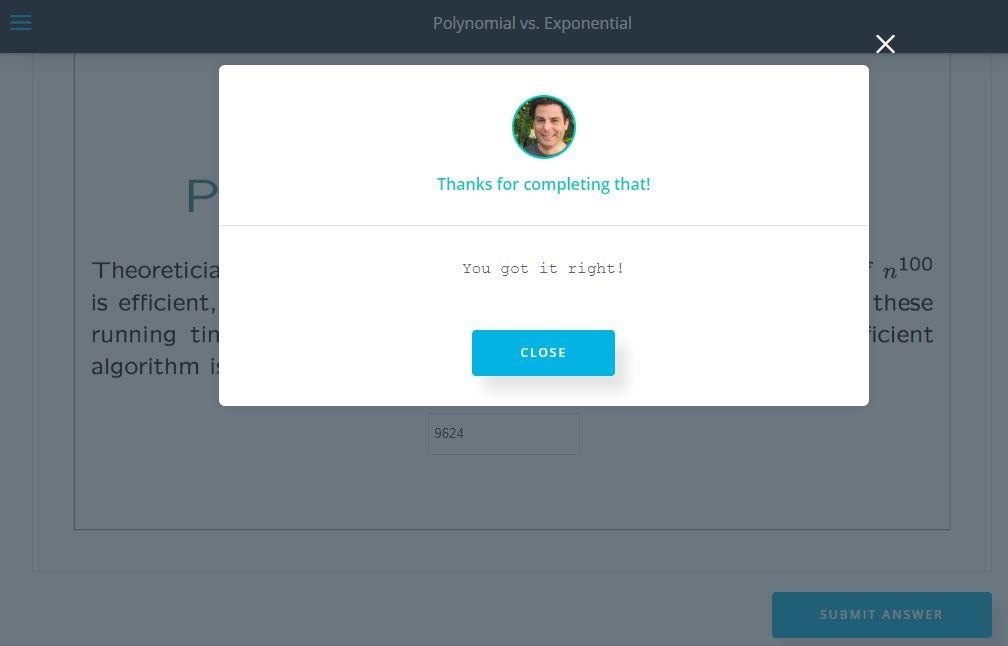
return True

return False

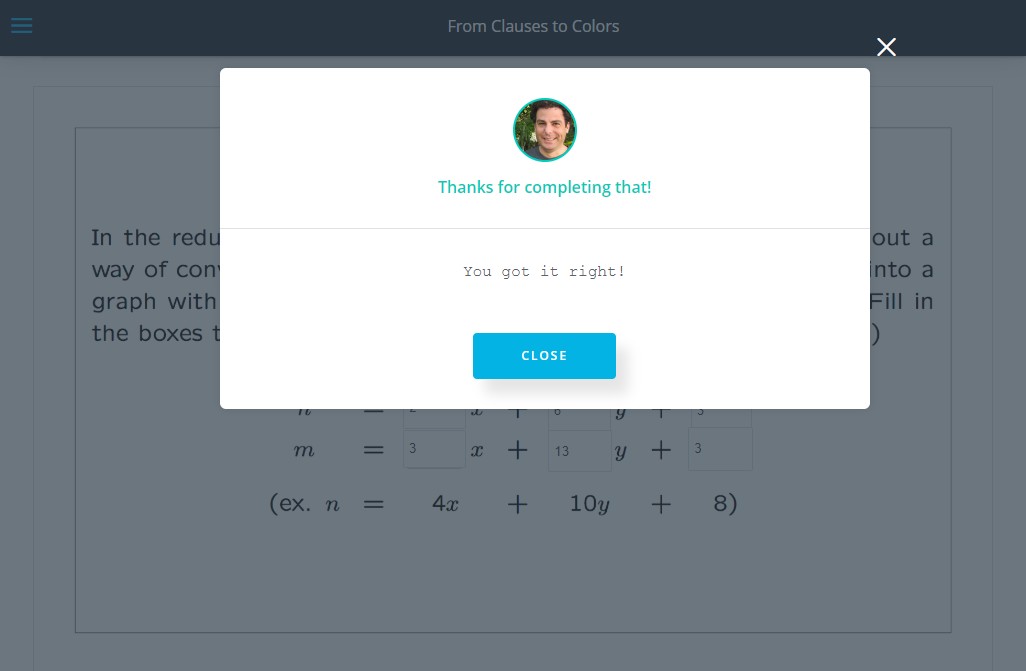


* 1. Polynomial vs . Exponential

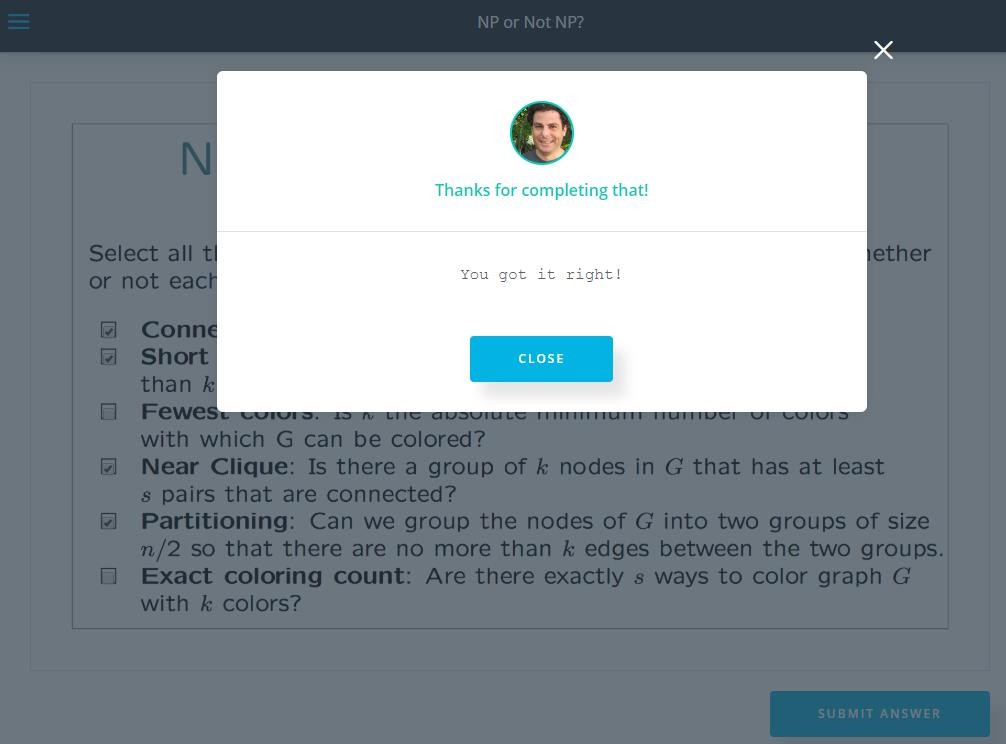
}



* 1. From Clauses to Colors



