Algorítmica Carlos Quesada Pérez

2º GII Miguel Ángel López Sánchez

Grupo B3 David Serrano Domínguez

**Práctica 3 - Algoritmos Greedy**

**Ejercicio 1**

**(1.1) Diseño de componentes**

* Lista de Candidatos (Abiertos): Sensores/nodos con conexión al actual (o a los antecedentes del actual) que aún no han sido visitados. En la implementación se ordenan de menor a mayor para facilitar su selección.
* Lista de Candidatos Usados (Cerrados): Sensores visitados de los cuales se han obtenido sus conexiones y la velocidad de dichas conexiones.
* Criterio de Selección: Se seleccionará el sensor con la mejor velocidad de transmisión (la menor) de entre los disponibles en cada momento.
* Criterio de factibilidad: Un sensor se insertará en la solución si pertenece a la red de nodos sensor que conecta al origen con el destino y, además, tiene la velocidad mínima de conexión.
* Función Objetivo: Minimizar el tiempo de velocidad de transmisión.

**(1.2) Diseño del algoritmo**

solución algoritmo(problema p, nodo Origen) {

C: Conjunto nodos Cerrados (sin repetidos), inicialmente vacío

A: Conjunto nodos Abiertos, ordenados por tiempo de transmisión de menor a mayor

nodo Actual = Origen

A := A UNION {Actual} // Se inserta el nodo Actual en Abiertos (sólo para la

// primera iteración).

while( A no esté vacío AND actual no sea destino) {

// Se extrae el nodo actual de abiertos, puesto que ya ha sido visitado, y se

// inserta en Cerrados

A := A - {Actual}

C := C UNION {Actual}

// Del grafo del problema, obtiene el resto de sensores con conexión hasta el

// actual y los inserta en la lista de Abiertos.

for(i:0 to p.numeroNodos) {

bool: tieneConexion := p.grafo(nodo actual, nodo i)

if (tieneConexion) A := A UNION {p.grafo(nodo i)}

}

// Eliminamos de Abiertos todos los nodos candidatos que se encuentran en

// Cerrados (ya han sido visitados).

while( {A.top} pertenece a C AND A no está vacía)

A := A - {A.top}

// Se comprueba primero si la lista de nodos candidatos, Abiertos, está vacía.

// En caso contrario, se escoge el siguiente nodo a procesar, mediante el

// criterio de selección escogido, el cual es el menor tiempo de transmisión.

// Como nuestro conjunto de nodos Abiertos está ordenado, esto implica

// escoger el primer nodo de la lista.

if ( A no está vacía )

Actual := A.top

}

// Si al salir del bucle, actual es el destino, hemos encontrado la solución. En caso contrario, la solución no ha sido encontrada

if (actual es destino)

return solucion valida

else

return solucion no válida

}

**(1.3) Estudio de optimalidad (demostración o contraejemplo)**

En este caso, el algoritmo propuesto e implementado cumple la condición de optimalidad, de encontrar el camino con la mínima velocidad de transmisión. Esto ocurre debido a que, de la forma en la que actúa (genera en cada iteración del algoritmo todos los posibles caminos desde el nodo procesado en cada momento) y la forma en que almacena los nuevos nodos procesados (los ordena de menor a mayor en una cola de prioridad según su velocidad total, partiendo del nodo de origen). El algoritmo continuará su ejecución hasta llegar a procesar el nodo objetivo. Si esto ocurre, el camino obtenido hasta llegar al objetivo será el de menor coste/velocidad puesto que se van procesando nodos y caminos de menor a mayor velocidad.

Por tanto, la idea es la siguiente: si en cada iteración el algoritmo procesa el camino más rápido desde el nodo/sensor origen hasta el nodo actual, en cuanto se procese el nodo/sensor destino, ya habrá obtenido su camino más veloz.

