

# PracticasDeGrafos2024.pdf



**Madao**



**Estructuras de Datos no Lineales**



**2º Grado en Ingeniería Informática**



**Escuela Superior de Ingeniería  
Universidad de Cádiz**

Formamos  
**talento** para un futuro  
**Sostenible**



MÁSTER EN

**Big Data &  
Business Analytics**

**EOI** Escuela de  
organización  
industrial

[saber más](#)

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

pierdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

PRACTICAS DE GRAFOS

ESTRUCTURA DE DATOS NO LINEALES

PRACTICAS DE GRAFOS

AUTOR/A: MADAO

WUOLAH

WUOLAH

# Índice general

















---

<b>I Práctica 6</b>	<b>1</b>
1. Ejercicio 2	2
2. Ejercicio 3	3
3. Ejercicio 4	4
 <b>II Práctica 7</b>	 <b>6</b>
4. Ejercicio 1	7
5. Ejercicio 2	9
6. Ejercicio 6	12
7. Ejercicio 7	14
8. Ejercicio 8	15
9. Ejercicio 9	17
10.Ejercicio 10	20
11.Ejercicio 11	25
12.Ejercicio 12	27
13.Ejercicio 13	29
 <b>III Práctica 8</b>	 <b>32</b>
14.Ejercicio 1	33
15.Ejercicio 2	36
16.Ejercicio 3	38
17.Ejercicio 4	39
18.Ejercicio 5	40
19.Ejercicio 6	41

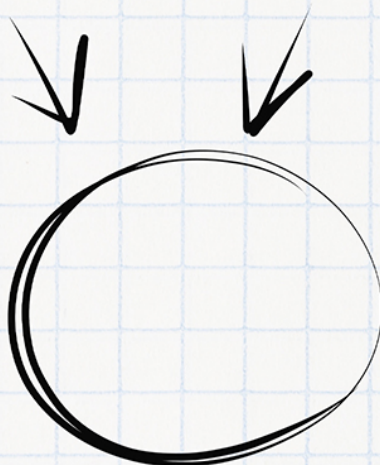


# Imagínate aprobando el examen

## Necesitas tiempo y concentración

Planes	 PLAN TURBO	 PLAN PRO	 PLAN PRO+
 Descargas sin publi al mes	10 	40 	80 
 Elimina el video entre descargas			
 Descarga carpetas			
 Descarga archivos grandes			
 Visualiza apuntes online sin publi			
 Elimina toda la publi web			
 Precios <span>Anual <input type="checkbox"/></span>	0,99 € / mes	3,99 € / mes	7,99 € / mes

Ahora que puedes conseguirlo,  
¿Qué nota vas a sacar?



# WUOLAH



## Estructuras de Datos no Line...



Banco de apuntes de la

**Comparte estos flyers en tu clase y consigue más dinero y recompensas**

- 1** Imprime esta hoja
- 2** Recorta por la mitad
- 3** Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
- 4** Llévate dinero por cada descarga de los documentos descargados a través de tu QR



# Parte I

## Práctica 6

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

perdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH

## 1. Ejercicio 2

Definiremos el pseudocentro de un grafo conexo como el nodo del mismo que minimiza la suma de las distancias mínimas a sus dos nodos más alejados. Definiremos el diámetro del grafo como la suma de las distancias mínimas a los dos nodos más alejados del pseudocentro del grafo. Dado un grafo conexo representado mediante matriz de costes, implementa un subprograma que devuelva la longitud de su diámetro.

```
size_t diametroPseudocentroGrafo(const GrafoP<size_t> &G)
{
    matriz<size_t> D;
    matriz<GrafoP<size_t>::vertice> V;
    D = Floyd(G, V);
    // Calcular el pseudocentro
    size_t pseudoCentro;
    float mediaPseudocentro = GrafoP<size_t>::INFINITO;
    for (size_t i = 0; i < D.dimension(); i++)
    {
        float mediaLocal = 0;
        for (size_t j = 0; j < D.dimension(); j++)
        {
            mediaLocal += D[i][j];
        }
        mediaLocal /= G.numVert();
        if (mediaLocal < mediaPseudocentro)
        {
            mediaPseudocentro = mediaLocal;
            pseudoCentro = i;
        }
    }
    // Diametro: longitud de los nodos mas lejanos
    size_t maxuno = 0;
    size_t maxdos = 0;
    for (size_t i = 0; i < D.dimension(); i++)
    {
        if (D[pseudoCentro][i] > maxdos)
        {
            maxdos = D[pseudoCentro][i];
            if (maxdos > maxuno)
                std::swap(maxdos, maxuno);
        }
    }
    return (maxuno + maxdos);
}
```



## 2. Ejercicio 3

---

3. Tu empresa de transportes “PEROTRAVEZUNGRAFO S.A.” acaba de recibir la lista de posibles subvenciones del Ministerio de Fomento en la que una de las más jugosas se concede a las empresas cuyo grafo asociado a su matriz de costes sea acíclico. ¿Puedes pedir esta subvención? Implementa un subprograma que a partir de la matriz de costes nos indique si tu empresa tiene derecho a dicha subvención.

```
bool subvencion(const GrafoP<size_t> &G)
{
    matriz<size_t> D;
    matriz<GrafoP<size_t>::vertice> V;
    D = Floyd(G, V);
    bool res = true;
    // Recorremos V para comprobar si hay ciclos
    // Usando un vector booleano de vertices visitados que se pasa por
    ↪ referencia
    //
    for (size_t i = 0; i < V.dimension(); i++)
    {
        std::vector<bool> visitados(V.dimension(), false); // vector de
        ↪ booleanos inicializado a false
        if (hayCiclo(V, i, visitados))
            res = false;
    }
    return res;
}

bool hayCiclo(const matriz<GrafoP<size_t>::vertice> &V, size_t vertice,
    ↪ std::vector<bool> &visitados)
{
    // Si ya hemos visitado el vertice, hay ciclo
    if (visitados[vertice])
        return true;
    visitados[vertice] = true; // Marcamos como
    ↪ visitado el vertice
    for (size_t i = 0; i < V.dimension(); i++) // Recorremos los
    ↪ vertices adyacentes a el vertice actual
        if (V[vertice][i] != GrafoP<size_t>::INFINITO) // Si hay arista
        ↪ entre los vertices y
            return (hayCiclo(V, i, visitados)); // Si hay ciclo en
            ↪ el vertice adyacente // Desmarcamos
            ↪ el vertice
    return false; // No hay ciclo
}
```

### 3. Ejercicio 4

---

Se necesita hacer un estudio de las distancias mínimas necesarias para viajar entre dos ciudades cualesquiera de un país llamado Zuelandia. El problema es sencillo pero hay que tener en cuenta unos pequeños detalles:

**typedef** GrafoP<size\_t>::vertice ciudad;

- a) La orografía de Zuelandia es un poco especial, las carreteras son muy estrechas y por tanto solo permiten un sentido de la circulación.

RECIBE UN GRAFO DIRIGIDO PONDERADO

- b) Actualmente Zuelandia es un país en guerra. Y de hecho hay una serie de ciudades del país que han sido tomadas por los rebeldes, por lo que no pueden ser usadas para viajar.

**typedef** std::vector<ciudad> ciudades\_rebeldes;

- c) Los rebeldes no sólo se han apoderado de ciertas ciudades del país, sino que también han cortado ciertas carreteras, (por lo que estas carreteras no pueden ser usadas).

