Prácticas Estructura de Datos No Lineales

Índice general

1. Prácticas Árboles

En este capítulo encontrarás todas las prácticas sobre los árboles binarios, generales, ABB, entre otros.

Práctica 1: Árboles Binarios I

Para poder resolver esta práctica es necesario haber visto el Tema 2: Árboles Binarios (para poder saber cual es la especificación del TAD Abin).

Además vamos a incluir los siguientes ficheros de cabecera:

```
#include <iostream>
#include "abin_E-S.h"
#include "abin.h" //contiene la especificación e implementación del TAD
```

Ejercicio 1: *Implementa un subprograma que calcule el número de nodos de un árbol binario.*

Para calcular el número de nodos de un árbol binario vamos a crear una función recursiva no final, la cual por cada iteración vaya incrementando su valor hasta que termine de recorrer todos los nodos del árbol, hacemos una función llamadora ya que el método `contarnodos()` solamente recibe el árbol y por tanto necesitamos especificar en que nodo comienza.

```
template <typename T> size_t contarnodos(const Abin<T>& A){
  //Comprobamos que el árbol no esté vacío
  if(A.raiz() == Abin<T>::NODO_NULO)
    return 0;
  else
    return contarnodos_rec(A.raiz(),A);
}

template <typename T> size_t contarnodos_rec(typename Abin<T>::nodo n, const
↪  Abin<T>& A){
  //Caso base
  if(n == Abin<T>::NODO_NULO)
    return 0; //si no hay nodo, devuelve 0
  else
    return 1+contarnodos_rec(A.hijoIzqdo(A),A) +
    ↪  contarnodos_rec(A.hijoDrcho(A),A);
}
```

Ejercicio 2: *Implementa un subprograma que calcule la altura de un árbol binario.*

La altura en un árbol es la longitud de la rama más larga, por lo que desde el nodo raíz iremos calculando la longitud de todas las ramas y nos quedaremos con la más grande.

Si el árbol está vacío su altura será -1, si solamente tiene el nodo raíz esta será 0.

```
template <typename T> size_t alturaAbin(const Abin<T>&A){
    return alturaAbin_rec(A.raiz() , A);
```

```
}

template <typename T> size_t alturaAbin_rec(typename Abin<T>::nodo n, const
↪  Abin<T>& A){
  //Caso base
  if(n == Abin<T>::NODO_NULO)
    return -1; //un nodo nulo no tiene altura, puesto que indica no existencia de
    ↪  nodo
  else
    return 1+std::max(alturaAbin_rec(A.hijoIzqdo(n),A),
      alturaAbin_rec(A.hijoDrcho(n),A));
}
```

Ejercicio 3: *Implementa un subprograma que, dados un árbol binario y un nodo del mismo, determine la profundidad de este nodo en dicho árbol.*

La profundidad es la longitud del único camino desde dicho nodo hasta la raíz del árbol, por tanto, una vez más vamos a implementar una función recursiva que nos calcule la profundidad de dicho nodo.

```
template <typename T> size_t profundidad_rec(typename Abin<T>::nodo n, const
↪  Abin<T>&A){
  if(n == A.raiz()) //la profundidad de la raiz es 0.
    return 0;
  else
    return 1+profundidad_rec(A.padre(n),A);
}
```

Ejercicio 4: *Añade dos nuevas operaciones al TAD árbol binario, una que calcule la profundidad de un nodo y otra que calcule la altura de un nodo en un árbol dado. Implementa esta operación para la representación vectorial (índices del padre, hijo izquierdo e hijo derecho).*

Este ejercicio tendremos que incluirlo en el fichero 'abin.vec_1' (las incluimos dentro de la clase y las implemetamos). Como en la implementación vectorial no podemos realizar la llamada mediante la estructura `celda`, por lo que tendremos que hacer uso del vector de nodos '`nodos`' para poder acceder al padre e hijos, siendo el acceso al padre (`nodos[n].padre`), acceso al hijo izquierdo (`nodos[n].hizq`) y acceso al hijo derecho (`nodos[n].hder`).

Como estamos dentro del TAD Abin_vec no hace falta hacer uso del operador de resolución de ámbito '::' para incluir el `NODO_NULO` o el nodo en sí.

```
//Profundidad de un nodo
template <typename T> size_t Abin<T>::profundidad_vec(nodo n){
  //La profundidad de la raiz es 0
  if(n == 0) return 0;
  else
    return 1+profundidad_vec(nodos[n].padre);
}

//Altura del árbol
template <typename T> size_t Abin<T>::altura_vec(nodo n)const{
  //La altura de un nodo nulo no existe, puesto que este indica no existencia de
  ↪  nodo
  if(n == NODO_NULO) return -1;
```

```
  else
    return 1+std::max(altura_vec(nodos[n].hizq),altura_vec(nodos[n].hder));
}
```

Ejercicio 5: *Repite el ejercicio anterior para la representación enlazada de árboles binarios (punteros al padre, hijo izquierdo e hijo derecho).*

Este ejercicio lo incluiremos en el fichero 'abin.h' y ahora el acceso al padre e hijos se realiza mediante punteros, de la forma `nodo→padre`(acceso al padre), `nodo→hizq`(acceso al hijo izquierdo) y `nodo→hder`(acceso al hijo derecho).

Al igual que en el ejercicio 4, no nos hace falta hacer uso del operador de resolución de ámbito, puesto que estamos dentro del contexto del TAD Abin (enlazado).

```
//Profundidad de un nodo
template <typename T> size_t Abin<T>::profundidad_enla(nodo n){
  //La profundidad de la raiz es 0
  if(n == r) return 0;
  else
    return 1+(profundidad_enla(n->padre));
}

//Altura del árbol
template <typename T> size_t Abin<T>::altura_enla(nodo n){
//La altura de un nodo nulo no existe, puesto que este indica no existencia de
↪  nodo
  if(n == NODO_NULO)
    return -1;
  else
    return 1+std::max(altura_enla(n->hizq),altura_enla(n->der));
}
```

Ejercicio 6: *Implementa un subprograma que determine el nivel de desequilibrio de un árbol binario, definido como el máximo desequilibrio de todos sus nodos. El desequilibrio de un nodo se define como la diferencia entre las alturas de los subárboles del mismo.*

Como hemos visto en la teoría, el desequilibrio del un árbol es la máxima diferencia de las alturas de los subárboles, por tanto, vamos a implementar una función recursiva que nos lo calcule.

También haremos uso del método altura del ejercicio 2 de esta misma práctica ya que nos ayudará a obtener la altura de cada subárbol.

```
template <typename T>
size_t desequilibrio(const Abin<T> &A){
    return desequilibrio_rec(A.raiz(), A);
}

template <typename T> size_t desequilibrio_rec(typename Abin<T>::nodo n, const
↪  Abin<T> &A){
  if (n == Abin<T>::NODO_NULO)
    return 0;
  else
```

```
    return std::max(std::abs(A.alturaAbin_rec(A.hijoIzqdo(n), A) -
    ↪  A.alturaAbin_rec(A.hijoDrcho(n) A)),
    ↪  std::max(desequilibrio_rec(A.hijoIzqdo(n), A),
    ↪  desequilibrio_rec(A.hijoDrcho(n), A)));
}
```

Ejercicio 7: *Implementa un subprograma que determine si un árbol binario es o no pseudo-completo. En este problema entenderemos que un árbol es pseudocompleto, si en el penúltimo nivel del mismo cada uno de los nodos tiene dos hijos o ninguno.*

Como tenemos que comprobar si tiene o no los dos hijos los nodos del último nivel, vamos a crear una función llamada esHoja la cual nos devolverá un valor booleano dependiendo si el nodo tiene algún nodo o ninguno, en el caso de que tenga al menos uno, devuelve false, de lo contrario devolverá true (no tiene ningún hijo).

Pero también necesitamos una manera de indicar si ese nodo tiene los dos hijos, para ello crearemos otra función llamada dosHijos que devuelve true en el caso de que ese nodo tenga tanto hijo izquierdo, como derecho, en el caso contrario devuelve falso.

Como estamos trabajando con un árbol binario cuya implementación es la enlazada, haremos uso del método que calcula la altura implementado en el ejercicio 5

```
template <typename T> bool esHoja(typename Abin<T>::nodo n, const Abin<T>& A){
  return (A.hijoIzqdo(n)==Abin<T>::NODO_NULO && A.hijoDrcho(n) ==
  ↪  Abin<T>::NODO_NULO);
}

template <typename T> bool dosHijos(typename Abin<T>::nodo n, const Abin<T>& A){
  return (A.hijoIzqdo(n)!=Abin<T>::NODO_NULO &&
  ↪  A.hijoDrcho(n)!=Abin<T>::NODO_NULO);
}

template <typename T> bool pseudocompleto(const Abin<T>& A){
  if(A.arbolVacio() || alturaAbin(A) == 0) return true;
  else  return pseudocompleto_rec(A.raiz(), A);
}

template <typename T> bool pseudocompleto_rec(typename Abin<T>::nodo n, const
↪  Abin<T>& A){
  if(A.altura_enla(n) == 1)
    return (dosHijos(n,A));
  else{
    if(A.alura_enla(A.hijoIzqdo(n))>A.altura_enla(A.hijoDrcho(n)))
      return pseudocompleto_rec(A.hijoIzqdo(n),A);
    else if(A.alura_enla(A.hijoIzqdo(n))<A.altura_enla(A.hijoDrcho(n)))
      return pseudocompleto_rec(A.hijoDrcho(n),A);
    else  return (pseudocompleto_rec(A.hijoIzqdo(n),A) &&
    ↪  pseudocompleto_rec(A.hijoDrcho(n),A));
  }
}
```

Práctica 2: Árboles Binarios II

En esta práctica seguiremos trabajando con los árboles binarios pero resolveremos ejercicios un poco más complejos.

Como fichros de cabecera seguiremos teniendo los mismos que en la práctica 1.

Ejercicio 1: *Dos árboles binarios son similares cuando tienen idéntica estructura de ramificación, es decir, ambos son vacíos, o en caso contrario, tienen subárboles izquierdo y derecho similares. Implementa un subprograma que determine si dos árboles binarios son similares.*

En este ejercicio vamos a realizar una comparación entre dos árboles binario comparando cada subárbol de cada uno para ver si tienen la misma ramificación, por ello vamos a necesitar dos árboles y dos nodos en la función recursiva, la cual devolverá true si ambos árboles tienen la misma ramificación y en el caso contrario devuelve false.

```
template <typename T> bool similares(const Abin<T>& A, const Abin<T>& B){
  return similares_rec(A.raiz(),B.raiz());
}

template <typename T> bool similares_rec(typename Abin<T>::nodo nA, typename
↪  Abin<T>::nodo nB, const Abin<T>& A, const Abin<T>& B){
  if(nA == Abin<T>::NODO_NULO || nb == Abin<T>::NODO_NULO)
    return nA == Abin<T>::NODO_NULO && nb == Abin<T>::NODO_NULO;
  else //Vamos a comprobar la ramificación de sus hijos
    return similares_rec(A.hijoIzqdo(nA), B.hijoIzqdo(nB),A,B) &&
      similares_rec(A.hijoDrcho(nA), B.hijoDrcho(nB),A,B);
}
```

Ejercicio 2: *Para un árbol binario B, podemos construir el árbol binario reflejado B^R cambiando los subárboles izquierdo y derecho en cada nodo. Implementa un subprograma que devuelva el árbol binario reflejado de uno dado.*

```
//Declaración adelanta del método reflejado_rec
template <typename T> void reflejado_rec(typename Abin<T>::nodo nA, typename
↪  Abin<T>::nodo nB,const Abin<T>& A, Abin<T>& B);

template <typename T> Abin<T> reflejado (const Abin<T> A){
  //Creamos el árbol binario a devolver
  Abin<T> B;
  if(!A.arbolVacio()){
    B.insertarRaiz(A.elemento(A.raiz())); //inicializamos el árbol B con la raiz
    ↪  de A
    reflejado_rec(A.raiz(),B.raiz(),A,B);
  }
  return B; //Devolvemos el árbol reflejado
}
template <typename T> void reflejado_rec(typename Abin<T>::nodo nA, typename
↪  Abin<T>::nodo nB,const Abin<T>& A, Abin<T>& B){
  //Comprobamos caso base, que el nodo nA no sea nulo
  if(nA!=Abin<T>::NODO_NULO){
    if(A.hijoIzqdo(nA)!=Abin<T>::NODO_NULO){ //Si A tiene Hizqdo, se inserta en
    ↪  Hder de B
```

```
      B.insertarHijoDrcho(nB,A.elemento(A.hijoIzqdo(nA)));
      reflejado_rec(A.hijoIzqdo(nA),nB,A,B);
    }
    if(A.hijoDrcho(nA)!= Abin<T>::NODO_NULO){//Si A tiene Hder, se inserta en
    ↪  Hizq de B
      B.insertarHijoIzqdo(nB,A.hijoDrcho(nA),A,B);
      reflejado_rec(A.hijoDrcho(nA),nB,A,B);
    }
    //Si los hijos de nA son hoja, no hace nada.
  }
  //Si nA es hoja, no hace nada.
}
```

Ejercicio 3: *El TAD árbol binario puede albergar expresiones matemáticas mediante un árbol de expresión. Dentro del árbol binario los nodos hojas contendrán los operandos, y el resto de los nodos los operadores.*

a) Define el tipo de los elementos del árbol para que los nodos puedan almacenar operadores y operandos.

b) Implementa una función que tome un árbol binario de expresión (aritmética) y devuelva el resultado de la misma. Por simplificar el problema se puede asumir que el árbol representa una expresión correcta. Los operadores binarios posibles en la expresión aritmética serán suma, resta, multiplicación y división.

Sea la operación aritmética → **7×(15 - 3,2)/2**

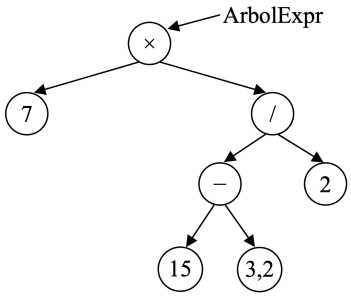


Figura 1.1: Ejemplo árbol binario de expresión (aritmética)

Nota: En el programa de prueba podemos usar las funciones rellenarAbin() de abin_E-S.h para introducir por teclado o desde un fichero el árbol de expresión a evaluar. Sin embargo, en este caso, será necesario sobrecargar los operadores utilizados internamente en dichas funciones, es decir >>, << y != para el tipo de los elementos del árbol.

Como vemos en la *Figura 1.1* los nodos que no son hojas son los que contienen los operadores aritméticos. Para poder introducir un número este nodo tiene que ser hoja por eso vamos a hacer uso de la función `esHoja()`.

```
template <typename T> bool esHoja(typename Abin<T>::nodo n, const Abin<T>& A){
  return (A.hijoIzqdo(n) == Abin<T>::NODO_NULO &&
  ↪  A.hijoDrcho(n)==Abin<T>::NODO_NULO);
}
```

Vamos a crear un tipo `Enum` que va a contener los operadores aritméticos.

```
enum operador{suma = '+', resta = '-', producto = '*', division = '/'};
```

Como vemos en el ejemplo del enunciado una expresión es la unión de un número con su operador

```
union expresion{
  double num;
  operador op;};
```

Como ya tenemos todos los componentes del ejercicio declarados, vamos a implementar la función que nos calcula la expresión aritmética.

```
double calcular(const Abin<expresion>& A){
  return calcular_rec(A.raiz(),A);
}

double calcular_rec(typename Abin<expresion>::nodo n, const Abin<expresion>& A){
  //Si el nodo es hoja, es el contenido del nodo es un número
  if(esHoja(n,A)){
    return A.elemento(n).num;
  }
  else{ //es un operador aritmético
    switch(A.elemento(n).op){
      case suma:
        return calcular_rec(A.hijoIzqdo(n),A) + calcular_rec(A.hijoDrcho(n),A);
        break;
      case resta:
        return calcular_rec(A.hijoIzqdo(n),A) - calcular_rec(A.hijoDrcho(n),A);
        break;
      case producto:
        return calcular_rec(A.hijoIzqdo(n),A) * calcular_rec(A.hijoDrcho(n),A);
        break;
      case division:
        return calcular_rec(A.hijoIzqdo(n),A) / calcular_rec(A.hijoDrcho(n),A);
        break;
    }
  }
}
```

Práctica 3: Árboles Generales

En esta práctica vamos a ver como trabajar con el TAD Agen con sus dos representaciones (enlazada y vectorial).

