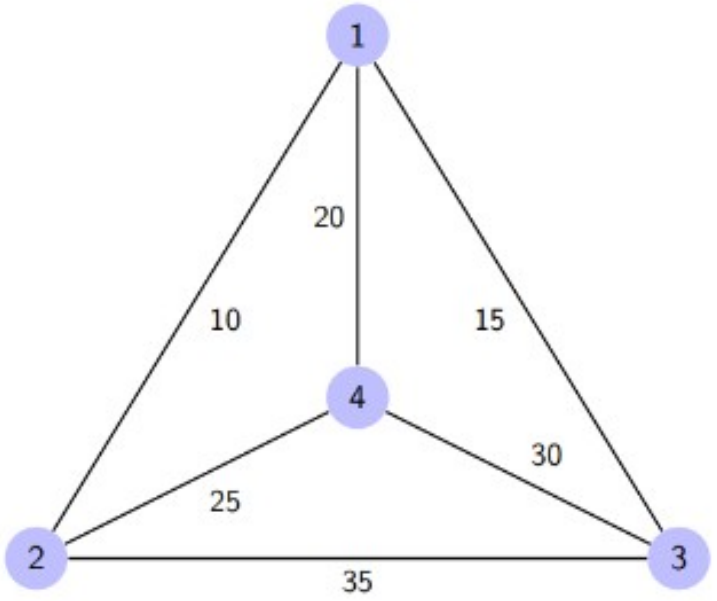
¿Qué es el problema del viajante?

El problema del viajante (TSP) es un problema combinatorio clásico de la informática teórica. El problema pide encontrar el camino más corto en un gráfico con la condición de visitar todos los nodos una sola vez y regresar a la ciudad de origen.
El planteamiento del problema proporciona una lista de ciudades junto con las distancias entre cada ciudad.

Objetivo: Para comenzar desde la ciudad de origen, visite otras ciudades solo una vez y regrese a la ciudad original nuevamente. Nuestro objetivo es encontrar el camino más corto posible para completar la ruta de ida y vuelta.

Aquí se proporciona un gráfico donde 1, 2, 3 y 4 representan las ciudades, y el peso asociado con cada borde representa la distancia entre esas ciudades.



Para el gráfico, la ruta óptima es seguir la ruta de costo mínimo: **1-2-4-3-1.** Y esta ruta más corta costaría
10+25+30+15 =80

Algoritmo para el problema del viajante

Utilizaremos el enfoque de programación dinámica para resolver el problema del viajante (TSP).
Antes de comenzar con el algoritmo, familiaricémonos con algunas terminologías:

- Un gráfico G=(V, E), que es un conjunto de vértices y aristas.
- V es el conjunto de vértices.
- E es el conjunto de aristas.
- Los vértices están conectados a través de aristas.
- Dist(i,j) denota la distancia no negativa entre dos vértices, i y j.

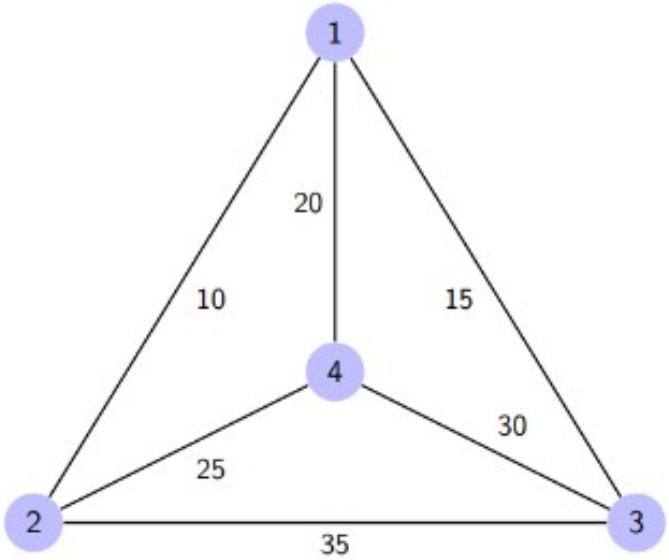
Supongamos que S es el subconjunto de ciudades y pertenece a {1, 2, 3,…, n} donde 1, 2, 3…n son las ciudades e i, j son dos ciudades en ese subconjunto. Ahora el costo (i, S, j) se define de tal manera como la longitud del camino más corto que visita el nodo en S, que tiene exactamente una vez el punto inicial y final como i y j respectivamente.

Por ejemplo, el costo (1, {2, 3, 4}, 1) denota la longitud del camino más corto donde:
- La ciudad inicial es 1.
- Las ciudades 2, 3 y 4 se visitan solo una vez
- El punto final es 1.

