Resolución del ejercicio de árboles(Práctica 3 ejercicio 4):

```
class Data():# la clase Data se utiliza como un objeto parámetro
        def __init_(self):
                self.mejor_camino               = []
                self.distancia_menor  = 9999999
                self.cantidad_de_sitios         = 0

        """ setters and getters"""

        def verificar(self,caminoNuevo, sitios,distancia):
                # se encarga de evaluar las condiciones del enunciado
                if (distancia < self.getDistancia()) or (distancia == self.getDistancia() and sitios >
                self.getCantidadDeSitios()):
                        self.setDistancia(distancia)
                        self.setCantidadDeSitios(sitios)
                        self.setMejorCamino(caminoNuevo)


class Turismo():
        def __init__(self,arbol):
                self.rutas = arbol # el arbol que se dispone

        def recorrer(self,rutas,camino,cantidad_de_sitios,distancia_camino,data):
                if rutas.esHoja():
                        # delego la responsabilidad a Data()
                        data.verificar(camino,cantidad_de_sitios,distancia)
                else:
                        cantidad_de_sitios += rutas.getDatoRaiz().getCantidadDeSitios()
                        distancia_camino   += rutas.getDatoRaiz().getDistancia()
                        camino.append(rutas.getDatoRaiz())
                        for hijo in rutas.getHijos():
                                self.recorrer(hijo,camino,cantidad_de_sitios,distancia_camino,data)
                        del camino[len(camino)-1]
                        # elimino el ultimo para que con la recursión me quede formado correctamente el camino


        def mejorCamino(self):# metodo pedido
                data = Data()
                self.recorrer(self.rutas,[], 0, 0, data)
                return data.getMejorCamino()
```

Resolución ejercicio de grafos (Práctica 5 ejercicio 4: con la modificación de devolver 2 en vez de 5):

```
class Maximo():
    def __init__(self):
        # utilizo una estructura de dic, se podría usar cualquier otra
        # donde pueda guardar el nombre de la empresa junta a la cantidad
        # de servicios que brinda directa o indirectamente
        self.empresas           = {}

    """ setters and getters"""

    def verificar(self,empresa,cantidad):
        empresas = self.getEmpresas()
        if len(empresas)<2:
            empresas[empresa.getNombre()] = cantidad
        else:
            # debo calcular la maxima para comparar con la nueva
            empresas.values().sort() # ordena el diccionario por valor, quedando el maximo primero
            if empresas.values()[0]< cantidad:# si el que esta primero es menor a cantidad
                del empresas[empresas.keys()[0]] # elimino el candidato a reemplazar
                empresas[empresa.getNombre()] = cantidad # agrego la nueva empresa
        self.setEmpresas(empresas)

    def getEmpresasMaximas(self): return self.empresas.keys()


class RedDeEmpresas():
    def bfs(self,vertice,visitados,maximos):
        # estrategia: contar todas las aristas de los no visitados
        cantidad = 0
        cola = Cola()
        visitados[vertice.getPosicion()] = True
        cola.poner(vertice)
        while not cola.esVacia():
            v = cola.sacar()
            for arista in v.obtenerAdyacentes():
                if not visitados[arista.verticeDestino().getPosicion()]:
                    cantidad +=1
                    cola.poner(arista.verticeDestino())
        # una vez calculada la cantidad de aristas
        # delego la responsabilidad a maximos
        maximos.verificar(vertice.getDato(),cantidad)
```

```
# metodo pedido
def topServicios(grafo):
        maximos         = Maximo()
        visitados       = []
        for i in grafo.listaDeVertices():visitados.append(False)
        for vertice in grafo.listaDeVertices():
                """ solucion con bfs """
                self.bfs(vertice,visitados,maximos)
                visitados = []# se desmarca todo de vuelta
                for i in grafo.listaDeVertices():visitados.append(False)
        return maximos.getEmpresasMaximas()
```

Resolución de ejercicio de Tiempo de ejecución (Practica 4 ejercicio 6: con la modificación del for para que sea más sencilla la resolución)

Dado el siguiente método, plantear y resolver la función de recurrencia:

```
def funcion(n):
        x=0
        if n <= 1:
                return 1
        else:
                for i in range(1,n):
                        x += 1
                return funcion(n/2) + funcion(n/2)
```

función de recurrencia:

$$T(n) = \begin{cases} cte1, & n<=1 \\ 2*T(n/2) + \sum_{i=1}^{n-1} cte2 + cte3 \end{cases}$$

resuelvo la sumatoria 1º

$$\sum_{i=1}^{n-1} cte2 = (n-1)*cte2 = n*cte2 - cte2$$

ahora reemplazo, descartando los negativos

$$T(n) = \begin{cases} cte1, & n<=1 \\ 2*T(n/2) + n*cte2 + cte3 \end{cases}$$

paso 1 : 2*T(n/2) +n*cte2 + cte3

paso 2:

2*[2*T(n/2/2) + n/2*cte2 + cte3] + n*cte2 + cte3

2^2 T(n/2^2) + 2*n*cte2 + 3*cte3

paso 3:

2^2 [2*T(n/2^2/2) + n/2/2 * cte2 + cte3] + 2*n*cte2 +3*cte3

2^3*T(n/2^3) + 3*n*cte2 + 7*cte3

paso k: $2^k * T(n/2^k) + k*n*cte2 + (2^k - 1)*cte3$

$n/2^k = 1$

k = $Log_2 n$

$2^{(\log n)} * T(n/2^{(\log n)}) + \log n * n * cte2 + 2^{(\log n)} - 1 * cte3$

n*cte1 + log n*n*cte2 + (n -1)*cte3

T(n) = O(n*Log_2n)