Como cabeceras de la práctica vamos a tener:

```
#include <iostream>
#include "agen.h" //Especificación e implementación de los métodos del TAD.
#include "agen_E-S.h"
```

Ejercicio 1: *Implementa un subprograma que dado un árbol general nos calcule su grado.*

Para saber cual es el grado de un árbol este es el máximo de los grados de los nodos que componen al árbol. Por tanto, para saber cuales el grado del Agen, tenemos que saber cual es el grado de cada nodo que lo compone y quedarnos con el máximo.

```
template <typename T> size_t gradoAgen(const Agen<T> &A){
  return gradoAgen_rec(A.raiz(),A);
}
template <typename T> size_t gradoAgen_rec(typename Agen<T>::nodo n, const
↪  Agen<T> &A){
  if(n == Agen<T>::NODO_NULO){
    return 0; //el grado de un nodo nulo (no existe nodo) es 0
  }
  else{
    size_t gradoMax = 0; //grado del árbol
    size_t nHijos = 0; //grado del nodo
    typename Agen<T>::nodo aux = A.hijoIzqdo(n);
    //Calculamos el grado del árbol.
    while(aux != Agen<T>::NODO_NULO){
      nHijos++;
      gradoMax = std::max(gradoMax,gradoAgen_rec(aux,A));
      aux = A.hermDrcho(aux);
    }
    return std::max(nHijos,gradoMax);
  }
}
```

Ejercicio 2: *Implementa un subprograma que dados un árbol y un nodo dentro de dicho árbol determine la profundidad de éste nodo en el árbol.*

Para calcular la profundidad de un nodo cualquiera en un árbol general, este se va a calcular de la misma forma que en los árboles binarios, ya que lo que tenemos que hacer es partir del nodo en cuestión e ir llamando al padre del mismo hasta llegar a la raiz.

```
template <typename T> size_t profundidadAgen_rec(typename Agen<T>::nodo n, const
↪  Agen<T> &A){
  if(n == A.raiz())
    return 0;
  else
    return 1+profundidadAgen_rec(A.padre(n),A);
}
```

Ejercicio 3: *Se define el desequilibrio de un árbol general como la máxima diferencia entre las alturas de los subárboles más bajo y más alto de cada nivel. Implementa un subprograma que calcule el grado de desequilibrio de un árbol general.*

Vamos a suponer que tenemos un método llamado `alturaAgen()`, el cual nos hará falta para poder calcular el desequilibrio de un árbol general (según la definició de desequilibrio dada).

Para calcular el desequilibrio vamos a recorrer todos los hermanos del hijo del nodo dado, y vamos calculando el desequilibrio máximo mediante la función `std::max()`.

```
template <typename T> size_t desequilibrioAgen(const Agen<T> &A){
  return profundidadAgen_rec(A.raiz(),A);
}

template <typename T> size_t desequilibrioAgen_rec(typename Agen<T>::nodo n,
↪  const Agen<T> &A){
  if(n == Agen<T>::NODO_NULO){
    return 0;
  }
  else{
    size_t maxDesequilibrio = 0;
    typename Agen<T>::nodo aux = A.hijoIzqdo(n);
    while (aux != Agen<T>::NODO_NULO){
      maxDesequilibrio = std::max(maxDesequilibrio,
      ↪  std::abs(alturaAgen_rec(aux,A) - alturaAgen(A.hermDrcho(aux),A)));

      aux = A.hermDrcho(aux);
    }
    return maxDesequilibrio;
  }
}
```

Ejercicio 4: *Dado un árbol general de enteros A y un entero x, implementa un subprograma que realice la poda de A a partir de x. Se asume que no hay elementos repetidos en A.*

Una poda significa que apartir de dicho nodo tenemos que eliminar todos sus desecendientes y a él mismo.

Para ello, primero tenemos que realizar la búsqueda del elemento 'x' en el Agen y a partir de su posición eliminar todos los desecendientes.

```
//Método buscar un nodo a partir de su contenido
template <typename T> typename Agen<T>::nodo buscar(T e, Agen<T> &A){
  return buscar_rec(e,A.raiz(),A);
}
template <typename T> typename Agen<T>::nodo buscar_rec(T e, typename
↪  Agen<T>::nodo n, Agen<T> &A){
  if(n==Agen<T>::NODO_NULO){
    return Agen<T>::NODO_NULO;
  }
  else if(A.elemento(n) == e){ //Lo hemos encontrado, lo devolvemos
    return n;
  }
  else{
```

```
    //Vamos a buscar en sus hermanos
    typename Agen<T>::nodo aux = A.hijoIzqdo(n);
    while(aux != Agen<T>::NODO_NULO){
      typename Agen<T>::nodo aux2 = buscar_rec(e,aux,A);
      if(aux2 != Agen<T>::NODO_NULO){
        return aux2;
      }
      aux = A.hermDrcho(aux);

    }
    return Agen<T>::NODO_NULO; //no existe entre los hermanos
  }
}

template <typename T> void podaAgen(T x, Agen<T> &A){
  typename Agen<T>::nodo n = buscar(x,A);
  return podaAgen_rec(n,A);
}

template <typename T> void podaAgen_rec(typename Agen<T>::nodo n, Agen<T>: &A){
  if(n != Agen<T>::NODO_NULO){
    //Mientras que n tenga hijos, podamos
    while(A.hijoIzqdo(n)!= Agen<T>::NODO_NULO){
      //Realizamos la llamada recursiva con el hijo (poda del subárbol del hijo)
      podaAgen_rec(A.hijoIzqdo(n),A);
      A.eliminarHijoIzqdo(n); //eliminamos al propio hijo
    }
  }
}
```

Práctica 4: Árboles Binarios de Búsqueda

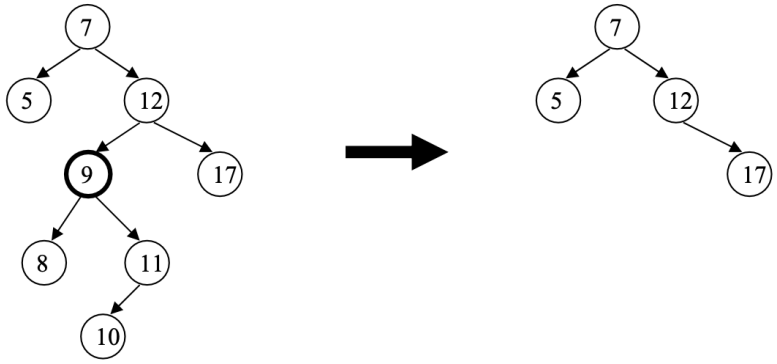
En esta práctica, veremos como trabajar con árboles binarios de búsqueda (ABB).

A diferencia de los árboles binarios o generales ahora no trabajamos con nodos, si no con subárboles del mismo.

Como ficheros de cabeceras tendremos:

```
#include <iostream>
#include "abb.h" //contiene Especificación e implementación métodos del TAD.
#include <vector> //para algunos ejercicios
```

Ejercicio 1: *Implementa una nueva operación del TAD Abb que tomando un elemento del mismo elimine al completo el subárbol que cuelga de él.* Ejemplo: Para el árbol binario de búsqueda de la figura se muestra la transformación si la entrada fuera el valor 9.



Suponemos que estamos dentro del TAD -¿ 'abb.h'

```
template <typename T> void deletSubarbol(Abb<T> A, const T& e){
  return deletSubarbol_rec(A.r, e);
}

template typename<T> void deletSubarbol_rec(typename Abb<T>::arbol *&a, const T&
↪  e){
  if(a != nullptr){
    if( e < a->elto){
      deletSubarbol_rec(a->izq.r,e);
    }
    else if(e > a->elto){
      deletSubarbol_rec (a->der.r, e);
    }
    else{
      delete a;
      a = nullptr;
    }
  }
}
```

Ejercicio 2: *Un árbol binario de búsqueda se puede equilibrar realizando el recorrido en inorden del árbol para obtener el listado ordenado de sus elementos y a continuación, repartir equitativamente los elementos a izquierda y derecha colocando la mediana en la raíz y construyendo recursivamente los subárboles izquierdo y derecho de cada nodo. Implementa este algoritmo para equilibrar un ABB.*

Para llevar a cabo este problema necesitamos almacenar todos los elementos de un Abb y mediante un recorrido en inorden realizar la inserción de dicho elementos de menera equitativa con el fin de obtener un ABB equilibrado. Aquí está el motivo de porque incluimos la cabecera `<vector>`.

Además vamos a construir el árbol equilibrado a partir del vector con los elementos, por eso el método `equilibrarABB_rec()` recibe dos enteros; posición inicio y fin del vector.

A partir de ahí se calcula la mediana que concuerda con el elemento que será la raiz del ABB.

```
template <typename T> void equilibrarABB(Abb<T> &A){
//Creamos el vector que contendrá los elementos del ABB
  vector<T> v;
//Recorrido en inorden
  inorden(A, v);
//Equilibramos todo el árbol.
  A.r = equilibrarABB_rec(v,0,v.size()-1);
}

template <typename T> Abb<T>::arbol* equilibrarABB_rec(std::vector<T> &v, size_t
↪  ini, size_t fin){
  if(ini > fin){
    return nullptr;
  }
  else{
    //sacamos la raiz
    size_t mediana = (ini+fin)/2;
    //creamos un nuevo árbol con el valor de la mediana como raiz
    typename Abb<T>::arbol* a = new typename Abb<T>::arbol(v[mediana]);
    //insertamos los elementos de forma equitativa
    a->izq.r = equilibrarABB_rec(v, ini,mediana-1);
    a->der.r = equilibrarABB_rec(v,mediana+1,fin);
    return a; //devolvemos el árbol equilibrado
  }
}
```

Ejercicio 3: *Dados dos conjuntos representados mediante árboles binarios de búsqueda, implementa la operación unión de dos conjuntos que devuelva como resultado otro conjunto que sea la unión de ambos, representado por un ABB equilibrado.*

```
template <typename T> Abb<T> Union(const Abb<T> &A, Abb<T>B){
  //Declaramos un Abb con el contenido de A que será el resultado de la unión de
  ↪  A y B.
  Abb<T> Res{A};
  while(!B.vacio()){
    //Insertamos los elemento que no se encuentren en A
    if(A.buscar(B.elemento()).vacio()){
```

```
      Res.insertar(B.elemento());
    }
    B.eliminar(B.elemento());
  }
  Equilibrar(Res);
  return Res;
}
```

Ejercicio 4: *Dados dos conjuntos representados mediante árboles binarios de búsqueda, implementa la operación intersección de dos conjuntos, que devuelva como resultado otro conjunto que sea la intersección de ambos. El resultado debe quedar en un árbol equilibrado.*

```
template <typename T> Abb<T> Intersec(const Abb<T> &A, Abb<T> B){
  //Declaramos un Abb vacio que será el resultado de la unión de A y B.
  Abb<T> Res;
  while(!B.vacio()){
    //insertamos los elementos que se encuentren en A
    if(!A.buscar(B.elemento()).vacio()){
      Res.insertar(B.elemento());
    }
    B.eliminar(B.elemento());
  }
  Equilibrar(Res);
  return Res;
}
```

Ejercicio 5: *Implementa el operador $\diamond$ para conjuntos definido como $A \diamond B = (A \cup B) - (A \cap B)$. La implementación del operador $\diamond$ debe realizarse utilizando obligatoriamente la operación $\in$, que nos indica si un elemento dado pertenece o no a un conjunto. La representación del tipo Conjunto debe ser tal que la operación de pertenencia esté en el caso promedio en $O(\log n)$.*

Para poder llevar a cabo este subprograma vamos a hacer uso de las dos funciones previamente implementadas (Unión e Intersección) de Abb.

```
//vamos a crearnos una función auxiliar que nos va a servir de ayuda para
↪  comprobar si un elemento existe o no en un conjunto.
template <typename T> bool existe(const T& e, const Abb<T> &A){
  return !A.buscar(e).vacio();
}
template <typename T> Abb<T> opRombo(const Abb<T> &A, Abb<T> B){
  //realizamos la unión e intersección de los árboles
  Abb<T> Union{Union(A,B)};
  Abb<T> Inserseccion{Intersec(A,B)};
  Abb<T> Diff{Union};

  while(!Intersección.vacio()){
    //Si encontramos el elemento en la intersección de A y B, se quita de la
    ↪  diferencia
    if(exite(Interseccion.elemento(),Union)){
      Diff.eliminar(Interseccion.elemento());
    }
    //Si no está, se elimina de la intersección
    Interseccion.eliminar(Interseccion.elemento());
```

```
    //Equilibramos el árbol despues de eliminar un elemento.
    Equilibrar(Interseccion);
  }
  //Equilibramos el árbol diferencia "Diff" ya que hemos eliminado del mismo.
  Equilibrar(Diff);
  return Diff;
}
```

Práctica 5: Árboles parcialmente ordenados y otros

En esta práctica trabajaremos con árboles parcialmente ordenados (APOs o montículos), con árboles B y con otros árboles vistos en prácticas anteriores.

Para poder realizar esta práctica vamos a poner como ficheros de cabeceras:

```
#include <iostream>
#include "abin.h"
#include "agen.h"
#include <vector> //para algunos ejercicios
#include <string.h> //para algunso ejercicios
```

Ejercicio 1: *Dado un árbol binario de enteros donde el valor de cada nodo es menor que el de sus hijos, implementa un subprograma para eliminar un valor del mismo preservando la propiedad de orden establecida. Explica razonadamente la elección de la estructura de datos.*

Nota: Se supone que en el árbol no hay elementos repetidos, y que el número de nodos del mismo no está acotado.

Según el enunciado tenemos un Abin que se comporta como un ABB debido a que el valor de los nodos que son descendientes son mayores que el del padre. Además si queremos que al eliminar cualquier nodo el árbol siga cumpliendo la propiedad de orden y sabiendo que no podemos eliminar un nodo que no sea hoja, tendremos que intercambiarlo por el nodo cuyo valor sea el menor de los dos hasta que el nodo a eliminar sea una hoja.