**typedef** std::pair<ciudad, ciudad> carretera;

**typedef** std::vector<carretera> carreteras\_cortadas;

- d) Pero el gobierno no puede permanecer impasible ante la situación y ha exigido que absolutamente todos los viajes que se hagan por el país pasen por la capital del mismo, donde se harán los controles de seguridad pertinentes.

ciudad capital;

Dadas estas cuatro condiciones, se pide implementar un subprograma que dados:

- el grafo (matriz de costes) de Zuelandia en situación normal,
- la relación de las ciudades tomadas por los rebeldes,
- la relación de las carreteras cortadas por los rebeldes
- y la capital de Zuelandia,

Calcule la matriz de costes mínimos para viajar entre cualesquiera dos ciudades zuelandesas en esta situación.

```

matriz<size_t> ZuelandiaUno(GrafoP<ciudad> &Z, ciudades_rebeldes cr,
↪ carreteras_cortadas cc, ciudad capital)
{
    for (size_t i = 0; i < Z.numVert(); i++)
        Z[i][capital] = 0; // camino directo a la capital
    // recorremos el vector de carreteras
    for (size_t i = 0; i < cc.size(); i++)
        Z[cc[i].first][cc[i].second] = GrafoP<size_t>::INFINITO;
    // recorremos el vector de ciudades rebeldes
    for (size_t i = 0; i < cr.size(); i++) // accedo al tamaño del vector
        ↪ de solo las ciudades que son rebeldes
        for (size_t j = 0; j < Z.numVert(); j++)
        {
            Z[cr[i]][j] = GrafoP<size_t>::INFINITO;
            Z[j][cr[i]] = GrafoP<size_t>::INFINITO;
        }
    for (size_t i = 0; i < Z.numVert(); i++)
        if (i != capital)
            for (size_t j = 0; j < Z.numVert(); j++)
                if (j != capital)
                    Z[i][j] = GrafoP<size_t>::INFINITO;
    matriz<size_t> D;
    matriz<ciudad> P;
    D = Floyd(Z, P);
    return D;
}

```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

pierdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH



## Parte II

### Práctica 7

## 4. Ejercicio 1

---

Tu agencia de viajes “OTRAVEZUNGRAFO S.A.” se enfrenta a un curioso cliente. Es un personaje sorprendente, no le importa el dinero y quiere hacer el viaje más caro posible entre las ciudades que ofertas. Su objetivo es gastarse la mayor cantidad de dinero posible (ojalá todos los clientes fueran así), no le importa el origen ni el destino del viaje. Sabiendo que es imposible pasar dos veces por la misma ciudad, ya que casualmente el grafo de tu agencia de viajes resultó ser acíclico, devolver el coste, origen y destino de tan curioso viaje. Se parte de la matriz de costes directos entre las ciudades del grafo. Controlar los INFINITOS que le entren para que no los coja como maximos

```
template <typename tCoste>
matriz<tCoste> FloydMaximo(const GrafoP<tCoste> &G,
                           matriz<typename GrafoP<tCoste>::vertice> &P)
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    const size_t n = G.numVert();
    matriz<tCoste> A(n); // matriz de costes mínimos
    const tCoste INFINITO = GrafoP<tCoste>::INFINITO;
    // Iniciar A y P con caminos directos entre cada par de vértices.
    P = matriz<vertice>(n);
    for (vertice i = 0; i < n; i++)
    {
        A[i] = G[i]; // copia costes del grafo
        A[i][i] = 0; // diagonal a 0
        P[i] = vector<vertice>(n, i); // caminos directos
    }
    // Calcular costes mínimos y caminos correspondientes
    // entre cualquier par de vértices i, j
    for (vertice k = 0; k < n; k++)
        for (vertice i = 0; i < n; i++)
            for (vertice j = 0; j < n; j++)
                tCoste ikj = maximoInteligente(A[i][k], A[k][j]);
    return A;
}
```



```
template <typename tCoste>
tCoste maximoInteligente(tCoste a, tCoste b)
{
    if (a == GrafoP<tCoste>::INFINITO)
        return b;
    if (b == GrafoP<tCoste>::INFINITO)
        return a;
    return std::max(a, b);
}
```

## 5. Ejercicio 2

---

Se dispone de un laberinto de  $N \times N$  casillas del que se conocen las casillas de entrada y salida del mismo. Si te encuentras en una casilla sólo puedes moverte en las siguientes cuatro direcciones (arriba, abajo, derecha, izquierda). Por otra parte, entre algunas de las casillas hay una pared que impide moverse entre las dos casillas que separa dicha pared (en caso contrario no sería un verdadero laberinto). Implementa un subprograma que dados

- $N$  (dimensión del laberinto)

Un `size_t`

- la lista de paredes del laberinto coordenadas de inicio y fin de la pared

```
struct coordenada
{
    size_t fila;
    size_t columna;
};
struct pared
{
    coordenada inicio, fin;
};
```

- la casilla de entrada

tipo `coordenada` par de `size_t`

- la casilla de salida

calcule el camino más corto para ir de la entrada a la salida y su longitud.

```
typedef GrafoP<size_t> laberinto;
typedef vector<pared> Paredes;
```

```
bool adyacentes(coordenada a, coordenada b)
{
    return ((valorAbsoluto(a.fila - b.fila) + valorAbsoluto(a.columna -
    ↪ b.columna)) == 1);
}
```

```
size_t valorAbsoluto(int a)
{
    return (a < 0) ? -a : a;
}
```

```
coordenada nodoACoordenada(laberinto::vertice n, size_t N)
{
```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

pierdo espacio



Necesito concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH

```
coordenada c;
c.fila = n / N;
c.columna = n % N;
return c;
}

laberinto::vertice coordenadaANodo(coordenada c, size_t N)
{
    return c.fila * N + c.columna;
}

vector<GrafoP<size_t>::vertice> laberinto(size_t N, Paredes p, coordenada
→ entrada, coordenada salida)
{
    GrafoP<size_t> G(N * N);
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            if (i > 0)
            {
                if (find(p.begin(), p.end(), pared{{i, j}, {i - 1, j}}) ==
→ p.end())
                {
                    G[coordenadaANodo({i, j}, N)][coordenadaANodo({i - 1,
→ j}, N)] = 1;
                }
            }
            if (i < N - 1)
            {
                if (find(p.begin(), p.end(), pared{{i, j}, {i + 1, j}}) ==
→ p.end())
                {
                    G[coordenadaANodo({i, j}, N)][coordenadaANodo({i + 1,
→ j}, N)] = 1;
                }
            }
            if (j > 0)
            {
                if (find(p.begin(), p.end(), pared{{i, j}, {i, j - 1}}) ==
→ p.end())
                {
                    G[coordenadaANodo({i, j}, N)][coordenadaANodo({i, j -
→ 1}, N)] = 1;
                }
            }
            if (j < N - 1)
```

```

    {
        if (find(p.begin(), p.end(), pared{{i, j}, {i, j + 1}}) ==
            ↪ p.end())
        {
            G[coordenadaANodo({i, j}, N)][coordenadaANodo({i, j +
            ↪ 1}, N)] = 1;
        }
    }
}

vector<GrafoP<size_t>::vertice> P;
matriz<GrafoP<size_t>::vertice> M;
Floyd(G, M);
P.push_back(coordenadaANodo(entrada, N));
GrafoP<size_t>::vertice v = coordenadaANodo(salida, N);
while (v != P.back())
{
    P.push_back(v);
    v = M[coordenadaANodo(entrada, N)][v];
}
return P;
}

```

## 6. Ejercicio 6

---

Al dueño de una agencia de transportes se le plantea la siguiente situación. La agencia de viajes ofrece distintas trayectorias combinadas entre  $N$  ciudades españolas utilizando tren y autobús. Se dispone de dos grafos que representan los costes (matriz de costes) de viajar entre diferentes ciudades, por un lado en tren, y por otro en autobús (por supuesto entre las ciudades que tengan línea directa entre ellas). Además coincide que los taxis de toda España se encuentran en estos momentos en huelga general, lo que implica que sólo se podrá cambiar de transporte en una ciudad determinada en la que, por casualidad, las estaciones de tren y autobús están unidas. Implementa una función que calcule la tarifa mínima (matriz de costes mínimos) de viajar entre cualesquiera de las  $N$  ciudades disponiendo del grafo de costes en autobús, del grafo de costes en tren, y de la ciudad que tiene las estaciones unidas.

