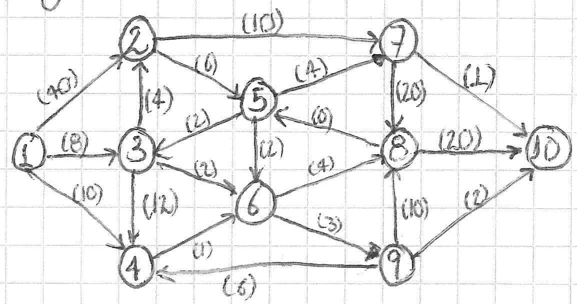
Taller 2 - Algoritmos

Jonathan Campo Rangel

1)



a) Dijkstra:

1) Iniciar distancias con valor ∞, excepto la del nodo inicial (=0)
2) a = nodo inicial
3) Visitar todos los vecinos de a y los marcamos al visitarlos (V_i)
4) Calcular distancia del nodo actual a sus vecinos. $dT(V_i) = Da + d(a, V_i)$. (Distancia total igual a distancia acumulada + distancia del nodo actual al vecino)
5) Si la distancia calculada es menor que la almacenada, se actualiza como nueva distancia tentativa.
6) Se marca como completado el nodo a.
7) Se toma un nuevo nodo inicial (el de menor distancia en D) y se repite desde el paso 3 mientras existan nodos no marcados.

Al terminar el algoritmo D tendrá las distancias entre los nodos.

b) Bellman - Ford

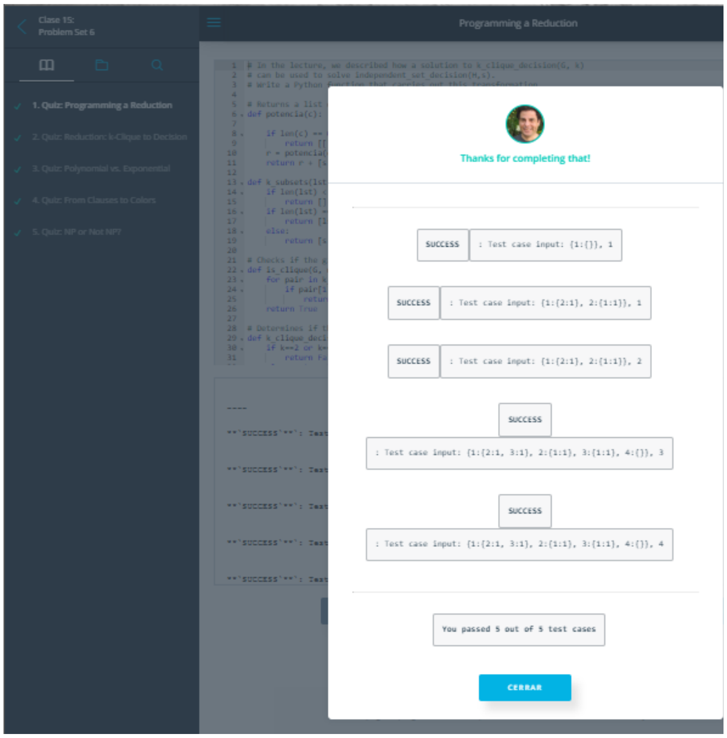
1) Iniciar distancias con valor ∞, excepto el nodo inicial.
2) Crear diccionario de Dist. Finales y de padres
3) Visitamos cada arista del grafo n-1 veces (n = # de nodos)
4) Comprobar que no hayan ciclos negativos
5) Se obtiene una lista de los vertices ordenados por la ruta mas corta.

c) Floyd - Warshal: Comparar todos los posibles caminos a través de un grafo entre cada par de vértices

1) Definir matrices de vértices C, D
2) Inicializar constante K=1
3) Seleccionar fila y columna K de C y para $i \neq j \neq K$:
4) Si $(C_{iK} + C_{Kj}) < C_{ij} \Rightarrow D_{ij} = D_{Kj} \wedge C_{ij} = C_{iK} + C_{Kj}$
5) Si no ⇒ No hay cambios
6) Si $K \leq n \Rightarrow K{+}{+}$; else ⇒ Stop.
7) C, al final, contiene los costos optimos para ir de un vértice a otro y D tiene los penúltimos vertices de los caminos optimos entre dos nodos.

