JUSTIFICACIÓN DE LA PRÁCTICA

- Comerciar -

- CONTEXTO DE LA FUNCIÓN

La función de carácter iterativo pertenece a la clase **Ciudad** y se llama desde el programa principal, usando la instrucción *comerciar* o *co*. Esta función no consta de ninguna función auxiliar asociada a la clase (u otras clases), y tiene como parámetro de entrada una ciudad y un inventario que nunca será modificado.

```
void comerciar(Ciudad& c, const Inventario& inv);
```

Esta función tiene como objetivo general intercambiar productos entre el parámetro implícito y otra ciudad, dándose todos los productos posibles recorriendo el inventario del parámetro implícito. La función basa su funcionamiento únicamente en modificar los inventarios de las ciudades, sin retornar nada al acabar de ejecutarse (función de tipo *void*).

```
/** @brief Comerciar
    Una ciudad le dara a la otra todos los productos que le sobren hasta alcanzar
    si es posible los que la otra necesite, y viceversa
    \pre c es una ciudad válida e inicializada, inv tiene al menos 1 elemento
    \post Se ha realizado el intercambio de productos entre el p.i. y c
*/
void comerciar(Ciudad& c, const Inventario& inv);
```

- PRECONDICIÓN:

La ciudad que forma parte del parámetro de entrada existe en la cuenca del río donde se trabaja en la función principal, con un identificador único asociado. Además, el inventario no está vacío, al menos tiene 1 elemento

- POSTCONDICIÓN:

Según el intercambio de productos entre el parámetro implícito y la Ciudad c se han modificado los inventarios de las ciudades añadiendo o quitando unidades de ciertos productos, pero nunca añadiendo productos al catálogo que no estaban antes de comerciar ni eliminando productos definitivamente del catálogo. Si no comercian las dos ciudades permanecen igual que antes.

- JUSTIFICACIÓN

En esta función el invariante representa a la garantía de que en esta función siempre se recorrerán productos que existan en el parámetro implícito, que en este caso sería:

it != prods_ciudad.end()

Al empezar esta función se entra en un bucle iniciando el iterator it = prods_ciudad.begin() (representa el primer producto de la ciudad). Si el parámetro implícito no tiene elementos, se cumple que prods_ciudad.begin() == prods_ciudad.end(), es decir, it == prods_ciudad.end(), y por tanto se sale del bucle y directamente no itera y acaba la función. Como toda la funcionalidad de comerciar está en el bucle, directamente no se comercia.

```
void Ciudad::comerciar(Ciudad& c, const Inventario& inv) {
    for (auto it = prods_ciudad.begin(); it != prods_ciudad.end(); ++it) {
        // Asignamos un id al producto que estamos tratando
        int id_prod = it->first;
        // Asignamos si a cada ciudad le falta o le sobra el producto
        int unidades1 = prods_ciudad[id_prod].first - prods_ciudad[id_prod].second;
        int unidades2 = c.consultar_reserva(id_prod) - c.consultar_faltante(id_prod);
        Producto p = inv.devolver_producto(id_prod);

        if (unidades1 > 0 and unidades2 < 0) {
            // Multiplicamos por -1 para obtener el valor absoluto de las unidades
            unidades2 = -1 * unidades2;
            int venta;
            // Si necesita más de las que puede dar, da todas las que puede, sino da todas las que necesita
            if (unidades1 < unidades2) venta = unidades1;
            else venta = unidades2;

            peso_total -= p.consultar_peso() * venta;
            volumen_total -= p.consultar_vol() * venta;
            prods_ciudad[id_prod].first -= venta;

            c.anadir_prod_reserva(p, venta);
        }
        else if (unidades1 < 0 and unidades2 > 0) {
            // Repetimos el proceso anterior pero cambiando unidades1 por unidades2
            unidades1 = -1 * unidades1;
            int compra;
            if (unidades2 < unidades1) compra = unidades2;
            else compra = unidades1;

            peso_total += p.consultar_peso() * compra;
            volumen_total += p.consultar_vol() * compra;
            prods_ciudad[id_prod].first += compra;

            c.quitar_prod_reserva(p, compra);
        }
    }
}
```

Si existe algún producto en el inventario de la ciudad, se hace mínimo una iteración (1 por cada producto distinto del inventario).

Para comerciar hay dos condiciones: Si al parámetro implícito le sobran unidades y a la ciudad c le faltan o si al parámetro implícito le faltan unidades y a la ciudad c le sobran. Para representarlo, se restan las unidades que tiene cada ciudad en reserva menos las que necesita, que dan los productos que les sobran. Si este número es negativo (unidades1 < 0, unidades2 < 0) significa que la ciudad tiene menos productos de los que necesita, y por tanto le faltan productos. Por otro lado, si este número es positivo quiere decir que le sobran unidades. Si da 0, quiere decir que ni le faltan ni le sobran o que la ciudad no tiene ese producto en su inventario, y por tanto no se comercia.