* 1. NP or not NP



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

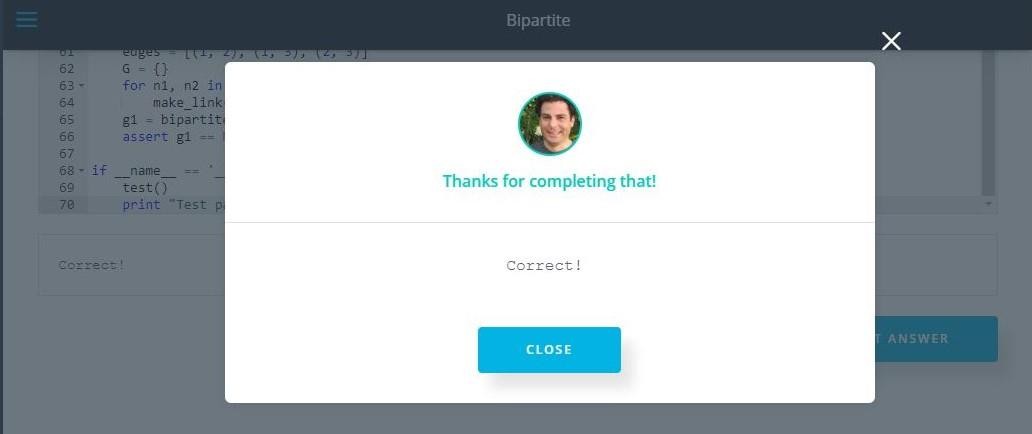
return False

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

def feel\_the\_love(G, i, j):