**(1.4) Ejemplo paso a paso de la explicación del funcionamiento del algoritmo para**

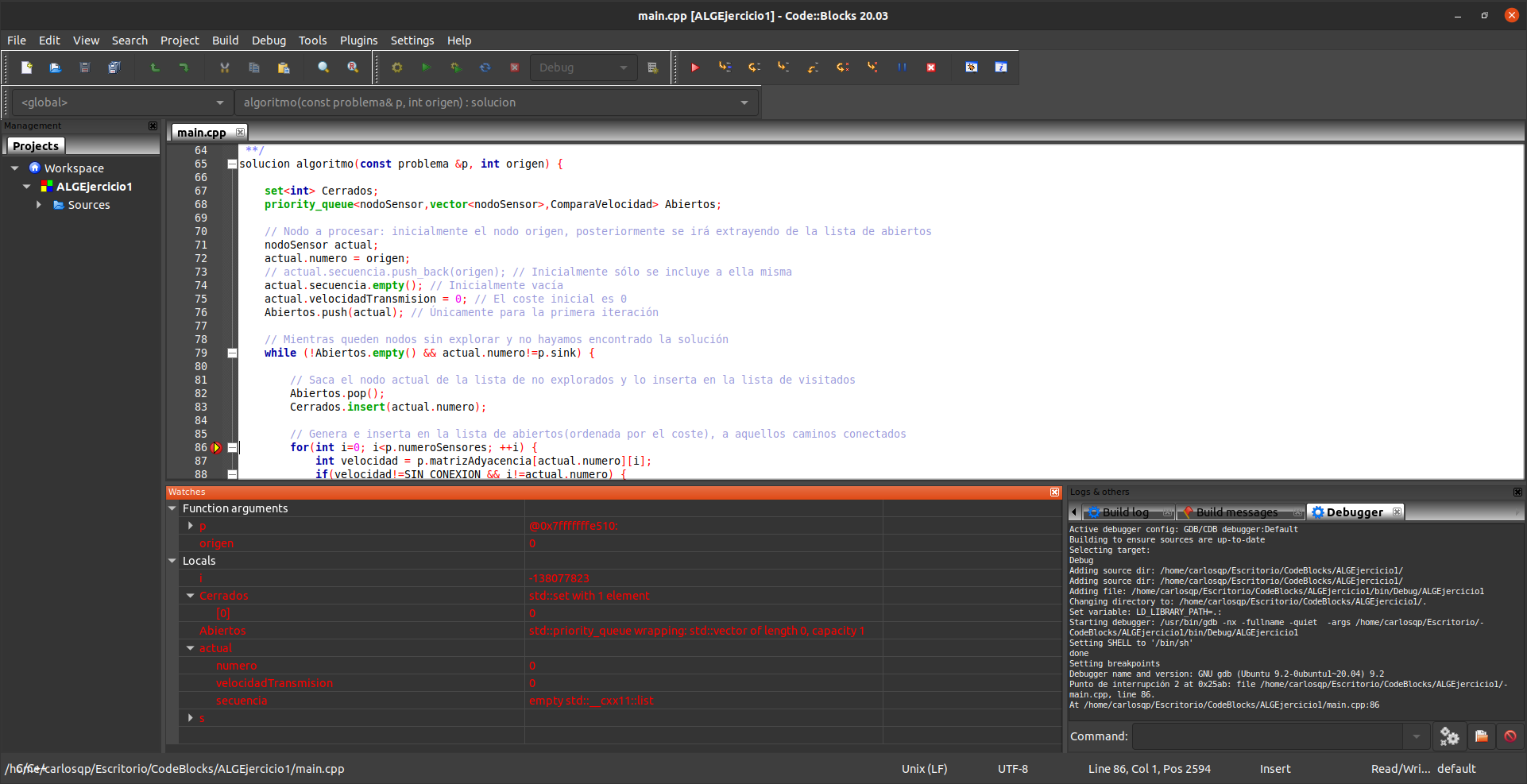
**una instancia pequeña propuesta por el estudiante.**

Supongamos el siguiente ejemplo:

Su fichero de datos a cargar sería el siguiente (en nuestra implementación):

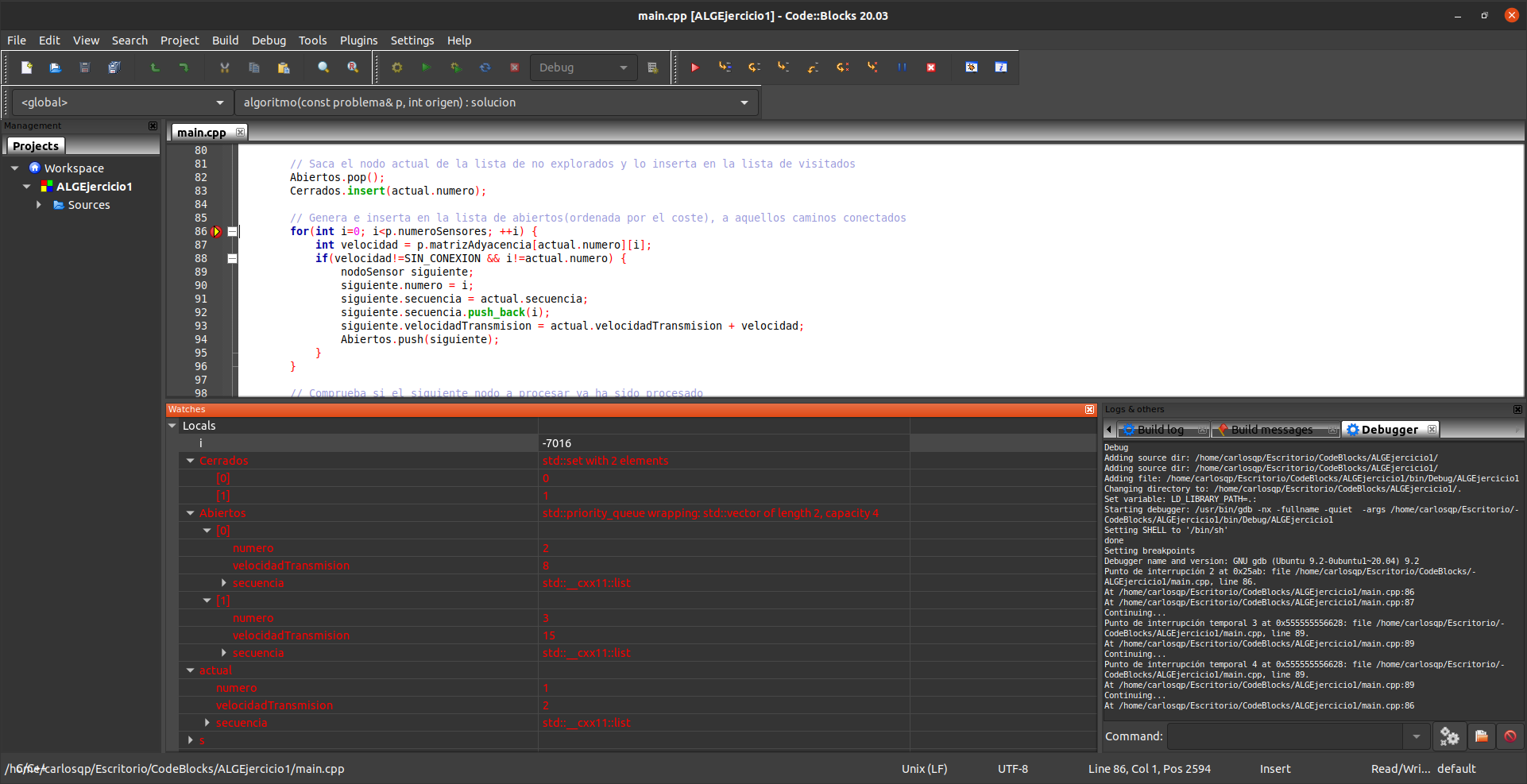
Conocidos Origen = 0 y Destino = 3, vamos a explicar cómo funciona nuestro algoritmo en cada iteración.

