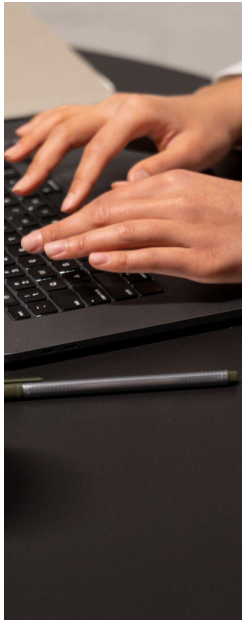
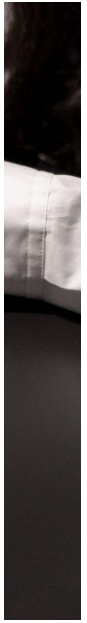
# Algoritmos Greedy (Voraces)

## ALGORÍTMICA

Azorín Martí, Carmen Cribillés Pérez, María Ortega Sevilla, Clara Torres Fernández, Elena

# ÍNDICE

# 01
# INTRODUCCIÓN

# Algoritmos Greedy

## CARACTERÍSTICAS

- Conjunto de Candidatos
- Conjunto Solución
- Función Selección
- Criterio de Factibilidad
- Función Objetivo

## CONTENEDORES

- Maximizar el número de contenedores cargados - ÓPTIMO
- Maximizar el número de toneladas cargadas - NO ÓPTIMO

## VIAJANTE DE COMERCIO

- Vecino más cercano
- Inserción más lejana
- Algoritmo Propio
- Ejemplos y Comparaciones

# Algoritmo Greedy

S=∅

Mientras S no sea una solución y C≠∅. Hacer:

    X=elemento de C que maximiza SELEC(X)

    C=C-{X}

    Si (S U {X}) es factible entonces S=SU{X}

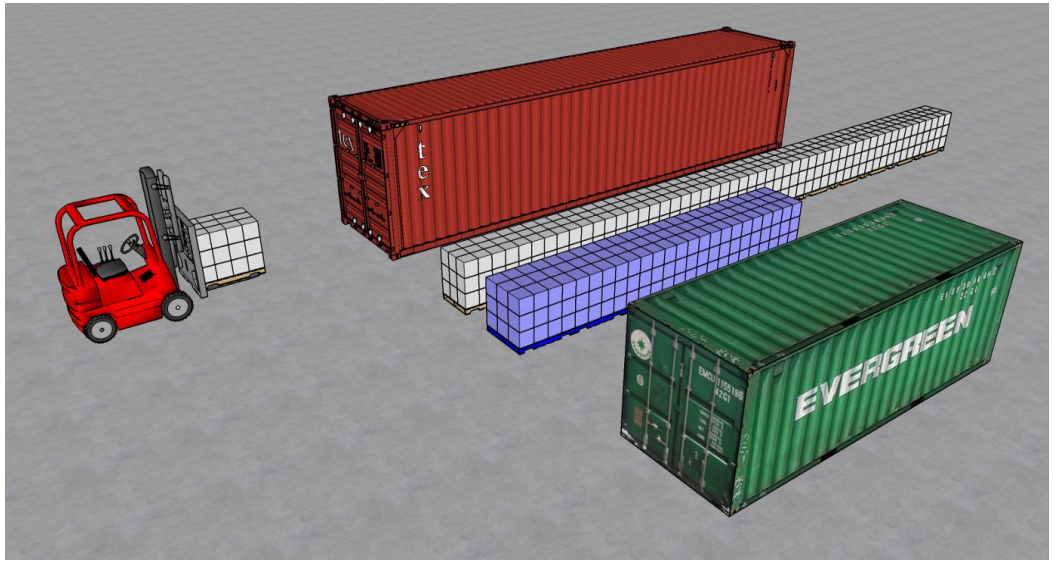
Si S es una solución entonces devolver S

en caso contrario: no hay sol

**S**: Conjunto Solución **C**: Conjunto Candidatos **SELEC()**: F. Selección

# 02

# CONTENEDORES




Tenemos un buque con capacidad de carga **K** toneladas y **n** contenedores cuyos respectivos pesos son **p1…pn**. Sabemos también que la capacidad del buque es menor que la suma total de los pesos de los contenedores.

- Maximizar número de ***contenedores***
- Maximizar número de ***toneladas***

# MAXIMIZAR NÚMERO DE CONTENEDORES

Cargas introducidas w[i]:

| 4 | 9 | 3 | 1 | 6 |
|---|---|---|---|---|

Cargas ordenadas de menor a mayor w[i]:

| 1 | 3 | 4 | 6 | 9 |
|---|---|---|---|---|

Capacidad máxima de carga: W=**6**

- **1+3=4<6**
- **1+3+4=8>6**

# PSEUDOCÓDIGO

Pedimos los datos de los pesos al usuario:

k: capacidad del buque

n: número de contenedores

w[i]: cada uno de los pesos de los contenedores, $1 \leq i \leq n$

Ordenamos los **pesos de los contenedores de menor a mayor** (C)

Aplicamos Greedy:

