PRÁCTICA 3:

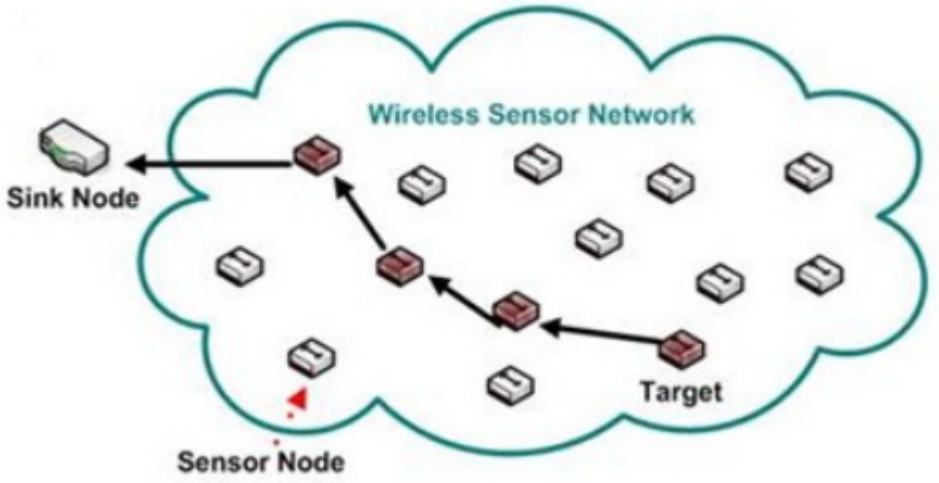
ALGORITMOS GREEDY

Por:

Carmona Molina, José Antonio
Manzano Mata, Nuria
Rodríguez Rodríguez, Antonio

Algoritmo 1:

Una red de sensores inalámbrica está compuesta por múltiples nodos sensores desplegados en un entorno (invernadero, campo de cultivo, instalación industrial, perímetro de vigilancia de seguridad, ...), cada uno equipado con un transmisor de datos inalámbrico de un alcance reducido. Cada cierto tiempo, cada sensor debe enviar los datos recolectados del entorno a un servidor de datos central (sink). Sin embargo, la distancia entre el nodo sensor y el servidor central puede ser elevada como para enviar todos los datos directamente, por lo que será necesario, en ocasiones, enviar los datos por otros nodos sensores que hagan de enlace intermedio (ver Figura 1). Nos interesa enviar los datos con la máxima velocidad posible por lo que, para cada par de nodos sensores de la red n i , n j (entre los que se incluye el servidor central), conocemos el tiempo de envío entre ambos nodos como t(n i , n j) – el tiempo que se tarda en enviar los datos desde el nodo n i al nodo n j) -. El valor t(n i , n j) podría tener valor infinito si la red inalámbrica no permite enviar datos directamente desde el nodo n i hasta el nodo n j . Se pide: desarrollar un algoritmo que nos permita conocer por cuáles nodos sensores intermedios debe enviar los datos cada nodo sensor, hasta llegar el servidor central, de modo que se tarde el mínimo tiempo en la transmisión desde cada nodo hasta el servidor central.

