```python
def dfs_path(G, source, target, visited):
    """
    Aquesta funció construeix el camí de source a target usant DFS.
    En cas que no existeixi, retornarà False.
    """

    # Si source és target, el camí serà simplement aquest node
    if source == target:
        return [target]

    # Si encara no hem visitat el node source, el visitem i seguim amb el DFS
    if source not in visited:
        visited.add(source)
        for nei in G.neighbors(source):

            # Comprovem que el pes de l'aresta sigui major que zero.
            # En el graf residual tindrem arestes 0 quan 'gastem' les unitats de flux
            # [(source, nei)] --> l'aresta que va de source a nei
            if G.edges()[(source, nei)]['weight'] > 0:
                pth = dfs_path(G, nei, target, visited)

                #Fen aquesta crida recursiva, anireem construint tot el path des de l'origen fins al desti
                if pth:
                    return [source] + pth
    return False


def ford_fulkerson(G, source='S', target='T'):
    """
    Implementació de l'algorisme ford-fulkerson
    """

    # Punt 1. Guardem el màxim flux que podem transportar. Inicialment és zero
    maxflow = 0

    # OPCIONA: Guardarem totes les iteracions de l'algorisme per veure com evoluciona el graf
    graph_states = [G.copy()]

    # Comencem aplicant un DFS, construint el path
    pth = dfs_path(G, source, target, set())

    # Punt 2. Si existeix el path
    while pth:

        # Punt 3. Trobem l'aresta de pes mínim dins el camí i actualitzem el flux màxim
        minflow = min([G.edges()[(pth[i], pth[i+1])]['weight'] for i in range(len(pth)-1)])
        maxflow += minflow

        # Punt 4. Actualitzem el valor de les arestes
        for i in range(len(pth)-1):

            # L'aresta (i, i+1) sempre existeix ja que forma part del path (gastem flux, anem endavant)
            G.edges()[(pth[i], pth[i+1])]['weight'] -= minflow

            # L'aresta (i+1, i) pot no existir ja que va en sentit contrari (recuperem flux, anem endarrera, graf contrari/residual)
            if (pth[i+1], pth[i]) not in G.edges():
                G.add_weighted_edges_from([(pth[i+1], pth[i], 0)])
            G.edges()[(pth[i+1], pth[i])]['weight'] += minflow

        graph_states.append(G.copy())

        # Punt 5. Tornem al Punt 2 mentre existeixi camí de l'origen al destí.
        pth = dfs_path(G, source, target, set())

    return maxflow, graph_states
```

```python
from collections import defaultdict

def change(X, P, coins = [.01, .02, .05, .1, .2, .5, 1, 2, 5, 10, 20, 50, 100, 200, 500]):
    """
    Soluciona el problema de retornar el canvi.

    Params
    ======
    :X: Preu
    :P: Pagament. Ha de ser superior o igual a X.
    :coins: Llista de monedes o bitllets de la moneda que estiguem considerant. Per defecte, euros.

    Returns
    =======
    :lst: Llista de monedes o bitllets i la quantitat de cada un d'ells amb el format següent. lst = [
        :value: és un valor existent dins la llista 'coins'
        :quantity: és el nombre de monedes/bitllets amb valor 'value'.

    """
    # Calculem el canvi
    v = P-X

    # Aquí guardarem la solució del problema com a diccionari. La clau serà la moneda/bitllet i el val
    solution = defaultdict(int)

    # Agafem la 'moneda' més gran, en aquest cas el bitllet de 500.
    n = len(coins) - 1

    # Comprovem si hem acabat, és a dir, si el valor que ens queda és inferior a la moneda més petita, l'al
    while v >= coins[0]:

        # Comprovem si podem pagar amb aquesta moneda. Si no podem, agafem la moneda següent.
        while coins[n] > v:
            n -= 1

        # Afegim a la solució
        solution[coins[n]] += 1

        # Modifiquem el valor que ens falta per retornar. Afegim el round a dos decimals per
        v = round(v - coins[n],2)

    # Imprimim amb el format que es demana
    lst = [(k, solution[k]) for k in solution]
    return lst
```

```python
def refill(K, stations):
    """
    Soluciona el problema de repostatge de vehicles.