**num_contenedores = 0**

Mientras num_contenedores no sea una solución y queden contenedores por ver:

num_toneladas += w[i]

si num_toneladas ≤ k **es factible** entonces **++num_contenedores**

Devolvemos num_contenedores

```
for(int i= 0; i < n; ++i){//O(n)
        cout << "Peso del contenedor " << (i+1) << ": ";//O(1)
        cin >> w[i];        //O(1)
        cout << endl;           //O(1)
}
sort(w,w+n);      //O(nlogn)
for(int i = 0; (i < n); ++i){     //O(n)
        num_toneladas += w[i];          //O(1)
        if(num_toneladas <= k)          //O(1)
                ++num_contenedores;  //O(1)
        else
                break;          //O(1)
}
```

# EFICIENCIA TEÓRICA

El algoritmo Greedy tiene eficiencia **O(nlogn)**

# ¿Es óptimo este algoritmo?

**Sí**, este algoritmo Greedy siempre proporciona la solución óptima.

Sea $T = \{c1, \ldots, cn\}$ el conjunto de contenedores y supongamos, sin pérdida de generalidad, que $p1 \leq p2 \leq \ldots \leq pn$. La solución que proporciona el algoritmo Greedy viene dada por $S = \{c1, \ldots, cm\}$ de modo que: $\sum_{i=1}^{m} pi \leq K$ y $\sum_{i=1}^{m+1} pi > K$

En consecuencia tenemos que: $\sum_{i=1}^{m} pi + \sum_{i=1}^{m+1} pi > K \ \forall Q \subseteq T \setminus S,\ Q \neq \emptyset$ , con lo que queda demostrado que los contenedores del conjunto $S$ se corresponden con la solución óptima, no se podrían meter más.

# MAXIMIZAR NÚMERO DE TONELADAS

Cargas introducidas w[i]:

| | | | | | | |
|---|---|---|---|---|---|---|
| **2** | **10** | **20** | **40** | **15** | **35** | **25** |

Cargas ordenadas de mayor a menor w[i]:

| | | | | | | |
|---|---|---|---|---|---|---|
| **40** | **35** | **25** | **20** | **15** | **10** | **2** |

Capacidad máxima de carga: W=**30**

- **40,35>6**
- **25<30**
- **25+20=45>30**

# PSEUDOCÓDIGO

Pedimos los datos de los pesos al usuario:

k: capacidad del buque

n: número de contenedores

w[i]: cada uno de los pesos de los contenedores, $1 \leq i \leq n$

Ordenamos los p**esos de los contenedores de mayor a menor** (C)

Aplicamos Greedy:

**suma_toneladas = 0**

Mientras suma_toneladas no sobrepase k y queden contenedores por ver:

si (w[i]+suma_toneladas) ≤ k **es factible** entonces

**suma_toneladas += w[i]**

Devolvemos num_toneladas

```
for(int i= 0; i < n; ++i){   //O(n)
        cout << "Peso del contenedor " << (i+1) << ": "; //O(1)
        cin >> w[i];          //O(1)
        cout << endl;         //O(1)
    }

sort(w,w+n,mayor);            //O(nlogn)

for(int i = 0; ((i<n)&&(suma_toneladas <= c)); ++i){   //O(n)
        if (w[i]<=c){                                  //O(1)
            if (w[i]+suma_toneladas <= c)              //O(1)
                suma_toneladas += w[i];                //O(1)
        else
            break;                                     //O(1)
        }
    }
```

