# Justificacíon de las operaciones de la práctica de PRO2

Las operaciones en cuestion son "comerciar" y "hacer_viaje", su equivalente en mi practica son `City::trade` y `River::do_trip` .

## Comerciar

La operacion iterativa esta programada con la fucion `City::trade()` . Su implementacion es la siguiente:

```
void City::trade(const Catalogue& catalogue, City& city){
  // PRE: this != city
  auto inv1It = this->_inventory.begin(); //por convenencia, ivn1it sera t                  //
  auto inv2It =  city._inventory.begin(); //por convenencia, inv2It sera c                  //
                                                                                            //
  const auto inv1ItEnd = this->_inventory.end();//por convenencia, inv1ItEnd sera fc        //      A
  const auto inv2ItEnd =  city._inventory.end();//por convenencia, inv1ItEnd sera ft        //
                                                                                            //
  int delta_weight = 0;                                                                     //
  int delta_volume = 0;                                                                     //
  /* Por definicion de la estructura de _inventory, todo elemento de inventory es un Market i
     relacionado a un producto,
  *     es decir una tupla de las cantidades que tiene y que quiere la ciudad de ese inventory,
  *     i.e (tiene, quiere).
  * Similarmente los iteradores (i1, i2, f1 y f2) de _inventory tienen la siguiente forma
  *  ( identificador del producto, ( tiene, quiere ) ).
  *
  */

  // Por convenencia definimos la proposiciones siguientes con a y b iteradores de inventarios
  // -    A(a,b) := «a es un producto anterior a b» <=> a[0] < b[0]
  // -    S(a)   := «Sobra producto a» <=> (a[1][0] - a[1][1]) >= 0
  // -    N(a)   := «Se necesita producto a» <=> (a[1][0] - a[1][1] <= 0 )
  // -    P(a,b) := « a es el mismo producto que b » <=> a[0] == a[1]
  // -    C(a)   := « a esta en el inventario de ambas ciudades »
  // -    E(a)   := «a esta equilibrado » <=>
  //              -> ( S(a) and S(b) ) or ( N(a) and N(b) )
  //  Cabe recalcar que:
  //  -  P(a,b) <=> P(b,a)
  //  -  P(a,b) => E(a) <-> E(b)
  //  -  P(a,b) => not A(a,b) and not A(b,a)

  // D := «delta_weight y delta_volume representan la diferencia de peso y de volumen total
  // de this»
```

```
// Invariante: I :=  D
//                and (para todo producto x anterior a t o a c, C(x) -> E(x) ) // X

//D es cierto porque no se ha hecho nada aun
//X es cierto porque no hay producto anterior que cumpla C(x)
while(inv1It != inv1ItEnd and inv2It != inv2ItEnd){//Guarda := B  := t != ft and c != fc

  if(inv1It->first == inv2It->first){ // Condicion: C1 := t[0] == c[0] <=> P(t, c)  and
    // C(t) and C(c)
    int productId = inv1It->first;
    int thisSurplus = inv1It->second.first - inv1It->second.second;
    int citySurplus = inv2It->second.first - inv2It->second.second;
    // => C1 and D and B
    if(thisSurplus > 0 and citySurplus < 0){// Condicion: C2 := S(t) and N(c)
      int given = min(thisSurplus,-citySurplus);
      inv1It->second.first -= given;
      inv2It->second.first += given;
      //=> S(t) and S(c)
      delta_weight -= given*catalogue.get_product(productId).first;
      delta_volume -= given*catalogue.get_product(productId).second;
      //=> D

      //=> S(t) and S(c) and C(c) and C(j) and D

    }else if(thisSurplus < 0 and citySurplus > 0){// Condicion: C3 := N(a)
      int given = min(-thisSurplus,citySurplus);
      inv1It->second.first += given;
      inv2It->second.first -= given;
      //=> S(t) and S(c) => E(t) => E(j)
      delta_weight += given*catalogue.get_product(productId).first;
      delta_volume += given*catalogue.get_product(productId).second;
      //=> D

      //=> S(t) and S(c) and C(c) and C(j) and D
    }// Else: N(t) and N(c) or S(t) and S(c)
    // => Else or (S(t) and S(c) or S(t) and S(c)) and D => E(c) and E(t) and D
    ++inv1It;// <= B
    ++inv2It;// <= B
    // A(t-1,t) and A(c-1,t) and E(c-1) and E(t-1) and D and B => I
  } else// not C1 = not P(t, c)

    if(inv1It->first < inv2It->first) //Condicion: C4 := A(t,c)
    //A(t,c)
    ++inv1It;//<= B
          // Dos casos:
           // A(t+1,c): A(t+1,c) and I and D and B => I // D no ha cambiado,
//                y B no nos afecta
           // P(t+1,c): P(t+1,c) and I and D and B => I // idem
   else/*if(inv1It->first > inv2It->first)*/// Condicion: A(c,t)
    ++inv2It; //<= B
      //Dos casos:
          //A(c+1,t): A(c+1,t) and I and D and B => I // idem
```

```
                //P(c+1,t): P(c+1,t) and I and D and B => I // idem
    }
    // not B and I
    // => Para todo producto x anterior a fc o ft si C(x) -> E(x) => Para todo x,
    // producto, C(x) -> E(x) puesto que no existe producto que cumpla C(x)
    // y sea anterior a fc y ft

    this->_total_product.first  += delta_weight; //
    this->_total_product.second += delta_volume; //
                                    // and D => T
     city._total_product.first  -= delta_weight; //
     city._total_product.second -= delta_volume; //
    // T and not B and I => POST
    // POST: Todos los cambios de volumen y peso han sido registrados and
    // Para todo x, producto, C(x) -> E(x)
}
```