Por tanto, nuestra función tendrá como parámetros de entrada el elemento del nodo (para poder buscarlo), el nodo (para poder trabajar con él) y el árbol.

```
//Métodos auxiliares
template <typename T> bool dosHijos(typename Abin<T>::nodo n, Abin<T> &A){
  return (A.hijoIzqdo(n)!=Abin<T>::NODO_NULO &&
  ↪  A.hijoDrcho(n)!=Abin<T>::NODO_NULO);
}

template <typename T> bool esHoja (typename Abin<T>::nodo n, const Abin<T> &A){
  return (A.hijoIzqdo(n) == Abin<T>::NODO_NULO && A.hijoDrcho(n) ==
  ↪  Abin<T>::NODO_NULO);
}

template <typename T> Abin<T>::nodo minimo (typename Abin<T>::nodo a,typename
↪  Abin<T>::nodo a const Abin<T> &A){
  if(a == Abin<T>::NODO_NULO) return b;
  if(b == Abin<T>::NODO_NULO) return a;
  else return (A.elemento(a) < A.elemento(b)) ? a : b;
}

template <typename T> void eliminar (const T& e, const Abin<T> &A){
  eliminar_rec(e, A.raiz(), A);
}
template <typename T> void eliminar_rec(const T& e, typename Abin<T>::nodo n,
↪  Abin<T> &A){
  if(n != Abin<T>::NODO_NULO){
```

```
    //Buscamos el nodo que tenga ese elemento
    if(A.elemento(n) == e){
      if(!esHoja(n,A)){
        //Si no es una hoja puede tener 1 ó 2 hijos
        if(dosHijos(n,A)){
          //Si tiene ambos hijos, intercambiamos con el menor de los 2 y el otro
          ↪  pasa a ser hijo
          typename Abin<T>::nodo hijo = minimo(A.hijoIzqdo(n),A.hijoDrcho(n),A);
          A.elemento(n) = A.elemento(hijo);
          eliminar_rec(A.elemento(hijo),hijo,A); //seguimos bajando
        }
        else{ //tiene 1 (puede ser el izquierdo o derecho)
          typename Abin<T>::nodo hijo = (A.hijoIzqdo(n) != Abin<T>::NODO_NULO) ?
          ↪  A.hijoIzqdo(n) : A.hijoDrcho(n);
          A.elemento(n) = A.elemento(hijo);
          eliminar_rec(A.elemento(hijo),hijo,A); //seguimos bajando
        }
      }
      else{
        //es hoja, eliminamos el Nodo
        typename Abin<T>::nodo padre = A.padre(n);
        if (A.hijoIzqdo(padre) == n) {
            A.eliminarHijoIzqdo(padre);
        } else {
            A.eliminarHijoDrcho(padre);
        }
      }
      }else {
        // Seguir buscando en los hijos
        eliminar_rec(e, A.hijoIzqdo(n), A);
        eliminar_rec(e, A.hijoDrcho(n), A);
      }
    //Si es nulo, no hace nada
  }
}
```

Ejercicio 2: *Un montículo min-max tiene una estructura similar a la de un montículo ordinario (árbol parcialmente ordenado), pero la ordenación parcial consiste en que los elementos que se encuentran en un nivel par (0, 2, 4,..) son menores o iguales que sus elementos descendientes, mientras que los elementos que se encuentran en un nivel impar (1, 3, 5,..) son mayores o iguales que sus descendientes. Esto quiere decir que para cualquier elemento e de un nivel par se cumple abuelo $\leq e \leq$ padre y para cualquier elemento e de un nivel impar padre $\leq e \leq$ abuelo.*
Implementa una operación de orden logarítmico para añadir un elemento a un montículo min-max almacenado en un vector de posiciones relativas.

Vamos a insertar ese elemento al final del APO, y lo iremos flotando hasta que se quede en la posición correta, por tanto tendremos que realizar una modificación en la función `flotar()`, ya que como hemos dicho antes, insertarémos por el final del APO.

```
template <typename T> void APOmM<T>::insertar(const T& e){
```

```
  assert(ultimo < maxNodos-1); //APOMinMax no lleno
  nodos[++ultimo] = e; //insertarmos al final e incrementamos el númoero de
  ↪  elementos.
  if(ultimo > 0)
    flotar(ultimo); //reordenamos el APO.
}

template <typename T> void APOmM<T>::flotar(nodo i){
  size_t profundidad = log2(i+1);
  T e = nodos[i];
  T aux;
  nodo abuelo;
  //como i no es raiz, tiene padre
  if(profundidad % 2 == 0){//nivel par
    if(nodos[padre(i)] < nodos[i]){ //Tenemos que reordenar respecto a los
    ↪  niveles impares
      Intercambio(nodos[i],nodos[padre(i)]);
      i = padre(i);
      profundidad--;
      abuelo = padre(padre(i));
      while(abuelo >= 0 && nodos[abuelo] < nodos[i]){
        Intercambio(nodos[i],nodos[abuelo]);
        i = abuelo;
        abuelo = padre(padre(i));
        profundidad-=2;
      }
    }
    else{ //Comprobamos si tenemos que reordenar respecto a los niveles pares del
    ↪  APO.
      abuelo = padre(padre(i));
      while(abuelo >= 0 && nodos[abuelo] > nodos[i]){
        Intercambio(nodos[abuelo],nodos[i]);
        i = abuelo;
        abuelo = padre(padre(i));
        profundidad-=2;
      }
    }
  }else{//nivel impar
    if(padre(i) >= 0 && nodos[padre(i)] > nodos[i]){//Tenemos que reordenar
    ↪  respecto a los niveles pares
      Intercambio(nodos[i],nodos[padre(i)]);
        i = padre(i);
        profundidad--;
        abuelo = padre(padre(i));
        while(abuelo >= 0 && nodos[abuelo] > nodos[i]){
          Intercambio(nodos[i],nodos[abuelo]);
          i = abuelo;
          abuelo = padre(padre(i));
          profundidad-=2;
        }
```

```
    }
    else{//Comprobamos si tenemos que reordenar respecto a los niveles pares del
    ↪  APO.
      abuelo = padre(padre(i));
      while(abuelo >= 0 && nodos[abuelo] < nodos[i]){//Reordenamos respecto
      ↪  niveles pares
        Intercambio(nodos[abuelo], nodos[i]);
        i = abuelo;
        abuelo = padre(padre(i));
        profundidad-=2;
      }
    }
  }
}
```

Ejercicio 3: *Implementa una operación de orden logarítmico para eliminar el elemento máximo de un montículo min-max definido como en el problema anterior.*

```
template <typename T> void APOmM::suprimirMax(){
  assert(ultimo > -1);
  if(--ultimo > -1){//APO no queda vacío
    if(ultimo >= 2){//Tiene dos hijos y al menos 1 nieto, el máximo está en uno
    ↪  de los 2 hijos.
      if(nodos[1] > nodos[2]){
        nodos[1] = nodos[ultimo+1];
        hundir(1);
      }else{
        nodos[2] = nodo[ultimo+1];
        hundir(2);
      }
    }else if(ultimo == 1){
      nodos[1] == std::min(nodos[1],nodos[2])M
    }
  }
}
```

Ejercicio 4: *Un árbol es estrictamente ternario si todos sus nodos son hojas o tienen tres hijos. Escribe una función que, dado un árbol de grado arbitrario, nos indique si es o no estrictamente ternario.*

Vamos a partir de un Agen, ya que son los únicos que pueden tener más de dos hijos (un hijo izquierdo y los hermanos de este).

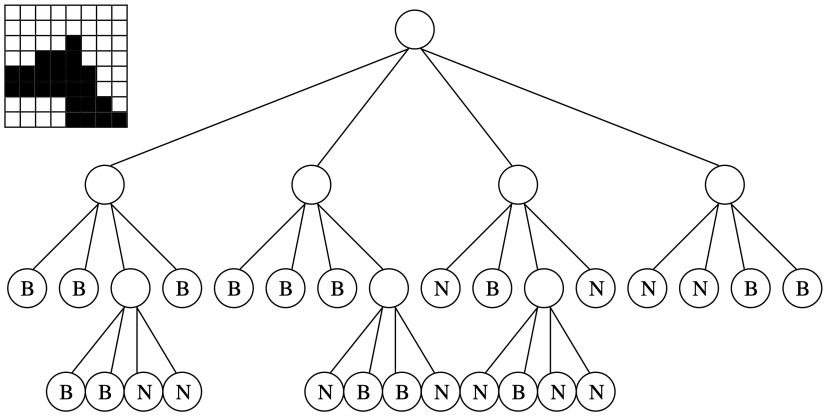
```
template <typename T> bool esTernario(const Agen<T> &A){
  return esTernario_rec(A.raiz(), A);
}

template <typename T> bool esTernario_rec(typename Agen<T>::nodo n, const Agen<T>
↪  &A){
  //n es un nodo nulo
  if(n == Agen<T>::NODO_NULO)
    return true;
  //n no tiene hijos.
```

```
  else if(A.hijoIzqdo(n) == Agen<T>::NODO_NULO && A.hermDrcho(n)==
  ↪  Agen<T>::NODO_NULO)
    return true;
  //n tiene hijos
  else if(A.hijoIzqdo(n) != Agen<T>::NODO_NULO && A.hermDrcho(n)!=
  ↪  Agen<T>::NODO_NULO){
    //llamamos a la función para sus hijos
    esTernario_rec(A.hijoIzqdo(n),A) && esTernario_rec(A.hermDrcho(n),A);
  }
  else //Tiene 1 hijo.
    return false;
}
```

Ejercicio 5: *Una forma de representar una figura plana en blanco y negro consiste en utilizar un árbol cuaternario en el que cada nodo o tiene exactamente cuatro hijos, o bien es una hoja. Un nodo hoja puede ser blanco o negro y un nodo interno no tiene color.*

Una figura dibujada dentro de un cuadrado de lado 2k se representa de la forma siguiente: Se divide el cuadrado en cuatro cuadrantes y cada uno se representa como un hijo del nodo raíz. Si un cuadrante está completamente negro corresponde a una hoja negra; si, por el contrario, el cuadrante está completamente blanco, éste corresponde a una hoja blanca; y si un cuadrante está parcialmente ocupado por negro y blanco, entonces corresponde a un nodo interno del árbol y este cuadrante se representa siguiendo el mismo método subdividiéndolo en otros cuatro cuadrantes. Como ejemplo se muestra una figura en blanco y negro y su árbol asociado, tomando los cuadrantes en el sentido de las agujas del reloj a partir del cuadrante superior izquierdo.



Implementa una función que dado un árbol de esta clase, con k+1 niveles, devuelva la figura asociada, representada como una matriz cuadrada de tamaño 2k en la que cada celda representa un punto blanco o negro.

Nota: Por simplificar el problema, se asume que en cada nodo del árbol se incluyen las coordenadas de la esquina superior izquierda y de la esquina inferior derecha del cuadrante que representa.

Partimos de la base que tenemos un árbol cuaternario (4 hijos por cada nodo) y sabemos que los nodos intermedios están vacios (no tienen un color asociado). El primer hijo del árbol, es decir, el hijo izquierdo del nodo raíz contempla los primeros cuatro cuadrados de la matriz (arriba a la izquierda), siendo estos todos "blancos"porque no tienen un color asociado.

Sabemos la dimensiones de la matriz que sería $2k$, es decir $2 * (NumNodos - 1)$, en este caso, como tenemos 33 nodos en el árbol, tenemos $2 * (33 - 1) = 64$ casillas, es decir una matriz de tamaño 8x8 = 64.

Como cada casilla tiene un color asociado o no, podemos creanos una estructura llamada "pixel"la cual contendrá la información de cada casilla junto con su çoordenada".

```
//Declaramos los tipos de datos a usar
typedef struct Pixel{
  std::string color;
  //Nos movemos en el sentidos de las agujas del reloj
  size_t x_[2]; //  (0,0) --> (0,1) |
  size_t y_[2]; //  (0,1) <-- (1,1) v
};
//Devolvemos la matriz por referencia
template <typename Pixel>
void Figura(typename Agen<Pixel>::nodo n, const Agen<Pixel> &A,
↪  vector<vector<Pixel>> &M){
  if(n != Agen<Pixel>::NODO_NULO){
    // Comprobamos si es un nodo intermedio
    if(A.elemento(n) == ""){
      Vamos a recorrer a sus hijos
      typename Agen<Pixel>::nodo hijo = A.hijoIzqdo(n);
      while(hijo != Agen<Pixel>::NODO_NULO){
        //Llamamos a la función con los hijos
        Figura(hijo,A,M);
        hijo = A.hermDrcho(hijo);
      }
    }
    else{
      if(A.elemento(n).color == "Blanco" || A.elemento(n).color == "Negro"){
      //Vamos ir insertando en la matriz los colores
        for(auto i = A.elemento(n).x_[0]; i <= A.elemento(n).x_[1]; i++)
          for(auto j = A.elemento(n).y_[1]; j <= A.elemento(n).y_[0]; j++)
            M[i][j] = A.elemento(n).color;
      }
    }
    //Si es otro color, no hace nada
  }
    //Si es nodo nulo, no se hace nada
}
```

Práctica Extra: Repaso árboles Binarios

Esta es una práctica de repaso de árboles binarios, es decir, de las prácticas 1 y 2.

Ejercicio 1: *Considere un árbol A, en el que puede suponer que no hay elementos repetidos. Dado un elemento x, devuelve el camino que existe entre la raíz y el nodo cuyo elemento es x, si existe.*

```
template <typename T> std::vector<T> camino(Abin<T> &A, const T& e){
  //Nos creamos el vector
  std::vector<T> camino_;
  if(!A.arbolVacio()){
    if(buscar_elemento(A.raiz(),A,e,camino_))
      std::reverse(camino_.begin(), camino_.end());
  }
    return camino_;
}
template <typename T> bool buscar_elemento(typename Abin<T>::nodo n, Abin<T>
↪  &A,const T& e, vector<T> &v){
  if(n != Abin<T>::NODO_NULO){
    if(A.elemento(n) == e){ //elemento encontrado, metemos en el vector, todos
    ↪  los nodos que lo preceden
      while(n != A.raiz()){
        v.push_back(A.elemento(n));
        n = A.padre(n);
      }
      v.push_back(A.elemento(A.raiz()));
      return true; //devolvemos true cuando hemos terminado de rellenar el camino
    }
    else{
      //Seguimos buscando
      return buscar_elemento(A.hijoIzqdo(n),A,e,v) &&
       ↪  buscar_elemento(A.hijoDrcho(n),A,e,v);
    }
  }
  return false; //no existe el elemento en el árbol
}
```

Ejercicio 2: *Implementa una función que, dado un árbol , devuelva el número de nodos prósperos que existen en él. Se considera que un nodo es próspero si es estrictamente más rico que sus ascendientes, pero menos rico que sus descendientes.*

```
template <typename T> size_t esProspero(const Abin<T> &A){
  return esProspero_rec(A.raiz(),A);
}

//Creamos un método auxiliar que compruebe si es más rico que todos sus
↪  ascendientes
template <typename T> bool MasRico(typename Abin<T>::nodo n, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO && A.padre(n) == Abin<T>::NODO_NULO) return false;
  ↪  //No existe el nodo en el árbol, ni su padre
```

```
  else{
    if(A.elemento(n) > A.elemento(A.padre(n)))
      return MasRico(A.padre(n),A);
    else return false;
  }
}

//Creamos un método auxiliar que compruebe si es menos rico que todos sus
↪  descendientes
template <typename T> bool MenosRico(typename Abin<T>::nodo n, const Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return 0; //nodo no existe
  else{
    if(A.elemento(n) < A.elemento(A.hijoIzqdo(n)) && A.elemento(n) <
    ↪  A.elemento(A.hijoDrcho(n)))
      return MenosRico(A.hijoIzqdo(n),A) && MenosRico(A.hijoDrcho(n),A);
    else
      return false;
  }
}

template <typename T> size_t esProspero_rec(typename Abin<T>::nodo n, const
↪  Abin<T> &A){
  if(n == Abin<T>::NODO_NULO) return 0;
  else{
    if(MasRico(n,A) && MenosRico(n,A))
      return 1 + esProspero_rec(A.hijoIzqdo(n),A) +
      ↪  esProspero_rec(A.hijoDrcho(n),A);
    else
      return 0 + esProspero_rec(A.hijoIzqdo(n),A) +
      ↪  esProspero_rec(A.hijoDrcho(n),A);
  }
}
```

Ejercico 3: *Implemente una función que, dado un árbol A y un elemento T, realice la poda del subárbol cuya raíz tiene por elemento T. Puede suponer que no hay elementos repetidos en el árbol.*

Para poder realizar la poda de un subárbol dado un elemento (no el nodo) tenemos que hacer:

1. Buscar el nodo que contenga ese elemento.
2. Recorrer los subárboles hasta llegar a los nodos hojas.
3. Eliminar los nodos hojas e ir eliminando sus ascendientes hasta el nodo con el elemento buscado.
4. Eliminar el nodo con el elemento buscado llamando al padre.

```
//Creamos los métodos para realizar la búsqueda del nodo
template <typename T> Abin<T>::nodo buscar_elto(const T& elto,const Abin<T> &A){
  //creamos el nodo donde se encuentra el elemento
  typename Abin<T>::nodo resultado = Abin<T>::NODO_NULO; //Lo inicializamos
  if(!A.arbolVacio())
    buscar_elto_rec(elto,A.raiz(),resultado,A);
```

```
  return resultado; //contiene el nodo donde se encuentra el elemento, si no
  ↪ existe el elemento, devuelve un nodo nulo.
}
template <typename T> bool buscar_elto_rec(const T& elto, typename Abin<T>::nodo
↪ n,typename Abin<T>::nodo &res, const Abin<T> &A){
  if(n != Abin<T>::NODO_NULO){
    if(A.elemento(n) == elto){
      res = n;
      return true;
    }
    return buscar_elto_rec(elto,A.hijoIzqdo(n),res,A) ||
    ↪ buscar_elto_rec(elto,A.hijoDrcho(n),res,A);
  }
  return false;
}

//Función auxiliar que nos comprueba si el nodo es una hoja o No
template <typename T> bool EsHoja(typename Abin<T>::nodo n , const Abin<T> &A){
  return (A.hijoIzqdo(n) == Abin<T>::NODO_NULO && A.hijoDrcho(n) ==
  ↪ Abin<T>::NODO_NULO);
}

//Vamos a realizar la función de poda
template <typename T> void poda(const T& elto, Abin<T> &A){
  if(!A.arbolVacio()){
    //Tenemos que buscar el nodo donde se encuentra dicho elemento
    typename Abin<T>::nodo n = buscar_elto(elto,A);
    if(n != Abin<T>::NODO_NULO){
      poda_rec(n,A);
      //Ahora tengo que eliminar a dicho nodo, llamando al padre
      typename Abin<T>::nodo padre = A.padre(n);
      if(padre != Abin<T>::NODO_NULO){
        if(n == A.hijoIzqdo(padre))
          A.eliminarHijoIzqdo(padre);
        else if(n == A.hijoDrcho(padre))
          A.eliminarHijoDrcho(padre);
      }else if(n == A.raiz() && EsHoja(n))
        A.eliminarRaiz();
    }
  }
}

template <typename T> void poda_rec(typename Abin<T>::nodo n, Abin<T> &A){
  if(n != Abin<T>::NODO_NULO){
    if(!EsHoja(n)){ //Si no es hoja no lo podemos eliminar, llamamos a sus
    ↪ descendientes
      poda_rec(A.hijoIzqdo(n),A);
      poda_rec(A.hijoDrcho(n),A);
    }
    else{ //Es hoja, eliminamos llamando al padre
```

```
      typename Abin<T>::nodo padre = A.padre(n);
      if(padre != Abin<T>::NODO_NULO){
        if(n == A.hijoIzqdo(padre))
          A.eliminarHijoIzqdo(padre);
        else if(n == A.hijoDrcho(padre))
          A.eliminarHijoDrcho(padre);
      }
    }
  }
}
```

2. Prácticas Grafos

En este capítulo vamos a realizar las prácticas correspondientes a los diferentes algoritmos de los Grafos (Dijkstra, Floyd, Kruskal, Prim..), con el fin de aprender cuando se usan cada uno y porqué.

Práctica 6: Problemas de Grafos I

Esta es la primera práctica de los grafos, donde realizaremos ejercicios con los algoritmos Dijkstra y Floyd.

Como cabeceras tendremos:

```
#include <iostream>
#include <vector>
#include "../grafoPMC.h"
#include "../alg_grafoPMC.h"
```

Ejercicio 1: *Añadir una función genérica, llamada DijkstraInv, en el fichero alg_grafoPMC.h para resolver el problema inverso al de Dijkstra, con los mismos tipos de parámetros y de resultado que la función ya incluida para éste. La nueva función, por tanto, debe hallar el camino de coste mínimo hasta un destino desde cada vértice del grafo y su correspondiente coste.*

Suponemos que estamos dentro del fichero `alg_grafoPMC.h` para poder incluir dicha función genérica para resolver el problema de Dijkstra pero de forma inversa.

Como sabemos Dijkstra funciona de la manera que nosotros le damos como parámetros de entrada el grafo, el vértice origen y el vector de vértices por los que pasa, donde desde el origen calcula todos los caminos de coste mínimos a todos los vértices del grafo.

Ahora esto es diferente ya que al pasarle el vértice destino, DijkstraInv nos calculará todos los caminos de coste mínimo desde cualquier vértice del grafo al vértice destino.

Por tanto, la implementación de DijkstraInv quedaría de la siguiente forma:

```
template <typename tCoste> vector<tCoste> DijkstraInv(const GrafoP<tCoste>& G,
↪  typename GrafoP<tCoste>::vertice destino, vector<typename
↪  GrafoP<tCoste>::vertice> &P){
  typedef GrafoP<tCoste>::vertice vertice; //llamamos vertice a los vértices
  ↪  (para no tener que escribir todo).
  vertice v,w; //vertices del grafo
  const size_t n = G.numVert();
  vector<bool>S(n,false); //conjuto de vértices (vacío).
  vector<tCoste> D(n); //vector de coste minimo hasta destino.
  //Iniciamos D y P con los caminos directos hasta el destino.
  for(size_t i = 0; i < n; i++)
    D[i] = G[i][destino];
  D[destino]=0;
  P = vector<vertice>(n,destino);
  //calculamos los caminos de costes mínimos.
  S[destino] = true; //incluimos en S el vértice destino
  for(size_t i = 1; i <= n-2; i++){
    //vamos a coger el vértice "w" no incluido en S con menor coste desde el
    ↪  destino.
    tCoste CosteMin = GrafoP<tCoste>::INFINITO;
    for(v = 0; v < n; v++){
      if(!S[v] && D[v] <= CosteMin)
        CosteMin = D[v];
        w = v;
```

```
      }
      S[w]=true; //incluimos vertice w en S
    for(v = 0; v < n; v++)
      if(!S[v]){
        tCoste vwd = suma(G[v][w],D[w]); //el coste de ir desde v a destino
        ↪ pasando por w.
        if(vwd < D[v]){ //si el coste es menor, se guarda el coste y se le asigna
        ↪ el vértice
          D[v] = vwd;
          P[v] = w;
      }
    }
  }
  return D; //devolvemos el camino de costes mínimo.
}

//Función calcular el camino pero de manera inversa.
//Esta función devuelve el camino de coste mínimo entre los vértices v y destino,
↪ a partir de un vector P (obtenido en la función anterior DijkstraInv).

template <typename tCoste> typename GrafoP<tCoste>::tCamino camino_inv(typename
↪ GrafoP<tCoste>::vertice destino, typename GrafoP<tCoste>::vertice v, const
↪ vector<typename GrafoP<tCoste>::vertice> &P){
//Nos creamos un camino nuevo
  typename GrafoP<tCoste>::tCamino C;
  C.insertar(v,C.fin());
  do{
    C.insertar(P[v],C.fin());
    v = P[v];
  }while(v != destino);
  return C;
}
```

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.

Sabemos que partimos de un grafo conexo (todos sus vértices están unidos con aristas con otros vértices). Para calcular el diametro de un grafo, es decir, la suma de las distancias mínimas a los dos nodos más alejados del grafo haremos uso del algoritmo de Floyd.

Recibimos como parámetros el Grafo G en cuestión y un vértice que será nuestro centro.

Finalmente, este subprograma devolverá el valor del diámetro del grafo G.

```
template <typename tCoste> tCoste diametro(const GrafoP<tCoste> &G){
  //renombramos los vértices y el infinito.
  typename GrafoP<tCoste>::vertice vertice;
  typename GrafoP<tCoste>::INFINITO infinito;
```

```
  //creamos una matriz de vértices y dos vértices cualesquiera
  matriz<vertice> M; //parámetro de Floyd
  vertice a, b;
  //Realizamos Floyd
  matriz<tCoste>costesminimo = Floyd(G,M);

  tCoste primer_mayor = std::numeric_limits<tCoste>::min(), diametro = infinito,
  ↪  segundo_mayor = std::numeric_limits<tCoste>::min();
  size_t n = costesminimo.dimension();

  for(auto i = 0; i < n-1; i++){
    primer_mayor = segundo_mayor = std::numeric_limits<tCoste>::min();
    for(auto j = 0; j <= n; j++){
      //Vamoa a obtener los dos vértices con mayor distancia.
      if(costesminimo[i][j] > primer_mayor){
        segundo_mayor = primer_mayor;
        b = a;
        primer_mayor = costesminimo[i][j];
        a = j;
      }
      else if(costesminimo[i][j] > segundo_mayor){
        segundo_mayor = costesminimo[i][j];
        b = j;
      }
    }
  }
  //ahora nos vamos a quedar con las sumas más pequeñas debido a que estamos
  ↪  calculando las distancias mínimas
  if(suma(primer_mayor,segundo_mayor) < diametro){
    diametro = suma(primer_mayor, segundo_mayor);
  }
  return diametro;
}
```

Ejercicio 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.

Como queremos saber si vamos a recibir o no la subvención este subprograma va a devolver un booleano.

Además tendremos que comprobar si el grafo tiene ciclos o no, lo cual es una condición necesaria para saber si recibimos o no la subvención.

```
//Primero vamos a crear la función que nos cumpruebe si hay ciclo o no.
template <typename tCoste> bool hayCiclo(const matriz<GrafoP<tCoste>::vertice>
↪  &M, GrafoP<tCoste>::vertice v, std::vector<bool> &visitados){
```

```
  //Vamos a trabajar con un vector de booleano "visitados", donde si hemos
  ↪  visitados dicho vértice previamente, encontramos que hay un ciclo.
  if(visitados[v])
    return true; //si está visitado, hay ciclo
  else{
    visitados[v] = true; //se visita
    for(auto i = 0; i < M.dimension(); i++){ //si hay camino, se va al siguiente
    ↪  vértice
      if(M[v][i] != GrafoP<tCoste>::INFINITO){
        return hayCiclo(M,i,visitados);
      }
    }
  }
  return false; //devuelve falso, si no hay ciclo.
}

//Ahora vamos a implementar el subprograma que mediante un valor booleano nos
↪  indicará si recibimos o no la subvención, teniendo en cuenta la función
↪  anterior "hayCiclo".

template <typename tCoste> bool recibe_subvencion(const GrafoP<tCoste> &G){
  //Lo primero que vamos a hacer es crearnos una matriz de costes y de vértices
  ↪  para hacer uso de Floyd.
  matriz<tCoste> CostesMinimos;
  matriz<GrafoP<tCoste>::vertice>V;
  CostesMinimos = Floyd(G,V);
  bool subvencion = true; //recibe subvención
  //Vamos a ver si hay ciclos en el grafo, para ello recorremos la matriz V
  ↪  haciendo uso de la función anterior.
  for(auto i = 0; i < V.dimension(); i++){
    std::vector<bool> visitados(V.dimension(),false);
    if(hayCiclo(V,i,visitados))
      subvencion = false; //no recibe subvención
  }
  return subvencion;
}
```

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:*

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

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.

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).

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.

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.

Si leemos el enunciado vemos que contamos con un Grafo ponderado y dirigido, un vector de ciudades que han sido tomadas por los rebeldes, una ciudad sería la capital de Zuelandia debido a que todos los viajes tienen que pasar si o si por la capital.

Suponemos que cada vértice del grafo es una ciudad y por ende, un par de ciudades da como resultado una carretera (que sería una arista del grafo), además vamos a tener un vector de las carreteras que están cortadas.

Sabiendo todo esto, vamos a realizar el ejercicio haciendo uso de Floyd, debido a que no nos dicen ningún origen explicitamente, si no que todos los viajes tengan que pasar por la capital de Zuelandia.

Suponemos que nuestro tCoste será cualquier tipo de dato, en mi caso será size_t.

```
//vamos a declarar los tipos datos necesarios para el ejericio, como son las
↪  ciudades, carreteras..
typedef GrafoP<size_t>::vertice ciudad;
typedef std::pair<ciudad,ciudad> carretera;
typedef std::vector<ciudad> ciudades_rebeldes;
typedef std::pair<ciudad,ciudad> carreteras_cortadas;
matriz<size_t> Zuelandia(GrafoP<size_t> &Z, ciudades_rebeldes cr,
↪  carreteras_cortadas cc, ciudad capitalZuelandia){
  //Renombramos el tipo de dato infinito
  const size_t INFINITO = GrafoP<size_t>::INFINITO;
  size_t num_ciudades = Z.numVert();

  // Aplicar las carreteras cortadas
  for (const auto& c : cc) {
      Z[c.first][c.second] = INFINITO;
  }

  // Aplicar las ciudades rebeldes
```

```
  for (const auto& ciudadrebelde : cr) {
      for (size_t i = 0; i < num_ciudades; ++i) {
          Z[ciudadrebelde][i] = INFINITO;
          Z[i][ciudadrebelde] = INFINITO;
      }
  }

  // Asegurar que todos los caminos pasen por la capital
  for (size_t i = 0; i < num_ciudades; ++i) {
      if (i != capitalZuelandia) {
          for (size_t j = 0; j < num_ciudades; ++j) {
              if (j != capitalZuelandia) {
                  Z[i][j] = INFINITO;
              }
          }
      }
  }

  // Calcular la matriz de costes mínimos usando Floyd
  matriz<size_t> CostesMinimos;
  matriz<ciudad> M;
  CostesMinimos = Floyd(Z, M);

  return CostesMinimos;
}
```

Ejercicio 5: *Escribir una función genérica que implemente el algoritmo de Dijkstra usando un grafo ponderado representado mediante listas de adyacencia.*

Vamos a calcular los caminos de coste mínimo entre el origen y todos los vértice del grafo (Dijkstra) pero ahora partiendo desde una lista de adyacencia que indica si dos vértices de un grafo están adyacentes (."l lado"), esta contiene tanto la lista de los vértices adyacentes al vértice v, como el peso de la arista que unen a dichos vértices.

```
template <typename tCoste> vector<tCoste> DijkstraLista(const GrafoP<tCoste> &G,
↪  typename GrafoP::vertice origen, vector<typename GrafoP<tCoste>::vertice>
↪  &P){
  //Alias de los tipos de datos usados
  typedef GrafoP<tCoste>::vertice vertice;
  typedef GrafoP<tCoste>::INFINITO INFINITO;
  vertice v,w; //Declaramos dos vértices

  const size_t n = G.numVert();
  vector<bool> S(n,false);  //conjunto de vértices vacío.
  vector<tCoste> D(n,INFINITO); //vector de costes mínimos desde origen

  //Iniciamos los vectores D y P con los caminos directos desde el origen
  for(const auto& arista : G.adyacentes(origen)){
    vertice i = arista.v;
    D[i] = arista.c;
  }
  D[origen] = 0; //el coste de ir origen-origen es 0
```

```
  P = std::vector<vertice>(n,origen);
  //Calculamos los caminos de coste mínimo desde origen hasta cada vértice del
  ↪  grafo
  S[origen]=true; //incluimos el origen en el conjunto de vértices.
  for(size_t i = 0; i <= n-2; i++){
    //Cogemos el vértice w no incluido en S con menor coste
    tCoste CosteMin = INFINITO;
    for(v = 0; v < n; v++){
      if(!S[v] && D[v] < INFINITO){
        CosteMin = D[v];
        w = v;
      }
    }
    S[w] = true; //Incluimos ese vértice en el conjunto
    // Recalcular coste hasta cada v no incluido en S a través de w
    for(const auto& arista : G.adyacentes(w)){
      vertice u = arista.v;
      tCoste peso = arista.c;
      if(!S[u] && D[w] + peso < D[u]){
        D[u] = D[w] + peso;
        P[u] = w;
      }
    }
  }
  return D;
}
```

Práctica 7: Problemas de Grafos II

Esta práctica es una continuación de la anterior.