    Params
    ======
    :K: quilòmetres que pot fer el cotxe amb el dipòsit ple.
    :stations: Punt quilomètric on es troba cada benzinera. L'últim element d'aquesta llista és el

    Returns
    =======
    :exists_solution: Si existeix o no solució al problema (True/False)
    :num_stops: Nombre de parades que hem de fer.
    :stops: Parades on ens aturarem (punt quilomètric).
    """

    car = 0             # Posició actual del cotxe
    petrol = K          # Combustible restant
    total_distance = 0 # Distància que hem recorregut fins al moment
    stops = []          # Benzineres on anirem repostant

    # Comprovem si encara no hem arribat al final
    while (car < len(stations)):

        # Modifiquem la benzina que ens queda i comprovem si ens hem quedat sense. En aquest cas, l
        petrol-=(stations[car]-total_distance)
        if petrol < 0:
            return False, len(stops), stops

        # Actualitzem la distància recorreguda.
        total_distance=stations[car]

        # Comprovem si encara estem en una benzinera (el proper valor de la llista no es la destina
        # Si no podem arribar a la propera benzinera amb la benzina que ens queda, repostem.
        if (car < len(stations)-1) and (petrol < (stations[car+1]-stations[car])):
            petrol = K
            stops.append(stations[car])

        # Movem el cotxe a la benzinera següent.
        car+=1

    return True, len(stops), stops
```

```python
def union_find(lst):
    """
    Implementació de l'algorisme union-find.

    Params
    ======
    :lst: Llista de parelles amb les connexions que volem realitzar.
    """

    # Agafem tots els nodes únics
    list1, list2 = zip(*lst)
    unique_nodes = set(list1+list2)

    # Inicialitzem les dues variables rank i parent que ens serviran per anar construïn
    rank = defaultdict(int)
    parent = {n: n for n in unique_nodes}

    # Aquesta variable no forma part de l'algorisme. Guardarem totes les modificacions
    # forma, podrem veure com anem connectant tots els nodes.
    parent_states = []

    # Recorrem totes les parelles
    for node1, node2 in lst:

        # Busquem el node arrel de cada un dels nodes
        parent1 = find(parent, node1)
        parent2 = find(parent, node2)

        # Si tenen el mateix pare, no fem res i continuem
        if parent1 == parent2:
            continue

        # Els unim
        union(parent, rank, parent1, parent2)

        # Guardem totes les versions del diccionari parent per poder-les mostrar
        parent_states.append(parent.copy())

    return parent, parent_states
```

```python
def detect_cycles(lst):
    """
    Detecta si un graf conté cicles

    Params
    ======
    :lst: Llista d'arestes del graf
    """

    # Agafem tots els nodes únics
    list1, list2 = zip(*lst)
    unique_nodes = set(list1+list2)

    # Inicialitzem les dues variables rank i parent que ens serviran per anar construïn
    rank = defaultdict(int)
    parent = {n: n for n in unique_nodes}

    # Recorrem totes les parelles
    for node1, node2 in lst:

        # Busquem el node arrel de cada un dels nodes
        parent1 = find(parent, node1)
        parent2 = find(parent, node2)

        # OBSERVACIÓ: L'única cosa que hem de canviar respecte l'algorisme union-find d
        if parent1 == parent2:
            return True

        # Els unim
        union(parent, rank, parent1, parent2)

    return False
```

```python
def knapsack(K, E):
    """
    Implementació del problema de la motxilla.

    Params
    ======
    :K: Pes màxim que la motxilla pot carregar
    :E: Elements disponibles representats com una llista de tuples E=[(w,v)] on:
        :w: Pes de l'objecte.
        :v: Valor de l'objecte.

    Returns
    =======
    :selected_elems: La llista dels elements escollits.
    :total_weight: El pes total dels objectes que hem afegit.
    :total_value: El valor total dels objectes que hem afegit.