### Justificacion de finalizacion

En cada iteracion decrece la suma de la distancia entre inv1It y inv1ItFin mas inv2It y inv2ItFin. Porque siempre incrementamos como minimo uno de los dos inv1It o inv2It

## Hacer viaje

La operacion auxiliar esta programada con la funcion `River::_find_best_path` y su implementacion es la siguiente:

```
River::Path River::_find_best_path(Ship ship, const Catalogue& catalogue, const string& cityId){
    // PRE: cityId es una ciudad valida o "#" para indicar que no hay ciudad
    // POST: Se devuelve la mejor ruta (River::Path) desde cityId o un camino vacio
    // si no se ha podido vender o comprar algo

    /* Segun el enunciado
    * - Una ruta acaba en la ciudad más alejada de la desembocadura en la que haya
       comprado o vendido algo.
    * - [La mejor ruta es la mas provechosa]
    * - Si dos rutas son igual de provechosas, se escoge la más corta en número de ciudades
       visitadas;
    * - Si tienen la misma longitud se escoge la que pasa por la ciudad que está a mano
       derecha mirando río arriba.
    * - [El provecho es] el numero de productos vendidos y comprados

    * Segun la definicion de River::Path,
    * La longitud de la ruta se guarda en length
    * El provecho de la ruta se guarda en totalTransacion.
    * La ultima ciudad de una ruta de guarda en finalCity
    *
    * Segun la definicion de Ship
```

```
* la cantidad que el barco puede comprar se obtiene con get_demand_amount()
* La cantidad que el barco puede  vender se obtiene con get_supply_amount()
*
*
* Esto implica que:
* - si el provecho es 0, no se han podido vender o comprar productos
*/
if(cityId == "#")return River::Path(); // Si no hay ciudad, no se puede vender o
// comprar algo, devolvemos camino vacio => POST
// NOT cityId == "#" AND PRE => cityId es una ciudad valida
if(ship.get_supply_amount() == 0 and ship.get_demand_amount() == 0)return River::Path();
// El barco no puede ni comprar ni vender, no se puede vender o comprar algo,
// devolvemos camino vacio => POST

int transactionSum = _transaction(ship, catalogue, cityId, false, true);
// transacionSum = el mayor numero de productos que se pueden comprar y vender en esta
// ciudad al barco
River::Path  leftPath = _find_best_path(ship, catalogue, get_city(cityId).get_left());
                    // cityId es una ciudad valida => PRE de get_city(cityId)
                    // POST get_left() => su return  es o una ciudad valida o "#"
//                      si no hay ciudad => PRE de _find_best_path
                    // POST de _find_best_path => su return es el mejor camino
//                      desde la ciudad a la izquierda de esta
River::Path rightPath = _find_best_path(ship, catalogue, get_city(cityId).get_right());
                    // cityId es una ciudad valida => PRE de get_city(cityId)
                    // POST get_right() => su return es o una ciudad valida o "#"
//                   si no hay ciudad => PRE de _find_best_path
                    // POST de _find_best_path => su retun es el mejor camino desde
//                      la ciudad a la derecha de esta

bool chooseLeft;
if(leftPath.totalTransaction == rightPath.totalTransaction)
   // Ambos caminos son igual de provechozos
    chooseLeft = leftPath.length         < rightPath.length; // true si el de la
//izquierda es mas corto, false si el de la dercha es mas corto o si son igual de largos
else chooseLeft = leftPath.totalTransaction > rightPath.totalTransaction;// true si
// el de la izquierda es mas provechozo, falso si el de la derecha es mas provechozo

// => chooseLeft == true si el de la izquierda es mejor camino, chooseLeft == false
// si el de la derecha es mejor camino accessible desde cityId

if(chooseLeft){
   // chooseLeft: La izquierda es mejor camino
   if(transactionSum == 0){
      //If: En esta ciudad no se ha podido vender o comprar algo
      if(leftPath.totalTransaction == 0){
         //If: en el camino de la izquierda no se ha podido comprar o vender algo