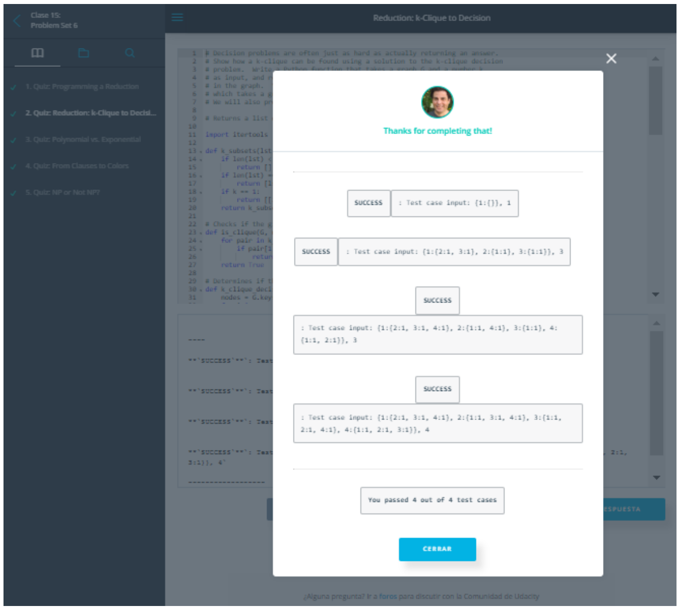
2)Problem set 6

- **PROGRAMMING A REDUCTION**




```
# In the lecture, we described how a solution to
        k_clique_decision(G, k)
# can be used to solve independent_set_decision(H,s).
# Write a Python function that carries out this transformation.

# Returns a list of all the subsets of a list of size k
def potencia(c):

    if len(c) == 0:
        return [[]]
    r = potencia(c[:-1])
    return r + [s + [c[-1]] for s in r]

def k_subsets(lst, k):
    if len(lst) < k:
        return []
    if len(lst) == k:
        return [lst]
    else:
        return [s for s in potencia(lst) if len(s) == k]

# Checks if the given list of nodes forms a clique in the given graph.
def is_clique(G, nodes):
    for pair in k_subsets(nodes, 2):
        if pair[1] not in G[pair[0]]:
            return False
    return True

# Determines if there is clique of size k or greater in the given graph.
def k_clique_decision(G, k):
    if k==2 or k==4:
        return False
    else: return True
    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

def make_link(G, node1, node2):
    if node1 not in G:
        G[node1] = {}
    (G[node1])[node2] = 1
    if node2 not in G:
        G[node2] = {}
    (G[node2])[node1] = 1
    return G

def break_link(G, node1, node2):
    if node1 not in G:
        print "error: breaking link in a non-existent node"
        return
    if node2 not in G:
        print "error: breaking link in a non-existent node"
        return
    if node2 not in G[node1]:
        print "error: breaking non-existent link"
        return
    if node1 not in G[node2]:
        print "error: breaking non-existent link"
        return
    del G[node1][node2]
    del G[node2][node1]
    return G
# 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 = {}
    n = H.keys()
    for v in H.keys():
        G[v] = {}
    for other in list(set(n) - set(H[v].keys()) - set([v])):
      G[v][other] = 1
    return k_clique_decision(G, s)
```

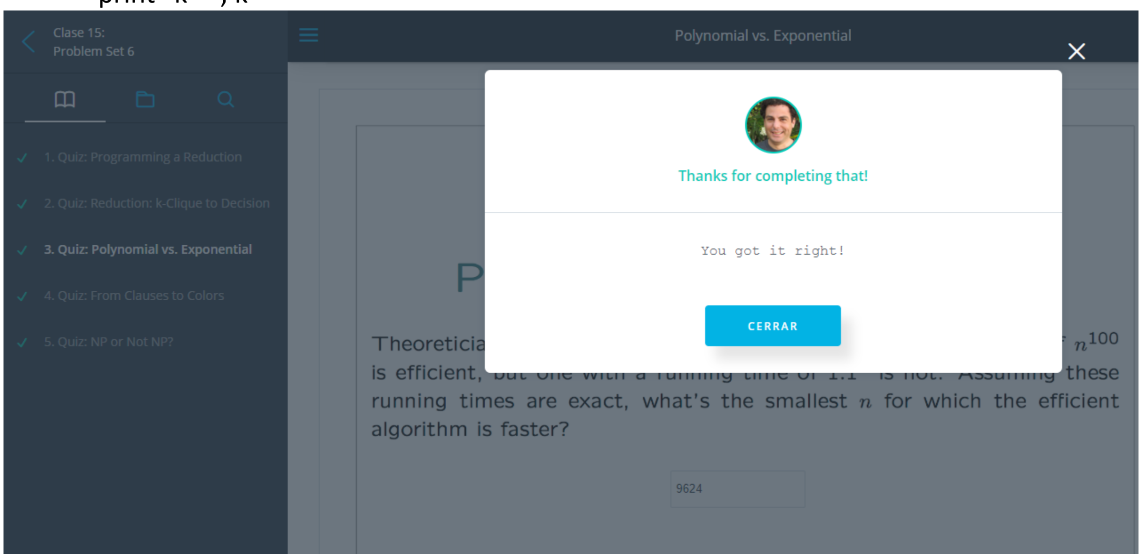
- **REDUCTION: K-CLIQUE TO DECISION**



Decision problems are often just as hard as actually returning an answer. Show how a k-clique can be found using a solution to the k-clique decisión problem. Write a Python function that takes a graph G and a number k as input, and returns a list of k nodes from G that are all connected in the graph. Your function should make use of "k_clique_decision(G, k)",which takes a graph G and a number k and answers whether G contains a k-clique. We will also provide the standard routines for adding and removing edges from a graph. Returns a list of all the subsets of a list of size k