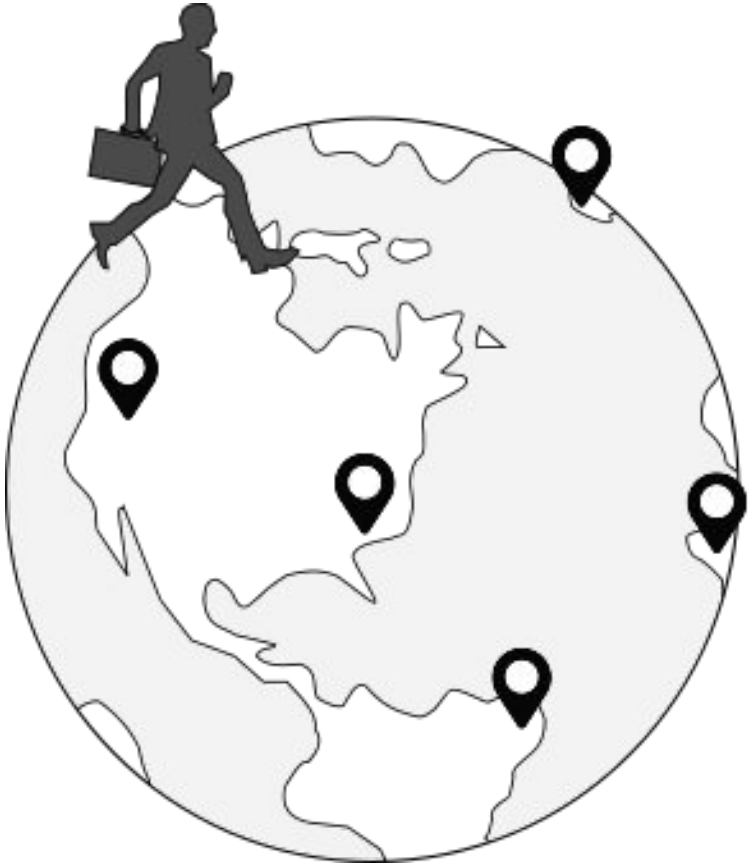
# EFICIENCIA TEÓRICA

El algoritmo Greedy tiene eficiencia **O(nlogn)**

# ¿Es óptimo este algoritmo?

En este caso este algoritmo Greedy **NO** es óptimo. ***CONTRAEJEMPLO***:

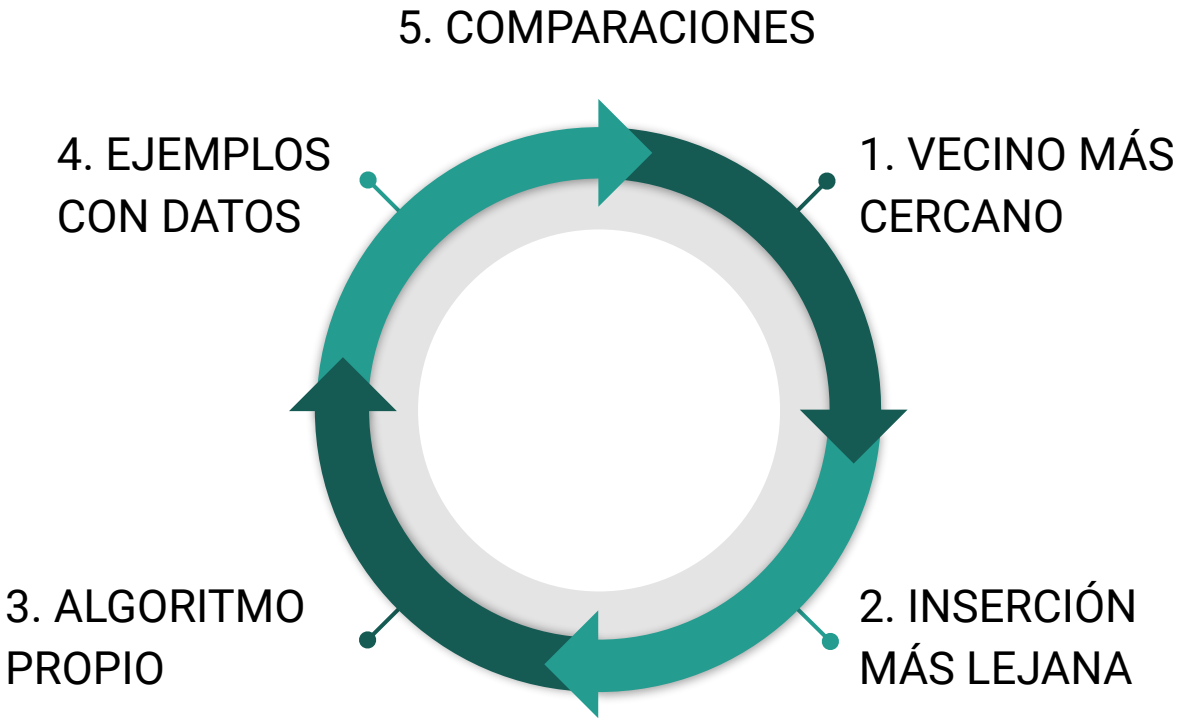

Podemos coger otras combinaciones con contenedores de pesos menores que dé más próximo a 30 que 25.



# 03

# PROBLEMA DEL VIAJANTE DE COMERCIO

Dado un conjunto de ciudades y las distancias entre ellas, hemos de averiguar la **ruta más corta** que visita todas las ciudades una única vez y volver al punto de partida.




# VIAJANTE COMERCIO

# Vecino más cercano

Función CalcularRecorrido(matriz)

1. Buscar una ciudad aleatoria *ciudad_random*
2. Actualizamos element a dicha ciudad y buscamos la más cercana no visitada que será nuestro destino (dst)
3. Marcamos dst como visitada
4. Comprobamos
    a. Si todas las ciudades se han visitado, termina
    b. Si no, vuelve al paso 2

```
while (!pila.isEmpty()){                    //O(n²)

    element = pila.peek();
    i = 1;
    min = Integer.MAX_VALUE;

    while (i <= nNodos){                                    //O(n)

        if (element!=i && ciudadesVisitadas[i] == 0){       //O(1)

            if (min > matriz[element][i]){                  //O(1)

                min = matriz[element][i];
                dst = i;
                minFlag = true;
            }
        }
        i++;
    }
```

# Eficiencia teórica

Eficiencia **O(n²)**

# Inserción más lejana

1. El recorrido parcial inicial se construye a partir de las tres ciudades más al este, al oeste y al norte.
2. El siguiente nodo a insertar es el de la primera fila que no ha sido visitada.
3. El nodo se inserta en la posición en la que provoque menor incremento de la distancia total recorrida.
    a. Para hacerlo se calcula cada posible posición y se guarda la que es mínima y su índice.