         //En Esta ciudad no se ha podido vender o comprar algo AND en el
         // mejor sub camino no se ha podido vender o comprar algo => no se ha podido vender o comprar
         return River::Path();// AND devolvemos camino vacio => POST
      }else{
         //Else: se ha podido comprar algo
```

```
        // en esta ciudad no se ha podido vender o comprar algo
        // AND lefPath.totalTransacion es la transacion total del camino de la izquierda
        // AND chooseLeft
        // => El mejor camino tiene totalTransacion igual que el camino de la izquierda

        // chooseLeft
        // => desde esta ciudad el mejor camino acaba en la misma ciudad
        // que el mejor sub camino, el de la izquierda

        // chooseLeft
        // => finalCity del mejor camino es igual que el camino de la izquierda

        // PRE AND chooseLeft
        // => el mejor camino desde cityId pasa por cityId AND el mejor sub
        // camino es el de la izquierda
        // => length del mejor camino es length del camino de la izquierda mas 1
        leftPath.length += 1;
        //; : leftPath es el mejor camino
        return leftPath;//devolvemos el mejor camino and Else => POST
    }
}else{
    // Else1: En esta ciudad se ha podido vender o comprar algo
    if(leftPath.totalTransaction == 0){// En el mejor sub camino no se ha
        // podido vender o comprar algo
        // => M: es la primera vez que se ha podido vender o comprar algo
        River::Path path;
        path.totalTransaction = transactionSum; // A
        path.finalCity = cityId; // B
        path.length = 1; // C
        // M  AND transactionSum => el mejor camino tiene totalTransaction
        // igual a transactionSum
        // F: M AND definicion de mejor camino => el mejor camino acaba en esta ciudad
        // F AND definicion de River::Path => el mejor camino tiene
        // finalCity igual a esta ciudad
        // M AND definicion de River::Path => el mejor camino tiene length igual a 1
        // A AND B AND C AND M AND F AND definicion de River::Path => path
        // es el mejor camino
        return path;// devolvemos el mejor camino => POST
    }
    //Else2: En el mejor sub camino se ha podido vender o comprar algo
    //Else : Else1 AND Else2
    leftPath.totalTransaction += transactionSum; // A
    leftPath.length += 1; // B
    // Else AND transactionSum AND chooseLeft => el mejor camino tiene
    // transactionTotal igual al totalTransaction del mejor sub camino, el izquierdo, mas transactionSum
    // Else AND definicion de River::Path => el mejor camino tiene
    // length igual al length del mejor subcamino, el izquierdo, mas 1
    // chooseLeft
    // => finalCity del mejor camino es igual que el camino de la izquierda
    // A AND B AND Else AND transactionSum AND chooseLeft => leftPath es el mejor camino
    return leftPath;// devolvemos el mejor camino AND Else => POST
}
```

```
    }else{
        /**
        * El procedimiento aqui es el mismo que ariba, pero con el mejor sub camino
            siendo la derecha
        */
        if(transactionSum == 0){
            if(rightPath.totalTransaction == 0){
                return River::Path();
            }else{
                rightPath.length += 1;
                return rightPath;
            }
        }else{
            if(rightPath.totalTransaction == 0){
                River::Path path;
                path.totalTransaction = transactionSum;
                path.finalCity = cityId;
                path.length = 1;
                return path;
            }
            rightPath.totalTransaction += transactionSum;
            rightPath.length += 1;
            return rightPath;
        }
    }
}
```

## Justificacion de finalizacion de la operacion

Cada llamada se hace sobre un cityId que esta mas al sur (post de City::get_left() y City::get_right()) y por lo tanto se acerca mas a las fuentes, y por lo tanto cada llamada es sobre una parte del rio mas pequeña