Estado inicial:



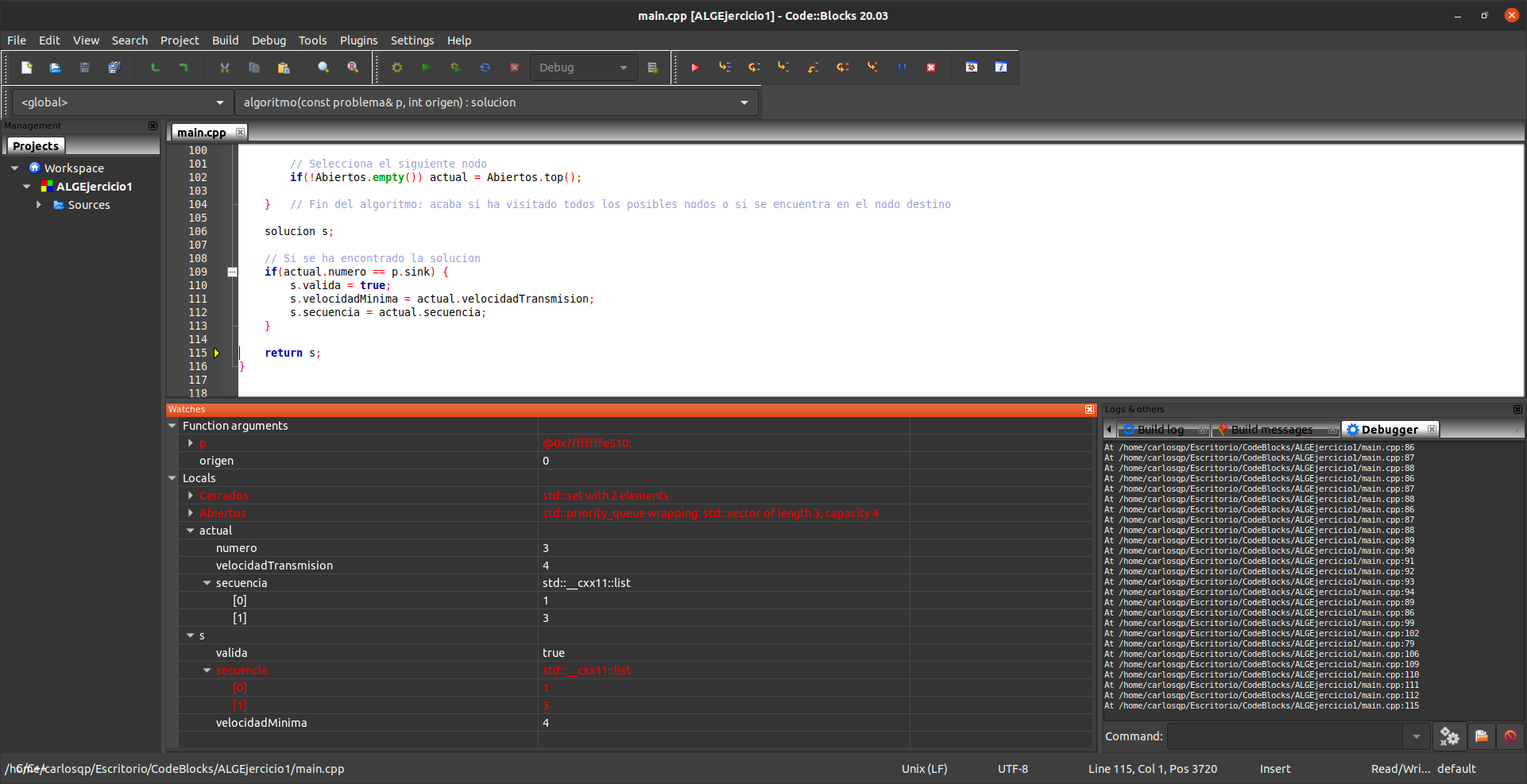
1. Inicialmente: se procesa el nodo origen. Se obtienen todos los nodos que tienen conexión con éste y se insertan en la lista de Abiertos, con sus velocidades correspondientes. Antes de ir a la siguiente iteración, se prepara el siguiente nodo a procesar, el cual es aquel que tiene un menor tiempo acumulado (en este caso el nodo 1, con tiempo 3 de transmisión 3).