Varias maneras:

- Ir siempre en tren  
Floyd(tren)
- Ir siempre en bus  
Floyd(bus)
- Ir en tren hasta la ciudad de cambio y luego en bus  
Columna de la ciudad de cambio en tren + fila de la ciudad de cambio en bus
- Ir en bus hasta la ciudad de cambio y luego en tren  
Columna de la ciudad de cambio en bus + fila de la ciudad de cambio en tren

La solución es el mínimo de las 4 opciones. Como nos pide una matriz el algoritmo involucrado SUELE ser Floyd

```
size_t minimo(size_t a, size_t b, size_t c, size_t d)
{
    return std::min(std::min(a, b), std::min(c, d));
}

matriz<size_t> tarifaMinima(const GrafoP<size_t> &tren, const
    ↪ GrafoP<size_t> &bus, size_t ciudadCambio)
{
    matriz<size_t> RES(tren.numVert()); // Asumo que tren y bus tienen el
    ↪ mismo número de vértices (mismas ciudades), sean conexas o no
    matriz<size_t> trenFloyd(tren.numVert());
    matriz<size_t> busFloyd(bus.numVert());
    matriz<size_t> verticesTren(tren.numVert());
    matriz<size_t> verticesBus(bus.numVert());
    trenFloyd = Floyd(tren, verticesTren);
    busFloyd = Floyd(bus, verticesBus);
```



```

for (size_t i = 0; i < tren.numVert(); i++)
{
    for (size_t j = 0; j < tren.numVert(); j++)
    {
        RES[i][j] = minimo(trenFloyd[i][j],
                           busFloyd[i][j],
                           trenFloyd[i][ciudadCambio] +
                           ↪ busFloyd[ciudadCambio][j],
                           busFloyd[i][ciudadCambio] +
                           ↪ trenFloyd[ciudadCambio][j]);
    }
}
return RES;
}

```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

perdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH

## 7. Ejercicio 7

Se dispone de dos grafos (matriz de costes) que representan los costes de viajar entre ciudades españolas utilizando el tren (primer grafo) y el autobús (segundo grafo). Los grafos representan viajes entre las mismas  $N$  ciudades. El objetivo es hallar el camino de coste mínimo para viajar entre dos ciudades. Se cretas del grafo, origen y destino, en las siguientes condiciones:

- La ciudad origen sólo dispone de transporte por tren.
- La ciudad destino sólo dispone de transporte por autobús.
- El sector del taxi, bastante conflictivo en nuestros problemas, sigue en huelga,

por lo que únicamente es posible cambiar de transporte en dos ciudades del grafo, cambio1 y cambio2, donde las estaciones de tren y autobús están unidas. Implementa un subprograma que calcule la ruta y el coste mínimo para viajar entre las ciudades Origen y Destino en estas condiciones. Varias formas:

- Ir en tren hasta la ciudad de cambio 1 y luego en bus.  
Columna de la ciudad de cambio 1 en tren + fila de la ciudad de cambio 1 en bus
- Ir en tren hasta la ciudad de cambio 2 y luego en bus.  
Columna de la ciudad de cambio 2 en tren + fila de la ciudad de cambio 2 en bus.

La solución es el mínimo de las 2 opciones. Se hace 2 Dijkstra: hasta cambio 1 y hasta cambio 2 con el grafo de tren. Se hace 2 DijkstraInverso desde cambio 1 y desde cambio 2 con el grafo de bus.

```
size_t minimo(size_t a, size_t b)
{
    return std::min(a, b);
}

size_t costeMinimoSiete(const GrafoP<size_t> &tren, const GrafoP<size_t>
→ &bus, size_t origen, size_t destino, size_t cambio1, size_t cambio2)
{
    vector<size_t> distanciasTren;
    vector<size_t> distanciasBus;
    vector<size_t> verticesTren;
    vector<size_t> verticesBus;
    distanciasTren = Dijkstra(tren, origen, verticesTren);
    distanciasBus = DijkstraInv(bus, destino, verticesBus);
    size_t a = distanciasTren[cambio1] + distanciasBus[cambio1];
    size_t b = distanciasTren[cambio2] + distanciasBus[cambio2];
    return minimo(a, b);
}
```

## 8. Ejercicio 8

---

“UN SOLO TRANSBORDO, POR FAVOR”. Este es el título que reza en tu flamante compañía de viajes. Tu publicidad explica, por supuesto, que ofreces viajes combinados de TREN y/o AUTOBÚS (es decir, viajes en tren, en autobús, o usando ambos), entre N ciudades del país, que ofreces un servicio inmejorable, precios muy competitivos, y que garantizas ante notario algo que no ofrece ninguno de tus competidores: que en todos tus viajes COMO MÁXIMO se hará un solo transbordo (cambio de medio de transporte). Bien, hoy es 1 de Julio y comienza la temporada de viajes. ¡Qué suerte! Acaba de aparecer un cliente en tu oficina. Te explica que quiere viajar entre dos ciudades, Origen y Destino, y quiere saber cuánto le costará. Para responder a esa pregunta dispones de dos grafos de costes directos (matriz de costes) de viajar entre las N ciudades del país, un grafo con los costes de viajar en tren y otro en autobús. Implementa un subprograma que calcule la tarifa mínima en estas condiciones. Mucha suerte en el negocio, que la competencia es dura.

Varias formas:

1. Ir en tren: Dijkstra(tren)
2. Ir en bus: Dijkstra(bus)
3. Ir en tren hasta la ciudad de cambio y luego en bus: Dijkstra(tren) + DijkstraInverso(bus)
4. Ir en bus hasta la ciudad de cambio y luego en tren: Dijkstra(bus) + DijkstraInverso(tren)

La solución es el mínimo de las 4 opciones

```
size_t costeMinimoOcho(const GrafoP<size_t> &tren, const GrafoP<size_t>
↪ &bus, size_t origen, size_t destino)
{
    vector<size_t> distanciasTren;
    vector<size_t> distanciasBus;
    vector<size_t> verticesTren;
    vector<size_t> verticesBus;
    distanciasTren = Dijkstra(tren, origen, verticesTren);
    distanciasBus = Dijkstra(bus, origen, verticesBus);
    size_t a = distanciasTren[destino];
    size_t b = distanciasBus[destino];
    size_t c;
    size_t d;
    for (size_t i = 0; i < tren.numVert(); i++)
    {
        c = distanciasTren[i] + DijkstraInv(bus, i, verticesBus)[destino];
        d = distanciasBus[i] + DijkstraInv(tren, i, verticesTren)[destino];
    }
}
```

```
    }  
    return minimo(a, b, c, d);  
}
```

## 9. Ejercicio 9

Se dispone de dos grafos que representan la matriz de costes para viajes en un determinado país, pero por diferentes medios de transporte (tren y autobús, por ejemplo). Por supuesto ambos grafos tendrán el mismo número de nodos (mismas ciudades),  $N$ . Dados ambos grafos, una ciudad de origen, una ciudad de destino y el coste del taxi para cambiar de una estación a otra dentro de cualquier ciudad (se supone constante e igual para todas las ciudades), implementa un subprograma que calcule el camino y el coste mínimo para ir de la ciudad origen a la ciudad destino.

Se resuelve haciendo un nuevo grafo de  $2*N$  nodos, donde los primeros  $N$  nodos

En una matriz quedarían:

	TREN	BUS
TREN	GRAFO TREN	DIAGONAL TAXI
BUS	DIAGONAL TAXI	GRAFO BUS

En el primer cuadrante se pone el grafo de tren. En el segundo cuadrante se pone la diagonal con el coste del taxi. En el tercer cuadrante se pone la diagonal con el coste del taxi. En el cuarto cuadrante se pone el grafo de bus.

Se hace 2 Dijkstra, uno desde la estación de tren de la ciudad origen y otro desde la estación de bus de la ciudad origen.

Los destinos son la estación de tren de la ciudad destino y la estación de bus de la ciudad destino.