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.

Partimos de que tenemos un grafo acíclico, la matriz de costes directos entre los vértices del mismo y queremos obtener los caminos con mayor coste entre las diferentes ciudades (vértices) del mismo, además no nos importa ni el origen ni destino del viaje, por lo que ya sabemos que algoritmo vamos a usar, en efecto Floyd.

Pero no vamos a hacer uso de un Floyd normal y corriente, debido a que este nos devuelve la matriz con los costes mínimos de todos los caminos posibles del grafo, ahora tenemos que devolver los caminos con mayor coste, por tanto, vamos a realizar una implementación de Floyd máximo para poder obtener el resultado esperado.

```
template <typename tCoste> tCoste MaxInteligente(tCoste a, tCoste b){
  if(a == GrafoP<tCoste>::INFINITO){
    return b;
  }
  if(b == GrafoP<tCoste>::INFINITO){
    return a;
  }
  return std::max(a,b);
}
template <typename tCoste> matriz<tCoste> Floyd_max(const GrafoP<tCoste> &G,
↪  matriz<typename GrafoP<tCoste>::vertice> &P){
  //Calculamos los caminos de coste máximo entre cada par de vértices del grafo G.
  typename GrafoP<tCoste>::vertice vertice;
  typename GrafoP<tCoste>::INFINITO INFINITO;
  const size_t n = G.numVert();
  matriz<vertice> A(n); //creamos la matriz con los costes máximos.
  //Inicializamos P y A.
  P = matriz<vertice>(n);
  for(size_t i = 0; i < n; i++){
    A[i] = G[i]; //Copiamos el coste del grafo
    A[i][i] = 0; //Diagonal principal a 0
    P[i] = vector<vertice>(n,i); //camino directos
  }
  //Calculamos los costes máximos y los caminos correspondientes
  for(vertice k = 0; k < n; k++)
    for(vertice i = 0; i < n, i++)
      for(vertice j = 0; j < n; j++){
        tCoste ikj = MaxInteligente(A[i][k],A[k][j]); //el coste de ir de i a j
        ↪  pasando por k.
```

```
        if(ikj > A[i][j] && ikj != INFINITO){
          A[i][j] = ikj;
          P[i][j] = k;
        }
      }
  return A; //devolvemos la matriz de costes máximos.
}

//Ya tenemos hecho el algoritmo Floyd_max, ahora vamos a calcular lo que nos pide
↪  el enunciado:
//Como tenemos que devolver el coste del camino entre ese origen-destino y ambas
↪  ciudades.
template <typename tCoste> tCoste viaje(const GrafoP<tCoste> &G, typename
↪  GrafoP<tCoste>::vertice &O, typename GrafoP<tCoste>::vertice &D){
  //Nos creamos la matriz para almacenar los costes máximos
  typename GrafoP<tCoste>::vertice vertice;
  matriz<vertice>P;
  matriz<tCoste>CosteMax;
  CosteMax = Floyd_max(G,P);
  typename GrafoP<tCoste>::INFINITO INFINITO;

  //nos creamos una variable que almacenará el coste del viaje del origen a
  ↪  destino
  tCoste viaje_max = std::numeric_limits<tCoste>::min();
  for(vertice i = 0; i < G.numVert(); i++)
    for(vertice j = 0; j <G.numVert(); j++){
      if(viaje_max < G[i][j] && G[i][j] != INFINITO){
        viaje_max = G[i][j]; //almacenamos el coste del viaje más costoso
        O = i; //almacenamos la ciudad origen del viaje
        D = j; //almacenamos la ciudad destino del viaje
      }
    }
  return viaje_max;
}
```

Ejercicio 2: *Se dispone de un laberinto de NxN 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),*
- *la lista de paredes del laberinto,*
- *la casilla de entrada, y*
- *la casilla de salida,*

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

Contamos con un laberinto de dos dimensiones NxN, donde conocemos la casilla de entrada

y la de salida. Ahora vemos que no tenemos un grafo de inicio, por tanto, a la función no podemos pasarle el grafo, si no que le pasamos la dimensión del laberinto y la casillas de salida y entrada.

Ahora encontramos un problema y es que nos piden que devolvamos el camino más corto de ir desde un origen a un destino, esto lo haremos mediante el uso de Dijkstra pero este algoritmo trabaja con grafos no con laberintos, por tanto, lo que vamos a hacer es convertir ese laberinto en un grafo, haciendo que esas casillas pasen a ser vértices del grafo, pero tenemos que buscar una función que eso se cumpla.

Como tenemos un laberinto de dimensión NxN, vamos a tener un grafo de N^2 nodos y una matriz de coste de N^2xN^2 posiciones.

Vamos a declarar los tipos de datos con los que vamos a trabajar:

Las casillas pues será una coordenada dentro del laberinto, es decir un par de dos enteros (por ejemplo) → `typedef std::pair<int,int> Casilla;` ó una estructura `struct Casilla{int fila_; int columna_;};`

Un laberinto tiene paredes, por lo que al tener casillas, una pared será un par de dos casillas → `std::pair<Casilla,Casilla> Pared;`, estas paredes estarán en aquellas casillas que sean adyacentes, por lo que necesitamos un método que dadas dos casillas nos indique si son o no adyacentes.

A la hora de trabjar con el grafo para poder calcular el camino más corto entre la salida y la entrada, tenemos que convertir esas casillas a vértices del mismo, por tanto haremos uso de una función auxiliar llamada `casillaAvertice();`.

```
//Declaramos los tipos específicos del laberinto:
typedef std::pair<int,int> Casilla;
typedef std::pair<Casilla,Casilla> Pared;
typedef std::vector<Pared> paredes;
typedef GrafoP<size_t>::vertice vertice;

//Método que convierte un vértice en una casilla
Casilla verticeAcasilla(vertice v, int N){
  Casilla C;
  C.first = v/N;
  C.second = v%N;
  return C;
}

//Método que comprueba la adyacencia de casillas
bool adyacentes(Casilla a, Casilla b){
  return (std::abs(a.first - b.first) + std::abs(a.second - b.second()) ==1);
}

void rellena_adyacentes(GrafoP<size_t> &G, int N){
  for(vertice v = 0; v < G.numVert() ; v++)
    for(vertice w = 0; w < G.numVert(); w++){
      if(v == w)
        G[v][w] = 0;
      else if(adyacentes(verticeAcasilla(v,N)), verticeAcasilla(w,N))
         G[v][w] = 1;
```

```
    }
}

//Método que nos convierte de una casilla a un vértice
vertice casillaAvertice(Casilla C, size_t N){
  return C.first*N + C.second;
}

//Método que construye las paredes
void construye_paredes(GrafoP<size_t> &L, const paredes &p, int N){
  for(auto &pared : p){
    vertice v = casillaAvertice(p.first,N);
    vertice w = casillaAvertice(p.second,N);
    L[v][w] = L [w][v] = GrafoP<size_t>::INFINITO;
  }
}

//Método que nos calcula el camino más corto del Laberinto junto con su costes
std::pair<vector<vertice>,size_t> laberinto(int N, const paredes &p, Casilla
↪  inicio, Casilla fin){
  //Creamos el grafo con dimensión NxN
  size_t D = N*N:
  GrafoP<size_t> Laberinto(D);
  //convertimos las casillas de inico y fin a vértices del grafo
  vertice ini_ = casillaAvertice(inicio,N),
          fin_ = casillaAvertice(fin,N);

  //Rellenamos el grafo con los vértices adyacentes
  rellena_adyacentes(Laberinto,D);
  //Construimos paredes
  construye_paredes(Laberinto,p,dimension);

  //Creamos el vector de vértice y de coste mínimo para calcular el camino mínimo
  ↪  entre la casilla de inicio y fin del Laberinto
  vector<vertice>P;
  vector<size_t>costemin = Dijkstra(Laberinto,ini_,P);
  return std::make_pair(P,costemin[fin_]);
}
```

Ejercicio 3: *Eres el orgulloso dueño de una empresa de distribución. Tu misión radica en distribuir todo tu stock entre las diferentes ciudades en las que tu empresa dispone de almacén.*

Tienes un grafo representado mediante la matriz de costes, en el que aparece el coste (por unidad de producto) de transportar los productos entre las diferentes ciudades del grafo.

Pero además resulta que los Ayuntamientos de las diferentes ciudades en las que tienes almacén están muy interesados en que almacenes tus productos en ellas, por lo que están dispuestos a subvencionarte con un porcentaje de los gastos mínimos de transporte hasta la ciudad. Para facilitar el problema, consideraremos despreciables los costes de volver el camión a su base (centro de producción).

He aquí tu problema. Dispones de

- *el centro de producción, nodo origen en el que tienes tu producto (no tiene almacén),*
- *una cantidad de unidades de producto (cantidad),*
- *la matriz de costes del grafo de distribución con N ciudades,*
- *la capacidad de almacenamiento de cada una de ellas,*
- *el porcentaje de subvención (sobre los gastos mínimos) que te ofrece cada Ayuntamiento.*

Las diferentes ciudades (almacenes) pueden tener distinta capacidad, y además la capacidad total puede ser superior a la cantidad disponible de producto, por lo que debes decidir cuántas unidades de producto almacenas en cada una de las ciudades. Debes tener en cuenta además las subvenciones que recibirás de los diferentes Ayuntamientos, las cuales pueden ser distintas en cada uno y estarán entre el 0 % y el 100 % de los costes mínimos.
La solución del problema debe incluir las cantidades a almacenar en cada ciudad bajo estas condiciones y el coste mínimo total de la operación de distribución para tu empresa.

Partimos de que tenemos un grafo (matriz de costes), tenemos una ciudad origen (centro de producción), la cantidad de producto y el porcentaje de subvneción que ofrece cada ayuntamiento.

Como tenemos un origen (indicado explícitamente) haremos uso de Dijkstra, no hacemos DijkstraInv ya que nos dice que el coste de volver es despreciable por tanto, asumimos que es 0. Luego de hacer Dijkstra (coste mínimo bruto), aplicaremos el porcentaje de subvención. Finalmente, cogemos el que menor coste tenga y rellenamos la matriz de costes de más barato a más caro. No buscamos el más barato directamente, ni el que mayor subvención recibe, si no el que más barato le salga al camón después de realizar el cálculo de los costes mínimos ya aplicada la subvención (coste mínimo neto).

```
% //Definimos los tipos
% typedef GrafoP<size_t>::vertice vertice;
% typedef std::vector<size_t> cantidades;
% typedef std::vector<size_t> porcentajes;

% //Nos creamos un método el cual nos servirá para aplicar las subvenciones
% void subvencion(cantidades &v, porcentajes &subvencion){
%   for(size_t i = 0; i < v.size(); i++){
%     v[i] -= v[i] * (subvenciones[i]/100);

%   }
% }
```

Ejercicio 4: *Eres el orgulloso dueño de la empresa "Cementos de Zuelandia S.A". Empresa dedicada a la fabricación y distribución de cemento, sita en la capital de Zuelandia. Para la distribución del cemento entre tus diferentes clientes (ciudades de Zuelandia) dispones de una flota de camiones y de una plantilla de conductores zuelandeses.*

El problema a resolver tiene que ver con el carácter del zuelandés. El zuelandés es una persona que se toma demasiadas "libertades" en su trabajo, de hecho, tienes fundadas sospechas de que tus conductores utilizan los camiones de la empresa para usos particulares (es decir indebidos, y a tu costa) por lo que quieres controlar los kilómetros que recorren tus camiones.

Todos los días se genera el parte de trabajo, en el que se incluyen el número de cargas de cemento (1 carga = 1 camión lleno de cemento) que debes enviar a cada cliente (cliente =

ciudad de Zuelandia). Es innecesario indicar que no todos los días hay que enviar cargas a todos los clientes, y además, puedes suponer razonablemente que tu flota de camiones es capaz de hacer el trabajo diario.

Para la resolución del problema quizá sea interesante recordar que Zuelandia es un país cuya especial orografía sólo permite que las carreteras tengan un sentido de circulación.

Implementa una función que dado el grafo con las distancias directas entre las diferentes ciudades zuelandesas, el parte de trabajo diario, y la capital de Zuelandia, devuelva la distancia total en kilómetros que deben recorrer tus camiones en el día, para que puedas descubrir si es cierto o no que usan tus camiones en actividades ajenas a la empresa.

Partimos de que tenemos un grafo con los costes directos, la cantidad de cemento que se reparte a cada ciudad, una ciudad origen (que es la capital) y nos piden devolver la distancia en kilómetros que deben de recorrer los camiones, es decir, el coste mínimo de un dia de trabajo.

Para ello vamos a hacer uso de Dijkstra para la ida de los camiones y DijkstraInv para la vuelta de los mismos, la suma de ambos costes serán los kilómetros que hace cada camión.

```
//Definimos los tipos que vamos a usar
typedef double km;
typedef GrafoP<km>::vertice vertice; //cada vertice = ciudad
typedef size_t carga;
typedef std::vector<cargar> vector_cargas; //número de carga de cada ciudad

//Vamos a realizar el cálculo de los kilómetros de cada camiones
km km_camiones(const GrafoP<km> &G, const vector_carga &cargas, vertice capital){
  //Vamos a crearnos el vector de vértices para hacer Dijkstra
  std::vector<vertice>P;
  std::vector<km> km_ida = Dijkstra(G,capital,P);
  std:vector<km> km_vuelta = DijkstraInv(G,capital,P);
  km total_km = 0;

  //Vamos a sumar el número de km para obtener cuantos hacen
  for(size_t i = 0; i < cargas.size(); i++){
    total_km += (ida[i] + vuelta[i]) * cargas[i];
  }
  return total_km;
}
```

Ejercicio 5: *Se dispone de tres grafos que representan la matriz de costes para viajes en un determinado país pero por diferentes medios de transporte, por supuesto todos los grafos tendrán el mismo número de nodos. El primer grafo representa los costes de ir por carretera, el segundo en tren y el tercero en avión. Dado un viajero que dispone de una determinada cantidad de dinero, que es alérgico a uno de los tres medios de transporte, y que sale de una ciudad determinada, implementar un subprograma que determine las ciudades a las que podría llegar nuestro infatigable viajero.*

Para empezar tenemos 3 grafos que representan 3 maneras diferentes de transportarse. Sabemos que tenemos una ciudad origen donde se empieza el viaje y que el viajero es alérgico a uno de los medios de transporte (a uno de los 3 grafos).

Sabiendo esto, tenemos que devolver las ciudades a las que puede viajar el viajero.

Haremos uso de Dijkstra para hacer este ejercicio, ya que contamos con una ciudad origen.

```
//Definimos los tipos de datos a usar
typedef GrafoP<size_t>::vertice vertice;

//Como tenemos 3 grafos (3 medios de transporte), podemos hacer un super grafo y
↪  rellenarlo, para poder trabajar más comodamente.
void rellena_supergrafo(const GrafoP<size_t> &trans1, const GrafoP<size_t>
↪  &trans2, const GrafoP<size_t> &supergrafo){
  for(vertice v = 0; v < trans1.numVert(); v++)
    for(vertice w = 0; w < trans2.numVert(); w++){
      supergrafo[v][w] = std::min(trans1[v][w],trans2[v][w]);
    }
}

//Como tenemos que devolver las ciudades a las que puede ir nuestro viajero
↪  alérgico vamos a crear un método que nos devuelva en un vector dichas
↪  ciudades.

std::vector<bool> ciudades_alcanzables(const GrafoP<size_t> &carretera, const
↪  GrafoP<size_t> &tren, const GrafoP<size_t> &avion, const string alergia,
↪  vertice origen, size_t presupuesto){
  //Nos creamos el supergrafo
  GrafoP<size_t> G(G.numVert());
  //Vamos a crearnos el vector que nos dirá a que ciudades podemos importa
  std::vector<bool> alcanzables(G.numVert(),false);
  //Nos creamos los vectores para hacer Dijkstra
  std::vector<vertice> P(G.numVert());
  std::vector<size_t> costes_minimos(G.numVert());

  //Ahora vamos a rellenar el supergrafo dependiendo de la alergia que tenga
  if(alergia == "carretera")
    rellena_supergrafo(tren,avion,G);
  else if(alergia == "tren")
    rellena_supergrafo(carretera,avion,G);
  else if(alergia == "avion")
    rellena_supergrafo(carretera,tren,G);
  //else si la string no es una de esas alergias no hace nada.

  //Calculamos los costes mínimo
  costes_minimos = Dijkstra(G,origen,P);

  //Ahora vamos a quedarnos con las ciudades a las que puede ir el viajero
  for(vertice i = 0; i < costes_minimos.size(); i++){
    if(costes_minimos[i] <= presupuesto)
      alcanzables[i] = true;
  }

  //Ahora ponemos la ciudad origen como alcanzable
  alcanzables[origen] = true;
```

```
  //devolvemos las ciudades a las que puede ir el viajero
  return alcanzables;
}
```

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.