Algoritmo y estructuras tras la primera iteración:



1. En el segundo paso, se procesa el nodo 1 puesto que es el que tiene un menor tiempo de transmisión. De la misma manera, se obtienen todas sus conexiones con los tiempos ACUMULADOS de transmisión y se insertan en Abiertos de forma ascendente. El siguiente nodo a procesar es el nodo 3, el cual ya se encontraba en Abiertos (puesto que tenía conexión con el nodo 0) pero esta vez con menor tiempo de transmisión (tiempo = 6).

Algoritmo y estructuras tras la segunda iteración:



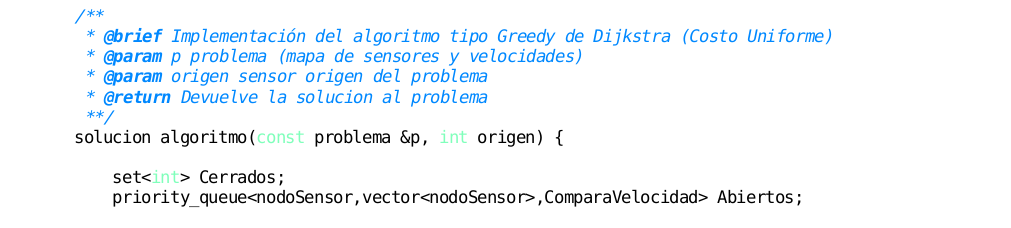
1. En el tercero, el nodo a procesar es el nodo objetivo, por lo que no se entra en la iteración puesto que ya tenemos su camino más veloz, el cual es la solución.

Algoritmo y estructuras tras la tercera iteración (no la llega a realizar, la devuelve en ‘s’):

1. Finalmente, se devuelve y el problema queda resuelto con el resultado esperado.

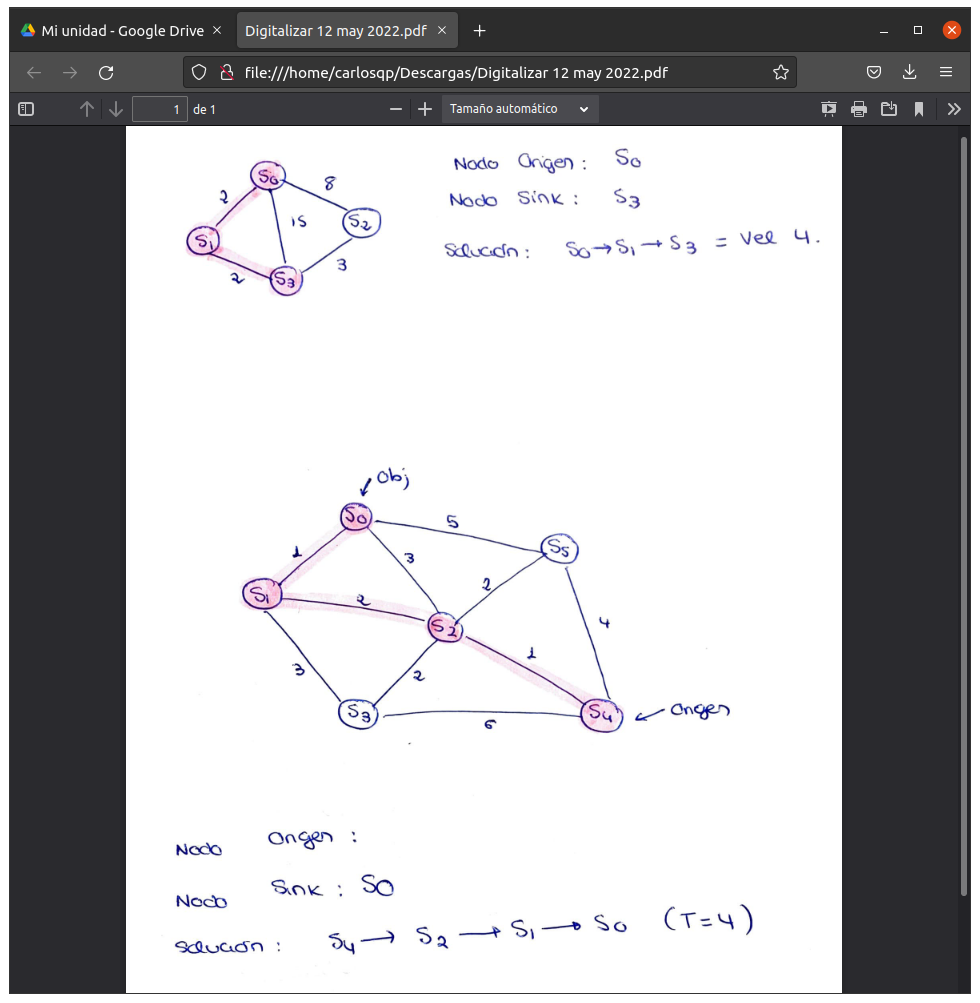
**(1.5) Correcto funcionamiento de la implementación.**

Implementación:

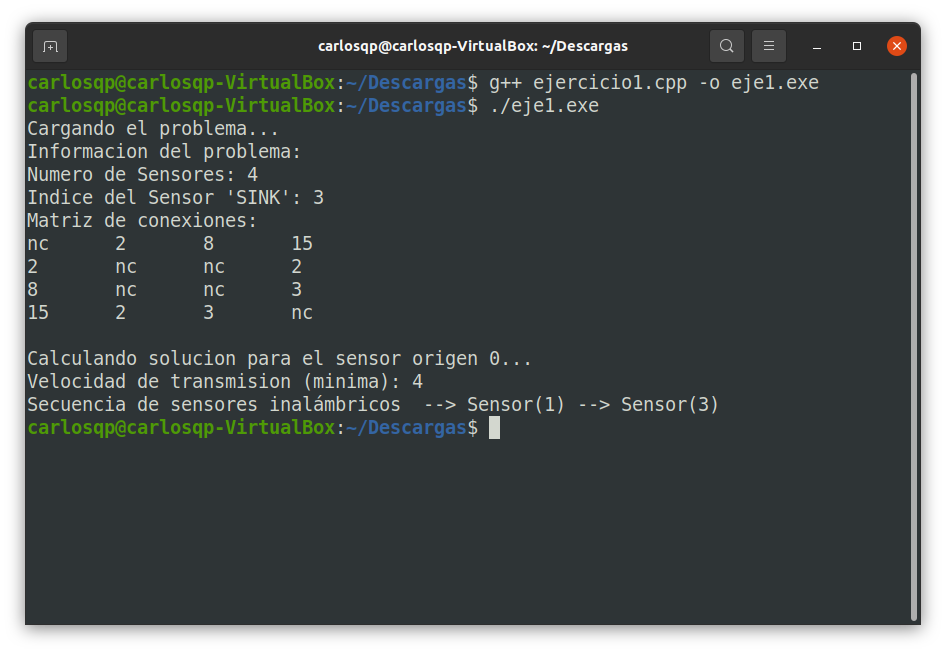
****

Ejemplo 1 (descrito detalladamente en el apartado 1.4).

Problema:

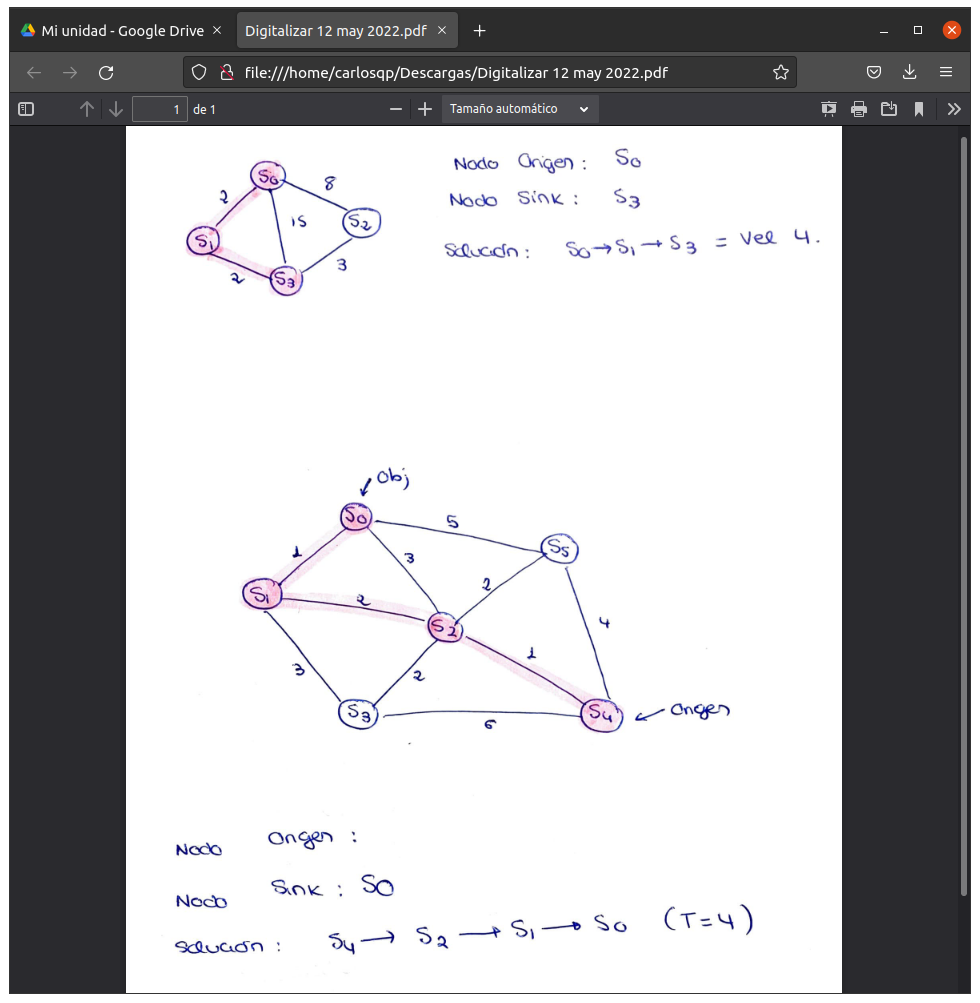


Salida:

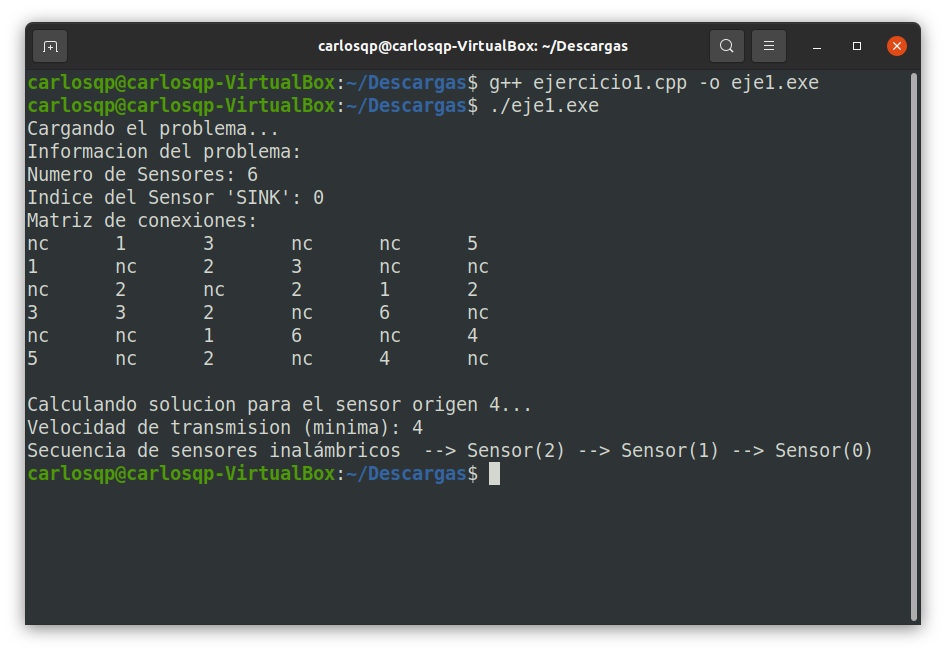


Ejemplo 2.

Problema:



Salida:



**Ejercicio 2**

**(2.1) Diseño de componentes**

-Lista de Candidatos: Serán las gasolineras en las que podrá el autobús repostar

-Lista de Candidatos usados: Serán las gasolineras que se han considerado usar para insertarse en la solución, haya sido o no factible su inserción

-Criterio de selección: Se seleccionará la gasolinera la cual se encuentre más lejos de la gasolinera actual y que podamos llegar con el depósito lleno, para así de esta manera maximizar el uso de la gasolina y minimizar el número de paradas

-Criterio de factibilidad, Una gasolinera se insertará si es la más lejana y posible de llegar con el deposito lleno para asi no hacer mayor cantidad de paradas

-Función Objetivo: Minimizar el número de paradas en gasolineras que componen la solución

**(2.2) Diseño del algoritmo**

solución algoritmo(problema p){

Conjunto nodos cerrados

Conjunto nodos abiertos

int distanciaRecorrida

nodo siguiente;

nodo actual

while(!Abierto.empty() && distanciaRecorrida!=distanciaTotal){

Abiertos.pop();

Cerrado.insert(actual);

for(i hasta numeroGasolineras)

if(distanciaASiguienteGasolinera<=distanciaConDepositoLleno) AND !(distanciaASiguienteSiguienteGasolinera<=distanciaConDepositoLLeno) AND !(distanciaConDepositoLLeno+distanciaRecorrida <= distanciaTotal)

modificamos distancia recorrida

metemos parámetros de siguiente gasolinera

Abiertos.push(siguiente)

}

mientras siguiente sea repetido lo eliminamos de abiertos

si !Abiertos.empty() actual = Abiertos.top()

}

solución s;

si distancia por recorrer llegamos con depósito lleno{

solución es válida

solución asignamos secuencia de nodos recorridos

solución asignamos número de paradas hechas

}

return s;

}

**(2.3) Estudio de optimalidad (demostración o contraejemplo)**

En este caso nuestro algoritmo si es óptimo nuestro algoritmo ya que conseguimos hacer el mínimo número de paradas para llegar a nuestro destino, ya que conseguimos maximizar el uso de combustible agotandolo máximo posible y por tanto consiguiendo recorrer la mayor distancia posible con el mismo, esto se puede ver claro con un ejemplo:

Si nuestro autobús puede recorrer con el combustible al máximo 100km, entonces si por el camino tenemos dos gasolineras una a 50 kilómetros, otra a 90km y otra a 135km nuestro algoritmo irá a la de 90km, ya que si vamos a la de 50km no usamos al máximo el combustible y a la de 135km no llegamos, por tanto la mejor opción será la de 90km.

Esto demuestra que nuestro algoritmo es el más óptimo porque consigue el menor número de paradas posibles entre el origen y el destino.