    """

    # Opció 1: Escollim l'element de valor més elevat sense tenir en compte el pes
    def find_element_best_value(weight, K, E):
        candidates = [e for e in E if e[0] <= K-weight]
        if len(candidates)>0:
            candidates = sorted(candidates, key=lambda x: -x[1])
            return candidates[0]
        return False

    # Opció 2: Escollim l'element que té un millor equilibri entre pes i valor
    def find_element_best_ratio(weight, K, E):
        candidates = [e for e in E if e[0] <= K-weight]
        if len(candidates)>0:
            candidates = sorted(candidates, key=lambda x: -x[1]/x[0])
            return candidates[0]
        return False


    # Assignem una de les múltiples funcions que podem definir.
    find_element = find_element_best_ratio

    # Inicialitzem el pes actual de la motxilla, el valor acumulat fins el moment i els ítems que hi hem carre
    total_weight = 0
    total_value  = 0
    selected_elems = []

    # Escollim el millor element seguin una política
    elem = find_element(total_weight, K, E)

    # Mentre existeixin elements que poguem afegir...
    while elem:

        # Eliminem l'element de la llista
        E.remove(elem)

        # Modifiquem els valors que emmagatzemen la informació
        total_weight += elem[0]
        total_value += elem[1]
        selected_elems.append(elem)

        # Busquem un nou element
        elem = find_element(total_weight, K, E)


    return selected_elems, total_weight, total_value
```

```python
def dfs(G, visited, current_node):
    # Versió recursiva del DFS.

    # Si el node actual no està visitat, l'afegim
    if current_node not in visited:
        print(current_node, end=' ')
        visited.add(current_node)

        # Per a cada veí del node actual, cridem de nou a la funció DFS per seguir visitant
        for nei in G.neighbors(current_node):
            dfs(G, visited, nei)
```

```python
# Implementació del DFS. Funció auxiliar per a resoldre el problema de les components connexes.
# En aquest cas, passarem una llista extra 'cc' que conté els nodes visitats dins de la component actual
# En canvi, 'visited' contindrà tots els nodes visitats fins el moment (és a dir, de totes les components con
def aux_dfs(G, node, visited, cc):
    if node not in visited:
        visited.add(node)
        cc.append(node)
        for nei in G.neighbors(node):
            # Crida recursiva
            visited, cc = aux_dfs(G, nei, visited, cc)
    return visited, cc

def connected_components(G):

    # Per practicar amb llistes i conjunts, podem definir 'visited' com a conjunt (set()) i
    # la llista de components connexes com a llista ([]).
    visited = set()    # Conté tots els nodes visitats fins al moment
    connected_lst = [] # Contindrà les components connexes (llista de llistes)
    for n in G.nodes():
        if n not in visited:
            cc = [] # Inicialitzem la component connexa i visitem tots els seus nodes
            visited, cc = aux_dfs(G, n, visited, cc)
            connected_lst.append(cc)

    return len(connected_lst), connected_lst
```

```python
# VERSIÓ 2
# Funció DFS auxiliar, molt similar a la del primer exercici. Passem com a paràmetre el grup que toca assignar al node.
# Farem el mateix que en la solució anterior amb una millora. Parem l'execució si trobem una contradicció.

def dfs_paint_v2(G, n, grup):

    # Comprovem si ja hem visitat el node actual n, igual que abans
    # Li assignem un grup en cas que no ho haguem fet encara
    if 'grup' not in G.nodes[n]:
        G.nodes[n]['grup']=grup

        # Per cada veí del node, si ja està pintat, comprovem si son o no del mateix color.
        # Si ho son, ja hem acabat i l'algorisme ha de retornar False
        for nei in G.neighbors(n):
            if 'grup' in G.nodes[nei]: # Comprovem si el veí està visitat
                if G.nodes[nei]['grup']==G.nodes[n]['grup']: # Comprovem si són del mateix color
                    return False

