Ejercicio 1

`def createGraph(List, List):`

```
def createGraph(ListVertices, ListAristas):
    longVert = linkedlist.lengthList(ListVertices) + 1
    Grafo = [ [] for _ in range(longVert) ]

    for i in range(len(Grafo)):
        Grafo[i] = []

    currentNode = ListAristas.head
    while currentNode != None:
        Grafo[currentNode.value[0]].append(currentNode.value[1])
        Grafo[currentNode.value[1]].append(currentNode.value[0])
        currentNode = currentNode.nextNode
    return Grafo
```

Ejercicio 2

`def existPath(Grafo, v1, v2):`

```
def existPath(Grafo, v1, v2):
    visitados = [False] * len(Grafo)-1
    return searchPath(Grafo, v1, v2, visitados)

def searchPath(Grafo, v, v2, visitados):
    visitados[v] = True

    if v == v2:
        return True

    for u in Grafo[v]:
        if not visitados[u]:
            if searchPath(Grafo, u, v2, visitados):
                return True
    return False
```

Ejercicio 3

`def isConnected(Grafo):`

```python
def isConnected(Grafo):
    longVert = len(Grafo)-1
    visitados = []

    searchIfConnected(visitados, Grafo, 0)

    if len(visitados) == longVert:
        return True
    return False

def searchIfConnected(visitados, Grafo, vertice):
    visitados.append(vertice)
    for element in Grafo[vertice]:
        if element not in visitados:
            searchIfConnected(visitados, Grafo, element)
```

Ejercicio 4

`def isTree(Grafo):`

```python
def isTree(Grafo):
    if isConnected(Grafo) == False:
        return False

    visitado = [False] * (len(Grafo)-1)
    if hasCycle(Grafo,0,visitado,-1) == True:
        return False
    return True

def hasCycle(Grafo,currentNode,visitado,parent):
    visitado[currentNode] = True

    for i in range(len(Grafo[currentNode])):
        aux = Grafo[currentNode][i]

        if not visitado[aux]:
            if hasCycle(Grafo, aux, visitado, currentNode):
                return True

        elif aux != parent:
            return True
    return False
```

Ejercicio 5

`def isComplete(Grafo):`

```
def isComplete(Grafo):
    vertices = len(Grafo)-1

    for i in range(vertices):
        if vertices-1 != len(Grafo[i]):
            return False
    return True
```

Ejercicio 6

`def convertTree(Grafo):`

```
def convertTree(Grafo):
    if isTree(Grafo)==True:
        return []
    edges = []

    Grafo, edges = convertToBFSTreeEdges(Grafo, 0)

    return Grafo, edges

def convertToBFSTreeEdges(Grafo, v):
    n = len(Grafo)
    BFS_tree = [[] for _ in range(n)]
    color = ['blanco' for _ in range(n)]
    parent = [-1 for _ in range(n)]
    edges = []

    color[v] = 'gris'
    cola = [v]

    while cola:
        u = cola.pop(0)
        for w in Grafo[u]:
            if color[w] == 'blanco':
                color[w] = 'gris'
                BFS_tree[u].append(w)
                BFS_tree[w].append(u)
                parent[w] = u
                cola.append(w)
            elif color[w] == "gris" and parent[u] != w:
                edges.append((u,w))
        color[u] = 'negro'

    return BFS_tree, edges
```

Parte 2

Ejercicio 7

`def countConnections(Grafo):`

```python
def countConnections(Grafo):
    n = len(Grafo)-1
    visitado = [False] * n
    componentes = 0

    def visit(u):
        visitado[u] = True
        for v in Grafo[u]:
            if not visitado[v]:
                visit(v)

    for u in range(n):
        if not visitado[u]:
            componentes += 1
            visit(u)

    return componentes
```

Ejercicio 8

`def convertToBFSTree(Grafo, v):`

```python
def convertToBFSTree(Grafo, v):
    n = len(Grafo)
    BFS_tree = [[] for _ in range(n)]
    color = ['blanco' for _ in range(n)]

    color[v] = 'gris'
    cola = [v]

    while cola:
        u = cola.pop(0)
        for w in Grafo[u]:
            if color[w] == 'blanco':
                color[w] = 'gris'
                BFS_tree[u].append(w)
                BFS_tree[w].append(u)
                cola.append(w)
        color[u] = 'negro'

    return BFS_tree
```

Ejercicio 9

`def convertToDFSTree(Grafo, v):`

```python
def convertToDFSTree(Grafo, v):
    if not isConnected(Grafo):
        return []
    visitados = [False] * len(Grafo)
    arbol = [[] for _ in range(len(Grafo))]
    padres = [None for _ in range(len(Grafo))]

    convertToDFSTreeR(Grafo, v, visitados, arbol, padres)

    return arbol

def convertToDFSTreeR(Grafo, u, visitados, arbol, padres):
    visitados[u] = True
    for v in Grafo[u]:
        if not visitados[v]:
            arbol[u].append(v)
            arbol[v].append(u)
            padres[v] = u
            convertToDFSTreeR(Grafo, v, visitados, arbol, padres)
        elif v != padres[u] and v in padres:
            continue
        elif v != padres[u]:
            arbol[u].append(v)
            arbol[v].append(u)
    return
```

Ejercicio 10

`def bestRoad(Grafo, v1, v2):`

```python
def bestRoad(Grafo, v1, v2):
    n = len(Grafo)
    visitado = [False for _ in range(n)]
    previo = [-1 for _ in range(n)]
    distancia = [float('inf') for _ in range(n)]

    distancia[v1] = 0
    visitado[v1] = True
    cola = []
    cola.append(v1)

    while cola:
        u = cola.pop()
        for v in Grafo[u]:
            if not visitado[v]:
                visitado[v] = True
                previo[v] = u
                cola.append(v)
                if v == v2:
                    camino = [v2]
                    while previo[v2] != -1:
                        camino.append(previo[v2])
                        v2 = previo[v2]
                    camino.reverse()
                    return camino
    return []
```

Ejercicio 12

Demuestre que si el grafo G es un árbol y se le agrega una arista nueva entre cualquier par de vértices se forma exactamente un ciclo y deja de ser un árbol.

Por propiedad, un árbol de n vértices siempre tendrá n-1 aristas. Si tenemos un árbol de m vértices y le agregamos una arista más, esta tendrá m aristas, por lo que niega esta propiedad y es falso, por lo que demuestra que deja de ser un árbol y se convierte en un grafo.

Ejercicio 13

Demuestre que si la arista (u,v) no pertenece al árbol BFS, entonces los niveles de u y v difieren a lo sumo en 1.

El algoritmo BFS va recorriendo de a un nivel por vez, tomando a todos los vértices adyacentes a cada uno, y cuando haya recorrido todos, pasa a los siguientes.

Supongamos que la arista (u,v) no pertenece al árbol BGS, si los niveles de u y v difieren en más de 1.
El árbol BFS está construido de tal manera que siempre se selecciona el camino más corto desde el nodo origen hasta cualquier otro nodo en el grafo, entonces esto implica que la arista (u,v) no podría ser una arista válida en el grafo, ya que si lo fuera, entonces el camino más corto desde el onoro origen hasta v tendría que pasar por u, lo cual es contradictorio a la premisa en cuestión.

Por lo tanto, si la arista (u,v) no pertenece al árbol BFS, entonces los niveles de u y v difieren a lo sumo en 1.