Una vez entrado a la condición cumplida, se multiplica por -1 el parámetro negativo para hacer su valor absoluto, en unidades. Como no se pueden comprar ni vender más unidades de las que hay disponibles, se escoge el parámetro que en valor absoluto sea menor.

Para realizar la compraventa, se incorpora o retira el nuevo peso y volumen de lo comprado o vendido, para actualizar correctamente las ciudades. Se usan los métodos de la clase ciudad *quitar_prod_reserva(const Producto& p, int unidades)* para quitar unidades de un producto p y *anadir_prod_reserva(const Producto& p, int unidades)* para realizar la acción contraria a la anterior, añadir unidades de un producto p al inventario de la ciudad correspondiente.

Pasando eso en cada bucle y cubriendo todas las posibilidades, al llegar al final del inventario finaliza el bucle y por tanto finaliza la función, habiendo comerciado correctamente si se diera el caso.

JUSTIFICACIÓN DE LA PRÁCTICA

Hacer viaje

- CONTEXTO DE LA FUNCIÓN

Esta función de carácter mayoritariamente recursivo pertenece a la clase **Río** y se llama desde el programa principal poniendo como input *hacer_viaje* o *hv*. Está compuesta por un esquema interno que incluye la llamada a dos funciones auxiliares: una que planea la ruta por donde se ha de realizar el viaje, de carácter recursivo, y otra que se encarga de realizar la compraventa con el barco y las ciudades, almacenadas en una pila. Las dos se encuentran entre los atributos privados del parámetro implícito.

```
/** @brief Hacer viaje
    El barco va desde la desembocadura hasta los nacimientos de los afluentes buscando rutas
    para comprar y vender. Seguirá la ruta donde pueda comprar y vender más productos
    \pre "cierto"
    \post Devuelve el número de productos que el barco ha comprado y vendido por la ruta
*/
int hacer_viaje(Barco& b, const Inventario& inv);
```

La función que justificamos a continuación será *planear_viaje*, y está en la parte privada de la implementación de la clase *Río*. Usando como parámetros de entrada un árbol binario con las ciudades, un barco, una pila para guardar las ciudades de la ruta y dos enteros por referencia, esta función devuelve el número de productos que se compran y venden en esa ruta. Este resultado luego no se acaba usando en la función *hacer_viaje*, pero es útil a la hora de hacer las llamadas recursivas, y esa es la razón por la cual devuelve un entero.

```
int Rio::hacer_viaje(Barco& b, const Inventario& inv) {
    // Reinicializamos la pila de la ruta
    stack<string> vacio;
    ruta = vacio;
    // Planeamos la ruta del viaje
    int c1 = b.consultar_prod_compra();
    int c2 = b.consultar_prod_venta();
    int d = planear_viaje(cuenca, b, ruta, c1, c2);
    // Calculamos los productos vendidos despues de hacer el viaje y los devolvemos
    d = hacer_viaje_priv(b, ruta, inv);
    return d;
}
```

Los enteros c1 y c2 tienen la cantidad de productos que puede comprar y vender el barco, para no tener que modificar el barco desde *planear_viaje*. El verdadero número que es retornado es el que devuelve la otra función auxiliar, *hacer_viaje_priv*. Debido al carácter iterativo de ésta última, no se justifica aquí.

```
/** @brief Planear ruta sobre la cuenca
    \pre "t" y "b" no están vacíos, "r" está vacío, compra_barco >= 0, venta_barco >= 0
    \post Se han añadido a "r" el nombre de las ciudades con las que el barco ha de comprar y vender productos,
          además de devolver el número de nodos que ha de visitar la ruta
*/
int planear_viaje(const BinTree<string>& t, const Barco& b, stack<string>& r, int& compra_barco, int& venta_barco);
```

- PRECONDICIÓN

El árbol del parámetro de entrada y el barco no están vacíos y están inicializados. La pila r entrará vacía, y almacenará *strings*.
compra_barco y *venta_barco* coinciden con la cantidad de productos que el barco ha de comprar y vender, y ambos son mayores o igual a cero.

- POSTCONDICIÓN

La pila se habrá modificado con la ruta que ha de seguir el barco para comprar y vender la máxima cantidad de productos posible. Si no ha de vender nada la pila seguirá vacía, pero si ha de comprar o vender algo tendrá almacenados en orden desde la desembocadura hacia el nacimientos de los ríos el nombre de las ciudades por las cuales ha de pasar. Además, esta función devuelve el número de productos que se pueden comprar y vender por la ruta adherida al stack.