Por tanto la solución es el mínimo de 4 valores:

1. Dijkstra(tren)[destinoTren]
2. Dijkstra(tren)[destinoBus]
3. Dijkstra(bus)[destinoTren]
4. Dijkstra(bus)[destinoBus]

```
typedef std::pair<vector<size_t>, size_t> resultado;
```

```
// Función:
```

```
resultado costeMinimoNueve(const GrafoP<size_t> &tren, const
↳ GrafoP<size_t> &bus, size_t origen, size_t destino, size_t taxi)
{
    // Crear grafo de 2*N nodos
    GrafoP<size_t> G(2 * tren.numVert());

    // Rellenar grafo
    for (size_t i = 0; i < tren.numVert(); i++)
    {
```



Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

perdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH

```
for (size_t j = 0; j < tren.numVert(); j++)
{
    // Primer cuadrante (grafo de tren)
    if (i < tren.numVert() && j < tren.numVert())
    {
        G[i][j] = tren[i][j];
    }
    // Segundo o tercer cuadrante (diagonal con el coste del taxi)
    else if ((i < tren.numVert() && j >= tren.numVert()) || (i >=
    → tren.numVert() && j < tren.numVert()))
    {
        if (i + tren.numVert() == j || j + tren.numVert() == i)
            G[i][j] = taxi;
        else
            G[i][j] = GrafoP<size_t>::INFINITO;
    }
    // Cuarto cuadrante (grafo de bus)
    else
    {
        G[i][j] = bus[i - tren.numVert()][j - tren.numVert()];
    }
}

// Dijkstra desde la estación de tren de la ciudad origen
vector<size_t> distanciasTren;
vector<size_t> verticesTren;
size_t origenTren = origen;
size_t destinoTren = destino;
distanciasTren = Dijkstra(G, origenTren, verticesTren); // el camino
→ de costes minimos de Tren

// Dijkstra desde la estación de bus de la ciudad origen
vector<size_t> distanciasBus;
vector<size_t> verticesBus;
size_t origenBus = origen + tren.numVert();
size_t destinoBus = destino + tren.numVert();
distanciasBus = Dijkstra(G, origenBus, verticesBus); // Camino de
→ costes minimos de Bus

// Calcular el coste mínimo
size_t a = distanciasTren[destinoTren];
size_t b = distanciasTren[destinoBus];
size_t c = distanciasBus[destinoTren];
size_t d = distanciasBus[destinoBus];

size_t min = minimo_inteligente(a, b, c, d);
```

```

// Devolver el camino y el coste mínimo
vector<size_t> verticesCamino = (min == a || min == b) ? verticesTren
↪ : verticesBus;

vector<size_t> camino;
size_t i = (min == a || min == c) ? destinoTren : destinoBus;
while (i != origen)
{
    camino.push_back(i);
    i = verticesCamino[i];
}
camino.push_back(origen);
camino.reserve(camino.size());
return {camino, min};
}

size_t minimo_inteligente(size_t a, size_t b, size_t c, size_t d)
{
    // Hacer el minimo pero si es infinito no lo coja
    size_t min = a;
    if (b != GrafoP<size_t>::INFINITO && b < min)
        min = b;
    if (c != GrafoP<size_t>::INFINITO && c < min)
        min = c;
    if (d != GrafoP<size_t>::INFINITO && d < min)
        min = d;
    return min;
}

```

## 10. Ejercicio 10

Se dispone de tres grafos que representan la matriz de costes para viajes en un determinado país, pero por diferentes medios de transporte (tren, autobús y avión). Por supuesto los tres grafos tendrán el mismo número de nodos,  $N$ . Dados los siguientes datos:

- los tres grafos,
- una ciudad de origen,
- una ciudad de destino,
- el coste del taxi para cambiar, dentro de una ciudad, de la estación de tren a la de autobús o viceversa (taxi-tren-bus) y
- el coste del taxi desde el aeropuerto a la estación de tren o la de autobús, o viceversa (taxi-aeropuerto-tren/bus)

y asumiendo que ambos costes de taxi (distintos entre sí, son dos costes diferentes) son constantes e iguales para todas las ciudades, implementa un subprograma que calcule el camino y el coste mínimo para ir de la ciudad origen a la ciudad destino.

La idea es igual que la anterior, pero ahora el grafo es de  $3*N$  nodos. En una matriz quedarían:

	TREN	BUS	AVION
TREN	GRAFO TREN	TAXI TREN-BUS	TAXI TREN-AVION
BUS	TAXI BUS-TREN	GRAFO BUS	TAXI BUS-AVION
AVION	TAXI AVION-TREN	TAXI AVION-BUS	GRAFO AVION

Hacemos 3 Dijkstra, uno desde la estación de tren de la ciudad origen, otro desde la estación de bus de la ciudad origen y otro desde el aeropuerto de la ciudad origen.

Los destinos son la estación de tren de la ciudad destino, la estación de bus de la ciudad destino y el aeropuerto de la ciudad destino.

Por tanto la solución es el mínimo de 9 valores:

1. Dijkstra(tren)[destinoTren]
2. Dijkstra(tren)[destinoBus]
3. Dijkstra(tren)[destinoAvion]
4. Dijkstra(bus)[destinoTren]
5. Dijkstra(bus)[destinoBus]
6. Dijkstra(bus)[destinoAvion]
7. Dijkstra(avion)[destinoTren]
8. Dijkstra(avion)[destinoBus]

## 9. Dijkstra(avion)[destinoAvion]

```
// Función:
resultado costeMinimoDiez(const GrafoP<size_t> &tren, const
→ GrafoP<size_t> &bus, const GrafoP<size_t> &avion, size_t origen,
→ size_t destino, size_t taxi, size_t taxiAvion)
{
    // Crear grafo de 3*N nodos
    GrafoP<size_t> G(3 * tren.numVert());

    // Rellenar grafo
    for (size_t i = 0; i < G.numVert(); i++)
    {
        for (size_t j = 0; j < G.numVert(); j++)
        {
            // Primer cuadrante (grafo de tren)
            if (i < tren.numVert() && j < tren.numVert())
            {
                G[i][j] = tren[i][j];
            }
            // Segundo cuadrante (taxi tren-bus)
            else if (i < tren.numVert() && j >= tren.numVert() && j < 2 *
→ tren.numVert())
            {
                if (j - tren.numVert() == i)
                    G[i][j] = taxi;
                else
                    G[i][j] = GrafoP<size_t>::INFINITO;
            }
            // Tercer cuadrante (taxi tren-avion)
            else if (i < tren.numVert() && j >= 2 * tren.numVert())
            {
                if (j - 2 * tren.numVert() == i)
                    G[i][j] = taxiAvion;
                else
                    G[i][j] = GrafoP<size_t>::INFINITO;
            }
            // Cuarto cuadrante (taxi bus-tren)
            else if (i >= tren.numVert() && i < 2 * tren.numVert() && j <
→ tren.numVert())
            {
                if (i - tren.numVert() == j)
                    G[i][j] = taxi;
                else
                    G[i][j] = GrafoP<size_t>::INFINITO;
            }
            // Quinto cuadrante (grafo de bus)
```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

perdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH



```
else if ((i >= tren.numVert() && i < 2 * tren.numVert()) && (j
↳ >= tren.numVert() && j < 2 * tren.numVert()))
{
    G[i][j] = bus[i - tren.numVert()][j - tren.numVert()];
}
// Sexto cuadrante (taxi bus-avion)
else if (i >= tren.numVert() && i < 2 * tren.numVert() && j >=
↳ 2 * tren.numVert())
{
    if (i - tren.numVert() == j - 2 * tren.numVert())
        G[i][j] = taxiAvion;
    else
        G[i][j] = GrafoP<size_t>::INFINITO;
}
// Séptimo cuadrante (taxi avion-tren)
else if (i >= 2 * tren.numVert() && j < tren.numVert())
{
    if (i - 2 * tren.numVert() == j)
        G[i][j] = taxiAvion;
    else
        G[i][j] = GrafoP<size_t>::INFINITO;
}
// Octavo cuadrante (taxi avion-bus)
else if (i >= 2 * tren.numVert() && j >= tren.numVert() && j <
↳ 2 * tren.numVert())
{
    if (i - 2 * tren.numVert() == j - tren.numVert())
        G[i][j] = taxiAvion;
    else
        G[i][j] = GrafoP<size_t>::INFINITO;
}
// Noveno cuadrante (grafo de avion)
else
{
    G[i][j] = avion[i - 2 * tren.numVert()][j - 2 *
↳ tren.numVert()];
}
}

// Dijkstra desde la estación de tren de la ciudad origen
vector<size_t> distanciasTren;
vector<size_t> verticesTren;
size_t origenTren = origen;
size_t destinoTren = destino;
distanciasTren = Dijkstra(G, origenTren, verticesTren); // el camino
↳ de costes minimos de Tren
```



```

// Dijkstra desde la estación de bus de la ciudad origen
vector<size_t> distanciasBus;
vector<size_t> verticesBus;
size_t origenBus = origen + tren.numVert();
size_t destinoBus = destino + tren.numVert();
distanciasBus = Dijkstra(G, origenBus, verticesBus); // Camino de
↳ costes minimos de Bus

// Dijkstra desde el aeropuerto de la ciudad origen
vector<size_t> distanciasAvion;
vector<size_t> verticesAvion;
size_t origenAvion = origen + 2 * tren.numVert();
size_t destinoAvion = destino + 2 * tren.numVert();
distanciasAvion = Dijkstra(G, origenAvion, verticesAvion); // Camino
↳ de costes minimos de Avion

// Calcular el coste mínimo
size_t a = distanciasTren[destinoTren];
size_t b = distanciasTren[destinoBus];
size_t c = distanciasTren[destinoAvion];
size_t d = distanciasBus[destinoTren];
size_t e = distanciasBus[destinoBus];
size_t f = distanciasBus[destinoAvion];
size_t g = distanciasAvion[destinoTren];
size_t h = distanciasAvion[destinoBus];
size_t i = distanciasAvion[destinoAvion];

size_t min = minimo_inteligente(a, b, c, d, e, f, g, h, i);

// Devolver el camino y el coste mínimo
vector<size_t> verticesCamino = (min == a || min == b || min == c) ?
↳ verticesTren : (min == d || min == e || min == f) ? verticesBus

vector<size_t> camino;
size_t i = (min == a || min == d || min == g) ? destinoTren : (min ==
↳ b || min == e || min == h) ? destinoBus
:

while (i != origen)
{
    camino.push_back(i);
    i = verticesCamino[i];
}

```

```

    camino.push_back(origen);
    return {camino, min};
}

size_t minimo_inteligente(size_t a, size_t b, size_t c, size_t d, size_t
→ e, size_t f, size_t g, size_t h, size_t i)
{
    // Hacer el minimo pero si es infinito no lo coja
    size_t min = a;
    if (b != GrafoP<size_t>::INFINITO && b < min)
        min = b;
    if (c != GrafoP<size_t>::INFINITO && c < min)
        min = c;
    if (d != GrafoP<size_t>::INFINITO && d < min)
        min = d;
    if (e != GrafoP<size_t>::INFINITO && e < min)
        min = e;
    if (f != GrafoP<size_t>::INFINITO && f < min)
        min = f;
    if (g != GrafoP<size_t>::INFINITO && g < min)
        min = g;
    if (h != GrafoP<size_t>::INFINITO && h < min)
        min = h;
    if (i != GrafoP<size_t>::INFINITO && i < min)
        min = i;
    return min;
}

```

# 11. Ejercicio 11

Disponemos de tres grafos (matriz de costes) que representan los costes directos de viajar entre las ciudades de tres de las islas del archipiélago de las Huríes (Zuelandia). Para poder viajar de una isla a otra se dispone de una serie de puentes que conectan ciudades de las diferentes islas a un precio francamente asequible (por decisión del Prefecto de las Huríes, el uso de los puentes es absolutamente gratuito). Si el alumno desea simplificar el problema, puede numerar las  $N_1$  ciudades de la isla 1, del 0 al  $N_1-1$ , las  $N_2$  ciudades de la isla 2, del  $N_1$  al  $N_1+N_2-1$ , y las  $N_3$  de la última, del  $N_1+N_2$  al  $N_1+N_2+N_3-1$ . Disponiendo de las tres matrices de costes directos de viajar dentro de cada una de las islas, y la lista de puentes entre ciudades de las mismas, calculad los costes mínimos de viajar entre cualesquiera dos ciudades de estas tres islas.

!!! QUE DISFRUTÉIS EL VIAJE !!!

Se resuelve haciendo un nuevo grafo de  $3*N$  nodos, donde los primeros  $N$  nodos son las ciudades de la isla 1, los siguientes  $N$  nodos son las ciudades de la isla 2 y los últimos  $N$  nodos son las ciudades de la isla 3.

En una matriz quedarían:

	ISLA 1	ISLA 2	ISLA 3
ISLA 1	GRAFO ISLA 1	PUENTES ISLA 1-2	PUENTES ISLA 1-3
ISLA 2	PUENTES ISLA 2-1	GRAFO ISLA 2	PUENTES ISLA 2-3
ISLA 3	PUENTES ISLA 3-1	PUENTES ISLA 3-2	GRAFO ISLA 3

```
// Estructura puente
typedef std::pair<size_t, size_t> puente;

// Función:
matriz<size_t> costeMinimoOnce(const GrafoP<size_t> &isla1, const
    ↪ GrafoP<size_t> &isla2, const GrafoP<size_t> &isla3, const
    ↪ vector<puente> &puentes)
{
    GrafoP<size_t> G(isla1.numVert() + isla2.numVert() + isla3.numVert());
    for (size_t i = 0; i < G.numVert(); i++)
    {
        for (size_t j = 0; j < G.numVert(); j++)
        {
            // Si es el primer cuadrante
            if (i < isla1.numVert() && j < isla1.numVert())
                G[i][j] = isla1[i][j];
            // Si es el quinto cuadrante
            else if (i >= isla1.numVert() && i < isla1.numVert() +
                ↪ isla2.numVert() && j >= isla1.numVert() && j <
                ↪ isla1.numVert() + isla2.numVert())
                G[i][j] = isla2[i - isla1.numVert()][j - isla1.numVert()];
```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

pierdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH



```
// Si es el noveno cuadrante
else if (i >= isla1.numVert() + isla2.numVert() && j >=
    ↪ isla1.numVert() + isla2.numVert())
    G[i][j] = isla3[i - isla1.numVert() - isla2.numVert()][j -
    ↪ isla1.numVert() - isla2.numVert()];
else
    G[i][j] = GrafoP<size_t>::INFINITO;
}
}

for (puente p : puentes)
{
    G[p.first][p.second] = 0;
    G[p.second][p.first] = 0;
}

matriz<size_t> M, P;
M = Floyd(G, P);
return M;
}
```

## 12. Ejercicio 12

---

El archipiélago de Grecoland (Zuelandia) está formado únicamente por dos islas, Fobos y Deimos, que tienen  $N1$  y  $N2$  ciudades, respectivamente, de las cuales  $C1$  y  $C2$  ciudades son costeras (obviamente  $C1 \leq N1$  y  $C2 \leq N2$ ). Se desea construir un puente que una ambas islas. Nuestro problema es elegir el puente a construir entre todos los posibles, sabiendo que el coste de construcción del puente se considera irrelevante. Por tanto, escogeremos aquel puente que minimice el coste global de viajar entre todas las ciudades de las dos islas, teniendo en cuenta las siguientes premisas:

1. Se asume que el coste de viajar entre las dos ciudades que una el puente es 0.
2. Para poder plantearse las mejoras en el transporte que implica la construcción de un puente frente a cualquier otro, se asume que se realizarán exactamente el mismo número de viajes entre cualesquiera ciudades del archipiélago. Por ejemplo, se considerará que el número de viajes entre la ciudad P de Fobos y la Q de Deimos será el mismo que entre las ciudades R y S de la misma isla. Dicho de otra forma, todos los posibles trayectos a realizar dentro del archipiélago son igual de importantes.