# Eficiencia teórica

```
void CalcularRecorrido(int matrix[][], int dim){
  Declaración de variables  ⋯  //O(1)
    for(int i=0; i < dim; i++)  //O(n)
        recorrido[i]=-1;
    for(int i=0; i < dim; i++)  //O(n)
        visitado[i]=0;

  Declaración de variables  ⋯  //O(1)

    if(west_city != -1){  //O(1)
        recorrido[contr]=west_city;
        contr++;
        visitado[west_city]=1;
        contv++;
    }
    if(north_city != -1){  //O(1)
      Similar a west_city
        pero con north_city ⋯
    }
```

```
for(int i =0; i < contr-1; i++){⋯}  //O(n)
                ⋯
for(int i=0; i<dim; i++){  //O(n²)
    if(visitado[i]!=1){  //O(n)
        Declaración e inicialización de variables  ⋯ //O(1)
        for(int k=1; k < contr-1; k++){  //O(n)
            Código con un if ⋯  //O(1)
        }
        Actualizar variables  ⋯  //O(1)

        if(suma_aux < min_suma){  ⋯}  //O(1)
                ⋯

        for(int j = contv; j > index_min; j--){⋯}  //O(n)
            Actualizar variables  ⋯  //O(1)
    }
}
    Imprimir por pantalla los resultados  ⋯  //O(n)
}
```

Eficiencia **O(n²)**

# Algoritmo Propio

1. El recorrido parcial inicial se construye a partir de cuatro ciudades distintas buscadas de forma aleatoria.
2. El siguiente nodo a insertar es el que se acerque más a alguno de los nodos ya insertados.
3. El nodo se inserta en la posición en la que provoque menor incremento de la distancia total recorrida.
    a. Para hacerlo se calcula cada posible posición y se guarda la que es mínima y su índice.

# Eficiencia teórica

```
void CalcularRecorrido(int matriz[][]){
                    …
    for (int j=0; j<num_nodos;++j){              //O(n)
            visitados[j]=false;
    }
                    …
    while(contr < 4){                            //O(1)
    random = rand.nextInt(num_nodos-1);
    if(!visitados[random]){···}                  //O(1)
    }

    for(int i = 0; i < contr-1; i++){···}        //O(n)
                    …
    while (contr < num_nodos){                   //O(n³)
                    …
            for(int i = 0; i < contr-1; i++){    //O(n²)
            for(int j = 0; j < num_nodos; j++){          //O(n)
            if(!visitados[j]){                   //O(1)
                    …
            if(dist_total + ps < dist_menor){···}  //O(1)
            }
            }
    }
```

```
for(int j = 0; j < num_nodos; j++){          //O(n)
            if(!visitados[j]){               //O(1)
                    if(dist_total + ps < dist_menor){...}
            }
    }
                    …
    for(int j = contr; j > nodo_menor; j--){···}  //O(n)
                            …
            for(int i = 0; i < path.length; i++){···} //O(n)
}
```

Eficiencia **O(n³)**

# COMPARACIÓN ENTRE HEURÍSTICAS:

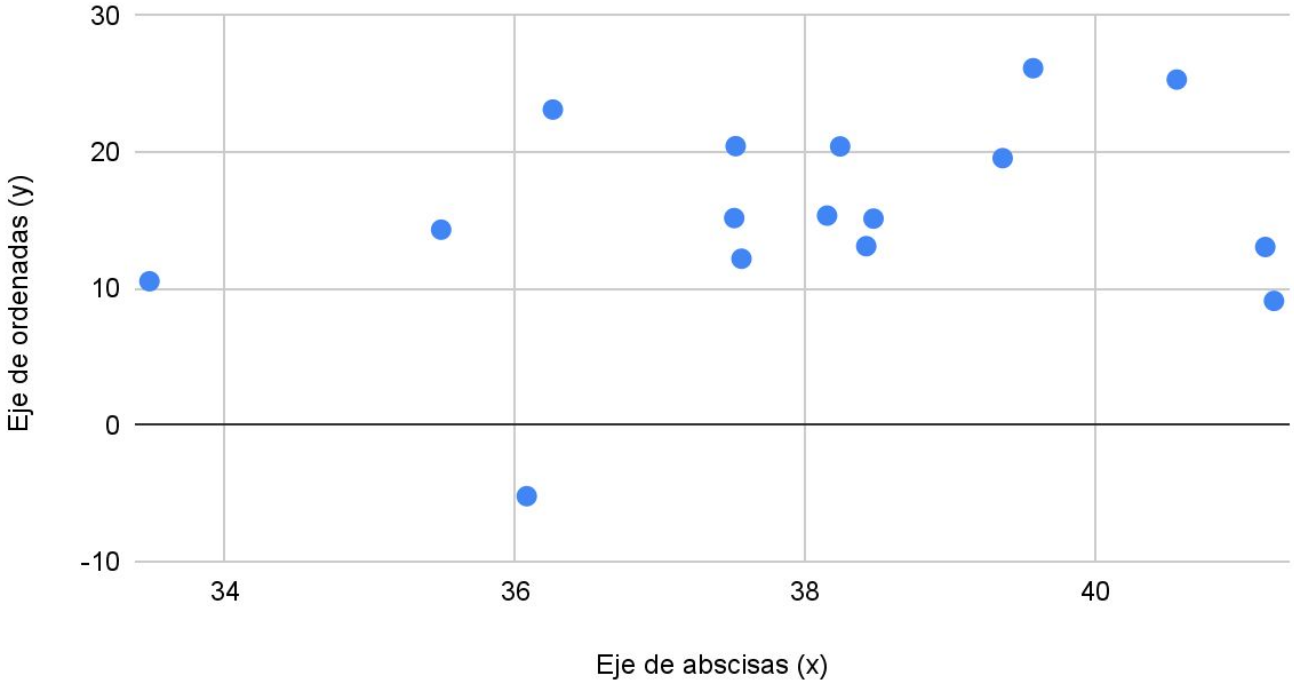
- Vecino más cercano
- Inserción más lejana
- Algoritmo Propio