            # Si el veí no està visitat, l'explorem.
            # Observeu que al cridar aquesta funció estem comprovant si retornarà False.
            # En cas que sigui així, hem d'acabar l'execució
            else:
                if not dfs_paint_v2(G, nei, grup*-1): # És a dir
                    return False

    return True

def es_bipartit_v2(G):
    # Agafem un node qualsevol, en aquest cas estem triant el primer de la llista
    n = list(G.nodes())[-1]

    # Apliquem DFS per assignar els grups, començant per assignar-li el grup 1 al primer node
    return dfs_paint_v2(G,n,1)
```

```python
def flood_fill(matrix, start_x, start_y, new_color):
    # Dimensions de la imatge
    width = len(matrix)
    height = len(matrix[0])
    visited=set()

    def fill(x,y,start_color,color_to_update):
        if (x,y) not in visited and 0<=x<width and 0<=y<height:
            t=(x,y)
            visited.add(t)
            if np.all(matrix[x][y]==start_color):
                matrix[x][y]=color_to_update
            for (u,v) in [(x+1,y),(x-1,y),(x,y+1),(x,y-1),(x-1,y-1),(x-1,y+1),(x+1,y-1),(x+1,y+1)]:
                fill(u,v,start_color,color_to_update)



    start_color = matrix[start_x][start_y].copy() # Agafem el color de la zona seleccionada
    fill(start_x,start_y,start_color,new_color)   # Fem la primera crida a l'algorisme amb el node inicial


newdata = data.copy()             # Fem una còpia per no modificar la 'data' actual
newcolor = [255,255,0]            # Color (RGB) que usarem per pintar. Són tres valors de 0 a 255. (RGB = Red
startpoint = (0, len(newdata)-1)  # Cantonada superior dreta

flood_fill(newdata, startpoint[0], startpoint[1], newcolor)
print(newdata[startpoint[0]][startpoint[1]])

plt.imshow(newdata)
plt.show()
```

```python
import networkx as nx
from matplotlib import pyplot as plt

# Observeu que ara el nostre DFS necessita un paràmetre extra: previous_node.
# Aquest és el node amb el que cridem el DFS a la iteració següent.
def dfs_cycles(G, visited, current_node, previous_node):
    # Com sempre, visitem els nodes que no estiguin visitats
    if current_node not in visited:
        visited.add(current_node)

        # Per a cada veí del node actual comprovem si ja ha estat visitat
        for nei in G.neighbors(current_node):

            # En el cas que no, seguim explorant amb el DFS.
            # Observeu que al cridar la funció, aprofitem per comprovar si ja s'ha t
            # En cas que sigui aixi, acabem.
            if nei not in visited:
                if dfs_cycles(G, visited, nei, current_node):
                    return True

            # Si el veí ja havia estat visitat, comprovem si es tracta d'un cicle.
            # Per fer-ho, s'ha de satisfer que el pare del node actual i el veí no c
            # Si fóssin el mateix, vol dir que no es un cicle sino que es una aresta
            # recorrent en ambdós sentits
            elif previous_node != nei:
                return True

    # Si no s'ha trobat cap cicle, es retornarà False.
    return False

def cycles(G):
    # Triem un node qualsevol (estem suposant que el graf té una sola component conn
    n = list(G.nodes())[0]

    # Inicialitzem un conjunt on guardem els nodes que ja han estat visitats.
    visited = set()

    return dfs_cycles(G, visited, n, None)
```

```python
def ud_euler_path(G):
    #comporvem que te cami euleria
    out_deg = defaultdict(lambda:[])
    in_deg = defaultdict(lambda:[])
    for i in G.nodes:
        for j in G.neighbors(i):
            out_deg[i].append(j)
            in_deg[j].append(i)
    start_node = 0
    fouls = 0
    for i in G.nodes:
        if not len(out_deg[i])%2==0:
            start_node = i
            fouls+=1

        if fouls>2:
            return None
    final_path = []
    stack = [start_node]
    H = G.copy()
    while len(stack)!=0:
        cn = stack[-1]
        stack =stack[:-1]
        count = 0
        aux(H, stack, cn, final_path)
    return final_path

def aux(G, stack, cn, final_path):
    count = 0
    for nei in G.neighbors(cn):
        if 'weight' not in G[cn][nei]:
            count+=1
            stack.append(nei)
            G[cn][nei]['weight'] = -1
            aux(G, stack, nei, final_path)

    if count==0:
        final_path.append(cn)
```

```python
def dfs_aux(G,v,dest,visited):
    if v == dest:
        return visited

    for u in G.neighbors(v):
        if u not in visited:
            visited.add(u)
            visited = dfs_aux(G,u,dest,visited)

    return visited

def exist_path(G, source, dest, visited):
    visited.add(source)
    visited = dfs_aux(G,source,dest,visited)

    return dest in visited, visited
```

```python
from collections import defaultdict

def union(parent, rank, node1, node2):
    """
    Operació d'unió de dos nodes. Al finalitzar aquesta funció, s'haurà assignat un del
    dos nodes d'entrada com a node pare de l'altre. Decidim quin en funció del su rang.