Vemos que tenemos dos medios de transporte (tren y bus) para viajar entre distintas ciudades. Estos dos medios de transporte son dos grafos que tienen el mismo número de vértices. Los viajes los podemos realizar de varias maneras:

- Viaje entero en bus.
- Viaje entero en tren.
- Tren → ciudad transbordo → bus.
- Bus → ciudad transbordo → tren.

Encontramos 4 maneras diferentes de realizar un viaje debido a que no nos obligan cambiar de medio de transporte.

Para calcular la tarifa mínima, vamos a hacer uso de Floyd, ya que no tenemos un origen definido, si no una ciudad donde se puede hacer transbordo, por tanto haremos Floyd para cada medio de transporte.

```
//Definimos los tipos de datos a usar
typedef size_t coste;
typedef GrafoP<coste>::vertice vertice; //Equivale a una ciudad del Grafo

//Vamos a obtener la matriz de costes mínimos de realizar los viajes
matriz<coste> tarifa_minima(const GrafoP<coste> &Tren, const GrafoP<coste> &Bus,
↪  vertice ctransbordo){
  //Creamos el supergrafo
  matriz<vertice> M(Tren.numVert());
  //Realizamos Floyd para calcular los costes mínimos de cada grafo dado
  matriz<coste> matriz_minima_bus = Floyd(Bus,M);
  matriz<costa> matriz_minima_tren = Floyd(Tren, M);
  matriz<coste> tarifa(Tren.numVert());

  //Ahora vamos a rellenar la matriz de las tarifas minimas
  for(size_t i = 0; i < tarifa.dimension(); i++)
    for(size_t j = 0; j < tarifa.dimension(); j++){
      coste bus_tren = matriz_minima_bus[i][ctransbordo] +
      ↪  matriz_minima_tren[ctransbordo][j];
```

```
      coste tren_bus = matriz_minima_tren[i][ctransbordo] +
      ↪  matriz_minima_bus[ctransbordo][j];
      tarifa[i][j] =
      ↪  std::min(std::min(matriz_minima_tren[i][j],matriz_minima_bus[i][j]),std::min(matriz_m
      ↪  matriz_minima_tren[i][j]));
    }
    return tarifa;
}
```

Ejercicio 7: *Se dispone de dos grafos (matriz de costes) que representan los costes de viajar entre N ciudades españolas utilizando el tren (primer grafo) y el autobús (segundo grafo). Ambos grafos representan viajes entre las mismas N ciudades.*

Nuestro objetivo es hallar el camino de coste mínimo para viajar entre dos ciudades concretas 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.

Partimos de que tenemos 2 Grafos (matriz de coste) para los viajes en tren y en bus, donde ambos tienen el mismo número de vértices.

Ahora nos obligan a que el viaje se empieza en tren y termina en bus, por tanto, nos obligan a hacer un transbordo.

Es decir, nuestros viajes serían:

- Tren(origen) → transbordo(ciudad cambio1) → Bus(destino).
- Bus(destino) → transbordo(ciudad cambio2) → Tren(origen).

Como tenemos un origen, haremos uso de Dijkstra y como tenemos un destino haremos DijkstraInv.

Donde el coste mínimo de viajar desde la ciudad Origen a Destino es el mínimo de las dos opciones vistas anteriormente.

```
size_t ViajeTren_Bus(const GrafoP<size_t> &Tren, GrafoP<size_t> &Bus, size_t
↪  origen, size_t destino, size_t cambio1, size_t cambio2){
  //Creamos los vectores (vértices y coste minimo) para hacer uso de Dijkstra e
  ↪  DijkstraInv
  vector<size_t> CosteTren;
  vector<size_t> CosteBus;
  vector<size_t> VerticesTren;
  vector<size_t> VerticesBus;

  CosteTren = Dijkstra(Tren,origen,VerticesTren);
  CosteBus = DijkstraInv(Bus,destino,VerticesBus);
```

```
  //nos creamos dos "ciudades" que serán representadas con dos enteros
  size_t ciudad1 = CosteTren[cambio1]+CosteBus[cambio1];
  size_t ciudad2 = CosteTren[cambio2]+CosteBus[cambio2];
  return std::min(ciudad1,ciudad2);
}
```

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.

Como en el ejercicio anterior, contamos con 2 grafos que representan los costes de ir en tren y bus, además contamos con dos ciudades, una de origen del viaje y la destino del mismo.

En este ejercicio vemos que los viajes pueden ser o no combinados, es decir, nuestros viajes son:

- Viaje1: Todo en Tren → Dijkstra(tren).
- Viaje2: Todo en Bus → Dijkstra(bus).
- Viaje3: Empiezo con Tren y termino con Bus → Dijkstra(tren) + DijkstraInv(bus).
- Viaje4: Empiezo con Bus y termino con Tren → Dijkstra(bus) + DijkstraInv(tren).

Por tanto, la tarifa mínima será el menor coste de estas 4 opciones:

```
size_t ViajeTren_Bus2(const GrafoP<size_t> &Tren, const GrafoP<size_t> &Bus,
↪  size_t origen, size_t destino){
  //Vamos a crear los vectores (vértices y costes) para realizar Dijkstra y
  ↪  DijkstraInv.
  vector<size_t> VerticesTren;
  vector<size_t> VerticesBus;
  vector<size_t> CostesMinimosTren = Dijkstra(Tren,origen,VerticesTren);
  vector<size_t> CostesMinimosBus = Dijkstra(Bus,origen,VerticesBus);
  //ahora vamos a calcular los costes para las 4 opciones de viaje:
  size_t viaje1 = CostesMinimosTren[destino],
         viaje2 = CostesMinimosBus[destino];
  size_t viaje3, viaje4;
  for(size_t i = 0; i < Tren.numVert(); i++){
    viaje3 = CostesMinimosTren[origen] +
    ↪  DijkstraInv(Bus,destino,VerticesBus)[destino];
```

```
    viaje4 = CostesMinimosBus[origen] +
    ↪  DijkstraInv(Tren,destino,VerticesTren)[destino];
  }
  //Devolvemos el mínimo de los 4:
  return std::min(viaje1,viaje2,viaje3,viaje4);
}
```

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, 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.*

Ahora contamos de nuevo con los dos grafos (costes de ir en tren y bus), la ciudades origen y destino del viaje, pero ahora tenemos un coste adicional que sería el taxi que nos permite cambiarnos de medio de transporte.

Nuestros viajes quedan de la forma:

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

Donde el primer cuadrante, sería el grafo del tren, segundo y tercer cuadrante en su diagonal se pondría el coste del taxi y en el cuarto cuadrante estaría el grafo del bus.

Ahora tendríamos los siguientes viajes:

- Viaje1: Todo en Tren → Dijkstra(origenTren)[destinoTren].
- Viaje2: Tren-Bus → Dijkstra(origenTren)[destinoBus].
- Viaje3: Todo Bus → Dijkstra(origenBus)[destinoBus].
- Viaje4: Bus-Tren → Dijkstra(origenBus)[destinoTren].

Esto lo podemos hacer haciendo un supergrafo, que nos dará como resultado la matriz anterior.

```
//Vamos a crearnos un método que calcule el mínimo de los 4 viajes:
size_t minimo (size_t a, size_t b, size_t c, size_t d){
  size_t minimo = a;
  typedef GrafoP<size_t>::INFINITO INFINITO;
  if(b != INFINITO && b < minimo)
     minimo = b;
  if(c != INFINITO && c < minimo)
    minimo = c;
  if( d != INFINITO && d < minimo)
    minimo = d;
  return minimo;
}
//Definimos los tipos de datos a usar
typedef std::pair<vector<size_t>, size_t> CaminoCosteViaje;
```

```
CaminoCosteViaje Viaje(const GrafoP<size_t> &Tren, const GrafoP<size_t> &Bus,
↪  size_t origen, size_t destino, size_t costeTaxi){
  //Creamos el supergrafo de tamaño 2*N
  GrafoP<size_t> G(2*Tren.numVert());
  //Rellenamos el supergrafo:
  for(size_t i = 0; i < Tren.numVert(); i++)
    for(size_t j = 0; j < Tren.numVert(); j++){
      if(i < Tren.numVert() && j < Tren.numVert()){ //Primer cuadrante
        G[i][j] = Tren[i][j];
      }
      //Segundo y Tercer cuadrante
      else if(i < Tren.numVert() && j >= Tren.numVert() || i >= Tren.numVert() &&
      ↪  j < Tren.numVert()){
        //Solamente tiene coste las diagonales (Donde se encuentran un Tren con
        ↪  un Bus)
        if(i + Tren.numVert() == j || i == Tren.numVert() + j){
          G[i][j] = costeTaxi;
        }
        else{
          G[i][j] = GrafoP<size_t>::INFINITO;
        }
      }
      else{//Cuarto cuadrante
        G[i][j] = Bus[i - Tren.numVert()][j - Tren.numVert()];
      }
    }
  //Ahora que tenemos relleno el supergrafo, vamos a calcular los costes minimos
  ↪  de los viajes
  //Primero creamos los vectores (vertices y costes) para hacer Dijkstra
  vector<size_t> CostesMinimosTren;
  vector<size_t> CostesMinimosBus;
  vector<size_t> VerticesTren;
  vector<size_t> VerticesBus;
  size_t COT = origen; //Ciudad Origen Tren
  size_t CDT = destino; //Ciudad Destino Tren
  size_t COB = origen + tren.numVert(); //Ciudad Origen Bus
  size_t CDB = destino + tren.numVert(); //Ciudad Destino Bus


  //Hacemos Dijkstra con el Grafo de Tren
  CostesMinimosTren = Dijkstra(Tren,COT,VerticesTren);
  CostesMinimosBus = Dijkstra(Bus,COB, VerticesBus);

  //Calculamos los 4 viajes diferentes
  size_t viaje1 = CostesMinimosTren[CDT];
  size_t viaje2 = CostesMinimosTren[CDB];
  size_t viaje3 = CostesMinimosBus[CDB];
  size_t viaje4 = CostesMinimosBus[CDT];

  //Ahora nos quedamos con el mínimo de los 4
```

```
  size_t viaje = minimo(viaje1,viaje2,viaje3,viaje4);

  //Obtenemos el camino para poder devolverlo
  vector<size_t> verticesCamino = (viaje == viaje1|| viaje == viaje2) ?
  ↪  VerticesTren : VerticesBus;
  size_t i = (viaje == viaje1 || viaje == viaje3) ? CDT : CDB;
  //Creamos el vector
  vector<size_t> camino;
  while(i != origen){
    camino.push_back(i);
    i = verticesCamino[i];
  }
  //Guardamos el origen y le damos la vuelta al vector
  camino.push_bac(origen);
  camino.reverse(camino.size());
  return{camino,viaje};
}
```

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.

En este ejercicio contamos ahora con 3 grafos (tren, bus y avión), que representan las 3 maneras de poder viajar entre ciudades de un mismo país, también contamos con la ciudad origen y destino del viaje y a diferencia de otros ejercicios tenemos el coste del taxi que nos lleva desde el aeropuerto a una estación de tren o bus, o viceversa y el coste del taxi de ir de una estación de tren a una de bus y viceversa.

Por tanto, para realizar este ejercicio, podemos hacer un supergrafo del tamaño 3*N para poder calcular los viajes que se pueden hacer en dicho país, para ello haremos uso de Dijkstra ya que tenemos una ciudad origen y un destino, sean los viajes posibles:

- Viaje1: Todo Tren → Dijkstra(Origentren)[DestinoTren]
- Viaje2: Tren-Bus → Dijkstra(OrigenTren)[DestinoBus]
- Viaje3: Tren-avion → Dijkstra(OrigenTren)[DestinoAvion]
- Viaje4: Todo Bus → Dijkstra(OrigenBus)[DestinoBus]

- Viaje5: Bus-Tren → Dijkstra(OrigenBus)[DestinoTren]
- Viaje6: Bus-Avion → Dijkstra(OrigenBus)[DestinoAvion]
- Viaje7: Todo Avion → Dijkstra(OrigenAvion)[DestinoAvion]
- Viaje8: Avion-Tren → Dijkstra(OrigenAvion)[DestinoTren]
- Viaje9: Avion-Bus → Dijkstra(OrigenAvion)[DestinoBus]

Finalmente, nuestra matriz (supergrafo) quedaría:

	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

```
//Definimos el tipo de dato que vamos a devolver
typedef std::pair<vector<vertice>,size_t> CosteViaje;

//Función auxiliar:
size_t minimo (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){
  size_t min = a;
  typedef GrafoP<size_t>::INFINITO Infinito;
  if(b != Infinito && b < min)
    min = b;
  if(c != Infinito && c < min)
    min = c;
  if(d != Infinito && d < min)
    min = d;
  if(e != Infinito && e < min)
    min = e;
  if(f != Infinito && f < min)
    min = f;
  if(g != Infinito && g < min)
    min = g;
  if(h != Infinito && h < min)
    min = h;
  if(i != Infinito && i < min)
    min = i;
  return min;
}

CosteViaje Viaje(GrafoP<size_t> &Tren, GrafoP<size_t> &Bus, GrafoP<size_t>
↪  &Avion, size_t Origen, size_t Destino, size_t CTaxi, size_t CTaxiAeropuerto){
  //Primero de todos creamos el supergrafo y lo rellenamos
  GrafoP<size_t> S(3*Tren.numVert());

  for(size_t i = 0; i < S.numVert(); i++){
    for(size_t j = 0; j < S.numVert(); j++){
      if(i < Tren.numVert() && j < Tren.numVert()){ //1º cuadrante
        S[i][j] = Tren[i][j];
```

```
}
else if(i < Tren.numVert() && j >= Tren.numVert() && j < 2*Tren.numVert()){
↪  //2º Cuadrante
  //Solamente se rellena la diagonal, si no infinito
  if(j - Tren.numvert() == i){
    S[i][j] = CTaxi;
  }else{
    S[i][j] = GrafoP<size_t>::INFINITO;
  }
}
else if(i < Tren.numVert() && j >= 2*Tren.numVert()){ //3º Cuadrante
  //Solamente se rellena la diagonal, si no infinito
  if(i == j - 2*Tren.numVert()){
    S[i][j] = CTaxiAeropuerto;
  }
  else{
    S[i][j] = GrafoP<size_t>::INFINITO;
  }
}
else if(i >= Tren.numVert() && i < 2*Tren.numVert() && j < Tren.numVert()){
↪  //4º Cuadrante
  //Solamente se rellena la diagonal, si no infinito
  if(j == i - Tren.numVert()){
    S[i][j] = CTaxi;
  }
  else{
    S[i][j] = GrafoP<size_t>::INFINITO;
  }
}
else if(i >= Tren.numVert() && i < 2*Tren.numVert() && (j >= Tren.numVert()
↪  && j < 2*Tren.numVert())){
  //5º Cuadrante
  S[i][j] = Bus[i-Tren.numVert()][j-Tren.numVert()];
}
else if(i >= Tren.numVert() && i < 2*Tren.numVert() && j >
↪  2*Tren.numVert()){ //6º Cuadrante
  //Solamente se rellena la diagonal, si no infinito
  if(i - Tren.numVert() == j - 2*Tren.numVert()){
    S[i][j] = CTaxi;
  }
  else{
    S[i][j] = GrafoP<size_t>::INFINITO;
  }
}
else if(i >= 2*Tren.numVert() && j < Tren.numVert() ){//7º Cuadrante
  //Solamente se rellena la diagonal, si no infinito
  if(i - 2*Tren.numVert() == j){
    S[i][j] = CTaxiAeropuerto;
  }
  else{
```

```
        S[i][j] = GrafoP<size_t>::INFINITO;
      }
    }
    else if(i >= 2*Tren.numVert() && j >= Tren.numVert() && j <
    ↪  2*Tren.numVert()){//8º Cuadrante
      //Solamente se rellena la diagonal, si no infinito
      if(i - 2*Tren.numVert() == j - Tren.numVert()){
        S[i][j] = CTaxiAeropuerto;
      }
      else{
        S[i][j] = GrafoP<size_t>::INFINITO;
      }
    }
    else{ //9º Cuadrante
      S[i-2*Tren.numVert()][j-2*Tren.numVert()] = Avion[i][j];
    }
  }
}
//Ya rellenado el supergrado, podemos realizar el cálculo de los costes mínimos,
↪  para ello creamos los vectores(coste y vértices) para realizar Dijkstra.


//Dijkstra desde Ciudad Origen Tren
vector<size_t>VerticesTren;
size_t OrigenTren = Origen;
size_t DestinoTren = Destino;
vector<size_t>CostesMinimosTren = Dijkstra(S,OrigenTren,VerticesTren);

//Dijkstra desde ciudad Origen Bus
vector<size_t>VerticesBus;
size_t OrigenBus = Origen + Tren.numVert();
size_t DestinoBus = Destino + Tren.numVert();
vector<size_t>CostesMinimosBus = Dijkstra(S,OrigenBus,VerticesBus);

//Dijkstra desde ciudad Origen Avion
vector<size_t> VerticesAvion;
size_t OrigenAvion = Origen + 2*Tren.numVert();
size_t DestinoAvion = Destino + 2*Tren.numVert();
vector<size_t> CostesMinimosAvion = Dijkstra(S,OrigenAvion,VerticesAvion);



//Vamos a calcular el camino más corto (menor coste):
//---Viajes OrigenTren---
size_t viaje1 = CostesMinimosTren[DestinoTren];
size_t viaje2 = CostesMinimosTren[DestinoBus];
size_t viaje3 = CostesMinimosTren[DestinoAvion];

//---Viajes OrigenBus---
size_t viaje4 = CostesMinimosBus[DestinoBus];
```

```
  size_t viaje5 = CostesMinimosBus[DestinoTren];
  size_t viaje6 = CostesMinimosBus[DestinoAvion];

  //---Viajes OrigenAvion---
  size_t viaje7 = CostesMinimosAvion[DestinoAvion];
  size_t viaje8 = CostesMinimosAvion[DestinoTren];
  size_t viaje9 = CostesMinimosAvion[DestinoBus];

  //Ahora nos vamos a quedar con el mínimo de los 9 viajes posibles.
  size_t viajeminimo =
  ↪  minimo(viaje1,viaje2,viaje3,viaje4,viaje5,viaje6,viaje7,viaje8,viaje9);

  //Vamos a obtener el camino del viaje minimo.
  vector<size_t>verticescamino = (viajeminimo == viaje1 || viajeminimo == viaje2
  ↪  || viajeminimo == viaje3) ? VerticesTren : (viajeminimo == Viaje4 ||
  ↪  viajeminimo == viaje5 || viajeminimo == viaje6 )? VerticesBus :
  ↪  VerticesAvion;

  vector<size_t>camino; //camino del viaje minimo
  size_t ciudad = (viajeminimo == viaje1 || viajeminimo == viaje5|| viajeminimo ==
  ↪  viaje8) ?DestinoTren : (viajeminimo == viaje2 || viajeminimo == viaje4 ||
  ↪  viajeminimo == viaje9) ? DestinoBus : DestinoAvion; //obtenemos el destino

  while(ciudad != Origen){
    camino.push_back(ciudad);
    ciudad = verticescamino[ciudad];
  }
  camino.push_back(origen);
  camino.reverse(camino.size());

  //Devolvemos el camino + el coste
  return{camino,viajecamino};
}
```

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 !!!

Contamos con 3 grafos (3 islas del archipiélago), la lista de puentes de cada isla. Como tenemos

3 grafos podemos hacer uso de un supergrafo (3*N) quedando la matriz de la manera:

	Isla1	**Isla2**	**Isla3**
Isla1	GRAFO ISLA1	PUENTES ISLA1-2	PUENTES ISLA1-3
Isla2	PUENTES ISLA2-1	GRAFO ISLA2	PUENTES ISLA2-3
Isla3	PUENTES ISLA3-1	PUENTES ISLA3-2	GRAFO ISLA3

Como no contamos ni con un origen ni destino, haremos uso de Floyd, para obtener la matriz de costes minimos. Una cosa a tener en cuenta son los puentes, que lo vamos a representar como un par de dos ciudades/vértices, además el uso de estos es gratis.

```
//Definimos los tipos de datos a usar
typedef std::pair<size_t,size_t> puente;

matriz<size_t> ViajesIslas(GrafoP<size_t> &Isla1, GrafoP<size_t> &Isla2,
↪  GrafoP<size_t> &Isla3, const vector<puente> &puentes){
  //Creamos el supergrafo
  GrafoP<size_t> S(Isla1.numVert() + Isla2.numVert() + Isla3.numVert());
  //Rellenamos el supergrafo
  for(size_t i = 0; i < S.numVert(); i++){
    for(size_t j = 0; j < S.numVert(); j++){
      //Como no tenemos que incluir el coste de los puentes, solamente buscamos
      ↪  los Cuadrantes de los Grafos, lo demás en infinito (no hay puentes
      ↪  todavia).
      if(i < Isla1.numVert() && j < Isla.numVert()){ //1º Cuadrante
        S[i][j] = Isla1[i][j];
      }
      else if(i >= Isla1.numVert() && i < Isla1.numVert() + Isla2.numVert() && j
      ↪  >= Isla1.numVert() && j < Isla1.numVert() + Isla2.numVert()){//5º
      ↪  Cuadrante
        S[i][j] = Isla2[i - Isla1.numVert()][j - Isla1.numVert()];
      }
      else if(i >= Isla1.numVert() + Isla2.numVert() && j >= Isla1.numVert() +
      ↪  Isla2.numVert()){//9º Cuadrante
        S[i][j] = Isla3[i - Isla1.numVert() - Isla2.numVert()][j -
        ↪  Isla1.numVert() - Isla2.numVert()];
      }
      else{ //no hay puentes todavía
        S[i][j] = GrafoP<size_t>::INFINITO;
      }
    }
  }
  //Una vez relleno el supergrafo, vamos a incluir los puentes.
  for(auto& p : puentes){
    // = 0 significa que hay camino directo.
    G[p.first][p.second] = 0;
    G[p.second][p.first] = 0;
  }

  //Ahora declaramos las matrices (CostesMinimos y vértices) para calcular Floyd.
  matriz<size_t>CostesMinimos,Vertices;
  return CostesMinimos = Floyd(S,Vertices);
```

}

Ejercicio 12: *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 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 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.

Partimos de que tenemos dos grafos (ambas islas) y dos conjuntos que son las ciudades costeras de cada isla.

Para poder construir el puente que minimice el coste global de viajar podemos calcular primero los costes mínimos de cada isla con floyd y luego quedarnos con el mínimo de ambas islas, construyendo el puente entre esas dos ciudades, donde un puente será un par de dos ciudades de diferentes islas.

```
//Definimos los tipos de datos a usar
typedef std::pair<size_t, size_t>puente;

puente PuenteFobosDeimos(const GrafoP<size_t> &Fobos, const GrafoP<size_t>
↪  &Deimos, const vector<size_t> &costerasFobos, const vector<size_t>
↪  &costerasDeimos){
  //Definimos las matrices (Costes y vértices) para hacer uso de floyd
  matriz<size_t>verticesFobos, CostesMinimosFobos, verticesDeimos,
  ↪  CostesMinimosDeimos;
  //Obtenemos los costes mínimos de cada isla.
  CostesMinimosFobos = Floyd(Fobos,verticesFobos);
  CostesMinimosDeimos = Floyd(Deimos,verticesDeimos);

  //La ciudad costera elegida será la mínima de la media de los costes de ir a
  ↪  dichas ciudades costeras.
  size_t CiudadCosteraMinimaFobos = GrafoP<size_t>::INFINITO;
  for(size_t i = 0; i < costerasFobos.size(); i++){//comparamos todas las
  ↪  ciudades costeras con las ciudades del grafo
    size_t media = 0;
    for(size_t j = 0; j < Fobos.numVert(); j++){
      media += Fobos[j][costerasFobos[i]];
    }
    if(media < CiudadCosteraMinimaFobos)
      CiudadCosteraMinimaFobos = costerasFobos[i];
```

```
  }

  size_t CiudadCosteraMinimaDeimos = GrafoP<size_t>::INFINITO;
  for(size_t i = 0; i < costerasDeimos.size(); i++){
    size_t media = 0;
    for(size_t j = 0; j < Deimos.numVert(); j++){
      media += Deimos[j][costerasDeimos[i]];
    }
    if(media < CiudadCosteraMinimaDeimos){
      CiudadCosteraMinimaDeimos = costerasDeimos[i];
    }
  }
  //Devolvemos el puente de ambas ciudades costeras que por lo general cuestan
  ↪  menos ir desde cualquier otra ciudad de su isla.
  return {CiudadCosteraMinimaFobos,CiudadCosteraMinimaDeimos};
}
```

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 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 archipélago. 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.

Ahora en este ejercicio tenemos 3 grafos (3 islas), el conjunto de las ciudades costeras de cada isla y queremos obtener el conjunto de puentes que podemos construir en el archipiélago.

Como queremos el mínimo número de puentes posibles se elegirá las ciudades costeras cuyo coste medio de ir desde cualquier otra ciudad de la isla sea el menor, es decir, para conseguir esto, vamos a realizar Floyd para cada isla con el fin de tener la matriz de costes mínimos de cada uno y luego seleccionamos la ciudad costera con menor coste de todas.

Un puente, igual que en el ejercicio anterior, será un par de dos ciudades o vértices de dos islas diferentes.

```
//Definimos los tipos de datos a usar
```

```cpp
typedef std::pair<size_t,size_t> puente;

vector<puente> PuentesArchipielago(const GrafoP<size_t> &Isla1, const
↪  GrafoP<size_t> &Isla2, const GrafoP<size_t> &Isla3, const vector<size_t>
↪  &costerasIsla1, const vector<size_t> &costerasIsla2, const vector<size_t>
↪  &costerasIsla3 ){
  //Vamos a declarar las matrices(CostesMinimos y vértices) para poder realizar
  ↪  Floyd.
  matriz<size_t>CostesMinimosIsla1,CostesMinimosIsla2,CostesMinimosIsla3,
                VerticesIsla1,VerticesIsla2,VerticesIsla3;
  CostesMinimosIsla1 = Floyd(Isla1,VerticesIsla1);
  CostesMinimosIsla2 = Floyd(Isla2,VerticesIsla2);
  CostesMinimosIsla3 = Floyd(Isla3,VerticesIsla3);

  //Como hemos dicho antes, tenemos que quedarnos con la ciudad costera cuyo
  ↪  coste medio sea el menor.

  size_t CiudadCosteraMinimaIsla1 = GrafoP<size_t>::INFINITO;
  size_t CiudadCosteraMinimaIsla2 = GrafoP<size_t>::INFINITO;
  size_t CiudadCosteraMinimaIsla3 = GrafoP<size_t>::INFINITO;

  //Obtenemos la costera mínima de Isla1 comparandola con todas las ciudades de
  ↪  la Isla:
  for(size_t i = 0; i < costerasIsla1.size(); i++){
    size_t media = 0;
    for(size_t j = 0; j < Isla1.numVert(); j++){
      media += Isla1[j][costerasIsla1[i]];
    }
    if(media < CiudadCosteraMinimaIsla1){
      CiudadCosteraMinimaIsla1 = costerasIsla1[i];
    }
  }

  //Obtenemos la costera mínima de Isla2 comparandola con todas las ciudades de
  ↪  la Isla:
  for(size_t i = 0; i < costerasIsla2.size(); i++){
    size_t media = 0;
    for(size_t j = 0; j < Isla2.numVert(); j++){
      media += Isla2[j][costerasIsla2[i]];
    }
    if(media < CiudadCosteraMinimaIsla2){
      CiudadCosteraMinimaIsla2 = costerasIsla2[i];
    }
  }
  //Obtenemos la costera mínima de Isla3 comparandola con todas las ciudades de
  ↪  la Isla:
  for(size_t i = 0; i < costerasIsla3.size(); i++){
    size_t media = 0;
    for(size_t j = 0; j < Isla3.numVert(); j++){
      media += Isla3[j][costerasIsla3[i]];
```

```
    }
    if(media < CiudadCosteraMinimaIsla3){
      CiudadCosteraMinimaIsla3 = costerasIsla3[i];
    }
  }
  //Obtenidas las ciudades costeras cuyo coste medio es el mínimo, vamos crear
  ↪  los puentes y devolverlos en el vector.
  vector<puente> puentes;
  puentes.push_back({CiudadCosteraMinimaIsla1,CiudadCosteraMinimaIsla2});
  puentes.push_back({CiudadCosteraMinimaIsla1,CiudadCosteraMinimaIsla3});
  puentes.push_back({CiudadCosteraMinimaIsla2,CiudadCosteraMinimaIsla3});

  return puentes; //devolvemos el conjunto de los puentes.
}
```

Práctica 8: Problemas de Grafos III

Esta es la última práctica de grafos. Donde trabajaremos con los algoritmos vistos en las prácticas 6 y 7, y además con nuevos algoritmos como son el TAD Partición, Kruskal o Prim y el TAD APO.

Para la resolución de esta práctica tendremos como ficheros de cabeceras:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cmath>
#include "../grafoMA.h"
#include "../grafoPMC.h"
#include "../alg_grafoMA.h"
#include "../alg_grafoPMC.h"
#include "../particion.h"
```

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.

No sabemos ni el número de islas ni de ciudades de las mismas. Sabemos que las ciudades de una isla se conectan mediante carreteras y estas ciudades están formadas por dos coordenadas cartesianas.

También contamos con una matriz de adyacencia, es decir, que podemos saber que ciudades están en dicha isla. Mediante esto y el Grafo dado, tenemos que devolver las islas del archipiélago y el coste de viajar entre dos ciudades de una misma isla.