Con los datos proporcionados:

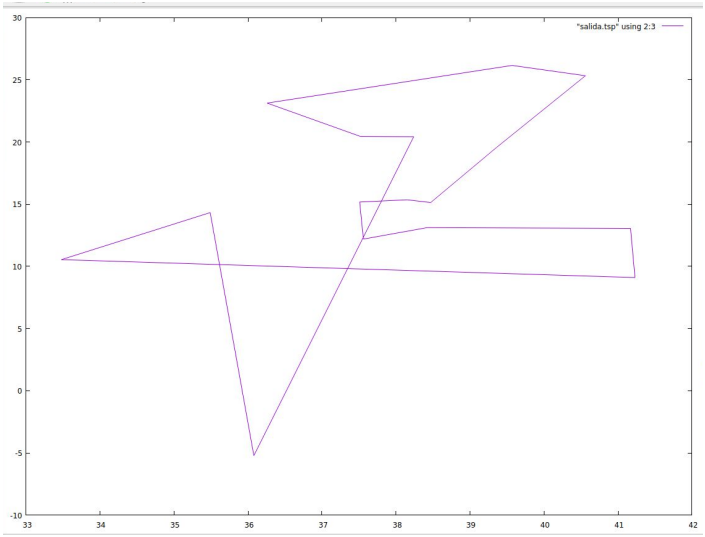
- 16 ciudades
- 29 ciudades
- 76 ciudades