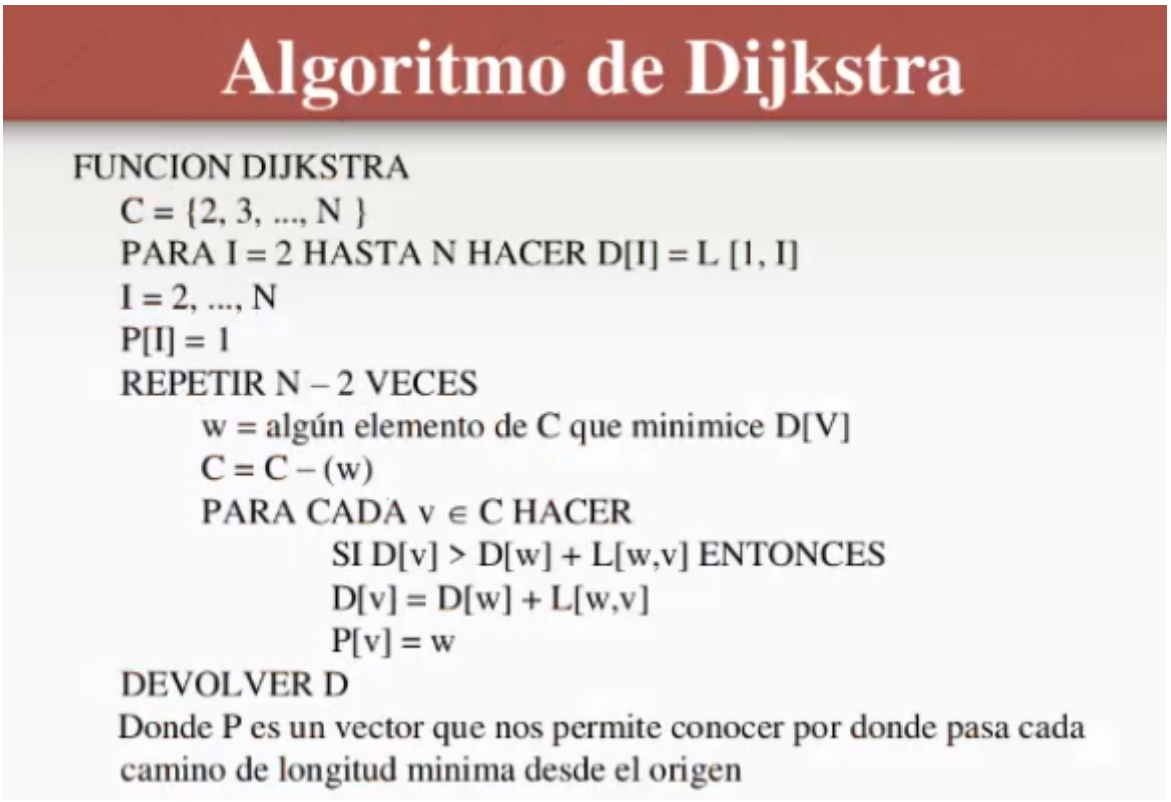


Diseño de componentes:

- Lista de candidatos: Todos los posibles caminos desde el servidor al sensor objetivo.
- Lista de candidatos usados: Camino que minimiza el tiempo de transmisión (con los respectivos nodos por los que pasa).
- Criterio de selección: Camino que presente un tiempo mínimo de transmisión desde el sensor objetivo al servidor central.
- Criterio de factibilidad: Inserta todos los sensores que componen el camino cuyo tiempo de transmisión hasta llegar al servidor sea mínimo.
- Función solución: Recorrido posible hasta llegar al servidor central desde el sensor objetivo.
- Función objetivo: Minimizar el tiempo de transmisión entre servidor central y sensor objetivo.

Diseño del Algoritmo:

Para comenzar con la implementación de nuestro algoritmo, en primer lugar decidimos que tipo de algoritmo se adapta mejor a la resolución de nuestro problema. Tras decidir que el algoritmo a utilizar sería un Dijkstra, empleamos el siguiente pseudocódigo explicado en clases de teoría como base de nuestro código



Este algoritmo resuelve la casuística de los problemas para encontrar el camino mínimo entre dos puntos dados. Y teniendo en cuenta que buscamos encontrar el camino con menor tiempo de transmisión entre dos puntos, podemos adaptar el algoritmo Dijkstra con relativa facilidad.

Finalmente la implementación del programa quedó de la siguiente forma:

Clase Problema:

```
// Valor Infinito
#define INF 1e20

// Cadena nula
```

```
#define SINNOMBRE ""

class Problema
{

private:
   double **t_transmission; // Tiempo de transmisión (ns)
    string **aristas;         // Nombre de las aristas (sensor x a sensor
y)
   string *sensor;           // Nombres de los sensores
   int n;                    // Número de sensores

   void Inicializar() {

       n = 0;
       t_transmission = 0;
       aristas = 0;
       sensor = 0;
   }

   void Liberar() { // Libera la memoria e inicializa a NULL todo

       if (n > 0) {
           for (int i = 0; i < n; i++){
               delete[] aristas[i];
               delete[] t_transmission[i];
           }

           delete[] t_transmission;
           delete[] aristas;
           delete[] sensor;
           Inicializar();
       }
   }

   void Reservar() {

       if (n > 0) {

           sensor = new string[n];
           aristas = new string *[n];
```

```
            t_transmission = new double *[n];
            for (int i = 0; i < n; i++) {

                sensor[i] = SINNOMBRE;
                aristas[i] = new string[n];
                t_transmission[i] = new double[n];

                for (int j = 0; j < n; j++)
                {
                    aristas[i][j] = SINNOMBRE;
                    t_transmission[i][j] = INF;
                }
            }
        }
    }

public:
    Problema() { // Crear un problema vacío
        Inicializar();
    }

    Problema(const Problema &otro) {

        Inicializar();
        *this = otro;
    }

    Problema &operator=(const Problema &otro) {

        Liberar();

        n = otro.n;
        Reservar();
        for (int i = 0; i < n; i++)
            sensor[i] = otro.sensor[i];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                aristas[i][j] = otro.aristas[i][j];
                t_transmission[i][j] = otro.t_transmission[i][j];
            }
```

```
        }

        return *this;
    }

    ~Problema() {
        Liberar();
    }

    void setNumsensores(int otran) {
        if (otran >= 0) {
            Liberar();

            n = otran;
            Reservar();
        }
    }

    int getNumsensores() {
        return n;
    }

    void setSensor(int idx, string nombre) {
        if (idx >= 0 && idx < n)
            sensor[idx] = nombre;
    }

    string getSensor(int idx) {
        if (idx >= 0 && idx < n)
            return sensor[idx];
        return SINNOMBRE;
    }

    // Precondición: El tiempo debe ser > 0
      void setArista(int pOrigen, int pDestino, string nombre, double
coste) {

         if (pOrigen >= 0 && pDestino >= 0 && pOrigen < n && pDestino < n
&& coste > 0) {
            t_transmission[pOrigen][pDestino] = coste;
            aristas[pOrigen][pDestino] = nombre;
```

```
        }
    }

    double gett_transmission(int pOrigen, int pDestino) {
         if (pOrigen >= 0 && pDestino >= 0 && pOrigen < n && pDestino <
n)
            return t_transmission[pOrigen][pDestino];
        return INF;
    }

    string getArista(int pOrigen, int pDestino) {
         if (pOrigen >= 0 && pDestino >= 0 && pOrigen < n && pDestino <
n)
            return aristas[pOrigen][pDestino];
        return SINNOMBRE;
    }

    bool leerFichero(const char *nombrefich) {
        ifstream fich;
        int numSensores;
        double t_trans;
        char nombres[1001];
        char c;

        fich.open(nombrefich);
        if (!fich)
            return false;

        fich >> numSensores;
        fich.get(c);
        if (numSensores <= 0) {
            fich.close();
            return false;
        }

        Liberar();
        n = numSensores;
        Reservar();

        for (int i = 0; i < n; i++) { // Leemos los sensores
            fich.getline(nombres, 1000);
```

```
            if (!fich.eof())
                setSensor(i, nombres);
            else {
                Liberar();
                fich.close();
                return false;
            }
        }

        // Leemos las aristas
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {

                fich >> t_trans;
                if (fich.eof() && i != n - 1 && j != n - 1) {
                    Liberar();
                    fich.close();
                    return false;
                }

                if (t_trans > -1) {
                    fich.getline(nombres, 1000);
                    setArista(i, j, nombres, t_trans);
                }
            }
        }

        fich.close();
        return true;
    }
};
```