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:

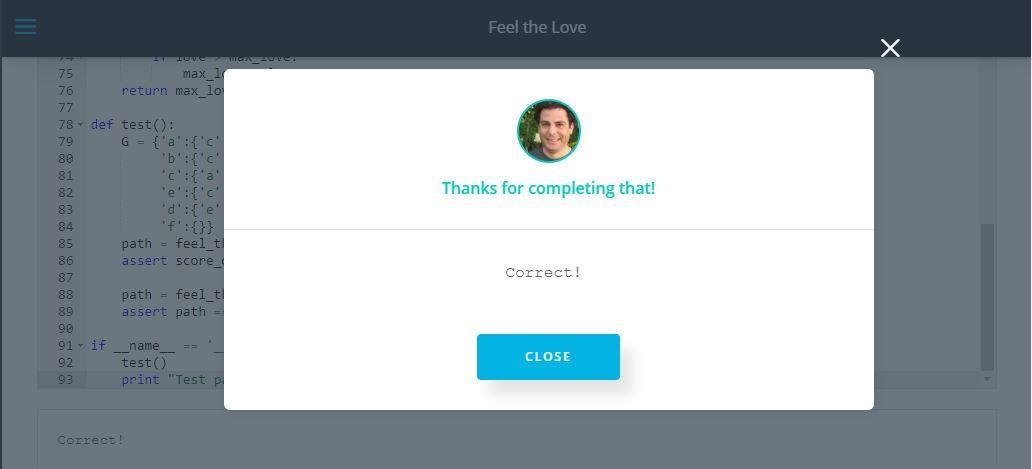
del dist\_so\_far[k] final\_dist[k] = w

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

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])

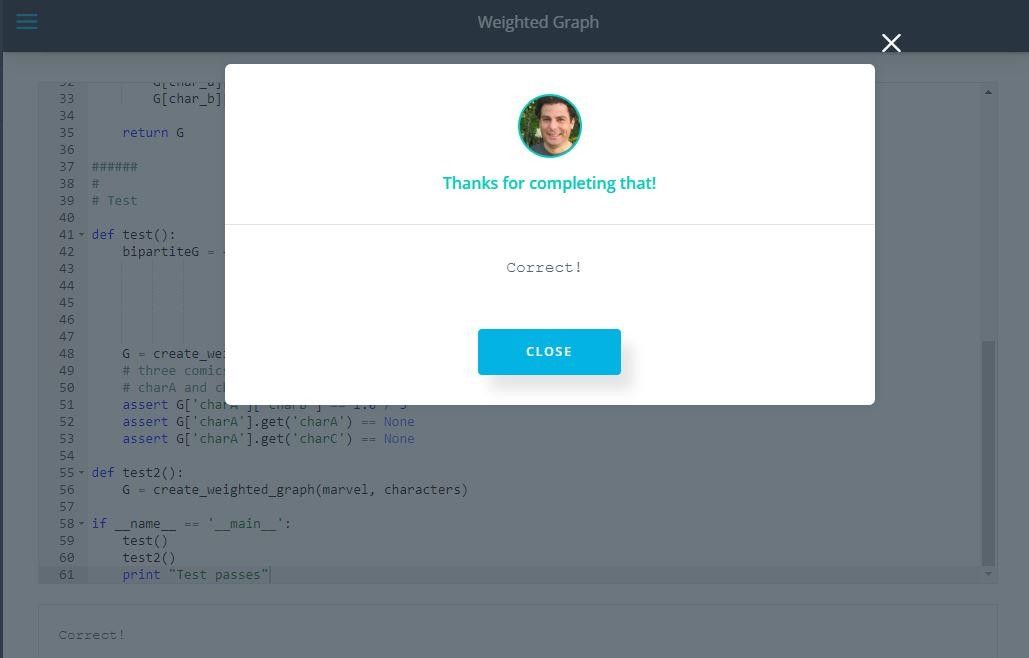
inter\_book\_num = float(len(a\_books.intersection(b\_books))) if inter\_book\_num == 0:

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

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()