El algoritmo de programación dinámica sería:

•Establezca el costo (i, , i) = 0, lo que significa que comenzamos y terminamos en i, y el costo es 0.
•Cuando |S| > 1, definimos costo(i, S, 1) = ∝ donde i !=1 . Porque inicialmente, no sabemos el costo exacto para llegar de la ciudad i a la ciudad 1 a través de otras ciudades.
•Ahora debemos comenzar en 1 y completar el recorrido. Necesitamos seleccionar la siguiente ciudad de tal manera:
costo(i, S, j)=costo mínimo (i, S−{i}, j)+dist(i,j) donde i∈S e i≠j
Para la figura dada, la matriz de adyacencia sería la siguiente:

dist(yo,j)	1	2	3	4
1	0	10	15	20
2	10	0	35	25
3	15	35	0	30
4	20	25	30	0



Veamos cómo funciona nuestro algoritmo:

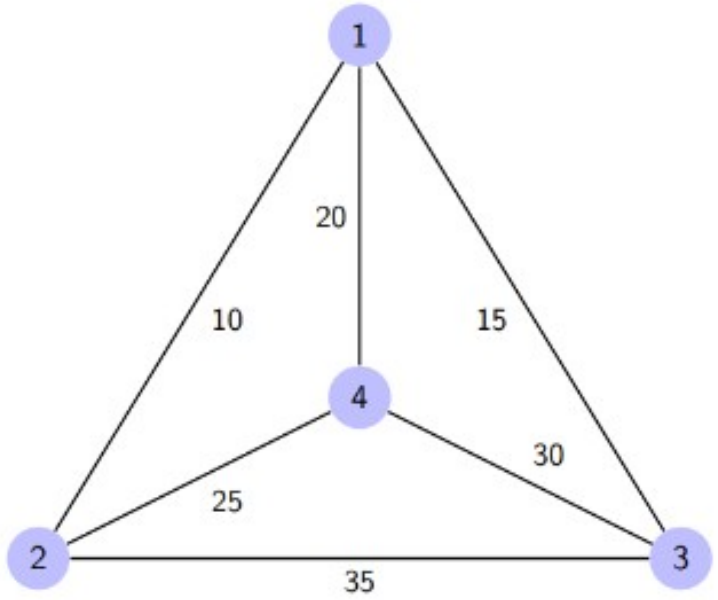
Paso 1) Estamos considerando nuestro viaje comenzando en la ciudad 1, visitar otras ciudades una vez y regresar a la ciudad 1.

Paso 2) S es el subconjunto de ciudades. Según nuestro algoritmo, para todo $|S| > 1$, fijaremos el coste de distancia $(i, S, 1) = \propto$. Aquí, coste (i, S, j) significa que empezamos en la ciudad i, visitamos las ciudades de S una vez y ahora estamos en la ciudad j. Fijamos este coste de ruta como infinito porque aún no conocemos la distancia. Por tanto, los valores serán los siguientes:
Costo $(2, \{3, 4\}, 1) = \propto$; la notación indica que comenzamos en la ciudad 2, pasamos por las ciudades 3, 4 y llegamos a 1. Y el costo del camino es infinito. Similarmente-
$costo(3, \{2, 4\}, 1) = \propto$
$costo(4, \{2, 3\}, 1) = \propto$

Paso 3) Ahora, para todos los subconjuntos de S, necesitamos encontrar lo siguiente:
$costo(i, S, j) = costo\ mínimo\ (i, S-\{i\}, j) + dist(i,j)$, donde $j \in S$ e $i \neq j$
Eso significa la ruta de costo mínimo para comenzar en i, pasar por el subconjunto de ciudades una vez y regresar a la ciudad j. Considerando que el viaje comienza en la ciudad 1, el costo óptimo del camino sería = costo(1, {otras ciudades}, 1).