Dadas las matrices de costes directos de Fobos y Deimos y las listas de ciudades costeras de ambas islas, implementa un subprograma que calcule las dos ciudades que unirá el puente.

```
puente costeMinimoDoce(const GrafoP<size_t> &fobos, const GrafoP<size_t>
→ &deimos, const vector<size_t> &costerasFobos, const vector<size_t>
→ &costerasDeimos)
{
    // Floyd a fobos
    matriz<size_t> Mfobos, Pfobos;
    Mfobos = Floyd(fobos, Pfobos);

    // Floyd a deimos
    matriz<size_t> Mdeimos, Pdeimos;
    Mdeimos = Floyd(deimos, Pdeimos);

    size_t minFobos = GrafoP<size_t>::INFINITO;

    // desde cualquier ciudad de fobos a sus costeras
    // La ciudad elegida es el minimo de la media de los costes de ir a
    → las costeras
    for (size_t i = 0; i < costerasFobos.size(); i++)
    {
        size_t media = 0;
        for (size_t j = 0; j < fobos.numVert(); j++)
        {
```

```

        media += Mfobos[j][costerasFobos[i]];
    }
    if (media < minFobos)
        minFobos = costerasFobos[i];
}

size_t minDeimos = GrafoP<size_t>::INFINITO;
// desde cualquier ciudad de deimos a sus costeras
// La ciudad elegida es el minimo de la media de los costes de ir a
↳ las costeras
for (size_t i = 0; i < costerasDeimos.size(); i++)
{
    size_t media = 0;
    for (size_t j = 0; j < deimos.numVert(); j++)
    {
        media += Mdeimos[j][costerasDeimos[i]];
    }
    if (media < minDeimos)
        minDeimos = costerasDeimos[i];
}

return {minFobos, minDeimos};
}

```

## 13. Ejercicio 13

---

El archipiélago de las Huríes acaba de ser devastado por un maremoto de dimensiones desconocidas hasta la fecha. La primera consecuencia ha sido que todos y cada uno de los puentes que unían las diferentes ciudades de las tres islas han sido destruidos. En misión de urgencia las Naciones Unidas han decidido construir el mínimo número de puentes que permitan unir las tres islas. Asumiendo que el coste de construcción de los puentes implicados los pagará la ONU, por lo que se considera irrelevante, nuestro problema es decidir qué puentes deben construirse.

Las tres islas de las Huríes tienen respectivamente  $N_1$ ,  $N_2$  y  $N_3$  ciudades, de las cuales  $C_1$ ,  $C_2$  y  $C_3$  son costeras (obviamente  $C_1 \leq N_1$ ,  $C_2 \leq N_2$  y  $C_3 \leq N_3$ ). Nuestro problema es elegir los puentes a construir entre todos los posibles. Por tanto, escogeremos aquellos puentes que minimicen el coste global de viajar entre todas las ciudades de las tres islas, teniendo en cuenta las siguientes premisas:

1. Se asume que el coste de viajar entre las ciudades que unan los puentes es 0.
2. La ONU subvencionará únicamente el número mínimo de puentes necesario para comunicar las tres islas.
3. Para poder plantearse las mejoras en el transporte que implica la construcción de un puente frente a cualquier otro, se asume que se realizarán exactamente el mismo número de viajes entre cualesquiera ciudades del archipiélago. Dicho de otra forma, todos los posibles trayectos a realizar dentro del archipiélago son igual de importantes.

Dadas las matrices de costes directos de las tres islas y las listas de ciudades costeras del archipiélago, implementad un subprograma que calcule los puentes a construir en las condiciones anteriormente descritas.

```
vector<puede> costeMinimoTrece(const GrafoP<size_t> &isla1, const
→ GrafoP<size_t> &isla2, const GrafoP<size_t> &isla3, const
→ vector<size_t> &costerasI1, const vector<size_t> &costerasI2, const
→ vector<size_t> &costerasI3)
{
    // Floyd a isla1
    matriz<size_t> Misla1, PIsla1;
    PIsla1 = Floyd(isla1, PIsla1);

    // Floyd a isla2
    matriz<size_t> Misla2, PIsla2;
    Misla2 = Floyd(isla2, PIsla2);

    // Floyd a isla3
    matriz<size_t> Misla3, PIsla3;
```



Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

pierdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH

```
Misla3 = Floyd(isla3, PIsla3);
```

```
size_t minIsla1 = GrafoP<size_t>::INFINITO;
```

```
// desde cualquier ciudad de isla1 a sus costeras  
// La ciudad elegida es el minimo de la media de los costes de ir a  
→ las costeras
```

```
for (size_t i = 0; i < costerasI1.size(); i++)  
{  
    size_t media = 0;  
    for (size_t j = 0; j < isla1.numVert(); j++)  
    {  
        media += Misla1[j][costerasI1[i];  
    }  
    if (media < minIsla1)  
        minIsla1 = costerasI1[i];  
}
```

```
size_t minIsla2 = GrafoP<size_t>::INFINITO;
```

```
// desde cualquier ciudad de isla2 a sus costeras  
// La ciudad elegida es el minimo de la media de los costes de ir a  
→ las costeras
```

```
for (size_t i = 0; i < costerasI2.size(); i++)  
{  
    size_t media = 0;  
    for (size_t j = 0; j < isla2.numVert(); j++)  
    {  
        media += Misla2[j][costerasI2[i];  
    }  
    if (media < minIsla2)  
        minIsla2 = costerasI2[i];  
}
```

```
size_t minIsla3 = GrafoP<size_t>::INFINITO;
```

```
// desde cualquier ciudad de isla3 a sus costeras  
// La ciudad elegida es el minimo de la media de los costes de ir a  
→ las costeras
```

```
for (size_t i = 0; i < costerasI3.size(); i++)  
{  
    size_t media = 0;  
    for (size_t j = 0; j < isla3.numVert(); j++)  
    {  
        media += Misla3[j][costerasI3[i];  
    }  
}
```

```

    }
    if (media < minIsla3)
        minIsla3 = costerasI3[i];
}

vector<punto> puentes;
puentes.push_back({minIsla1, minIsla2});
puentes.push_back({minIsla2, minIsla3});
puentes.push_back({minIsla1, minIsla3});

return puentes;
}

```

# Parte III

## Práctica 8

## 14. Ejercicio 1

---

El archipiélago de Tombuctú, está formado por un número indeterminado de islas, cada una de las cuales tiene, a su vez, un número indeterminado de ciudades. En cambio, sí es conocido el número total de ciudades de Tombuctú (podemos llamarlo  $N$ , por ejemplo).

Dentro de cada una de las islas existen carreteras que permiten viajar entre todas las ciudades de la isla. Se dispone de las coordenadas cartesianas  $(x, y)$  de todas y cada una de las ciudades del archipiélago.

Se dispone de un grafo (matriz de adyacencia) en el que se indica si existe carretera directa entre cualesquiera dos ciudades del archipiélago. El objetivo de nuestro problema es encontrar qué ciudades de Tombuctú pertenecen a cada una de las islas del mismo y cuál es el coste mínimo de viajar entre cualesquiera dos ciudades de una misma isla de Tombuctú.

Así pues, dados los siguientes datos:

- Lista de ciudades de Tombuctú representada cada una de ellas por sus coordenadas cartesianas.
- Matriz de adyacencia de Tombuctú, que indica las carreteras existentes en dicho archipiélago.