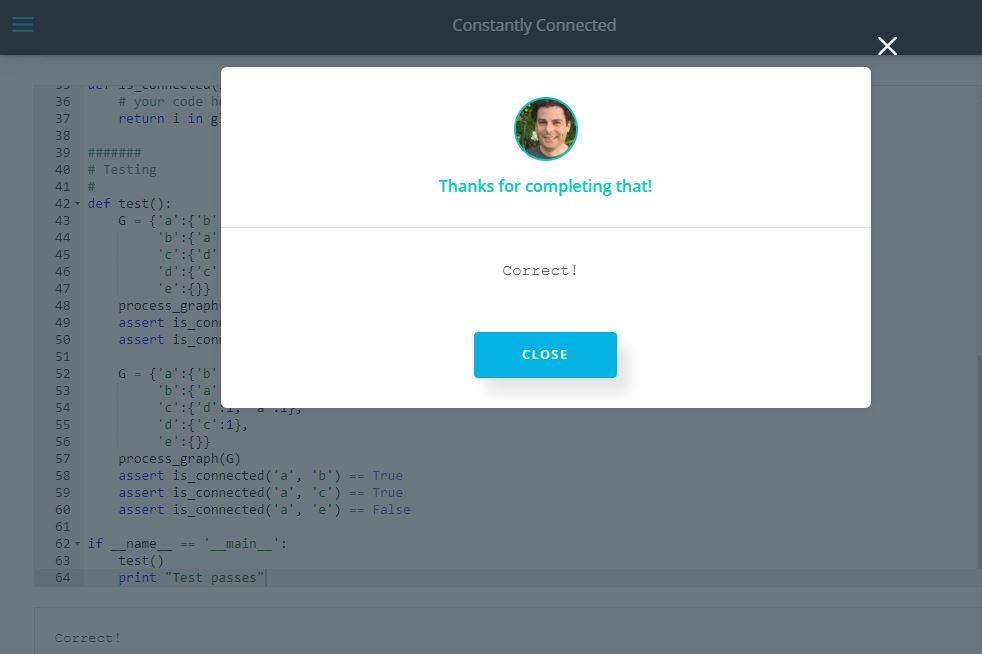
global\_G[node][new\_node] = 1

global\_G[new\_node][node] = 1

for x in global\_G[new\_node]:

if x not in global\_G[node]:

to\_visit += [x]



* 1. Distance Oracle 1

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:

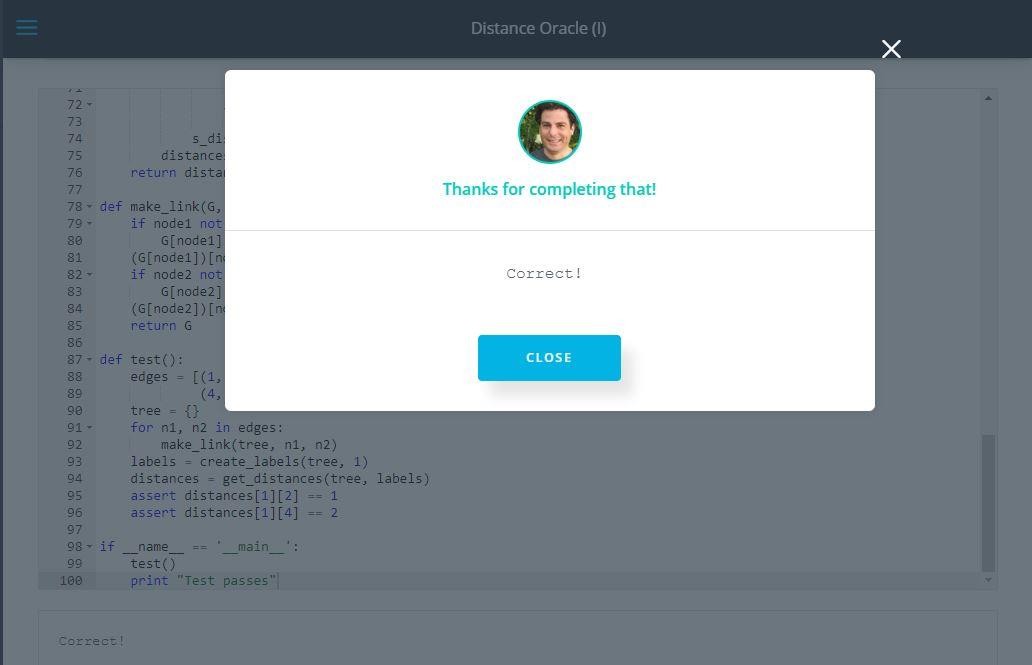
continue labels[child] = {child: 0}

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 1

def count\_nodes(treeG, node): 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):

