Justificación de la Función Iterativa comerciar_prod

```
void Ciudad::comerciar_prod(Ciudad& otra_ciudad, const Cjt_productos& p ){
   iterador it1 = productos.begin();
   iterador it2 = otra_ciudad.productos.begin();
   // Iteramos sobre los productos de ambas ciudades
   while(it1 != productos.end() and it2 != otra_ciudad.productos.end()) {

/* \pre Las ciudades existen en la cuenca.
       Los inventarios de ambas ciudades no están vacíos.
   \post Se efectúa el intercambio de productos entre ciudades dependiendo de las
necesidades de cada una.
*/

       if (it1->first < it2->first) {
          ++it1; // Avanzamos en el inventario de la primera ciudad
       }
       else if (it1->first > it2->first) {
          ++it2; // Avanzamos en el inventario de la otra ciudad
       }
       else {
          // Calculamos la cantidad que necesita cada ciudad
          int cant1 = it1->second.second - it1->second.first;
          int cant2 = it2->second.second - it2->second.first;
          if (cant1 * cant2 < 0) { // Si las necesidades son opuestas
             int intercambio = min(abs(cant1), abs(cant2)); // Calculamos la cantidad a
intercambiar
             bool poner = false;
             if (cant1 < cant2) {
                it1->second.first -= intercambio; // Actualizamos la primera ciudad
                comerciar_ajustes(it1->first, intercambio, poner, p);
                poner = true;
                it2->second.first += intercambio; // Actualizamos la otra ciudad
                otra_ciudad.comerciar_ajustes(it1->first, intercambio, poner, p);
             }
             else {
                poner = true;
                it1->second.first += intercambio; // Actualizamos la primera ciudad
                comerciar_ajustes(it1->first, intercambio, poner, p);
                poner = false;
                it2->second.first -= intercambio; // Actualizamos la otra ciudad
                otra_ciudad.comerciar_ajustes(it1->first, intercambio, poner, p);
             }
          }
          ++it1; // Avanzamos en ambos inventarios
          ++it2;
```

```
      }
   }
}

void Ciudad::comerciar_ajustes(const int& ident_prod, int intercambio, bool& poner, const
Cjt_productos& p) {
      int peso_prod = p.peso(ident_prod);
      int vol_prod = p.volumen(ident_prod);
      if(poner){
         peso_total += abs(intercambio)*peso_prod;
         volumen_total += abs(intercambio)*vol_prod;
      }
      else{
         peso_total -= abs(intercambio)*peso_prod;
         volumen_total -= abs(intercambio)*vol_prod;
      }
   }
```

Inicialización

Antes de la primera iteración del bucle while:

- it1 apunta al primer producto del inventario de productos de la primera ciudad.
- it2 apunta al primer producto del inventario de productos de la otra ciudad.
- Ningún producto ha sido procesado aún.

Estas inicializaciones aseguran que estamos listos para empezar a intercambiar productos entre las dos ciudades. Al entrar al while nos hemos asegurado de que hay productos en ambos inventarios de las ciudades, satisfaciendo así la invariante.

Condición de Salida

El bucle while termina cuando se cumpla alguna de estas dos condiciones:

- it1 alcanza el final del inventario de productos de la primera ciudad (it1 == productos.end()).
- it2 alcanza el final del inventario de productos de la otra ciudad (it2 == otra_ciudad.productos.end()).

Esto asegura que todos los productos hayan sido revisados y procesados correctamente.

Cuerpo del Bucle

En cada iteración del bucle:

1. Se comparan los productos apuntados por it1 y it2:
    - Si it1->first < it2->first, se avanza it1 al siguiente producto en la primera ciudad.
    - Si it1->first > it2->first, se avanza it2 al siguiente producto en la otra ciudad.
    - Si los productos son iguales (it1->first == it2->first), se calcula la cantidad necesaria para cada ciudad (cant1 y cant2).
2. Si las necesidades de los productos son opuestas (cant1 * cant2 < 0):
    - Se calcula la cantidad a intercambiar (intercambio) como el mínimo entre las necesidades absolutas.

- Dependiendo de qué ciudad necesita más del producto, se actualizan los inventarios y se realizan ajustes de comercio (comerciar_ajustes).

3. Se avanza en ambos iteradores (it1 y it2).

Finalización

El bucle finaliza cuando uno de los inventarios ha sido completamente recorrido:

- En este punto, todos los productos han sido comparados e intercambiados según las necesidades de las ciudades.
- Los inventarios y los totales de peso y volumen han sido actualizados según los cambios que se han realizado.