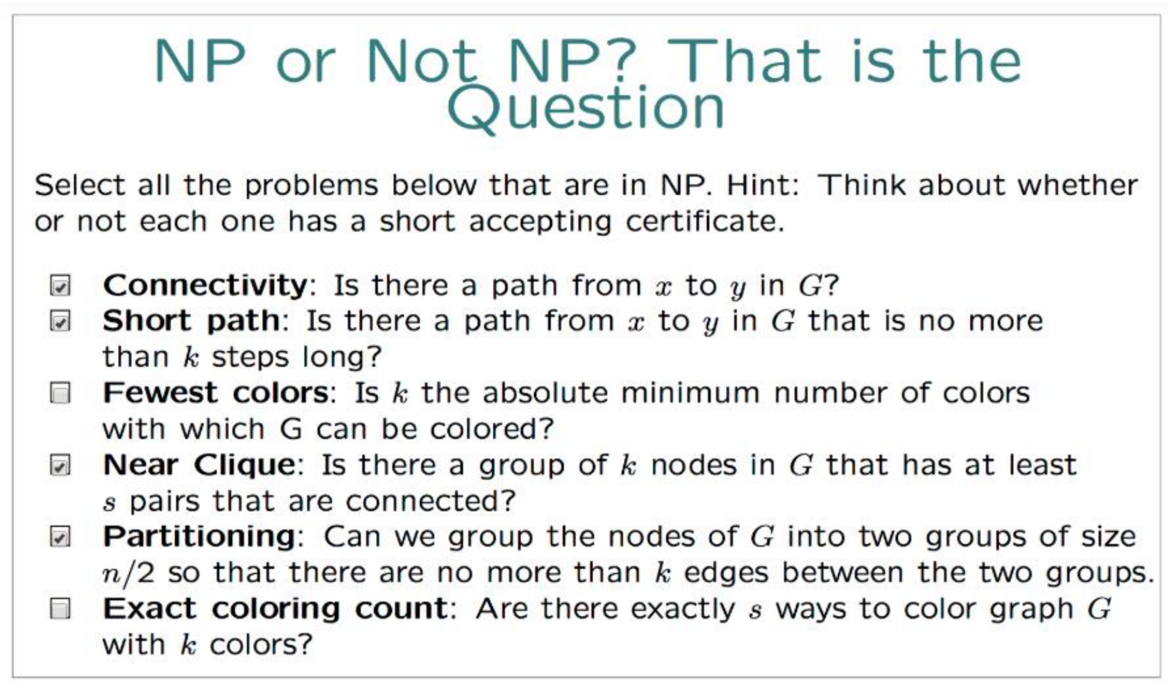
```
import itertools
def k_subsets(lst, k):
    if len(lst) < k:
        return []
    if len(lst) == k:
        return [lst]
    if k == 1:
        return [[i] for i in lst]
    return k_subsets(lst[1:],k) + map(lambda x: x + [lst[0]], k_subsets(lst[1:], k-1))

# Checks if the given list of nodes forms a clique in the given graph.
def is_clique(G, nodes):
    for pair in k_subsets(nodes, 2):
        if pair[1] not in G[pair[0]]:
            return False
    return True

# Determines if there is clique of size k or greater in the given graph.
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

def make_link(G, node1, node2):
    if node1 not in G:
        G[node1] = {}
    (G[node1])[node2] = 1
    if node2 not in G:
        G[node2] = {}
    (G[node2])[node1] = 1
    return G

def break_link(G, node1, node2):
    if node1 not in G:
        print "error: breaking link in a non-existent node"
        return
    if node2 not in G:
        print "error: breaking link in a non-existent node"
        return
    if node2 not in G[node1]:
        print "error: breaking non-existent link"
        return
    if node1 not in G[node2]:
        print "error: breaking non-existent link"
        return
    del G[node1][node2]
    del G[node2][node1]

    if G[node1] == {}:
        del G[node1]
    if G[node2] == {}:
        del G[node2]

    return G

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()
#This function is from
https://github.com/denversc/udacity/blob/master/
#Was used by: Joacampora
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)
```



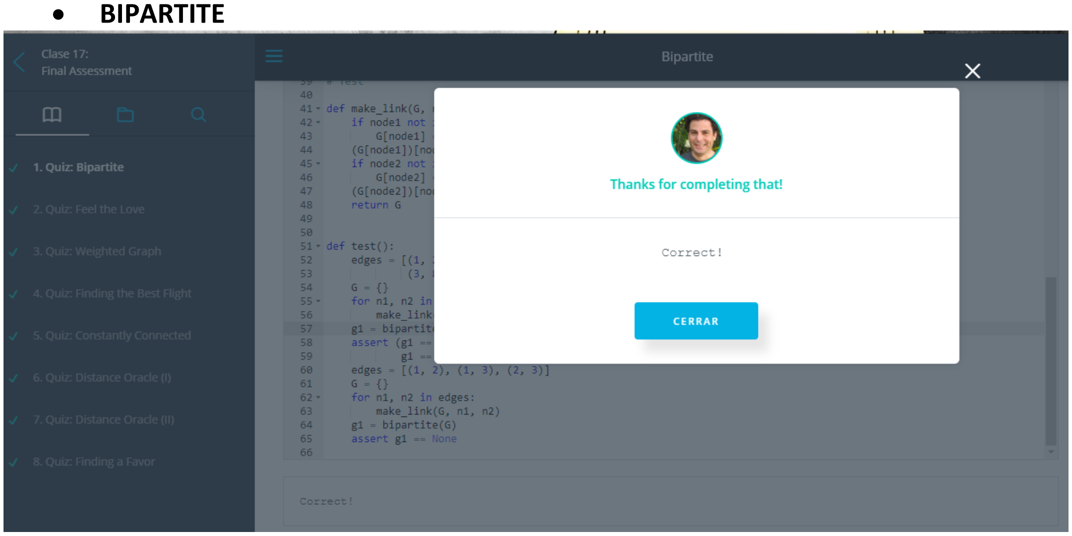
from
https://github.com/denversc/udacity/blob/master/

```
import decimal
decimal.getcontext().prec = 100
def polynomial(x):
    return decimal.Decimal(x)**100
def exponential(x):
    return
(decimal.Decimal(11)/decimal.Decimal(10))**x
def differ(x):
    return exponential(x) - polynomial(x)
if _name_ == '_main_':
    n   = 2
    step = 1
while differ(n) < 0:
    n   += step
    step *= 2
lower = n - step/2
upper = n
while True:
    n = int((upper+lower) /2)
    if differ(n-1) * differ(n) < 0:
        break
    if differ(n) > 0:
        upper = n
    else:
        lower = n
print ("n =", n)
```