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) if cnts[tmt\_root] - mcc >= mcc:

return tmt\_root cnts[mc] += cnts[tmt\_root] - mcc cnts[tmt\_root] -= mcc

return find\_cen(treeG, mc, cnts) 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)

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:

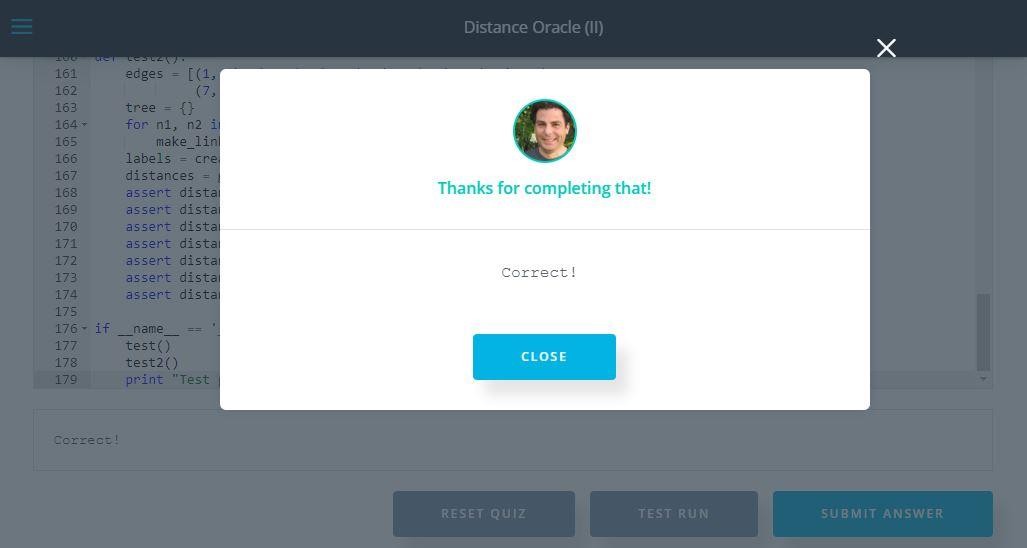
labels[neighbor] = {neighbor: 0} labels[neighbor][sub\_cen] = treeG[v][neighbor] + labels[v][sub\_cen]

cens\_nodes = {} labels = {}

tmt\_root = iter(treeG).next()

cnts = count\_nodes(treeG, tmt\_root) label\_tree(tmt\_root)

return labels



* 1. Finding a Favor

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) def dijkstra\_heap(G, a):

# Distance to the input node is zero, and it has

# no parent

first\_entry = (0, a, None) heap = [first\_entry]

# location keeps track of items in the heap # so that we can update their value later location = {first\_entry:0}

dist\_so\_far = {a:first\_entry}

final\_dist = {}

while len(dist\_so\_far) > 0:

dist, node, parent = heappopmin(heap, location) final\_dist[node] = (dist, parent)

del dist\_so\_far[node] for x in G[node]:

if x in final\_dist:

continue

new\_dist = G[node][x] + final\_dist[node][0] new\_entry = (new\_dist, x, node)

if x not in dist\_so\_far:

insert\_heap(heap, new\_entry, location) dist\_so\_far[x] = new\_entry

elif new\_entry < dist\_so\_far[x]:

decrease\_val(heap, location, dist\_so\_far[x], new\_entry) dist\_so\_far[x] = new\_entry

return final\_dist

def dijkstra\_list(G, a):

dist\_so\_far = {a:(0, None)} final\_dist = {}

while len(final\_dist) < len(G):

node, entry = min(dist\_so\_far.items(), key=itemgetter(1)) final\_dist[node] = entry

del dist\_so\_far[node] for x in G[node]:

if x in final\_dist:

continue

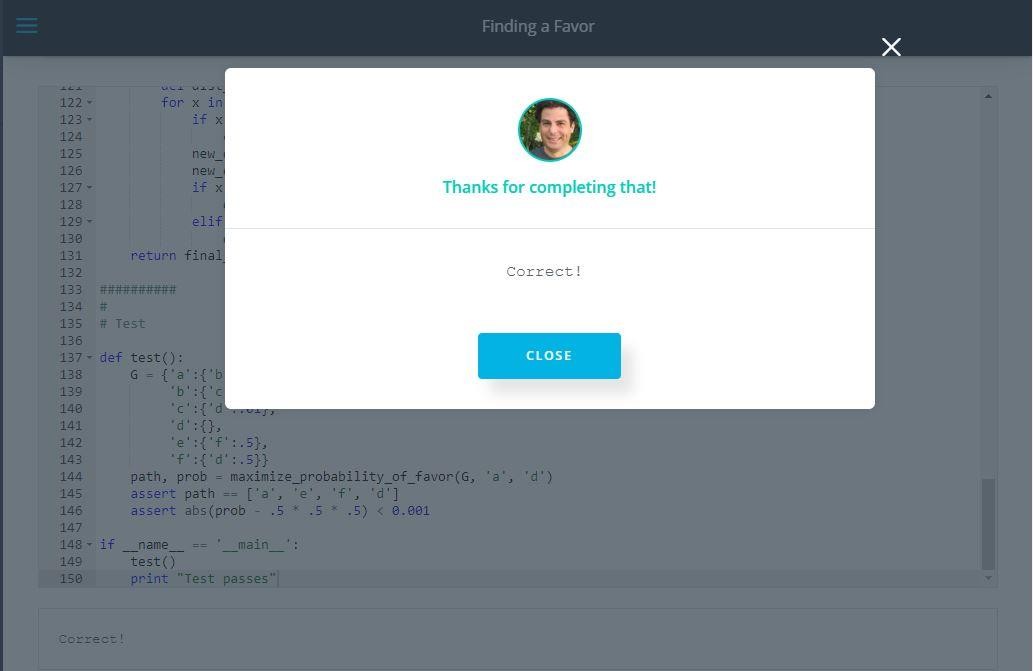
new\_dist = G[node][x] + final\_dist[node][0] new\_entry = (new\_dist, node)

if x not in dist\_so\_far:

dist\_so\_far[x] = new\_entry elif new\_entry < dist\_so\_far[x]:

dist\_so\_far[x] = new\_entry

return final\_dist



4.

1. Muestre que el problema cumple con la propiedad de subestructura óptima

Sea ***f(n)*** el costo mínimo de cubrir la cinta hasta el largo ***n***, se puede ver que esta solución se puede encontrar a partir de las soluciones para los largos ***n • 2*** y ***n • 3*** adicionando la ficha que cubra el largo restante y que tenga el mínimo costo.

1. Plantee una ecuación recursiva para resolver el problema

*f*(*n*) =

0 *si n* = 0

*min* (*c*2, *c*3) *si n* = 1 *o n* = 2

*e*.*o*.*c min*( *f*(*n* − 1) + *c*2, *f*(*n* − 1) + *c*3, *f*(*n* − 2) + *c*2, *f*(*n* − 2) + *c*3, *f*(*n* − 3) + *c*3)

1. 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

print (cubrir(5, 7, 10))

1. 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, 10)** | **0** | **5** | **5** | **7** | **10** | **12** | **14** | **17** | **19** | **21** | **24** |

5. Problema de cubrimiento de un tablero 3 xn con chas de domino

* 1. 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

- *D1* = 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

* 1. ¿Por qué En siempre es 0?

Para el caso E se tienen 2 casos por verificar:

1. **N** es impar: si **N** es impar entonces las líneas inferior y superior serán de tamaño impar y dado que solo podemos cubrirlas con dominós de tamaño 2, será imposible cubrirlas por completo.
2. **N** es par: si **N** es par entonces la fila de la mitad tendría un número impar de espacios por llenar, luego es imposible cubrir la figura.

Por tanto, es imposible cubrir una figura como la del caso E con dominós de 2x1 o 1x2.

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

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]

print(tiling(10)) print(tiling(50)) print(tiling(100))

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **10** | **50** | **100** |
| **tiling** | **571** | **156886956080403** | **31208688988045323113527764971** |