Justificación de la Función Recursiva Calcular_ruta_rec

```
int Cjt_rios::calcular_ruta_rec(vector<string>& mejor_ruta, Barco& b, int ya_comprado, int ya_vendido, const BinTree<string>& a) {
    //ya_comprado es lo que mis ascendientes ya han comprado
    //ya_vendido es lo que mis ascendientes ya han vendido
    // return tiene q ser lo que yo y mis hijos hemos comprado + lo que hemos vendido
    int max_compra = b.consultar_cantidad_compra();
    int max_venta = b.consultar_cantidad_venta();
    if(not a.empty() and (ya_comprado < max_compra or ya_vendido < max_venta)){
        //interaccion con la ciudad, a.value(), --> c, v a la ciudad (teniendo en cuenta el maximo del barco)
        map<string,Ciudad>::iterator it =  ciudades.find(a.value());
        int comprado_aqui = it->second.barco_puede_comprar(b.consultar_id_prod_comp(), (max_compra-ya_comprado));
        ya_comprado += comprado_aqui;
        int vendido_aqui = it->second.barco_puede_vender(b.consultar_id_prod_vend(), (max_venta-ya_vendido));
        ya_vendido += vendido_aqui;
        vector<string> ruta1, ruta2;
        int total1 = calcular_ruta_rec(ruta1,b,ya_comprado,ya_vendido,a.left());
        int total2 = calcular_ruta_rec(ruta2,b,ya_comprado,ya_vendido,a.right());
        //comparaciones de las dos rutas 1 y 2
        if (total1 > total2 or (total1 == total2 and ruta1.size() <= ruta2.size())) {
            mejor_ruta = ruta1;
            int total = total1 + comprado_aqui + vendido_aqui;
            if(total > 0) mejor_ruta.insert(mejor_ruta.begin(), a.value());
            return total;
        }
        else {
            mejor_ruta = ruta2;
            int total = total2 + comprado_aqui + vendido_aqui;
            if(total > 0) mejor_ruta.insert(mejor_ruta.begin(), a.value());
            return total;
        }
    }
```

```
    return 0;
}
```

Caso Sencillo

Si el árbol a está vacío o si el barco ha alcanzado las capacidades máximas de compra y venta (**ya_comprado >= max_compra** y **ya_vendido >= max_venta**):

1. La función devuelve 0 por alguna de las siguientes razones:
    - No hay más ciudades que visitar (a.empty()).
    - Ya se ha alcanzado la capacidad máxima del barco para comprar y vender productos (ya_comprado es mayor o igual a max_compra y ya_vendido es mayor o igual a max_venta).

Caso Recursivo

Si el subárbol izquierdo está vacío, tal como dicta el enunciado de la práctica, el subárbol derecho también lo estará. En caso de que el árbol no esté vacío y aún hay capacidad para comprar o vender:

1. Interacción con la ciudad actual:
    - Se busca la ciudad actual en el mapa ciudades usando a.value().
    - Se calcula comprado_aqui como la cantidad que el barco puede comprar en la ciudad actual.
    - Se actualiza ya_comprado sumando comprado_aqui.
    - Se calcula vendido_aqui como la cantidad que el barco puede vender en la ciudad actual.
    - Se actualiza ya_vendido sumando vendido_aqui.
2. Llamadas Recursivas:
    - Se calculan las rutas y totales para los subárboles izquierdo (total1 y ruta1) y derecho (total2 y ruta2) mediante llamadas recursivas:

Comparación de Rutas:

- Se compara total1 y total2.
- Se elige la ruta con el mayor total. En caso de empate, se elige la ruta con el camino más corto, como dice el enunciado de la práctica.
- La ruta elegida (mejor_ruta) se actualiza con la ciudad actual si el total es mayor que 0:

Condiciones

Después de procesar cada nodo, se realizan varias condiciones para determinar la mejor ruta:

1. Que no hayamos llegado a la capacidad máxima:
    - Continuar procesando si ya_comprado < max_compra o ya_vendido < max_venta.
2. Interacción con la ciudad actual:
    - Se realizan compras y ventas limitadas por la capacidad restante del barco.
3. Comparación de rutas:
    - Se elige la ruta con el mayor total de productos comprados y vendidos.
    - En caso de empate, se elige la ruta con el más corta.

Decrecimiento

En cada llamada recursiva, el algoritmo procesa un subárbol más pequeño, asegurando que eventualmente se llegue a un nodo vacío:

- El árbol a, se divide en subárboles izquierdo y derecho, que son más pequeños que el árbol original, repitiendo este proceso hasta que se llega al nodo vacio. Al encontrar este nodo vacío, la función devuelve un 0.