Cabe mencionar que empleamos la clase Problema dada por los profesores de la asignatura, utilizada con anterioridad para explicar un ejemplo con el algoritmo de Árbol Generador Minimal, como base para recolectar los datos de nuestro grafo y así poder trabajar con relativa facilidad con nuestro algoritmo Dijkstra.

No obstante, en este proyecto no entraremos en detalles sobre el funcionamiento de esta clase al darse por conocido siendo el caso base de ejemplo.

Función Dijkstra:

```
double Dijkstra(Problema p, int destino, int origen) {
   int n = p.getNumsensores();
   int sol[n] = {-1};
   double distancia[n], C[n];
   for (int i = 0; i < n; i++) {
       distancia[i] = p.gett_transmission(destino, i);
       C[i] = p.gett_transmission(destino, i);
       sol[i] = -1;
   }

   int cont = 1;
   while(cont < n) {
       int pos = 0;
       for (int i = 0; i < n; i++) {
           if (C[i] < distancia[cont]) {
               pos = i;
               C[pos] = INF;
               i = n;
           }
       }
       int aux = 0;
       for (int i = 0; i < n; i++) {
            if (distancia[i] > distancia[pos] + p.gett_transmission(pos,
i)) {
                distancia[i] = distancia[pos] + p.gett_transmission(pos,
i);
               if (aux != i) sol[aux] = -1;
               sol[i] = pos;
               aux = i;
           }
       }
       cont++;
   }

   cout << "Señal emitida desde: " << p.getSensor(origen) << endl;
   cout << "\nSensores intermedios: " << endl;
   int id = sol[origen];
   while (id != -1) {
       cout << p.getSensor(id) << endl;
```

```
        id = sol[id];
    }
    cout << endl;
    cout << "Sensor destino: " << p.getSensor(destino);

    return distancia[origen];
}
```

Este algoritmo conocido como "Dijkstra" o algoritmo de caminos mínimos funciona de tal manera que determina el camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene. Se trata de una especialización de la búsqueda de costo uniforme y, como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).

Main:

```
int main()
{
    Problema p;
    double s;

    p.leerFichero("sensores.dat");
    s = Dijkstra(p, 0, 3);

    cout << endl
         << "Tiempo: " << s << endl;

    return 0;
}
```

Iremos variando los parámetros usados en la función Dijkstra, según el nodo origen y el nodo destino entre los que queramos calcular el tiempo mínimo de transmisión.