- JUSTIFICACIÓN

En esta función tenemos dos casos base: que el nodo del árbol esté vacío o que el barco ya no pueda comprar ni vender más. En este caso, no se ha de añadir ninguna ciudad a la pila y tampoco se han de sumar unidades de compraventa.

```
// Caso base
if (t.empty()) return 0;
if (compra_barco == 0 and venta_barco == 0) return 0;
```

En el caso general, tenemos unos elementos determinados por recorrer, es decir, elementos finitos. Por tanto, usaremos el caso base para construir nuestra hipótesis de inducción:

- **TESIS DE INDUCCIÓN**: Si el nodo no está vacío entonces hay dos posibilidades:
  - El nodo no compra ni vende nada
  - El nodo compra y vende aunque sea, 1 producto

Si es el primer caso, dependerá de si los nodos anteriores compran o venden algo. Si es así, se añadirá a la ruta igualmente ya que será el nodo de la desembocadura. Si ninguno de los nodos anteriores compra o vende, entonces ese nodo no se añadirá a la ruta y devolverá 0 al acabar la función.
Si el nodo compra y vende algo, entonces éste se añadirá a la pila y se sumará lo que compra y vende a lo que han comprado y vendido los nodos anteriores en la ruta y se devolverá ese número.

- **HIPÓTESIS DE INDUCCIÓN**: Si el nodo está vacío o no puede comprar ni vender nada, devolverá 0.

.

```
// Calculo lo que tiene que comprar y vender el nodo actual
int compra = 0;
if (mapa_cuenca[id_ciudad].consultar_producto(id_prod_compra) and compra_barco > 0){
    if (sobrante_ciudad > 0 ) {
        compra = compra_barco;
        if (sobrante_ciudad < compra) {
            compra = sobrante_ciudad;
            compra_barco -= compra;
        }
        else compra_barco = 0;
    }
}
int venta=0;
if (mapa_cuenca[id_ciudad].consultar_producto(id_prod_venta) and venta_barco > 0){
    if (necesidad_ciudad > 0) {
        venta = venta_barco;
        if (necesidad_ciudad < venta) {
            venta = necesidad_ciudad;
            venta_barco -= venta;
        }
        else venta_barco = 0;
    }
}
```

Después de definir todas las variables locales de la función, se hace la compraventa entre el barco y el nodo que se está visitando. Si la función ha llegado a este punto, podemos asumir que por hipótesis de inducción el nodo no está vacío y el barco todavía puede comprar y vender algún producto.
Esta parte de la función compra o vende con la ciudad del nodo guardando en las variables *compra* y *venta* el mínimo valor entre los productos sobrantes de la ciudad y lo que el barco quiere comprar, para la variable *compra*, y los productos que la ciudad necesita y los que el barco ha de vender, para la variable *venta*. Como podemos ver, las variables *compra_barco* y *venta_barco* se actualizan, para las próximas llamadas recursivas (cada vez son más pequeñas).

```
// Miro cuantas unidades me entran de la ciudad izquiera y derecha por recursion
BinTree<string> tree_esq = t.left();
BinTree<string> tree_dre = t.right();
int compra_esq = compra_barco, compra_dre = compra_barco;
int venta_esq = venta_barco, venta_dre = venta_barco;
stack<string> aux_esq, aux_dre;
int prods_esq = planear_viaje(tree_esq, b, aux_esq, compra_esq, venta_esq);
int prods_dre = planear_viaje(tree_dre, b, aux_dre, compra_dre, venta_dre);
```

Ahora preparamos variables para las llamadas recursivas, una para la izquierda y una para la derecha.
Por hipòtesis de inducción, si la recursión en la izquierda y la derecha del árbol se cumple tal como se espera por tesis, entonces *prods_esq* y *prods_dre* contendrán el número de productos que se pueden comprar y vender por cada ruta posible, izquierda y derecha respectivamente.

```
// Defino la variable unidades que almacena cuantos productos se compran y se venden
int unidades = compra+venta;
// Si hay más productos en la izquierda guardo la recursion izquierda
if (prods_esq > prods_dre) unidades += planear_viaje(tree_esq, b, r, compra_barco, venta_barco);
// Si hay más en la derecha se guarda la recursion derecha
else if (prods_esq < prods_dre) unidades += planear_viaje(tree_dre, b, r, compra_barco, venta_barco);
// Si hay el mismo numero se queda el camino mas corto, que en caso de empate es el izquierdo
else if (aux_esq.size() > aux_dre.size()) unidades += planear_viaje(tree_dre, b, r, compra_barco, venta_barco);
else unidades += planear_viaje(tree_esq, b, r, compra_barco, venta_barco);
```

En la variable unidades almacenaremos los productos que se pueden comprar y vender en el árbol por la ruta que marca la pila r.
Las condiciones para escoger el nodo izquierdo y derecho definen qué ruta será escogida. En este caso, vemos que están todas las posibilidades cubiertas y por tanto no hay ningún camino indefinido.

```
// Si el nodo actual no compra ni vende unidades se retorna las unidades de la recursion
if (compra+venta == 0 and unidades == 0) return unidades;
else {
    ++unidades;
    r.push(t.value());
    return unidades;
}
```

Justo después de eso, llega la hora de hacer un return, que confirma la tesis de inducción.
Al haber un return bien puesto, se confirma que las llamadas recursivas al árbol derecho e izquierdo tiene final y son finitas: hasta que el nodo esté vacío o no pueda comprar ni vender nada (hipótesis de inducción). Por tanto, queda demostrado que la función *planear_viaje* termina de forma segura al terminar las llamadas recursivas y terminar la llamada principal.