# Ulysses 16 -> 16 ciudades

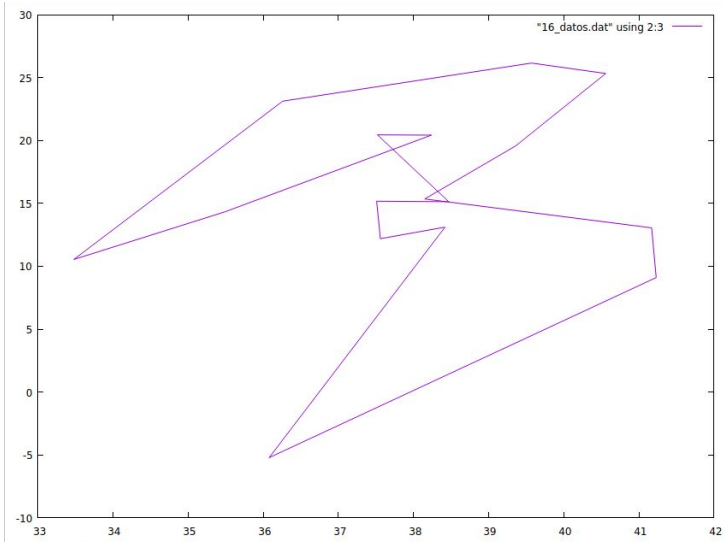
16 ciudades sin ordenar



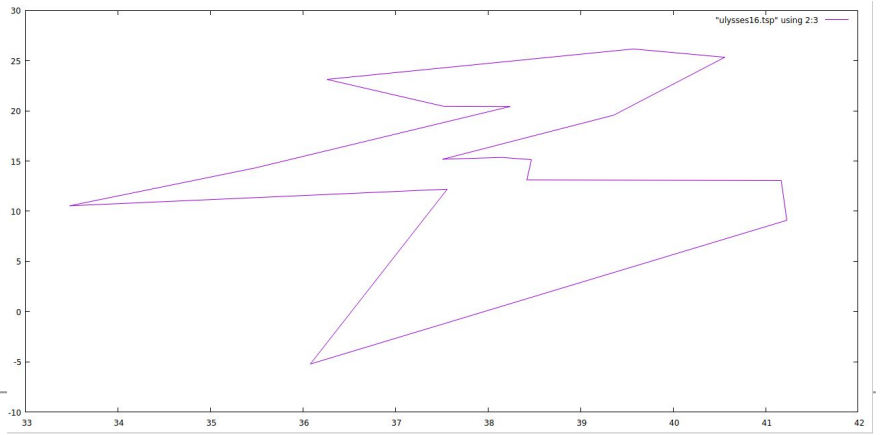
Vecino más cercano




Algoritmo propio



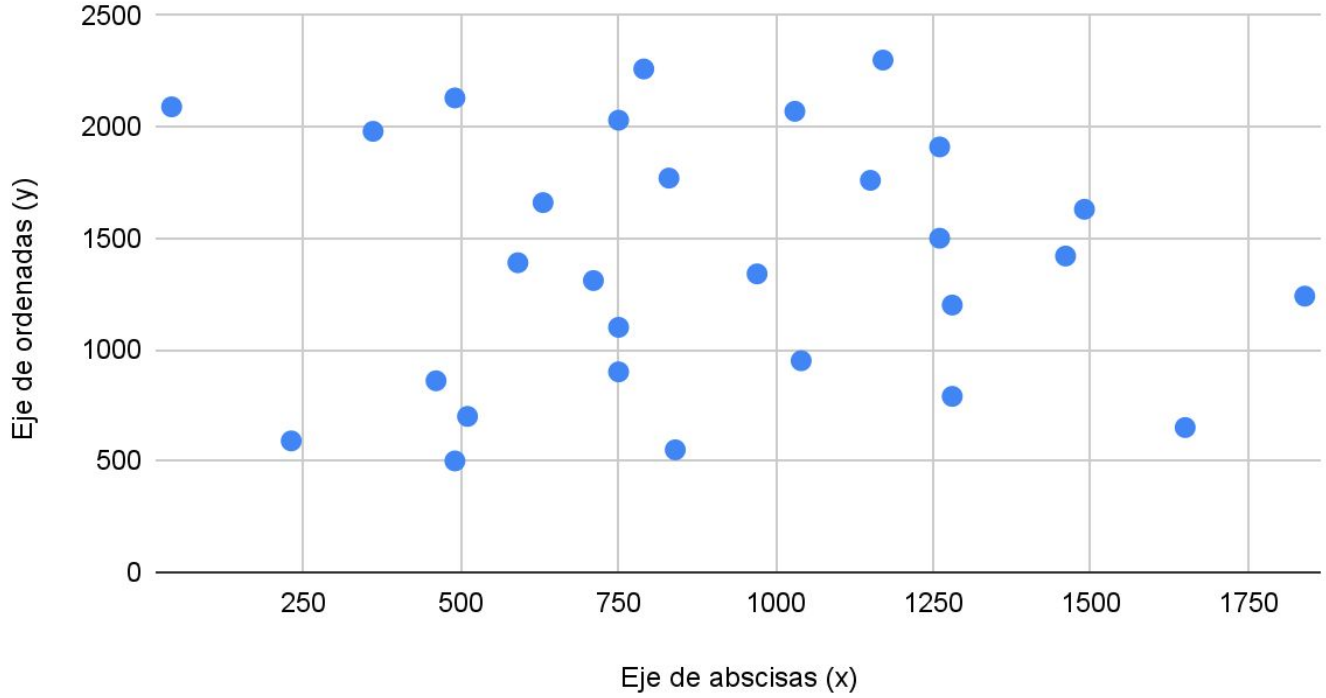

Inserción más lejana



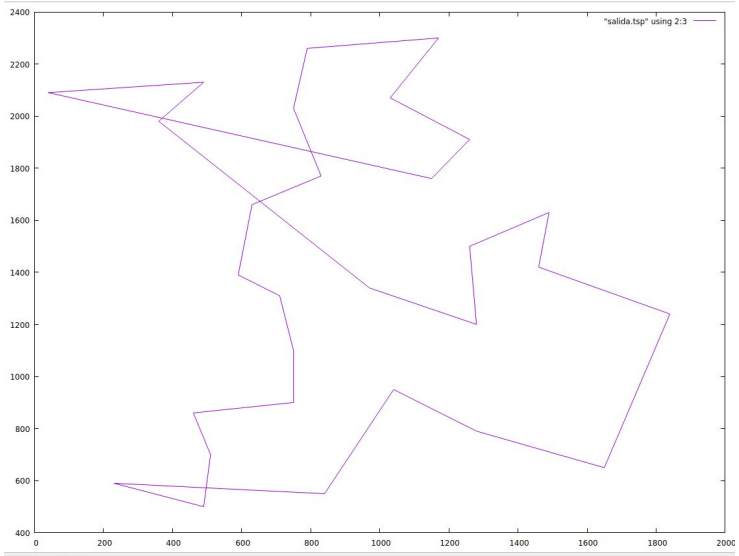

Algorítmica

29 datos sin conectar

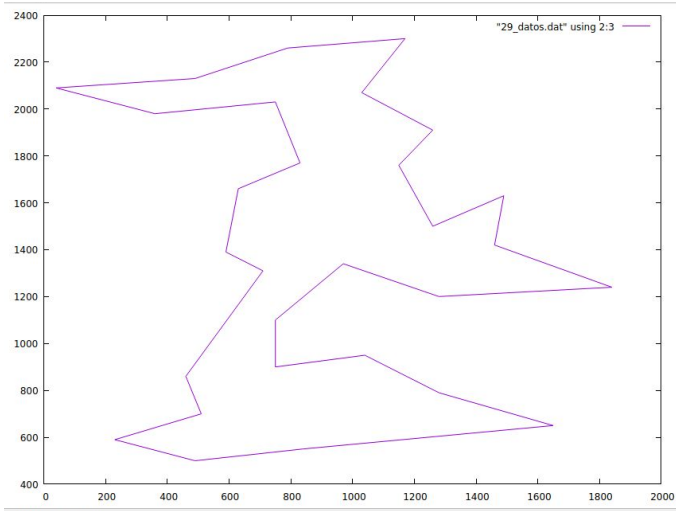
# Bayg 29 -> 29 ciudades



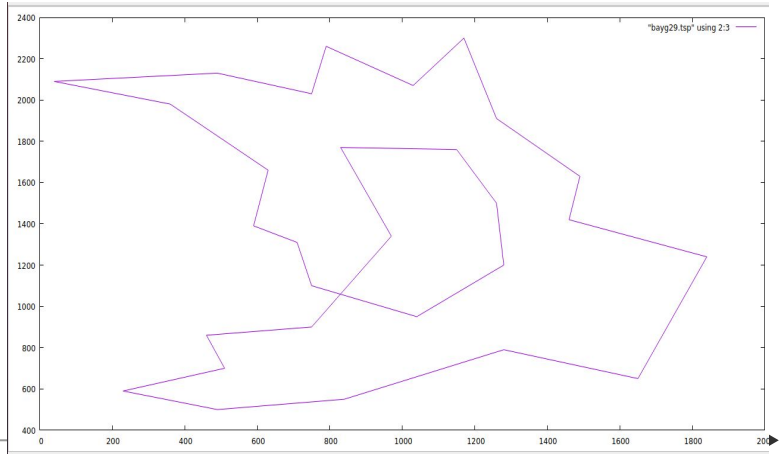
Vecino más cercano




Algoritmo propio



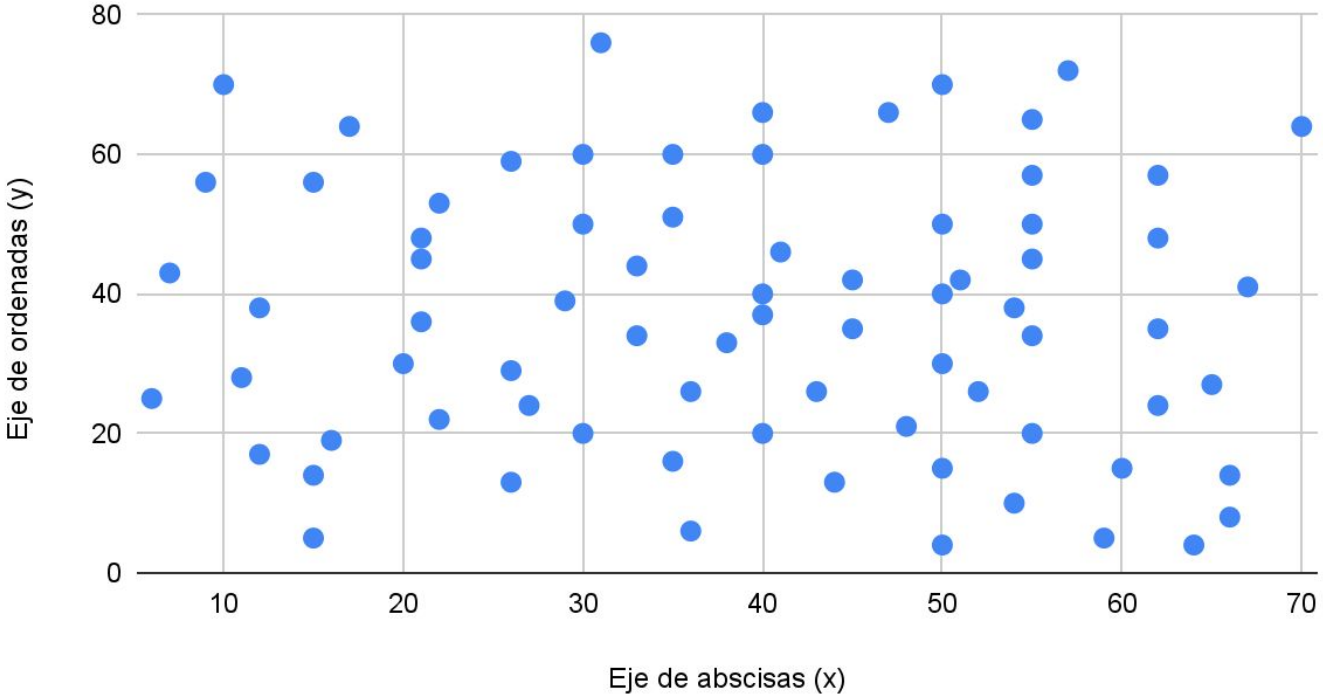