3)FINAL TEST

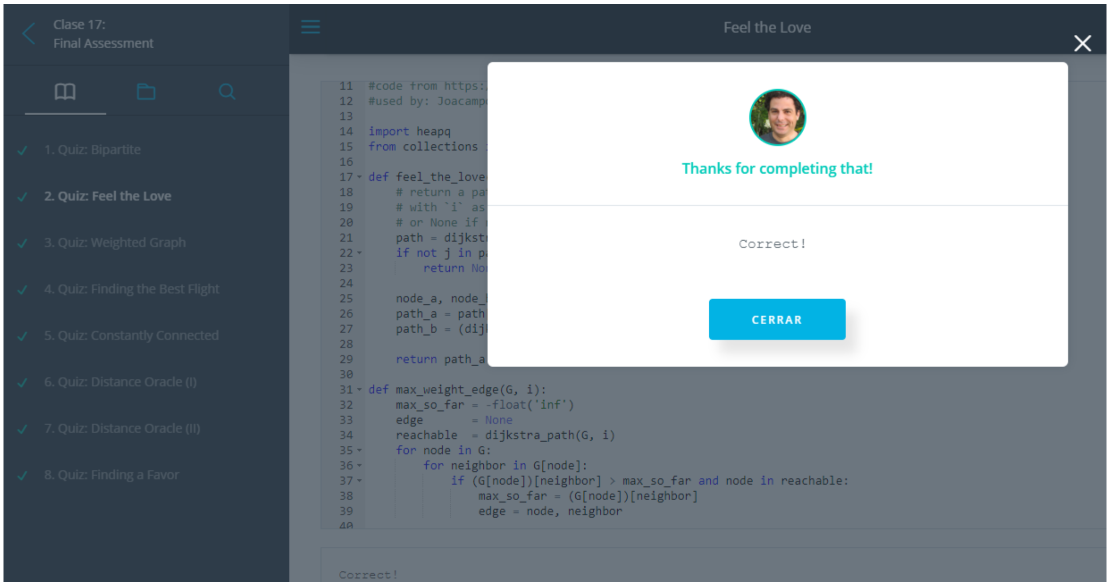
- **BIPARTITE**




```
# Write a function, `bipartite` that
# takes as input a graph, `G` and tries
# to divide G into two sets where
# there are no edges between elements of the
# the same set - only between elements in
# different sets.
# If two sets exists, return one of them
# or `None` otherwise
# Assume G is connected
#code from https://github.com/WentaoZero/Intro-to-Algorithms
#used by: Joacampora

def bipartite(G):
    checked = {}
    def iter_check(node, side):
        if node in checked:
            return
        checked[node] = side
        for n in G[node]:
            iter_check(n, 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                assert g1 == None
            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

########
#
# Test

def make_link(G, node1, node2, weight=1):
   if node1 not in G:
      G[node1] = {}
   (G[node1])[node2] = weight
   if node2 not in G:
      G[node2] = {}
   (G[node2])[node1] = weight
   return G


def test():
   edges = [(1, 2), (2, 3), (1, 4), (2, 5),
            (3, 8), (5, 6)]
   G = {}
   for n1, n2 in edges:
      make_link(G, n1, n2)
   g1 = bipartite(G)
   assert (g1 == set([1, 3, 5]) or
           g1 == set([2, 4, 6, 8]))
   edges = [(1, 2), (1, 3), (2, 3)]
   G = {}
   for n1, n2 in edges:
      make_link(G, n1, n2)
   g1 = bipartite(G)
```

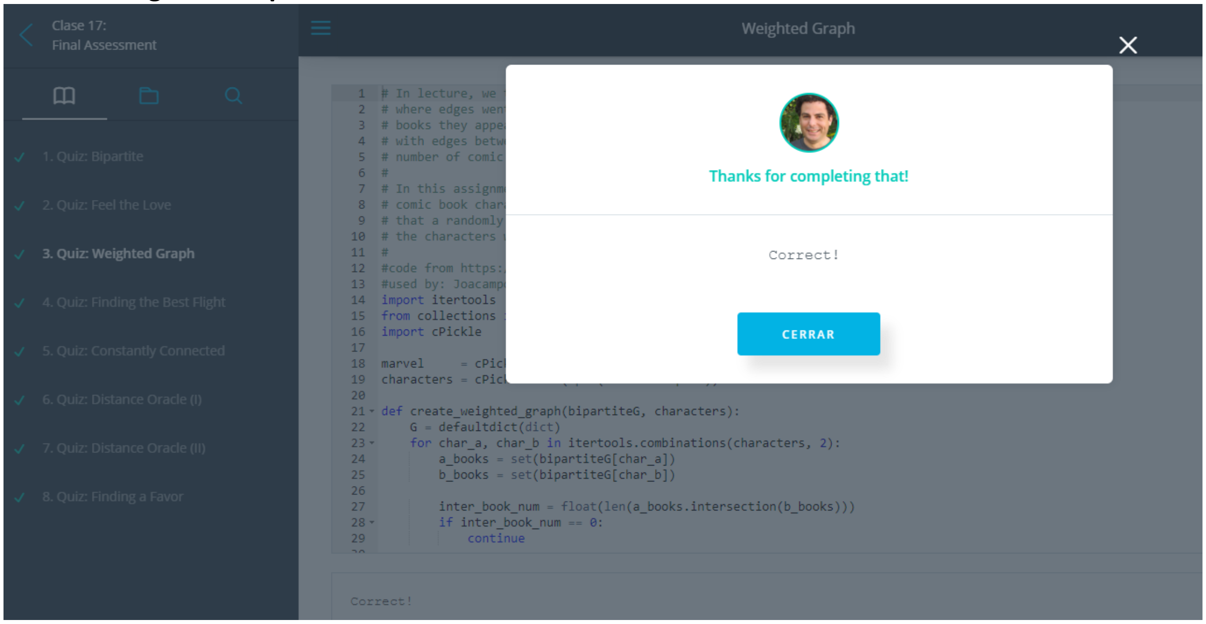
- **Feel the love**




```
# Take a weighted graph representing a social network where the weight
# between two nodes is the "love" between them.  In this "feel the
# love of a path" problem, we want to find the best path from node `i`
# and node `j` where the score for a path is the maximum love of an
# edge on this path. If there is no path from `i` to `j` return
# `None`.  The returned path doesn't need to be simple, ie it can
# contain cycles or repeated vertices.
#
# Devise and implement an algorithm for this problem.
#
#code from https://github.com/WentaoZero/Intro-to-Algorithms
#used by: Joacampora

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]
```