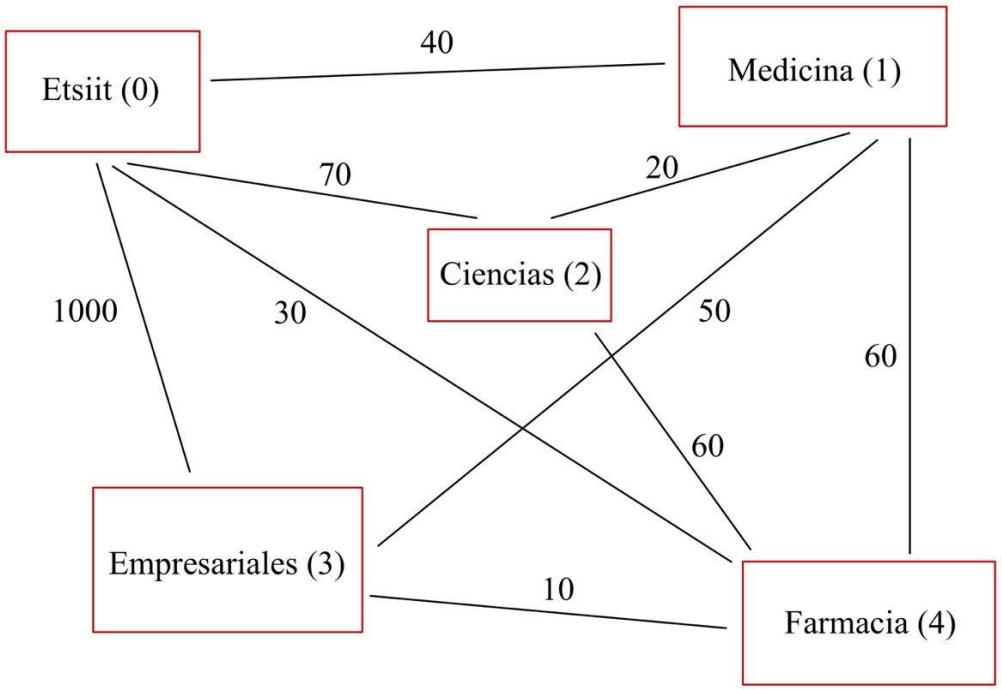
Análisis del algoritmo y estudio de la optimalidad:

En el estudio de la optimalidad, en este caso siempre obtendremos una solución óptima al emplear en la implementación del algoritmo un algoritmo de tipo Dijkstra.

Teniendo en cuenta la explicación del algoritmo dada en el apartado anterior (diseño del algoritmo), explicaremos a continuación el resultado obtenido para la ejecución del siguiente caso que hemos planteado y que nos muestra la primera ejecución de la captura anterior.

Presentamos el problema que sigue, en el cual queremos llegar desde la Facultad de Empresariales hasta la Etsiit, con ello en el main nos encontraremos la línea:

```
s = Dijkstra(p, 0, 3);
```



De este grafo sacamos el txt con los datos necesarios para meter la información a nuestro programa mediante la clase Problema que hemos cogido del ejemplo que teníamos de los profesores.
Quedando el mismo como sigue:

```
5
Etsiit
Medicina
Ciencias
Empresariales(origen)
Farmacia
-1
40 Etsiit-Medicina
70 Etsiit-Ciencias
1000 Etsiit-Empresariales(origen)
30 Etsiit-Farmacia
40 Etsiit-Medicina
-1
20 Medicina-campo
50 Medicina-Empresariales(origen)
60 Medicina-Farmacia
70 Etsiit-Ciencias
20 Medicina-campo
-1
-1
60 Ciencias-Farmacia
1000 Etsiit-Empresariales(origen)
50 Medicina-Empresariales(origen)
-1
-1
10 Empresariales(origen)-Farmacia
30 Etsiit-Farmacia
60 Medicina-Farmacia
60 Ciencias-Farmacia
10 Empresariales(origen)-Farmacia
-1

```

Una vez tenemos todos los datos ya disponibles para el correcto funcionamiento del algoritmo, llamamos a la función mediante la línea de código indicada anteriormente y el algoritmo procederá como sigue:

En primer lugar, el programa almacena todos los tiempos de transmisión entre sensores tanto en el vector distancia como en el vector auxiliar C[].
A continuación entraremos en un bucle que se ejecutará n veces siendo n el número total de sensores. Si encontramos algún elemento del vector auxiliar C[] que minimice el tiempo de transmisión sobre el que estamos iterando, guardaremos la posición de dicho elemento de C[] en nuestra variable “pos”, y seguidamente eliminaremos de C el elemento en cuestión (C[pos] = INF).
Posteriormente, iteramos sobre un bucle que irá comprobando si existe algún camino que minimice el tiempo de transmisión entre el servidor y el sensor objetivo e iremos insertando y eliminando sensores o almacenando dichos sensores según vayamos aceptando o rechazando el camino final que minimice el tiempo de transmisión, al mismo tiempo que iremos sumando los tiempos de transmisión entre los diferentes sensores que forman el camino final.
A continuación imprimimos el sensor de inicio, seguido de los sensores por los que hemos pasado en la elección de nuestro camino resultado y los cuales iremos recuperando mirando el sensor previo (cosa que hacemos en el último bucle ya fuera del bucle principal del programa); seguido del sensor destino.