Inserción más lejana



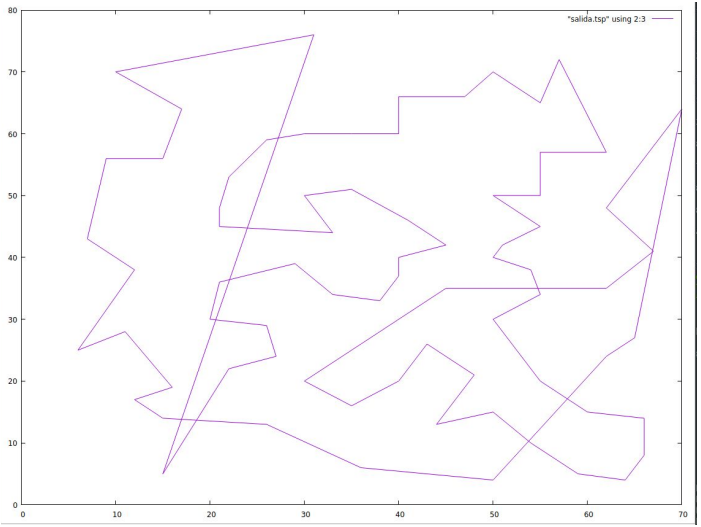

Algorítmica

# Eil 76 -> 76 ciudades

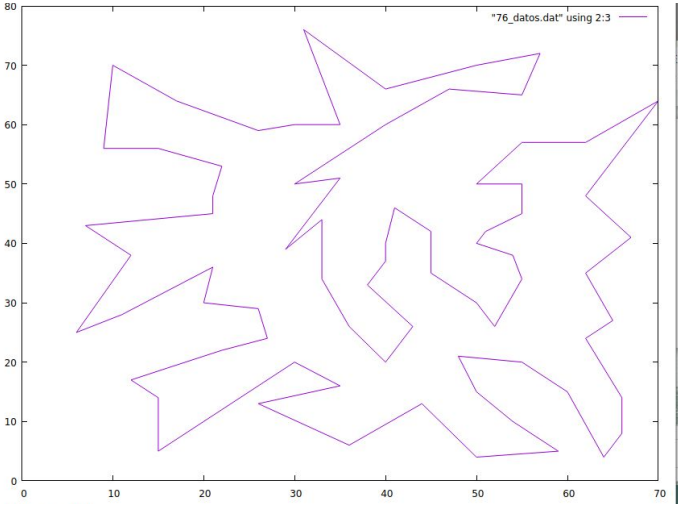
76 datos sin conectar



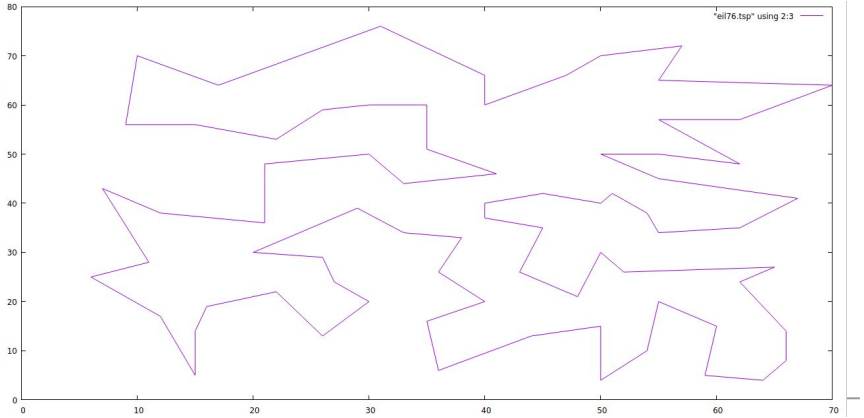
Vecino más cercano




Algoritmo propio




Inserción más lejana



# Comparativa

| *Número nodos* | Vecino más Cercano | Inserción más Lejana | Algoritmo Propio | |
|---|---|---|---|---|
| **16** | 79 | 69 | 83 | *Distancia* |
| | 35.83 | 27.02 | 33.98 | *Tiempo (ms)* |
| **29** | 10200 | 9735 | 9547 | *Distancia* |
| | 36.30 | 22.98 | 39.76 | *Tiempo (ms)* |
| **76** | 662 | 581 | 586 | *Distancia* |
| | 27.19 | 26.87 | 47.08 | *Tiempo (ms)* |

A por el 10 :()

# 04 CONCLUSIONES

Los **algoritmos Greedy** son:

- ➔ Eficientes
- ➔ Diseño e implementación sencillos
- ➔ No siempre dan con la solución óptima
  - ◆ Heurísticas Greedy
- ➔ Muy útiles en problemas sobre GRAFOS

# ¡GRACIAS!



**Azorín Martí, Carmen** **Cribillés Pérez, María** **Ortega Sevilla, Clara** **Torres Fernández, Elena**