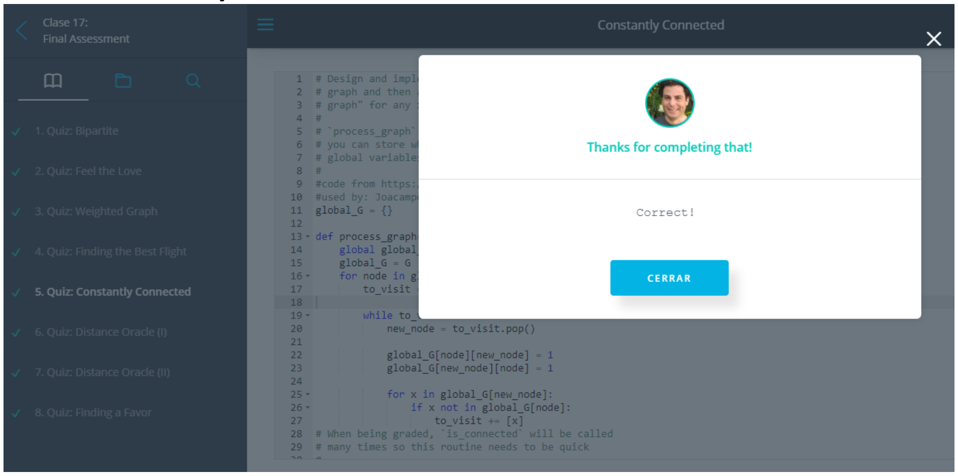
```
                        if neighbor not in dist_so_far or
nw < dist_so_far[neighbor]:
                                dist_so_far[neighbor] =
nw
                                heapq.heappush(heap,
(nw, neighbor))

        for node in final_path:
                final_path[node] += [node]
        return final_path

#########
#
# Test

def score_of_path(G, path):
        max_love = -float('inf')
        for n1, n2 in zip(path[:-1], path[1:]):
                love = G[n1][n2]
                if love > max_love:
                        max_love = love
        return max_love

def test():
        G = {'a':{'c':1},
                 'b':{'c':1},
                 'c':{'a':1, 'b':1, 'e':1, 'd':1},
                 'e':{'c':1, 'd':2},
                 'd':{'e':2, 'c':1},
                 'f':{}}
        path = feel_the_love(G, 'a', 'b')
        assert score_of_path(G, path) == 2

        path = feel_the_love(G, 'a', 'f')
        assert path == None

if __name__ == '__main__':
        test()
        print "Test passes"
```

- **Weighted Graph**



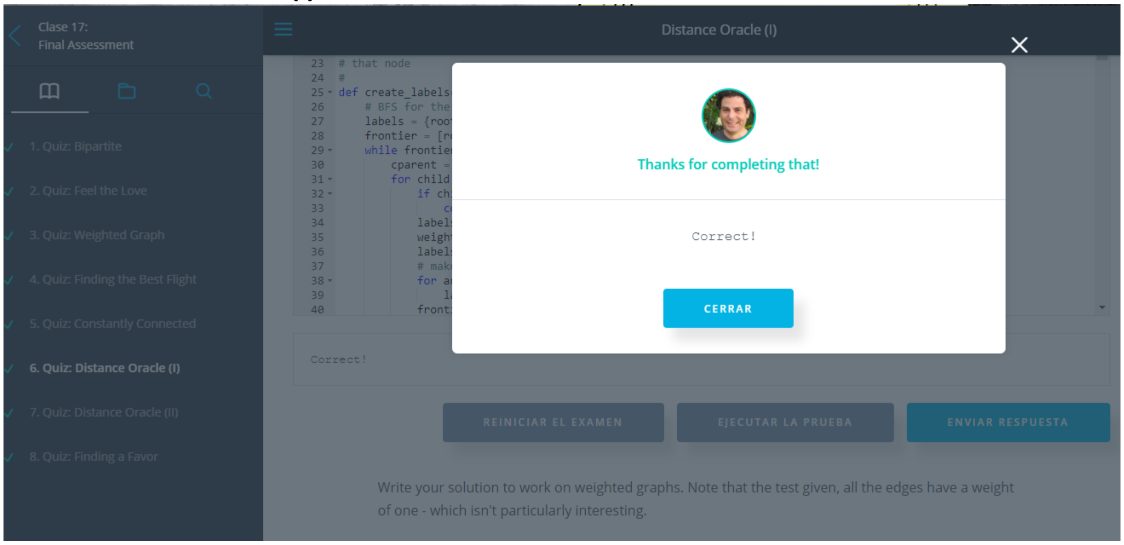

```
# In lecture, we took the bipartite Marvel graph,
# where edges went between characters and the comics
# books they appeared in, and created a weighted graph
# with edges between characters where the weight was the
# number of comic books in which they both appeared.
#
# In this assignment, determine the weights between
# comic book characters by giving the probability
# that a randomly chosen comic book containing one of
# the characters will also contain the other
#
#code from https://github.com/WentaoZero/Intro-to-Algorithms
#used by: Joacampora
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

######
#
# Test
def test():
	bipartiteG = {'charA':{'comicB':1, 'comicC':1},
				'charB':{'comicB':1, 'comicD':1},
				'charC':{'comicD':1},
				'comicB':{'charA':1, 'charB':1},
				'comicC':{'charA':1},
				'comicD': {'charC':1, 'charB':1}}
	G = create_weighted_graph(bipartiteG, ['charA', 'charB', 'charC'])
	# three comics contain charA or charB
	# charA and charB are together in one of them
	assert G['charA']['charB'] == 1.0 / 3
	assert G['charA'].get('charA') == None
	assert G['charA'].get('charC') == None

def test2():
	G = create_weighted_graph(marvel, characters)

if __name__ == '__main__':
	test()
	test2()
	print "Test passes"
```