Finalmente retornamos el tiempo total de transmisión del camino escogido, y el cuál optimiza la solución.

Extrapolando esto al ejemplo de la ejecución entre los sensores Empresariales y Etsiit, el código irá operando del mismo modo hasta obtener el resultado expuesto a continuación:

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio1
Señal emitida desde: Empresariales(origen)

Sensores intermedios:
Farmacia

Sensor destino: Etsiit
Tiempo transmisión: 40
```

Otros ejemplos de ejecución serían los que siguen:

```
s = Dijkstra(p, 0, 2);
```

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio1
Señal emitida desde: Ciencias

Sensores intermedios:
Medicina

Sensor destino: Etsiit
Tiempo transmisión: 60
```

```
s = Dijkstra(p, 2, 3);
```

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio1
Señal emitida desde: Empresariales(origen)

Sensores intermedios:
Medicina

Sensor destino: Ciencias
Tiempo transmisión: 70
```

Algoritmo 2:

Un autobús realiza una ruta determinada entre su origen y su destino (n kilómetros en total). Con el tanque de gasolina lleno, el autobús puede recorrer kilómetros sin parar. El conductor dispone de un listado con las gasolineras existentes en su camino, y el punto kilométrico donde se encuentran. Se pide: Diseñar un algoritmo greedy que determine en qué gasolineras tiene que repostar el conductor para realizar el mínimo número de paradas posible.

Diseño de componentes:

- Lista de candidatos: Todas las gasolineras en las que se puede realizar una parada.
- Lista de candidatos usados: Gasolineras donde hacemos parada.
- Criterio de selección: Seleccionaremos las gasolineras que permitan hacer una parada antes de superar los kilómetros de autonomía del autobús y tratando que la distancia entre parada y parada sea máxima.
- Criterio de factibilidad: Se inserta en la solución si la siguiente gasolinera está más lejos de lo que el autobús puede avanzar.
- Función solución: Lista de gasolineras en las que se deberá parar, intentando hacer el mínimo número de paradas.
- Función objetivo: Minimizar el número de paradas hasta llegar a su destino.

Diseño y análisis del Algoritmo y estudio de la optimalidad:

Hemos empleado un algoritmo Greedy en el que buscamos hacer el número mínimo de paradas en un trayecto.

```
void OrdenaBurbuja(vector<pair<string,double>> & v, int n) {

    int i, j, aux;
    string id;
    bool haycambios= true;

    i= 0;
    while (haycambios) {
        haycambios=false; // Suponemos vector ya ordenado
        for (j= n-1; j>i; j--) { // Recorremos vector de final a i

            if (v[j-1].second>v[j].second) { // Dos elementos consecutivos mal ordenados
              aux = v[j].second; // Los intercambiamos
              id = v[j].first;
              v[j].second = v[j-1].second;
              v[j].first = v[j-1].first;
              v[j-1].second = aux;
              v[j-1].first = id;
              haycambios= true; // Al intercambiar, hay cambio
            }
        }
        i++;
    }
}
```

```cpp
void Minimas_paradas(vector<pair<string,double>> gasolineras, double autonomia){
    OrdenaBurbuja(gasolineras, gasolineras.size());
    cout << "Salimos de: " << gasolineras.front().first << endl << "Pararemos en: \n";
    int cont = 0;
    for(int n = 0; n < gasolineras.size();) {
        int distancia = gasolineras[n].second - gasolineras[cont].second;
        if(distancia <= autonomia){
            n++;
        } else {
            if (gasolineras[n-1] != gasolineras[cont]){
                cout << "Gasolinera " << gasolineras[n-1].first << " en el km " << gasolineras[n-1].second << endl;
                cont = n-1;
            } else {
                cout << "No es posible completar el recorrido" << endl;
                exit(-1);
            }
        }
    }
    cout << gasolineras.back().first << " en el kilometro " << gasolineras.back().second << endl;
}
```

```cpp
int main(int argc, char** argv) {
    if (argc <= 1) {
        cerr<<"\nError: El programa se debe ejecutar de la siguiente forma.\n\n";
        cerr<<argv[0]<<" Autonomía gasolinera1 dist_origen1 identificador2 gasolinera2 ... gasolineraN dist_origenN\n\n";
        return 0;
    }
    vector<pair<string,double>> gasolineras;
    for (int i=2; i<argc; i+=2) {
        string gasolinera = argv[i];
        double dist = stod(argv[i+1]);
        pair<string, double> input;
        input.first = gasolinera;
        input.second = dist;
        gasolineras.push_back(input);
    }
    int autonomia = stod(argv[1]);
    Minimas_paradas(gasolineras, autonomia);
```