Implementen un subprograma que calcule y devuelva la distribución en islas de las ciudades de Tombuctú, así como el coste mínimo de viajar entre cualesquiera dos ciudades de una misma isla del archipiélago.

```
typedef std::pair<size_t, size_t> ciudad;

typedef struct
{
    vector<ciudad> ciudades;
    matriz<size_t> coste;
} isla;

typedef vector<isla> archipielago;

// Vector ciudad:
// [{0,1},{2,4},{3,5},{6,7},{8,9},{10,11},{12,13}]
// [ c1,   c2,   c3,   c4,   c5,   c6,   c7]

// Matriz de adyacencia:
//      [c1,   c2,   c3,   c4,   c5,   c6,   c7]
// c1 [ 0,   1,   0,   0,   0,   0,   0]
// c2 [ 1,   0,   1,   0,   0,   0,   0]
// c3 [ 0,   1,   0,   0,   0,   0,   0]
```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

perdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH

```
// c4 [ 0, 0, 0, 0, 1, 0, 0]
// c5 [ 0, 0, 0, 1, 0, 1, 0]
// c6 [ 0, 0, 0, 0, 1, 0, 1]
// c7 [ 0, 0, 0, 0, 0, 1, 0]
```

```
archipelago tombuctuUno(vector<ciudad> c, Grafo carreteras)
{
    Particion p(c.size());
    // [{c1},{c2},{c3},{c4},{c5},{c6},{c7}]
    for (size_t i = 0; i < c.size(); i++)
    {
        for (size_t j = i + 1; j < c.size(); j++)
        {
            if (carreteras[i][j])
            {
                p.unir(p.encontrar(i), p.encontrar(j));
            }
        }
    }
    // Ahora mismo en p tengo las islas ya formadas
    // [{c1,c2,c3},{c4,c5,c6,c7}]
    archipelago archipelagoRes;
    // recorrer la particion creando por cada subconjunto una isla
    // a cada isla le vamos a calcular el costeMinimo de viajar entre
    // → todas las ciudades de esta
    // Y la vamos a meter en res
    size_t representante = p.encontrar(0);
    isla islaAux;
    for (size_t i = 0; i < c.size(); i++)
    {
        // si el representante es el mismo, meto la ciudad
        if (representante == p.encontrar(i))
        {
            islaAux.ciudades.push_back(c[i]);
        }
        // Si ha cambiado de representante, meto la isla en res y creo una
        // → nueva isla
        else
        {
            // calculo el coste minimo de viajar entre todas las ciudades
            // → de la isla
            islaAux.coste = calculaCostes(islaAux.ciudades);
            archipelagoRes.push_back(islaAux);
            // una vez haya añadido todas las ciudades, limpio ese vector
            // → para saltar a la siguiente isla
            islaAux.ciudades.clear();
            islaAux.ciudades.push_back(c[i]);
        }
    }
}
```

```

        representante = p.encontrar(i);
    }
}
return archipielagoRes;
}
matriz<size_t> calculaCostes(vector<ciudad> c)
{
    // creamos un grafo en donde almacenaremos los costes de ir de una
    // ciudad a otra
    GrafoP<size_t> costes(c.size());

    matriz<size_t> res(costes.numVert(), vertices(costes.numVert()));
    for (size_t i = 0; i < c.size(); i++)
        for (size_t j = i + 1; j < c.size(); j++)
            costes[i][j] = distanciaEuclidea(c[i], c[j]);

    res = Floyd(costes, vertices);
    return res;
}

size_t distanciaEuclidea(ciudad a, ciudad b)
{
    return sqrt(pow(a.first - b.first, 2) + pow(a.second - b.second, 2));
}

```

## 15. Ejercicio 2

---

2. El archipiélago de Tombuctú<sup>2</sup> está formado por un número desconocido de islas, cada una de las cuales tiene, a su vez, un número desconocido de ciudades, las cuales tienen en común que todas y cada una de ellas dispone de un aeropuerto. Sí que se conoce el número total de ciudades del archipiélago (podemos llamarlo  $N$ , por ejemplo).

Dentro de cada una de las islas existen carreteras que permiten viajar entre todas las ciudades de la isla. No existen puentes que unan las islas y se ha decidido que la opción de comunicación más económica de implantar será el avión. Se dispone de las coordenadas cartesianas  $(x, y)$  de todas y cada una de las ciudades del archipiélago. Se dispone de un grafo (matriz de adyacencia) en el que se indica si existe carretera directa entre cualesquiera dos ciudades del archipiélago. El objetivo de nuestro problema es encontrar qué líneas aéreas debemos implantar para poder viajar entre todas las ciudades del archipiélago, siguiendo los siguientes criterios:

1. Se implantará una y sólo una línea aérea entre cada par de islas.
2. La línea aérea escogida entre cada par de islas será la más corta entre todas las posibles.

Así pues, dados los siguientes datos:

- Lista de ciudades de Tombuctú<sup>2</sup> representada cada una de ellas por sus coordenadas cartesianas.
- Matriz de adyacencia de Tombuctú que indica las carreteras existentes en dicho archipiélago.

Implementen un subprograma que calcule y devuelva las líneas aéreas necesarias para comunicar adecuadamente el archipiélago siguiendo los criterios anteriormente expuestos.

```
typedef vector<std::pair<ciudad, ciudad>> lineasAereas;

lineasAereas tombuctuDos(vector<ciudad> c, Grafo carreteras)
{
    archipelago archi = tombuctuUno(c, carreteras);
    lineasAereas lineasAereasRes;

    // Bucle para recorrer todas las islas
    for (size_t i = 0; i < archi.size(); i++)
    { // Bucle para recorrer la isla anterior con todas saltantose a
      ↪ si misma y a las anteriores
        for (size_t j = i + 1; j < archi.size(); j++)
        {
            size_t costeMin = GrafoP<size_t>::INFINITO;
```



```

        ciudad ciudadIsla1, ciudadIsla2;
        for (size_t k = 0; k < archi[i].ciudades.size(); k++)
        {
            for (size_t l = 0; l < archi[j].ciudades.size(); l++)
            {
                if (distanciaEuclidea(archi[i].ciudades[k],
                    ↪ archi[j].ciudades[l]) < costeMin)
                {
                    costeMin = archi[i].coste[k][l];
                    ciudadIsla1 = archi[i].ciudades[k];
                    ciudadIsla2 = archi[j].ciudades[l];
                }
            }
        }
        lineasAereasRes.push_back({ciudadIsla1, ciudadIsla2});
    }
}

return lineasAereasRes;
}

```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

pierdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH

## 16. Ejercicio 3

Implementa un subprograma para encontrar un árbol de extensión máximo. ¿Es más difícil que encontrar un árbol de extensión mínimo?

```
template <typename tCoste>
GrafoP<tCoste> KruskallMaximo(const GrafoP<tCoste> &G)
// Devuelve un árbol generador de coste máximo
// de un grafo no dirigido ponderado y conexo G.
{
    assert(!G.esDirigido());

    typedef typename GrafoP<tCoste>::vertice vertice;
    typedef typename GrafoP<tCoste>::arista arista;

    const size_t n = G.numVert();
    Particion p(n);
    vector<arista> aristas;
    for (vertice u = 0; u < n; ++u)
        for (vertice v = u + 1; v < n; ++v)
            if (G[u][v] != GrafoP<tCoste>::INFINITO)
                aristas.push_back(arista(u, v, G[u][v]));

    sort(aristas.begin(), aristas.end(), greater<arista>()); //
    ↪ ordenar por la arista mayor

    // Igual a Kruskall pero en vez de ordenar de menor a mayor,
    ↪ ordenamos de mayor a menor
    GrafoP<tCoste> A(n);
    for (size_t i = 0; i < aristas.size(); ++i)
    {
        vertice u = aristas[i].u, v = aristas[i].v;
        if (p.encontrar(u) != p.encontrar(v))
        {
            A[u][v] = A[v][u] = aristas[i].coste;
            p.unir(u, v);
        }
    }
    return A;
}
```

## 17. Ejercicio 4

---

4. La empresa EMASAJER S.A. tiene que unir mediante canales todas las ciudades del valle del Jerte (Cáceres). Calcula qué canales y de qué longitud deben construirse partiendo del grafo con las distancias entre las ciudades y asumiendo las siguientes premisas:

- El coste de abrir cada nuevo canal es casi prohibitivo, luego la solución final debe tener un número mínimo de canales.
- El Ministerio de Fomento nos subvenciona por Kms de canal, luego los canales deben ser de la longitud máxima posible.

```
GrafoP<size_t> Jerte(vector<ciudad> c)
{
    // Crear un grafo con las distancias entre las ciudades
    GrafoP<size_t> d(c.size());
    for (size_t i = 0; i < c.size(); i++)
        for (size_t j = i + 1; j < c.size(); j++)
            d[i][j] = distanciaEuclidea(c[i], c[j]);
    // Calcular el arbol de expansion MAXIMO
    return KruskallMaximo(d);
}
```