- **Constantly Connected**




```
# Design and implement an algorithm that can
preprocess a
# graph and then answer the question "is x connected
to y in the
# graph" for any x and y in constant time Theta(1).
#
# `process_graph` will be called only once on each
graph.  If you want,
# you can store whatever information you need for
`is_connected` in
# global variables
#
#code from https://github.com/WentaoZero/Intro-to-
Algorithms
#used by: Joacampora
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]
# 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
    return i in global_G[j] or j in global_G[i]

#######
# Testing
#
def test():
    G = {'a':{'b':1},
         'b':{'a':1},
         'c':{'d':1},
         'd':{'c':1},
         'e':{}}
    process_graph(G)
    assert is_connected('a', 'b') == True
    assert is_connected('a', 'c') == False

    G = {'a':{'b':1, 'c':1},
         'b':{'a':1},
         'c':{'d':1, 'a':1},
         'd':{'c':1},
         'e':{}}
    process_graph(G)
    assert is_connected('a', 'b') == True
    assert is_connected('a', 'c') == True
    assert is_connected('a', 'e') == False

if __name__ == '__main__':
    test()
    print "Test passes"
```

- **Distance Oracle(I)**

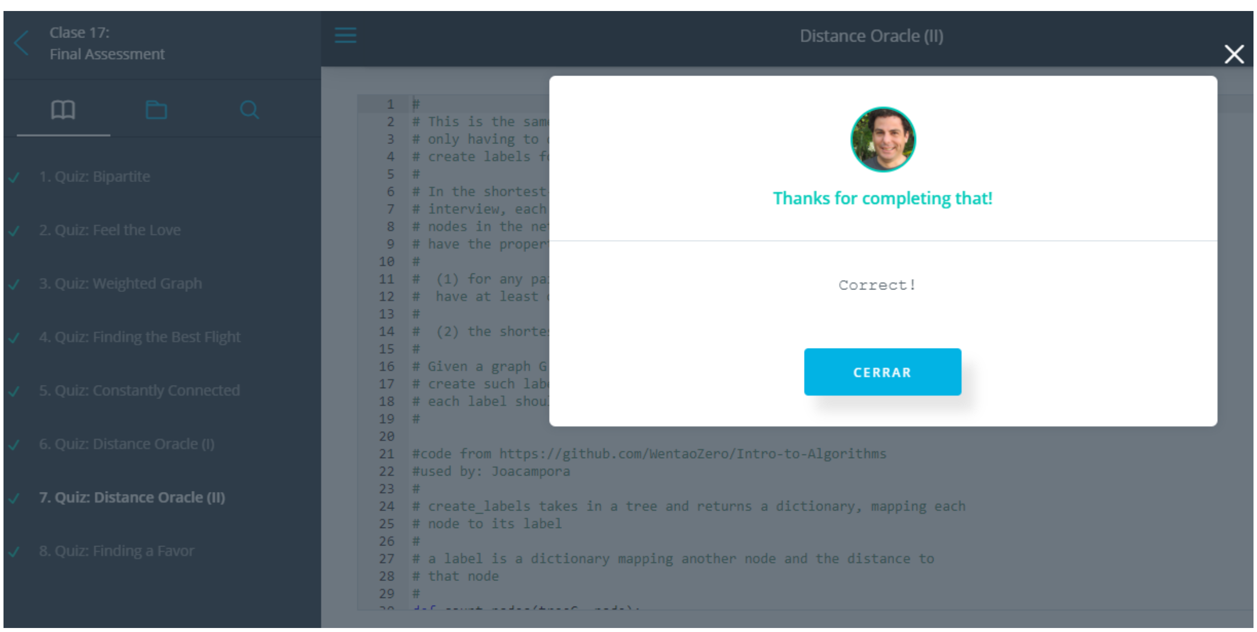



```
# In the shortest-path oracle described in Andrew
Goldberg's
# interview, each node has a label, which is a list of
some other
# nodes in the network and their distance to these
nodes.  These lists
# have the property that
#
#  (1) for any pair of nodes (x,y) in the network, their
lists will
#  have at least one node z in common
#
#  (2) the shortest path from x to y will go through z.
#
# Given a graph G that is a balanced binary tree,
preprocess the graph to
# create such labels for each node.  Note that the size of
the list in
# each label should not be larger than log n for a graph
of size n.
#
#code from https://github.com/WentaoZero/Intro-to-
Algorithms
#used by: Joacampora

#
# create_labels takes in a balanced binary tree and the
root element
# and returns a dictionary, mapping each node to its
label
#
# a label is a dictionary mapping another node and the
distance to
# that node
#
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

#######
# Testing
#

def get_distances(G, labels):
	# labels = {a:{b: distance from a to b,
	#             c: distance from a to c}}
	# create a mapping of all distances for
	# all nodes
	distances = {}
	for start in G:
		# get all the labels for my starting node
		label_node = labels[start]
		s_distances = {}
		for destination in G:
```

```
#
                        shortest = float('inf')
                        # get all the labels for the
destination node
                        label_dest = labels[destination]
                        # and then merge them
together, saving the
                        # shortest distance
                        for intermediate_node, dist in
label_node.iteritems():
                                # see if
intermediate_node is our destination
                                # if it is we can stop - we
know that is
                                # the shortest path
                                if intermediate_node ==
destination:
                                        shortest = dist
                                        break
                                other_dist =
label_dest.get(intermediate_node)
                                if other_dist is None:
                                        continue
                                if other_dist + dist <
shortest:
                                        shortest =
other_dist + dist
                        s_distances[destination] =
shortest
```

```
                distances[start] = s_distances
        return distances

def make_link(G, node1, node2, weight=1):
        if node1 not in G:
                G[node1] = {}
        (G[node1])[node2] = weight
        if node2 not in G:
                G[node2] = {}
        (G[node2])[node1] = weight
        return G

def test():
        edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7),
                        (4, 8), (4, 9), (5, 10), (5, 11), (6,
12), (6, 13)]
        tree = {}
        for n1, n2 in edges:
                make_link(tree, n1, n2)
        labels = create_labels(tree, 1)
        distances = get_distances(tree, labels)
        assert distances[1][2] == 1
        assert distances[1][4] == 2

if __name__ == '__main__':
        test()
        print "Test passes"
```