```cpp
//////////////////////////// USADO PARA DEBUGGER.
/*
    vector<pair<string,double>> ejemplo;
    ejemplo={{"salida", 0},{"a", 2},{"b", 4},{"c", 9},{"llegada", 14}};

    Minimas_paradas(ejemplo,5);

    return 0;
*/
}
```

A continuación, en el análisis del algoritmo y su explicación para un caso concreto, comprenderemos el funcionamiento del mismo, explicando detalladamente el mismo.

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio2

Error: El programa se debe ejecutar de la siguiente forma.

./ejercicio2 Autonomía gasolinera1 dist_origen1 identificador2 gasolinera2 ... g
asolineraN dist_origenN

antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio2 5 Origen 2 b 4 c 9 Destino 14
Salimos de: Origen
Pararemos en:
Gasolinera b en el km 4
Gasolinera c en el km 9
Destino en el kilometro 14
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio2 5 Origen 2 b 4 c 9 Destino 15
Salimos de: Origen
Pararemos en:
Gasolinera b en el km 4
Gasolinera c en el km 9
No es posible completar el recorrido
```

En la primera ejecución, ejecutamos sin argumentos lo que nos mostrará cuales son los que debemos de utilizar.
Seguidamente ejecutaremos en base a estos argumentos, indicando en primer lugar la autonomía de la que dispone el autobús (en los ejemplos de ejecución hemos supuesto 5km como caso hipotético) y colocando las gasolineras disponibles con sus respectivas distancias (en kilómetros) al punto de partida (origen).
Con ello nuestro algoritmo nos dirá en qué gasolineras debemos de hacer parada, recorriendo la máxima distancia posible entre parada y parada, hasta llegar a nuestro destino.
Si en algún caso nos encontrásemos que la distancia entre dos gasolinera consecutivas o entre el origen/destino y su respectiva gasolinera fuese mayor que la autonomía, diremos que no será posible completar el recorrido, ya que la autonomía del autobús no permite llegar de un punto al siguiente, quedándose parado en mitad del trayecto; tal y como vemos en la tercera ejecución de la captura de pantalla.

En el estudio de la optimalidad de este caso, hemos concluido que no hay contraejemplo, ya que el programa debería de dar siempre una solución óptima, el único ejemplo que nos ocurre que se acercaría a esta situación de contraejemplo podría ser cuando no es posible llegar de un punto a otro consecutivo haciendo imposible acabar el recorrido. Aún así, esto sería una situación provocada por factores externos al programa por lo que no podría considerarse exactamente como un contraejemplo.
Con lo que siempre nos arrojará un resultado óptimo.

Sabiendo cuál es el resultado que arroja el programa y las diferentes situaciones que se nos pueden plantear con respecto a los datos, a continuación analizaremos cómo opera el programa hasta llegar a dicha solución:

En primer lugar observando los parámetros de nuestra función, encontramos un vector de pair que almacena el nombre de cada gasolinera con su respectiva distancia al punto de origen; así como la autonomía del autobús.
Entrando en el algoritmo en sí, en primer lugar ordenamos las gasolineras en base a la distancia al punto de origen (de más cercanas a más lejanas), para ello hemos utilizado la función OrdenaBurbuja utilizada en prácticas anteriores, modificándola para que trabaje con vectores de pair en lugar de con vectores de enteros.
Una vez ordenadas nuestras gasolineras disponibles, hacemos un bucle que recorra todas las gasolineras e iremos avanzando de una gasolinera a otra, si la distancia es menor o igual a la autonomía entre gasolineras consecutivas. En el momento en el que esta distancia supere a la autonomía, debemos hacer una parada en la gasolinera anterior (ya que en caso contrario nos estaríamos pasando), y el bucle comenzará de nuevo desde la gasolinera en la que hemos parado (cont=n-1), e irá repitiendo el proceso bien hasta que hayamos llegado al destino o hasta que encuentre una situación en la que es imposible acabar el recorrido, esto último traducido a nuestro código ocurrirá cuando gasolineras[n-1] == gasolineras[cont], ya que en este caso la gasolinera anterior será la misma en la que ya habíamos parado, con lo que significaría que no hemos avanzado y con ello que es imposible llegar a la siguiente parada consecutiva.

Extrapolando esto a la segunda ejecución de la captura de pantalla, en primer lugar vemos que ordenaría las gasolineras que le pasamos como parámetro (aunque en este caso ya aparecen en orden), imprimimos el punto de partida (origen = 2), ahora pasaríamos por la gasolinera b=4, dónde haremos parada ya que la siguiente gasolinera c=9, se encuentra a una distancia de 7km desde el punto de partida. En segundo lugar pararemos también en la gasolinera c, ya que desde 4 hasta el destino en el kilómetro 14 hay una distancia de 10km (> 5km). Y en último lugar avanzamos desde la gasolinera c hasta el destino recorriendo una distancia igual a la autonomía (5km).
NOTA: Vamos imprimiendo por pantalla las gasolineras en las que vamos haciendo parada.