**(2.4) Ejemplo paso a paso de la explicación del funcionamiento del algoritmo para**

**una instancia pequeña propuesta por el estudiante.**

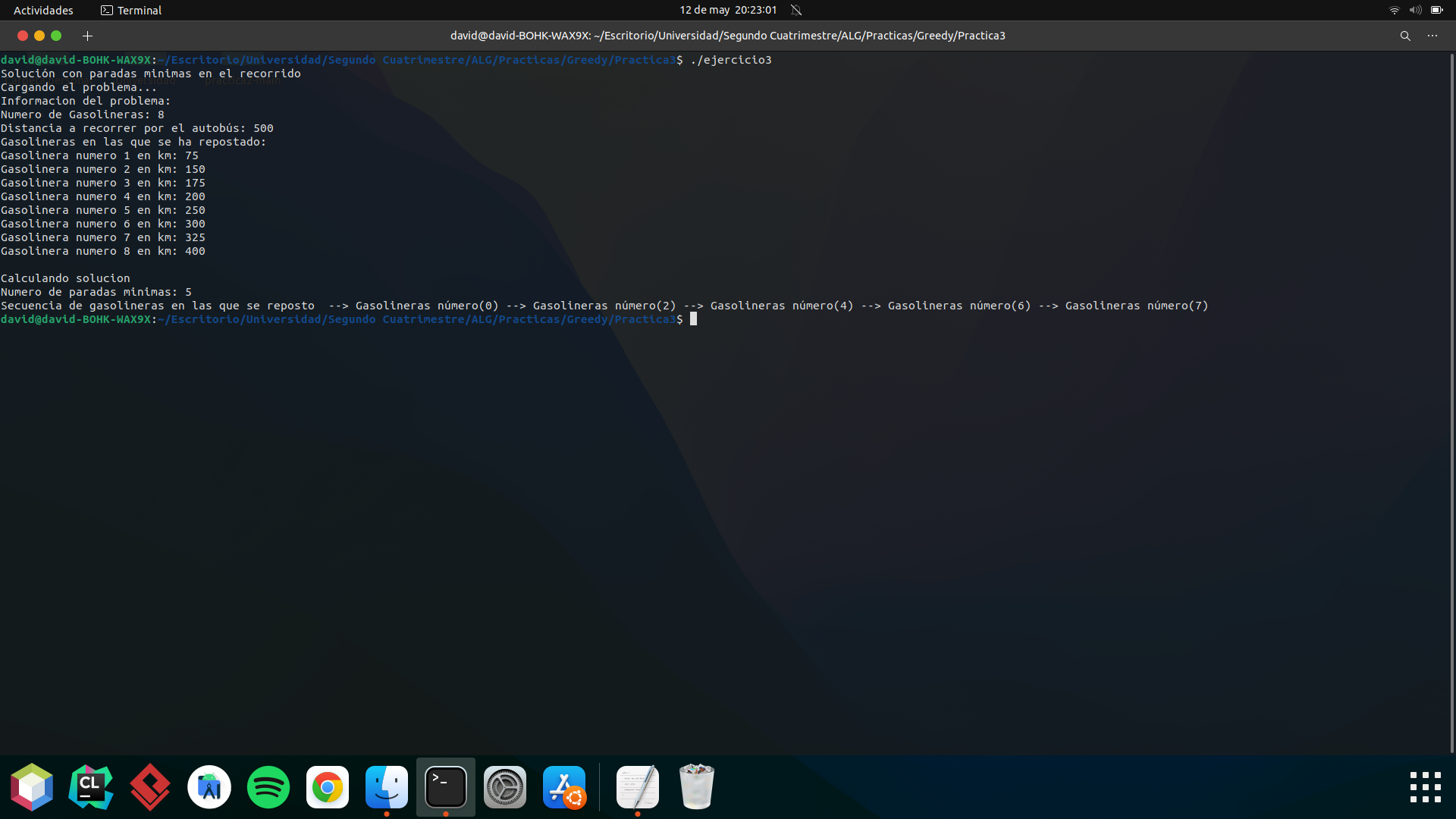
Suponemos un ejemplo en el cual tenemos que recorrer 500km y en el camino tenemos 8 gasolineras distintas(1:75km, 2:150km 3:175km 4:200km 5:250km 6:300km 7:325km 8:400km) y el deposito nos permite recorrer 100 km máximo, por tanto nuestro algoritmo lo que hará será es comprobar si la siguiente gasolinera es la más lejana o si es la siguiente y si la gasolina con el depósito lleno nos permite llegar a la misma, en caso de que si esta la agregaremos como una parada que forma parte de la solución y repetiremos el proceso una vez encontrar en la siguiente gasolinera, de esta manera maximizamos uso de gasolina y minimizamos el número de paradas. En este caso tendríamos que hacer 5 paradas mínimo y las gasolineras en concreto serían la 1,3,5,7 y 8

**(2.5) Correcto funcionamiento de la implementación.**

Implementación:



Captura de ejecución del ejemplo explicado paso a paso en el apartado 2.4:



**Ejercicio 3**

**(3.1) Diseño de componentes**

-Lista de Candidatos: Serán los contenedores disponibles que se pueden cargar al barco

-Lista de Candidatos usados: Serán los contenedores que han sido seleccionados o no seleccionados, es decir que se han comprobado ya.

-Criterio de selección: Se seleccionara el contenedor cuyo peso es mayor y no exceda el peso máximo del barco

-Criterio de factibilidad: Sólo si el tamaño disponible en el barco es mayor o igual al contenedor que se quiere elegir en ese momento, sin contar los que están ya dentro o fuera.

-Función Objetivo: Maximizar el peso del barco sin importar el número de contenedores. El objetivo es llegar lo más próximo al peso total soportado por el barco sin excederse pero llegando a su máximo.

**(3.2) Diseño del algoritmo**

Solucion algoritmo(problema p)

conjunto contenedores\_totales

conjunto contenedores\_en\_barco

k=peso\_barcoMAX

do{

while(contenedores\_totales.top().second > k) contenedores\_totales.pop();

contenedores\_en\_barco.push(contenedores\_totales.top())

k = k - contenedores\_totales.peso

cont.pop()

}while(contenedores\_totales.size() != 0 and k>0);

return contenedores\_en\_barco

**(3.3) Estudio de optimalidad (demostración o contraejemplo)**

En este caso, el algoritmo greedy no es **óptimo** ya que se puede encontrar el objetivo que se quiere que es maximizar el peso de los contenedores en el barco, no obstante, no siempre se cumplirá porque la heurística no es muy buena. Un caso en el que sí se podría optimizar de forma eficiente es en el caso de llenar el barco con el mayor peso posible y con el mayor número de contenedores posibles, en este caso, se añadirían de menor a mayor los contenedores para que se puedan introducir el mayor número y el mayor peso posible mientras la suma de todos esos contenedores no exceda el peso total del barco.