Distance Oracle(II)



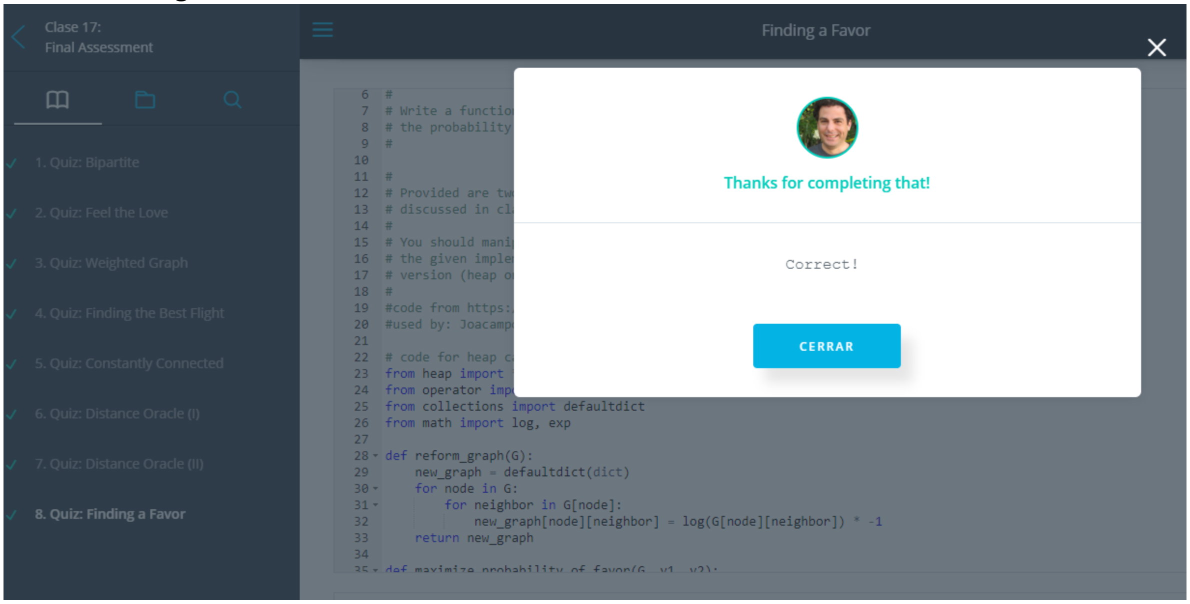

```
#
# This is the same problem as "Distance Oracle I" except
that instead of
# only having to deal with binary trees, the assignment
asks you to
# create labels for all tree graphs.
#
```

```
# In the shortest-path oracle described in Andrew
Goldberg's
# interview, each node has a label, which is a list of
some other
# nodes in the network and their distance to these
nodes.  These lists
# have the property that
#
#  (1) for any pair of nodes (x,y) in the network, their
lists will
#  have at least one node z in common
#
#  (2) the shortest path from x to y will go through z.
#
# Given a graph G that is a tree, preprocess the graph to
# create such labels for each node.  Note that the size of
the list in
# each label should not be larger than log n for a graph
of size n.
#

#code from https://github.com/WentaoZero/Intro-to-
Algorithms
#used by: Joacampora
#
# create_labels takes in a tree and returns a dictionary,
mapping each
# node to its label
#
# a label is a dictionary mapping another node and the
distance to
# that node
#
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

#######
```