A continuación mostramos algunos ejemplos más de ejecuciones para demostrar el correcto funcionamiento y cualquier duda que pueda quedar al respecto del funcionamiento del algoritmo.

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio2 6 Origen 0 b 3 c 6 Destino 10
Salimos de: Origen
Pararemos en:
Gasolinera c en el km 6
Destino en el kilometro 10
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio2 5 Origen 0 b 3 c 6 Destino 10
Salimos de: Origen
Pararemos en:
Gasolinera b en el km 3
Gasolinera c en el km 6
Destino en el kilometro 10
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio2 5 Origen 0 b 3 c 9 Destino 10
Salimos de: Origen
Pararemos en:
Gasolinera b en el km 3
No es posible completar el recorrido
```

Algoritmo 3:

Se tiene un buque mercante cuya capacidad de carga es de k toneladas, y un conjunto de contenedores c 1 , ... , c n cuyos pesos respectivos son p 1 ,..., p n (expresados también en toneladas). Teniendo en cuenta que la capacidad del buque podría ser menor que la suma total de los pesos de los contenedores, se pide: Diseñar un algoritmo que permita decidir qué contenedores hay que cargar para maximizar la suma de los pesos de los contenedores a transportar en el barco.

Diseño de componentes:

- Lista de candidatos: Todos los contenedores que podrían meterse en el barco.
- Lista de candidatos usados: Contenedores que han sido cargados en el barco.
- Criterio de selección: Seleccionar los contenedores que más llenan el barco sin sobrepasar su capacidad (los más grandes).
- Criterio de factibilidad: Se inserta el barco en la solución si al sumarlo no supera el peso máximo del barco.
- Función solución: Lista de contenedores que se han insertado en el barco.
- Función objetivo: Maximizar el peso de los contenedores que se meterán en el barco.

Diseño del Algoritmo:

```
void OrdenaBurbuja(vector<pair<string,double>> & v, int n) {

    int i, j, aux;
    string id;
    bool haycambios= true;

    i= 0;
    while (haycambios) {
        haycambios=false; // Suponemos vector ya ordenado
        for (j= n-1; j>i; j--) { // Recorremos vector de final a i

            if (v[j-1].second>v[j].second) { // Dos elementos consecutivos mal ordenados
              aux = v[j].second; // Los intercambiamos
              id = v[j].first;
              v[j].second = v[j-1].second;
              v[j].first = v[j-1].first;
              v[j-1].second = aux;
              v[j-1].first = id;
              haycambios= true; // Al intercambiar, hay cambio
            }
        }
        i++;
    }
}
```

```cpp
// ALGORITMO MAXIMIZACIÓN DEL PESO TIPO GREDDY.
void maximizacion_peso(vector<pair<string, double>> & pesoContenedores, double cargaMaxima) {
    int n = pesoContenedores.size();
    OrdenaBurbuja(pesoContenedores, n);
    double sumaPesos = 0;
    cout << "Carga máxima: " << cargaMaxima << endl;
    cout << "Contenedores cargados: ";
    while (sumaPesos < cargaMaxima && !pesoContenedores.empty()) {
        int aux = pesoContenedores.back().second;
        if (sumaPesos + aux <= cargaMaxima) {
            sumaPesos += aux;
            cout << pesoContenedores.back().first << ":" << aux << "T ";
        }
        pesoContenedores.pop_back();
    }
    cout << endl << "Peso total: " << sumaPesos << endl;
}
```

```cpp
int main (int argc, char *argv[]) {
    if (argc <= 1) {
        cerr<<"\nError: El programa se debe ejecutar de la siguiente forma.\n\n";
        cerr<<argv[0]<<" cargaMaxima identificador1 pesoContenedores1 identificador2 pesoContenedores2 ... identificadorN pesoContenedoresN\n\n";
        return 0;
    }
    vector<pair<string,double>> pesoContenedores;
    for (int i=2; i<argc; i+=2) {
        string id = argv[i];
        double aux = stod(argv[i+1]);
        pair<string, double> input;
        input.first = id;
        input.second = aux;
        pesoContenedores.push_back(input);
    }
    maximizacion_peso(pesoContenedores, stod(argv[1]));
}
```

En primer lugar, debemos de tener claro que estamos ante un algoritmo del tipo mochila fraccionada, en el que no podemos fraccionar (al no poder dividir el peso de los contenedores) con lo que no siempre obtendremos la solución más óptima, aunque sí la más rápida.
Sabiendo esto, diremos que este algoritmo funciona de manera que inicialmente se recibirán por parámetro un vector de pares con todos los pesos de los contenedores y la carga máxima que admite el buque. En primer lugar se ordenan los contenedores en función de su peso y se imprime por pantalla la carga máxima del buque. Dentro de un bucle se irán insertando los contenedores en el barco (incrementando la variable sumaPesos) mientras que no se iguale a la capacidad máxima permitida del buque o nos hayamos quedado sin contenedores disponibles, eliminando los elementos por los que ya se ha pasado, hayan sido o no insertados. Se van imprimiendo por pantalla los contenedores insertados y su peso, además de la carga total final.

Análisis del algoritmo y estudio de la optimalidad:

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio3

Error: El programa se debe ejecutar de la siguiente forma.

./ejercicio3 cargaMaxima identificador1 pesoContenedores1 identificador2 pesoCon
tenedores2 ... identificadorN pesoContenedoresN

antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio3 30 a 11 b 9 c 7 d 10
Carga máxima: 30
Contenedores cargados: a:11T d:10T b:9T
Peso total: 30
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio3 30 a 11 b 9 c 22 d 10
Carga máxima: 30
Contenedores cargados: c:22T
Peso total: 22
```

Podemos ver en esta fotografía en primer lugar, como nos pide por pantalla como debemos de escribir los parámetros del programa para poder ejecutarlo. Una vez pasados los parámetros de manera correcta nos encontramos con dos ejemplos:

- En el **primer ejemplo** vemos como, dado una carga máxima de 30 Toneladas que posee un barco, y diferentes contenedores con cargas de 11T el a, 9T el b, 7T el c y 10T el d. El resultado que nos ofrece este algoritmo es que la carga más óptima de dichos contenedores dentro del barco sería del contenedor a, d y b cuya suma total de toneladas asciende a 30 toneladas correspondiendo a la carga máxima del barco y a su vez, corresponden a los contenedores de mayor peso.
- En el **segundo ejemplo** pasa algo diferente y es que, poseemos la misma carga máxima del barco, pero, hay un contenedor , en concreto el c que ahora pesa 22T en vez de 7T, el resultado más óptimo y correcto debería de ser el mismo que en el resultado anterior, dado que aprovecharíamos la carga máxima completa de dicho barco y además, cargaríamos 3 contenedores a la vez; pero este algoritmo decide cargar solamente el contenedor c, coincidiendo que es el de mayor peso, es por ello, que encontraríamos un ejemplo de contraejemplo de este algoritmo Greedy.

En definitiva, podemos comprobar que en ambos casos el algoritmo recoge primero los valores más grandes y después los más pequeños, dando así resultados lo más rápidos posible, pero que no siempre serán los más correctos, para el aprovechamiento máximo de la carga del barco, pues como hemos dicho anteriormente, nos encontramos con un algoritmo de tipo mochila fraccionada, en el que no podemos fraccionar. A continuación podremos comprender mejor su funcionamiento.

En base a la explicación que hemos dado sobre el funcionamiento del algoritmo y el resultado de la segunda ejecución de la captura de pantalla anterior, trataremos de explicar paso a paso la obtención de este último:

En primer lugar tomamos la carga máxima como referencia (30 Toneladas).
En segundo lugar ordenamos los contenedores de menor a mayor con la función OrdenaBurbuja, tal y como ya explicamos en el ejercicio 2 para las distancias de las gasolineras.
Ahora entraremos en el bucle vamos insertando los contenedores, siempre y cuando no supere la suma de pesos de todos los contenedores el peso máximo, de mayor a menor (cogiendo el último elemento de nuestro vector de contenedores) y eliminaremos el último elemento de dicho vector, independientemente de si seleccionamos o descartamos dicho elemento.
Por ello, en este ejemplo de ejecución insertará con el orden 11T el a, 9T el b y 10T el d; y como completan la carga máxima no se contempla el contenedor c de 7T.
Del mismo modo en la tercera ejecución (contraejemplo), insertará sólo el mayor de 22T, mostrando el comportamiento Greddy de darnos la solución más rápida y no siempre la más óptima que en este caso sería la suma del resto de contenedores que nos daría la carga máxima del barco; pero que al tomar el contenedor más grande ya no se admite ninguno de los restantes.
NOTA: Vamos imprimiendo los contenedores que se cargan en el barco conforme los vamos seleccionando.

A continuación mostramos algunos ejemplos más de ejecuciones para demostrar el correcto funcionamiento y cualquier duda que pueda quedar al respecto del funcionamiento del algoritmo.

```
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio3 100 a 30 b 35 c 120 d 37 e 28
Carga máxima: 100
Contenedores cargados: d:37T b:35T e:28T
Peso total: 100
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio3 100 a 30 b 35 c 100 d 37 e 28
Carga máxima: 100
Contenedores cargados: c:100T
Peso total: 100
antonio@antonio-IdeaPad-Gaming-3-15ARH05:~/Descargas/Greedy/Ejercicios_definitiv
os$ ./ejercicio3 20 a 30 b 35 c 100 d 37 e 28
Carga máxima: 20
Contenedores cargados:
Peso total: 0
```