```
//Definimos los tipos de datos a usar
typedef std::pair<size_t,size_t> Ciudad; //ciudad con las coordenadas x , y

typedef struct Isla{
```

```
    vector<Ciudad> CiudadesIsla;
    matriz<size_t> CostesIsla;
};

typedef vector<Isla> Archipielago;

//Función que nos calcula la distancia euclidea de dos ciudades
size_t DistanciaEuclidea(Ciudad a, Ciudad b){
  return sqrt(pow(a.first-b.first,2) + (a.second-b.second,2));
}

//Función que nos calcula los costes minimos de cada isla dado su conjunto de
↪  ciudades
matriz<size_t> CalculaCostesIsla(vector<Ciudad> &ciudades){
  //Creamos el grafo para poder realizar el cálculo de los costes mínimos
  GrafoP<size_t> GIsla(ciudades.size());
  //Rellenamos el grafo con los costes (Distancia Euclídea)
  for(size_t i = 0; i < ciudades.size(); i++)
    for(size_t j =  i+1; j < ciudades.size(); j++){
      GIsla[i][j] = GIsla[j][i] = DistanciaEuclidea(ciudades[i],ciudades[j]);
    }
  //Rellenado el grafo, vamos a calcular sus costes mínimos, con Floyd por lo que
  ↪  creamos las matrices de vértices y costes
  matriz<size_t>Vertices(GIsla.numVert()),CostesMinimos(GIsla.numVert());
  return CostesMinimos = Floyd(GIsla,Vertices);
}

//Sabemos que partimos de una matriz de adyacencia (grafo no ponderado) y del
↪  conjunto de ciudades
Archipielago Tumbuctu1(const Grafo& G, vector<Ciudad> &ciudades){
  //Con la matriz de adyacencia sabemos si una ciudad está en la misma isla que
  ↪  otra si esta devuelve 1, si no devuelve 0.
  //Sabiendo esto podemos ir recorriendo la matriz e insertandon en cada
  ↪  subconjunto de la partición las ciuades.
  size_t Tama = G.numVert();
  Particion P(Tama);
  //Recorremos el grafo comprobando si hay carretera
  for(size_t i = 0; i < Tama; i++)
    for(size_t j = 0; j < Tama; j++){
      if(G[i][j] && P.encontrar(i) != P.encontrar(j)){ //Hay carretera y no están
      ↪  unidas (distinto representante), las unimos en el mismo subconjunto.
        P.unir(P.encontrar(i),P.encontrar(j));
      }
    }
  //Tenemos los sunconjuntos crados, por lo que vamos a insertar en cada isla sus
  ↪  ciudades el coste
  size_t representante_actual = P.encontrar(0);
  Isla isla;
  Archipielago archipielago; //contiene el número de islas
  for(size_t i = 0; i < ciudades.size(); i++){
```

```
    if(P.encontrar(i) == representante_actual){ //está en esa isla
      isla.CiudadesIsla.push_back(ciudades[i]);
    }
    else{ //si no tiene el mismo representante, metemos la isla en el
    ↪  archipiélago y cambiamos el representante
      isla.CostesIsla = CalculaCostesIsla(isla.CiudadesIsla); //gaurdamos los
      ↪  costes mínimos de la isla
      archipielago.push_back(isla); //almacenamos la isla en el archipiélago
      //limpiamos el vector de ciudades de la isla
      isla.CiudadesIsla.clear();
      isla.CiudadesIsla.push_back(ciudades[i]); //insertamos la nueva ciudad
      representante_actual = P.encontrar(i); //nuevo representante, de la nueva
      ↪  ciudad
    }
  }
  return archipielago;
}
```

Ejercicio 2: *El archipiélago de Tombuctú2 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 Tombuctú2 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.

Este ejercicio es casi parecido al anterior, con la diferencia que ahora buscamos las líneas aéreas necesarias para comunicar las diferentes islas del archipiélago.

Vamos a hacer uso del método implementado en el ejercicio anterior `Tombuctu1()`, y el método `DistanciaEuclidea()` para reutilizar código.

Ahora estas líneas aéreas será un par de dos ciudades de diferente isla, que mediante una matriz de booleanos vamos a comprobar si están ya unidas o no.

Además como hay un aeropuerto por cada ciudad, esto nos deja un número de $n(n-1)/2$ líneas aéreas posibles que equivale al número de nodos del grafo. Esto es un número enorme de posibilidades por lo que haremos uso de un APO donde guardaremos estas líneas aéreas y podremos acceder a la de menor coste en un tiempo de $O(1)$, ya que tenemos acceso al mínimo elemento.

```
//Definimos los tipos de datos a usar
typedef std::pair<size_t,size_t> Ciudad; //ciudad con las coordenadas x , y

typedef struct LineaAerea{
  size_t Origen_, Destino_, CosteViaje_;
};

typedef vector<LineaAerea> LineasAereasArchipielago;

typedef struct Isla{
   vector<Ciudad> CiudadesIsla;
   matriz<size_t> CostesIsla;
};

typedef vector<Isla> Archipielago;

LineasAereasArchipielago Tumbuctu(const Grafo& G, vector<Ciudad> &ciudades){
  //Creamos el archipiélago a partir del algoritmo del ejercicio 1.
  Archipielago archipielago = Tumbuctu1(G,ciudades);
  //Para ello, haremos uso del TAD APO, ya que el coste de encontrar el mínimo es
  ↪  de orden O(1).
  Apo<LineaAerea> A(archipielago.size()*(archipielago.size()-1));
  //Guardamos en el APO todas las líneas aéreas posible
  for(size_t i = 0; i < archipielago.size(); i++)
    for(size_t j = 0; j < archipielago.size(); j++){
      if(P.encontrar(i) != P.encontrar(j)){
        //Distinto representante, distinta isla -> linaea aerea
        //insertamos la linea aérea en el apo
        size_t representanteOrigen = P.encontrar(i);
        size_t representanteDestino = P.encontrar(j);
        A.insertar({representanteOrigen,representanteDestino,
          DistanciaEuclidea(ciudades[i],ciudades[j])});
      }
      //son de la misma isla, no hay linea aérea
    }
  //Tenemos el APO con las líneas aéreas, vamos a quedarnos con las menos costosas
  //Vamos a crearnos una matriz de adyacencia de las islas para saber que islas
  ↪  están unidas
  matriz<bool> ConexiónIsla(archipielago.size(),false);

  LineasAereasArchipielago
  ↪  vector_LA((archipielago.size()*(archipielago.size()-1))/2);
  size_t contador = 0;
  while(!A.vacio() && contador <
  ↪  (archipielago.size()*(archipielago.size()-1))/2){
```

```
    LineaAerea aux = A.cima(); //linea aérea menos costosa
    if(P.encontrar(aux.Origen_) != P.encontrar(aux.Destino_) &&
    ↪  ConexiónIsla[aux.Origen_][aux.Destino_] == false){
      //son islas diferentes y no están conectadas.
      ConexiónIsla[aux.Origen_][aux.Destino_] =
      ↪  ConexiónIsla[aux.Origen_][aux.Destino_] = true;
      //rellenamos el vector de las lineas aereas
      vector_LA[contador].Origen_ = aux.Origen_;
      vector_LA[contador].Destino_ = aux.Destino_;
      vector_LA[contador].CosteViaje_ = aux.CosteViaje_;
      contador++ ; //aumentamos el contador de las líneas aéreas.
    }
  }
  return vector_LA;
  hola
}
```

Ejercicio 3: *Implementa un subprograma para encontrar un árbol de extensión máximo. ¿Es más difícil que encontrar un árbol de extensión mínimo?*

Podemos hacer uso del algoritmo de Kruskal pero con la diferencia de que en vez de ordenar las aristas de menor a mayor, las ordenamos de mayor a menor, es decir, vamos a tener un algoritmo llamado Kruskal Máximo.

```
template <typename tCoste> GrafoP<tCoste> KruskalMaximo(const GrafoP<tCoste> &G){
  //Comprobamos si este grafo es dirigido o no
  assert(!G.esDirigido());
  typedef typename GrafoP<size_t>::vertice vertice;
  typedef typename GrafoP<size_t>::arista arista;
  const size_t n = G.numVert();
  Particion P(n);
  vector<arista>aristas;
  for(vertice v = 0; v < n; v++)
    for(vertice w = v+1; w < n; w++)
      if(G[v][w] != GrafoP<size_t>::INFINITO)
        aristas.push_back(arista(v,w,G[v][w]));
  //Ordenamos las aristas de mayor a menor, en vez de menor a mayor
  sort(aristas.begin(),aristas.end(),greater<arista>());
  GrafoP<tCoste>A(n); //Creamos el nuevo grafo
  for(size_t i = 0; i < aristas.size(); i++){
    vertice v = aristas[i].v, w = aristas[i].w;
    if(P.encontrar(v) != P.encontrar(w)){
      A[v][w] = A[w][v] = aristas[i].coste;
      P.unir(v,w);
    }
  }
  retrun A;
}
```

Ejercicio 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.*

Partimos del conjunto de ciudades de Cárceres, para calcular qué canales de tamaño máximo podemos construir haremos uso del algoritmo `KruskalMaximo()`, es decir, estamos buscando un árbol de expansión máximo.

```
//Definimos los tipos de datos a usar
typedef std::pair<size_t,size_t> ciudad;

GrafoP<size_t> EMASAJER(vector<ciudad>ciudades){
  //Vamos a crear el grafo a devolver
  GrafoP<size_t> Jerte(ciudades.size());

  //Rellenamos el Grafo
  for(size_t i = 0; i < ciudades.size(); i++)
    for(size_t j = i+1; j < ciudades.size(); j++)
      Jerte[i][j] = DistanciaEuclidea(ciudades[i],ciudades[j]);

  //Ahora calculamos el KruskalMaximo de Jerte y devolvemos el Grafo
  return KruskalMaximo(Jerte);
}
```

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.*

Este ejercicio es muy parecido al anterior pero ahora buscamos obtener el árbol de expansión mínimo, es decir, haremos uso de Kruskal.

```
//Definimos los tipos de datos a usar
typedef std::pair<size_t,size_t> ciudad;

size_t RETEUNI3(GrafoP<size_t> G){
  //Creamos el grafo a devolver
  GrafoP<size_t> F = Kruskal(G);
  size_t longitud = 0; //longitud de la fibra óptica necesaria
  //Rellenamos el grafo con los costes
  for(size_t i = 0; i < ciudades.size(); i++)
    for(size_t j = i+1; j < ciudades.size(); j++)
      longitud += F[i][j];
  return longitud;
}
```

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 m3/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 m3/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.

Tenemos que tener en cuenta que queremos devolver los canales y su longitud sabiendo que el caudal del mismo tiene que ser lo mayor posible y el mínimo número de canales posibles. Como queremos que el caudal sea lo mayor posible, vamos a hacer uso de `KruskalMaximo()`.

```
//Definimos los tipos de datos a usar
typedef struct{
  size_t caudal_, longitud_;
}canal;

GrafoP<canal> EMASAJER2(matriz<size_t>caudales, matriz<size_t>longitudes, size_t
↪  subvencionCaudal, size_t costeLongitud){
  //Creamos el grafo
  Grafop<size_t> Jerte(caudales.dimension());
  //Rellenamos el Grafo con los caudales y longitudes
  for(size_t i = 0; i < caudales.dimension(); i++)
    for(size_t j = i+1; j < caudales.dimension(); j++)
      Jerte[i][j] = caudales[i][j]*(subvencionCaudal/100) -
      ↪  longitudes[i][j]*costeLongitud;
  //Ahora hacemos KruskalMaximo del Grafo
  Jerte = KruskalMaximo(Jerte);
  //Ahora creamos el grafo que vamos a devolver
  GrafoP<canal> JerteFinal(Jerte.numVert());
  //Rellenamos el Grafo
  for(size_t i = 0; i < Jerte.numVert(); i++)
    for(size_t j = i+1; j < Jerte.numVert(); j++){
      if(Jerte[i][j]!= GrafoP<size_t>::INFINITO){
        JerteFinal[i][j].caudal_ =  caudales[i][j];
        JerteFinal[i][j].longitud_ = longitudes[i][j];
      }
    }
  return JerteFinal;
}
```

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:

1. *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).*
2. *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:

1. *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.*

Contamos con el conjunto de ciudades de las islas Fobos y Deimos, el conjunto de ciudaes costeras de ambas islas y la ciudad origen y destino del viaje.

Como vemos, Grecoland está destruido y debe de ser reconstruido sabiendo que el coste de construir un puente es más caro que una carretera y que se debe de hacer con el menor coste posible, por lo que haremos uso de Kruskal, para tener las ciudades de cada isla comunicadas con el menor coste posible.

Como queremos obtener el coste mínimo de viajar entre dos ciudades de Grecoland vamos a hacer uso de Dijkstra.

```
//Definimos los tipos de datos a usar
typedef std::pair<size_t,size_t>ciudad;
size_t Grecoland(vector<ciudad>ciudadesFobos,vector<ciudad>ciudadesDeimos,
↪  vector<ciudad>costerasFobos, vector<ciudad>costerasDeimos, size_t Origen,
↪  size_t Destino){
  //Primero vamos a crear los grafos de Fobos para poder calcular su árbol
  ↪  generador de coste mínimo (Kruskal)
  GrafoP<size_t>Fobos(ciudadesFobos.size());
  //Rellenamos el grafo con los costes (DistanciaEuclidea)
  for(size_t i = 0; i < ciudadesFobos.size(); i++)
    for(size_t j = i+1; j < ciudadesFobos.size(); j++){
      Fobos[i][j] = DistanciaEuclidea[ciudadesFobos[i],ciudadesFobos[j]];
    }
  //Ahora calculamos el árbol generador de coste mínimo
  Fobos = Kruskal(Fobos);
```

```
//Lo mismo para la isla de Deimos
GrafoP<size_t>Deimos(ciudadesDeimos.size());
for(size_t i = 0; i < ciudadesDeimos.size(); i++)
  for(size_t j = i+1; j < ciudadesDeimos.size(); j++)
    Deimos[i][j] = DistanciaEuclidea(ciudadesDeimos[i], ciudadesDeimos[j]);
//Ahora hacemos Kruskal
Deimos = Kruskal(Deimos);

//Ahora vamos a crear el supergrafo
GrafoP<size_t>grecoland(ciudadesFobos.size());
//Rellenamos el supergrafo primero con el grafo Fobos
for(size_t i = 0; i < ciudadesFobos.size(); i++)
  for(size_t j = 0; j < ciudadesFobos.size(); j++){
    grecoland[i][j] = Fobos[i][j];
  }
//Ahora incluimos en el supergrafo el grafo Deimos
for(size_t i = 0; i < ciudadesDeimos.size(); i++)
  for(size_t j = 0; j < ciudadesDeimos.size(); j++)
    grecoland[i+ciudadesFobos.size()][j+ciudadesFobos.size()] = Deimos[i][j];

//Ahora que tenemos el supergrafo relleno, es decir, el archipielago
↪  reconstruido con las carreteras, vamos a incluir los puentes.
for(size_t i = 0; i < ciudadesFobos.size(); i++)
  for(size_t j = 0; j < ciudadesDeimos.size(); j++){
    //Si i es costera y j es costera, se unen con un puente
    //Buscamos en el vector de las ciudades costeras si dicha ciudad costera
    ↪  existe en el vector de ciudades de la isla:
    if(find(costerasFobos.begin(),costerasFobos.end(), ciudadesFobos[i])!=
    ↪  costerasFobos.end() &&
    ↪  find(costerasDeimos.begin(), costerasDeimos.end(), ciudadesDeimos[j])
    ↪  != costerasDeimos.end()){

      size_t costePuente =
      ↪  DistanciaEuclidea(ciudadesFobos[i],ciudadesDeimos[j]);
      //Guardamos en el grafo el coste del puente (la unión entre las dos
      ↪  ciudades):
      grecoland[i][j+ciudadesFobos.size()] = costePuente;
      grecoland[j+ciudadesFobos.size()][i] = costePuente;
    }
  }
//Ahora hacemos Kruskal del grafo grecoland para quedarnos con los puentes con
↪  menor costes
grecoland = Kruskal(grecoland);

//Ya que tenemos las ciudades unidas mediante carreteras y puentes con el coste
↪  menor posible, podemos calcular el coste de ir de una ciudad origen a
↪  destino en nuestro archipielago Grecoland. Para ello vamos a declarar los
↪  vectores (coste y vértices) para poder hacer uso de Dijkstra
vector<size_t>vertices(grecoland.numVert());
vector<size_t>CosteMinimo = Dijkstra(grecoland, Origen, vertices);
```

```
  return costeMinimo[Destino]; //devolvemos el coste mínimo de ir desde el Origen
  ↪  a nuestro Destino.
}
```