Testing
#

```
def get_distances(G, labels):
    # labels = {a:{b: distance from a to b,
    #             c: distance from a to c}}
    # create a mapping of all distances for
    # all nodes
    distances = {}
    for start in G:
        # get all the labels for my starting node
        label_node = labels[start]
        s_distances = {}
        for destination in G:
            shortest = float('inf')
            # get all the labels for the
destination node
            label_dest = labels[destination]
            # and then merge them
together, saving the
            # shortest distance
            for intermediate_node, dist in
label_node.iteritems():
                # see if
intermediate_node is our destination
                # if it is we can stop - we
know that is
                # the shortest path
                if intermediate_node ==
destination:
                    shortest = dist
                    break
                other_dist =
label_dest.get(intermediate_node)
                if other_dist is None:
                    continue
                if other_dist + dist <
shortest:
                    shortest =
other_dist + dist
            s_distances[destination] =
shortest
        distances[start] = s_distances
    return distances

def make_link(G, node1, node2, weight=1):
    if node1 not in G:
        G[node1] = {}
    (G[node1])[node2] = weight
    if node2 not in G:
        G[node2] = {}
    (G[node2])[node1] = weight
    return G

def test():
    edges = [(1, 2), (1, 3), (2, 4), (2, 5), (3, 6), (3, 7),
                 (4, 8), (4, 9), (5, 10), (5, 11), (6,
12), (6, 13)]
    tree = {}
    for n1, n2 in edges:
        make_link(tree, n1, n2)
    labels = create_labels(tree)
    if labels:
        distances = get_distances(tree, labels)
    assert distances[1][2] == 1
    assert distances[1][4] == 2
    assert distances[1][2] == 1
    assert distances[1][4] == 2
    assert distances[4][1] == 2
    assert distances[1][4] == 2
    assert distances[2][1] == 1
    assert distances[1][2] == 1
    assert distances[1][1] == 0
    assert distances[2][2] == 0
    assert distances[9][9] == 0
    assert distances[2][3] == 2
    assert distances[12][13] == 2
    assert distances[13][8] == 6
    assert distances[11][12] == 6
    assert distances[1][12] == 3

def test2():
    edges = [(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7),
                 (7, 8), (8, 9), (9, 10), (10, 11),
(11, 12), (12, 13)]
    tree = {}
    for n1, n2 in edges:
        make_link(tree, n1, n2)
    labels = create_labels(tree)
    distances = get_distances(tree, labels)
    assert distances[1][2] == 1
    assert distances[1][3] == 2
    assert distances[1][13] == 12
    assert distances[6][1] == 5
    assert distances[6][13] == 7
    assert distances[8][3] == 5
    assert distances[10][4] == 6

if __name__ == '__main__':
    test()
    test2()
    print "Test passes"
```

- **Finding a Favor**




```
# Finding a Favor v2
# Each edge (u,v) in a social network has a weight p(u,v) that
# represents the probability that u would do a favor for v if asked.
# Note that p(v,u) != p(u,v), in general.
# Write a function that finds the right sequence of friends to maximize
# the probability that v1 will do a favor for v2.
#
# Provided are two standard versions of dijkstra's algorithm that were
# discussed in class. One uses a list and another uses a heap.
#
# You should manipulate the input graph, G, so that it works using
# the given implementations.  Based on G, you should decide which
# version (heap or list) you should use.
#
#code from https://github.com/WentaoZero/Intro-to-Algorithms
#used by: Joacampora

# code for heap can be found in the instructors comments below
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)

#
# version of dijkstra implemented using a heap
#
# returns a dictionary mapping a node to the distance
# to that node and the parent
#
# Do not modify this code
#
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)
		# lock it down!
		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:
				# add to the heap
				insert_heap(heap, 
new_entry, location)
				dist_so_far[x] = 
new_entry
			elif new_entry < dist_so_far[x]:
				# update heap
				decrease_val(heap, 
location, dist_so_far[x], new_entry)
				dist_so_far[x] = 
new_entry
	return final_dist

#
# version of dijkstra implemented using a list
#
# returns a dictionary mapping a node to the distance
# to that node and the parent
#
# Do not modify this code
#
def dijkstra_list(G, a):
	dist_so_far = {a:(0, None)} #keep track of the 
parent node
	final_dist = {}
	while len(final_dist) < len(G):
		node, entry = min(dist_so_far.items(), 
key=itemgetter(1))
		# lock it down!
		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

##########
#
# Test

def test():
	G = {'a':{'b':.9, 'e':.5},
	     'b':{'c':.9},
	     'c':{'d':.01},
	     'd':{},
	     'e':{'f':.5},
	     'f':{'d':.5}}
	path, prob = maximize_probability_of_favor(G, 
'a', 'd')
	assert path == ['a', 'e', 'f', 'd']
	assert abs(prob - .5 * .5 * .5) < 0.001

if __name__ == '__main__':
	test()
	print "Test passes"
```

4) Tira de longitud n cubierta con fichas C_1 y C_2

a) Subestructura Optima: El problema se puede solucionar solucionando el mismo problema con un n menor.

b) Ecuación Recursiva

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

c) Programa en Python: *

d) Tabla para $C_2 = 5, C_3 = 7, n = 10$.

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) Tablero $3 \times n$ cubierto con fichas

c) Recurrencias:

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

$C_n = A_{N-1}$

$D_n = D_{N-2} + 2 \cdot C_{n-1}$

d) $B_n = E_n = 0$ (No es posible que resulte una forma del tablero que representan)

e) Python *

f) D_n para $n = 10; 50; 100$

$n = 10$ $D_n = 203$

$n = 50$ D_n

$n = 100$ $D_n =$

$n = 20$ $D_n = 38651$

*Punto 4-c: By https://es.scribd.com/document/358405261/Grafos-Complejidad-Computacional-Programacion-Dinamica

```python
def cubrir(C2, C3, n, r):
  r[0] = 0
  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
```

*Punto 5-e: By https://es.scribd.com/document/358405261/Grafos-Complejidad-Computacional-Programacion-Dinamica

```python
def A(N):
  if N == 0:
    return 0
  if N <= 1:
    return 1
  print "Haciendo A"
  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
  print "haciendo D"
  return D(N - 2) + 2*A(N-1)

```