    Params
    ======
    :parent: Diccionari on emmagatzem quin node és pare de quin altre. Aquest diccionar
             ens permet saber l'estructura del graf que estem construint.

    :rank: Diccionari per saber el rang de cada node.
    :node1, node2: Els dos nodes que volem connectar
    """

    # Comprovem si el rang de node1 es menor o igual al del segon node.
    # En cas que sigui així, afegim el node2 com a node pare del node1
    if rank[node1] <= rank[node2]:
        parent[node1] = node2

        # Si els rangs fóssin iguals, incrementem en un el rang del node que ha esdevin
        if rank[node1]==rank[node2]:
            rank[node2]+=1

    # En cas que el rang del node1 sigui major que el del node2, node1 esdevé pare
    else:
        parent[node2] = node1


def find(parent, node):
    """
    Donat un node i el diccionari d'estructura del graf amb tots els nodes pare, retorn
    l'arrel del grup on 'node' pertany.

    Params
    ======
    :parent: Diccionari on cada node té associat un node pare.
    :node: Node del que volem comprovar quin és el node arrel del grup al que pertany.

    Returns
    =======
    :root: Node arrel del grup on 'node' pertany.

    """

    # Anem obtenint el pare del node d'entrada fins que aquest no en tingui cap
    # Això passarà quan el node pare sigui ell mateix.
    while parent[node] != node:
        node = parent[node]

    return node


def find_compressed(parent, node):
    if parent[node]!=node:
        parent[node] = find_compressed(parent, node)

    return parent[node]
```

```python
def floyd_warshall(G):

    # Nombre total de nodes
    n = len(G.nodes())

    # Diccionaris de conversió. A cada node li assignem un nombre enter des de 0 fins a n-1
    id2index = {k: v for v, k in enumerate(G.nodes())}
    index2id = {v: k for v, k in enumerate(G.nodes())}

    # Inicialitzem una matriu de distàncies de valor infinit
    dist = np.zeros((n,n))+np.inf

    # Recorrem totes les arestes del graf. Modifiquem la matriu inicial amb els pesos de les arestes.
    for u,v,p in G.edges(data=True):
        dist[id2index[u],id2index[v]] = p['weight']

    # La diagonal de la matriu ha de ser zero
    for i in range(n):
        dist[i,i] = 0

    # Fins aquí hem inicialitzat la matriu. Ara apliquem la segona part de l'algorisme on hem d'anar modificant la matriu.
    for k in range(n):
        for i in range(n):
            for j in range(n):
                # Si hem trobat un valor millor, l'actualitzem
                if(dist[i,j]>dist[i,k]+dist[k,j]):
                    dist[i,j] = dist[i,k]+dist[k,j]

    return dist, id2index
```

```python
G = nx.DiGraph()
G.add_weighted_edges_from([(1, 3, -2), (2, 1, 4),
                           (2, 3, 3),(4, 2, -1),(3, 4, 2)])


pos = nx.planar_layout(G)
nx.draw(G,pos,with_labels=True)

labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)

dist, id2index = floyd_warshall(G)

n1, n2 = 3, 2
print(f"El camí més curt entre {n1} i {n2} és {dist[id2index[n1],id2index[n2]]}")
```

```
El camí més curt entre 3 i 2 és 1.0
```