Un contraejemplo muy claro y sencillo sería el siguiente, tenemos un barco de 50 Toneladas, y los siguientes contenedores a cargar: {c1,70/c2,40/c3,30/c4,20/c5,5}. En este caso el algoritmo añadiría el c2 y el c5 ( peso Total = 45), cuando el mas optimo seria c3 y c4 ( peso Total = 50).

**(3.4) Ejemplo paso a paso de la explicación del funcionamiento del algoritmo para**

**una instancia pequeña propuesta por el estudiante.**

Proponemos un ejemplo el cual nos ayudará a entender el algoritmo perfectamente paso a paso, tenemos una estructura la cual tiene por cada posición, un nombre del contenedor, y un peso de dicho contenedor, es el siguiente:

*struct= {c1,100/c2,200/c3,1000/c4,300/c5,10000/c6,700/c7,100/c8,50/c9,150/c10,250/c11,150/*

*c12,350}*

y la cantidad máxima permitida por el barco que es: *k = 3000*.

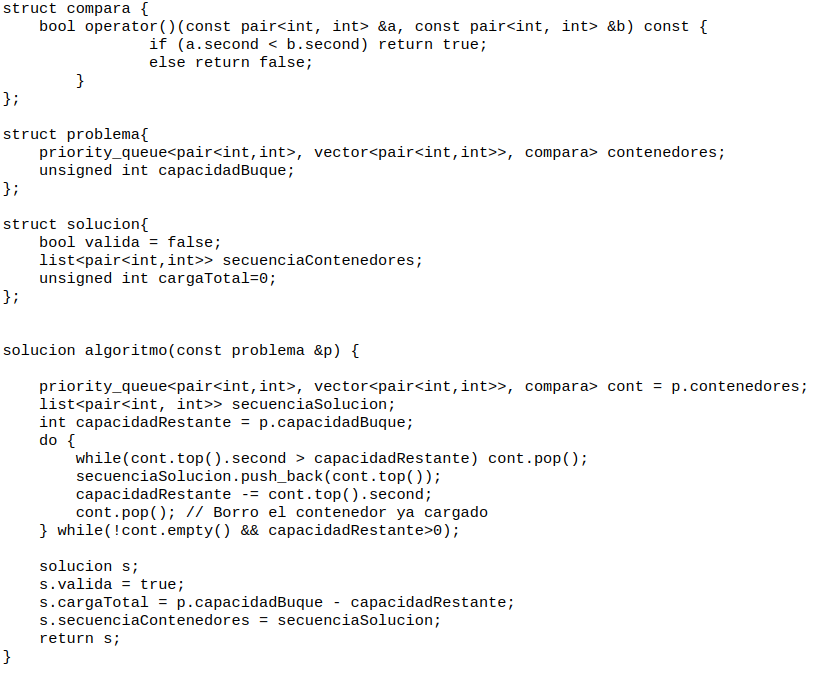
Ahora, mientras haya elementos dentro del *struct* y k sea mayor que 0*,* hará la siguiente comprobación: se hace una comprobación de todos los contenedores que se pueden introducir ahora mismo en el barco, es decir su peso menor que k, se introduce el primero y se resta a k, el peso de dicho contenedor que se ha introducido y así hasta que se llene el barco y no se exceda su peso.

En este caso, la estructura resultante sería: {c3,c6,c12,c4,c10,c2,c11,c8}

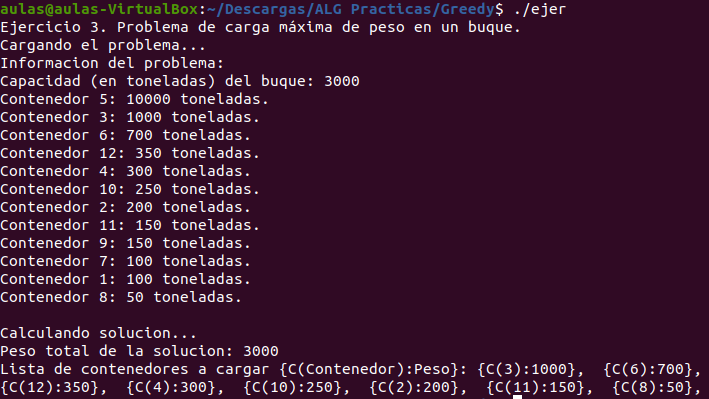
**(3.5) Correcto funcionamiento de la implementación.**

Disponemos de varios structs los cuales nos ayudan a tener el problema y la solución agrupados en conjuntos de datos para tener una implementación bien estructurada.

Disponemos también de un método “algoritmo” el cual devuelve un struct solución. Lo que hace es calcular el peso máximo a cargar en el barco con los contenedores disponibles en el struct problema.



Pruebas de ejecución del ejemplo propuesto antes:



Ejecución del contraejemplo: propuesto en los apartados anteriores:

