## 1. Método iterativo: Comerciar

**Precondición:** `l1=L1` (inventario del parámetro implícito) y `l2=L2` (inventario de Ciudad &c).

**Postcondición:** `l1` y `l2` son los inventarios resultantes de haber comerciado todos los productos con la otra ciudad.

**Coste:** Lineal respecto al tamaño más pequeño en relación al parámetro impícito y la Ciudad c.

**Invariante:** `it1` apunta a un elemento de `inventario` o a `inventario.end()`. Cada producto de `inventario` des del inicio hasta la posición anterior a la que marca `it1` ya ha sido comerciado con la Ciudad c. A partir de `it1` hasta el final del inventario los productos aún no han sido comerciados. La invariante se define de manera análoga para `it2` i `c.inventario`.

**Función de cota:** A cada iteración decrece el tamaño de la sublista entre `it1` y `inventario.end()` o entre `it2` y `c.inventario.end()`.

```
void Ciudad::comerciar(Ciudad &c) {
  map<Producto, Atributos>::iterator it1 = inventario.begin();
  map<Producto, Atributos>::iterator it2 = c.inventario.begin();
  while (it1 != inventario.end() and it2 != c.inventario.end())
  {
    if (it1->first < it2->first) ++it1;
    else if (it2->first < it1->first) ++it2;
    else // if (it1->first == it2->first)
    {
      int dar = it1->second.posee - it1->second.necesita;
      int rec = it2->second.necesita - it2->second.posee;
      if (dar > 0 and rec > 0) rec = min(dar, rec);
      else if (dar < 0 and rec < 0) rec = max(dar, rec);
      else rec = 0;

      it1->second.posee -= rec;
      it2->second.posee += rec;

      peso_total -= it1->first.consultar_peso()*rec;
      volumen_total -= it1->first.consultar_volumen()*rec;
      c.peso_total += it2->first.consultar_peso()*rec;
      c.volumen_total += it2->first.consultar_volumen()*rec;
      ++it1;
      ++it2;
    }
  }
}
```

**Justificación del método con respecto a la invariante.**

1 **Cuando se llega al bucle por primera vez, la invariante se cumple.**
Cuando se llega al bucle por primera vez, `it1=inventario.begin()` y `it2=c.inventario.begin()`. Por lo tanto, tal y como dice la invariante aún no se ha comerciado ningún producto entre las dos ciudades.

2 **Suponemos la invariante, y vemos que la condición del bucle se evalua en cierto. Entonces, después de ejecutar el cuerpo del bucle, la invariante se vuelve a cumplir.**
Si los productos apuntados por `it1` e `it2` son el mismo se calculan los sobrantes para el inventario del parámetro implícito y el inventario de la Ciudad c (`int dare` `int rec`, respectiv amente). Entonces, se calcula quien tiene que dar y recibir y las cantidades de productos y se realiza el cambio (asignaciones al mínimo/máximo/cero y resta/suma de la variable `rec`). A continuación se reajustan los pesos y volúmenes totales para ambas ciudades. Finalmente se mueven `it1` e `it2` a la siguiente posición, y entonces se cumple la invariante de nuevo.
Si los productos apuntados por `it1` e `it2` se mueve `it1` o `it2` a la siguiente posición, según el producto apuntado con identificador menor, y se procede a cumplir de nuevo la invariante.

3 **Suponemos la invariante, y que la condición de entrada al bucle falla. Entonces, vemos que la postcondición se cumple.**
Que la condición de entrada al bucle falle implica que o bien `it1` o bien `it2` son iguales al final de su respectiva lista. Siguiendo la invariante esto quiere decir que se ha iterado por todos los productos de la lista con menor tamaño, y todos ellos (los que también estan en la otra lista) han sido comerciados con la otra ciudad. Por tanto la Postcondición se cumple.

4 **Suponemos la invariante, y que la condición de entrada al bucle se cumple. Entonces, después de ejecutar el cuerpo del bucle la función de cota se evalua a algo estrictamente menor.**
A cada iteración del bucle o bien se mueve `it1` o bien `it2` a la siguiente posición de la lista. Cosa que hace que la función de cota se evalúe a algo estrictamente menor.

5 **La invariante implica que la función de cota no se evalua a algo menor que 0. El bucle no es infinito**
La función de cota se evalua en cero cuando `it1=inventario.end()`. Siguiendo la invariante la función de cota no puede evaluarse a nada menor debido a que `it1` no puede apuntar a nada más allá de la dirección de memoria de`inventario.end()`. Se define de maner a análoga para `it2`.

## 2. Método recursivo: Hacer Viaje

**Precondición:** comprar >= 0 y vender >= 0

**Postcondición:** La pila *ruta* es la ruta más corta y más a la derecha (mirando río arriba) en la que el barco vende/compra todos sus productos, si no se cumple, en la que vende/compra más productos.

**Coste:** Lineal respecto al número de ciudades

**Hipótesis de Inducción:** *ruta_izquierda* y *ruta_derecha* cumplen ambos la Postcondición del árbol, pero solo hasta el nodo actual.