## 18. Ejercicio 5

---

La nueva compañía de telefonía RETEUNI3 tiene que conectar entre sí, con fibra óptica, todas y cada una de las ciudades del país. Partiendo del grafo que representa las distancias entre todas las ciudades del mismo, implementad un subprograma que calcule la longitud mínima de fibra óptica necesaria para realizar dicha conexión.

```
size_t RETEUNI3(GrafoP<size_t> c)
{
    GrafoP<size_t> d = Kruskall(c);
    // Recorrer la matriz de distancias y sumarlas
    size_t res = 0;
    for (size_t i = 0; i < d.numVert(); i++)
        for (size_t j = i + 1; j < d.numVert(); j++)
            res += d[i][j];
    return res;
}
```

## 19. Ejercicio 6

La empresa EMASAJER S.A. tiene que unir mediante canales todas las ciudades del valle del Jerte (Cáceres), teniendo en cuenta las siguientes premisas:

- El coste de abrir cada nuevo canal es casi prohibitivo, luego la solución final debe tener un número mínimo de canales.
- El Ministerio de Fomento nos subvenciona por m<sup>3</sup>/sg de caudal, luego el conjunto de los canales debe admitir el mayor caudal posible, pero por otra parte, el coste de abrir cada canal es proporcional a su longitud, por lo que el conjunto de los canales también debería medir lo menos posible. Así pues, la solución óptima debería combinar adecuadamente ambos factores.

Dada la matriz de distancias entre las diferentes ciudades del valle del Jerte, otra matriz con los diferentes caudales máximos admisibles entre estas ciudades teniendo en cuenta su orografía, la subvención que nos da Fomento por m<sup>3</sup>/sg de caudal y el coste por km de canal, implementen un subprograma que calcule qué canales y de qué longitud y caudal deben construirse para minimizar el coste total de la red de canales.

```
typedef struct
{
    size_t caudal;
    size_t distancia;
} canal;

GrafoP<canal> ejercicio6(matriz<size_t> caudales, matriz<size_t>
→ distancias, size_t subvencionCaudal, size_t costeDistancia)
{
    GrafoP<size_t> aux(caudales.dimension());
    // rellenamos el grafo con los caudales y distancias
    for (size_t i = 0; i < caudales.dimension(); i++)
        for (size_t j = i + 1; j < caudales.dimension(); j++)
            aux[i][j] = caudales[i][j] * subvencionCaudal -
            → distancias[i][j] * costeDistancia;
    aux = KruskallMaximo(aux);
    GrafoP<canal> res(aux.numVert());
    for (size_t i = 0; i < aux.numVert(); i++)
        for (size_t j = i + 1; j < aux.numVert(); j++)
        {
            if (aux[i][j] != GrafoP<size_t>::INFINITO) // el infinito
            → es para comprobar que no existe camino
            {
                // a cada una de las aristas que quedan una vez
                → aplicado Kruskal le ponemos su caudal y distancia
                → para devolver el grafo que nos piden
            }
        }
    return res;
}
```

Importante

Puedo eliminar la publi de este documento con 1 coin

¿Cómo consigo coins? → Plan Turbo: barato  
→ Planes pro: más coins

pierdo  
espacio



Necesito  
concentración

ali ali ooh  
esto con 1 coin me  
lo quito yo...

WUOLAH



```
        res[i][j].caudal = caudales[i][j];  
        res[i][j].distancia = distancias[i][j];  
    }  
}  
return res;  
}
```

## 20. Ejercicio 7

---

El archipiélago de Grecoland (Zuelandia) está formado únicamente por dos islas, Fobos y Deimos, que tienen  $N_1$  y  $N_2$  ciudades, respectivamente, de las cuales  $C_1$  y  $C_2$  ciudades son costeras (obviamente  $C_1 \leq N_1$  y  $C_2 \leq N_2$ ). Se dispone de las coordenadas cartesianas  $(x, y)$  de todas y cada una de las ciudades del archipiélago. El huracán Isadore acaba de devastar el archipiélago, con lo que todas las carreteras y puentes construidos en su día han desaparecido. En esta terrible situación se pide ayuda a la ONU, que acepta reconstruir el archipiélago (es decir, volver a comunicar todas las ciudades del archipiélago) siempre que se haga al mínimo coste.

De cara a poder comparar costes de posibles reconstrucciones se asume lo siguiente:

- El coste de construir cualquier carretera o cualquier puente es proporcional a su longitud (distancia euclídea entre las poblaciones de inicio y fin de la carretera o del puente).
- Cualquier puente que se construya siempre será más caro que cualquier carretera que se construya.

De cara a poder calcular los costes de viajar entre cualquier ciudad del archipiélago se considerará lo siguiente:

- El coste directo de viajar, es decir de utilización de una carretera o de un puente, coincidirá con su longitud (distancia euclídea entre las poblaciones origen y destino de la carretera o del puente).

En estas condiciones, implementa un subprograma que calcule el coste mínimo de viajar entre dos ciudades de Grecoland, origen y destino, después de haberse reconstruido el archipiélago, dados los siguientes datos:

1. Lista de ciudades de Fobos representadas mediante sus coordenadas cartesianas.
2. Lista de ciudades de Deimos representadas mediante sus coordenadas cartesianas.
3. Lista de ciudades costeras de Fobos.
4. Lista de ciudades costeras de Deimos.
5. Ciudad origen del viaje.
6. Ciudad destino del viaje.

```
size_t Grecoland(vector<ciudad> fobos, vector<ciudad> deimos,
    ↪ vector<ciudad> costerasFobos, vector<ciudad> costerasDeimos,
    ↪ size_t origen, size_t destino)
{
    GrafoP<size_t> grecoland(fobos.size() + deimos.size());

    // Rellenar con los costes de ir a todas las ciudades de fobos
```



```

GrafoP<size_t> grafoFobos(fobos.size());
for (size_t i = 0; i < fobos.size(); i++)
    for (size_t j = i + 1; j < fobos.size(); j++)
        grafoFobos[i][j] = distanciaEuclidea(fobos[i], fobos[j]);
grafoFobos = Kruskall(grafoFobos);

// Rellenar con los costes de ir a todas las ciudades de deimos
GrafoP<size_t> grafoDeimos(deimos.size());
for (size_t i = 0; i < deimos.size(); i++)
    for (size_t j = i + 1; j < deimos.size(); j++)
        grafoDeimos[i][j] = distanciaEuclidea(deimos[i], deimos[j]);
grafoDeimos = Kruskall(grafoDeimos);

// Rellenamos el grafo con los costes de ir por fobos, de ir por
↳ deimos y el puente que hemos creado
for (size_t i = 0; i < fobos.size(); i++)
{
    for (size_t j = 0; j < fobos.size(); j++)
    {
        grecoland[i][j] = grafoFobos[i][j];
    }
}

for (size_t i = 0; i < deimos.size(); i++)
{
    for (size_t j = 0; j < deimos.size(); j++)
    {
        grecoland[i + fobos.size()][j + fobos.size()] =
            ↳ grafoDeimos[i][j];
    }
}

// Rellenamos el grafo con los puentes
for (size_t i = 0; i < fobos.size(); i++)
{
    for (size_t j = 0; j < deimos.size(); j++)
    {
        // Si i es costera y j es costera, creamos un puente
        if (find(costerasFobos.begin(), costerasFobos.end(),
            ↳ fobos[i]) != costerasFobos.end() &&
            find(costerasDeimos.begin(), costerasDeimos.end(),
            ↳ deimos[j]) != costerasDeimos.end())
        {
            size_t costePuente = distanciaEuclidea(fobos[i],
                ↳ deimos[j]);
            grecoland[i][j + fobos.size()] = costePuente;
            grecoland[j + fobos.size()][i] = costePuente;
        }
    }
}

```

```

        }
    }
}
grecoland = Kruskall(grecoland);

// Aplicamos Floyd para obtener el coste minimo de ir de una ciudad a
// otra
matriz<size_t> vertices(grecoland.numVert());
matriz<size_t> res = Floyd(grecoland, vertices);
// Convertir un tipo ciudad a un vertice

return res[origen][destino];
}

```