**Función de cota:** En cada paso recursivo el tamaño (número de nodos) del árbol *t* se hace más pequeño, hasta llegar al árbol vacío (caso base).

```
void Cuenca::buscar_ruta_recursiva(BinTree<string> t, stack<string> &ruta,
      int &comprar, int &vender) const {
  if (not t.empty())
  {
    int idc = barco.consultar_id_comprar();
    int idv = barco.consultar_id_vender();
    if (ciudades.at(t.value()).tiene_producto(productos.at(idc)))
    {
      pair<int, int> atributos = ciudades.at(t.value()).consultar_producto(
                    productos.at(barco.consultar_id_comprar()));
      // posee - necesita
      int diferencia = atributos.first - atributos.second;
      if (diferencia > 0) comprar -= min(diferencia, comprar);
    }
    if (ciudades.at(t.value()).tiene_producto(productos.at(idv)))
    {
      pair<int, int> atributos = ciudades.at(t.value()).consultar_producto(
                    productos.at(barco.consultar_id_vender()));
      // necesita - posee
      int diferencia = atributos.second - atributos.first;
      if (diferencia > 0) vender -= min(diferencia, vender);
    }

    // nota: siempre comprar >= 0 y vender >= 0
    if (comprar > 0 or vender > 0)
    {
      int ci = comprar;
      int vi = vender;
      stack<string> ruta_izquierda;
      buscar_ruta_recursiva(t.left(), ruta_izquierda, ci, vi);

      int cd = comprar;
      int vd = vender;
      stack<string> ruta_derecha;
      buscar_ruta_recursiva(t.right(), ruta_derecha, cd, vd);
```

```
      if (not (comprar == ci and ci == cd and vender == vi and vi == vd))
      {
        if (ci + vi < cd + vd)
        {
          comprar = ci;
          vender = vi;
          ruta = ruta_izquierda;
        }
        else if (cd + vd < ci + vi)
        {
          comprar = cd;
          vender = vd;
          ruta = ruta_derecha;
        }
        else // if (ci + vi == cd + vd)
        {
          if (ruta_izquierda.size() <= ruta_derecha.size())
          {
            comprar = ci;
            vender = vi;
            ruta = ruta_izquierda;
          }
          else
          {
            comprar = cd;
            vender = vd;
            ruta = ruta_derecha;
          }
        }
      }
      ruta.push(t.value());
    }
}
```

**Justificación del método con respecto a la hipótesis de inducción.**

1 **Suponemos que la Precondición es cierta al principio de la ejecución de la función y que las instrucciones nos traen los casos directos. Entonces se cumple la Postcondición:**
Tenemos dos casos base:
1. **El árbol está vacío**: El método no hace nada y se cumple la Post.
2. Después de ejecutar las instrucciones iniciales, donde se ajustan los valores de `comprar` y `vender` con respecto a la Ciudad del nodo actual **`comprar=0` y `vender=0`**. En este punto se han comprado y vendido todas las unidades que el barco quería comprar/vender y por tanto solo queda añadir la Ciudad (nodo) actual a la cima de la pila `ruta` para que se cumpla la Postcondición.

2 **Suponemos que la Precondición se cumple al principio de ejecutar la función y que las instrucciones nos llevan al caso recursivo. Entonces vemos que se cumple la Precondición adaptada a los parámetros de la llamada.**
La precondición se cumple ya que el caso recursivo se ejecutará si solo si`comprar > 0` o `vender > 0`.

3 **Suponemos que la Precondición se cumple al principio de la ejecución de la función y que las instrucciones nos traen al caso recursivo. Entonces, suponiendo que por Hipótesis de Inducción la Postcondición adaptada a los parámetros se cumple y que se ejecutan las instrucciones restantes después de la llamada recursiva, la Postcondición se cumple.**
En el caso que no se haya comprado o vendido nada más allá del nodo actual, se añade la Ciudad (nodo) actual a la cima de la pila `ruta` para que se cumpla la Postcondición.
De manera contraria, si se ha comerciado con el barco más allá del nodo actual se asigna la ruta en la que se haya comerciado más con el barco en total (ya sea la ruta del árbol izquierdo o derecho) a la ruta actual. Entonces, después de haber ajustado el número restante de productos para comprar y vender se añade la Ciudad (nodo) actual a la cima de la pila `ruta` para que se cumpla la Postcondición.
Finalmente, cabe la posibilidad de que ambas en la ruta izquierda y derecha queden al barco le queden el mismo número de productos para comprar y vender (en total). En este caso se asigna la ruta más corta de ambas a la ruta actual. Entonces, después de haber ajustado el número restante de productos para comprar y vender se añade la Ciudad (nodo) actual a la cima de la pila `ruta` para que se cumpla la Postcondición.

4 **Suponemos que la Precondición es cierta al principio de la ejecución de la función y que las instrucciones nos llevan al caso recursivo. Entonces vemos que la función de cota para los parámetros de la llamada es menor que la función de cota para los parámetros iniciales.**
Si se ejecuta el caso recursivo el parámetro de llamada sobre el que se realiza la recursión, el árbol `t`, se divide en `t.left()` y `t.right`. Entonces, en ser el parámetro de llamada para la siguiente recursión uno de estos dos árboles, con tamaño de una unidad menor respecto al árbol inicial, estos se acercan más al tamaño de un árbol vacío (condición que se ha de cumplir para la que la función de cota se evalúe a cero).

5 **Suponemos la Precondición. Entonces la función de cota se evalúa a mayor o igual que cero.**
La función de cota se evalúa al tamaño de un árbol, que por la naturaleza de éste solo puede ser >= 0. La mínima cantidad de nodos que pueden existir es que no exista ninguno, y por tanto que el árbol no exista (la función de cota se evalúa a cero).