Averigüemos cómo podríamos lograrlo:

Ahora S = {1, 2, 3, 4}. Hay cuatro elementos. Por lo tanto, el número de subconjuntos será 2 ^ 4 o 16. Esos subconjuntos son:



1) |S| = Nulo:
{Φ}

2) |S| = 1:
{{1}, {2}, {3}, {4}}

3) |S| = 2:
{{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}}

4) |S| = 3:
{{1, 2, 3}, {1, 2, 4}, {2, 3, 4}, {1, 3, 4}}

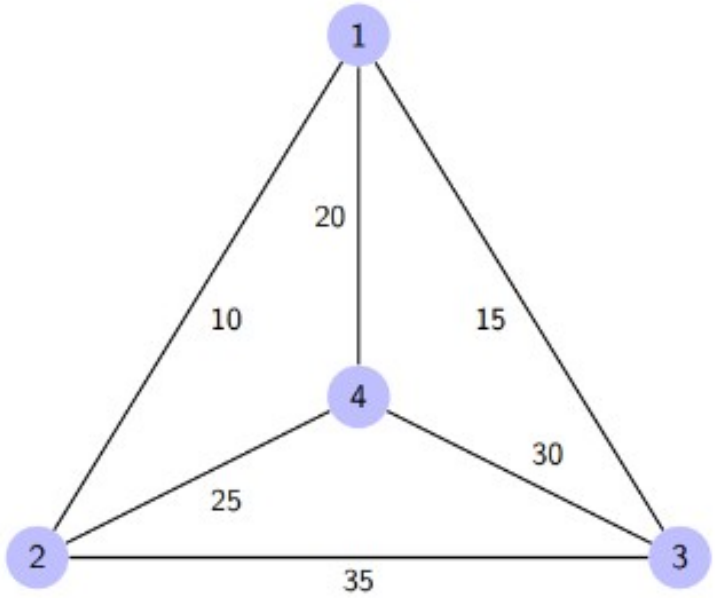
5) |S| = 4:
{{1, 2, 3, 4}}

Como comenzamos en 1, podríamos descartar los subconjuntos que contienen la ciudad 1.
El cálculo del algoritmo:

1) |S| =Φ:
costo (2, Φ, 1) = dist(2, 1) = 10
costo (3, Φ, 1) = dist(3, 1) = 15
costo (4, Φ, 1) = dist(4, 1) = 20

2) |S| = 1:
costo (2, {3}, 1) = dist(2, 3) + costo (3, Φ, 1) = 35+15 = 50
costo (2, {4}, 1) = dist(2, 4) + costo (4, Φ, 1) = 25+20 = 45
costo (3, {2}, 1) = dist(3, 2) + costo (2, Φ, 1) = 35+10 = 45
costo (3, {4}, 1) = dist(3, 4) + costo (4, Φ, 1) = 30+20 = 50
costo (4, {2}, 1) = dist(4, 2) + costo (2, Φ, 1) = 25+10 = 35
costo (4, {3}, 1) = dist(4, 3) + costo (3, Φ, 1) = 30+15 = 45

3) |S| = 2:
costo (2, {3, 4}, 1) = mínimo [dist[2,3]+Costo(3,{4},1) = 35+50 = 85,
dist[2,4]+Costo(4,{3},1) = 25+45 = 70] = 70
costo (3, {2, 4}, 1) = mínimo [dist[3,2]+Costo(2,{4},1) = 35+45 = 80,
dist[3,4]+Costo(4,{2},1) = 30+35 = 65] = 65
costo (4, {2, 3}, 1) = mínimo [dist[4,2]+Costo(2,{3},1) = 25+50 = 75
dist[4,3]+Costo(3,{2},1) = 30+45 = 75] = 75



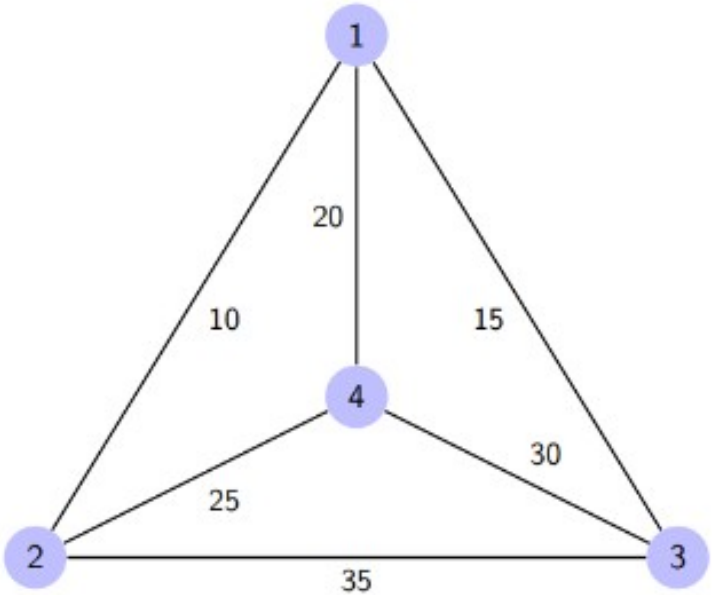
4) |S| = 3:

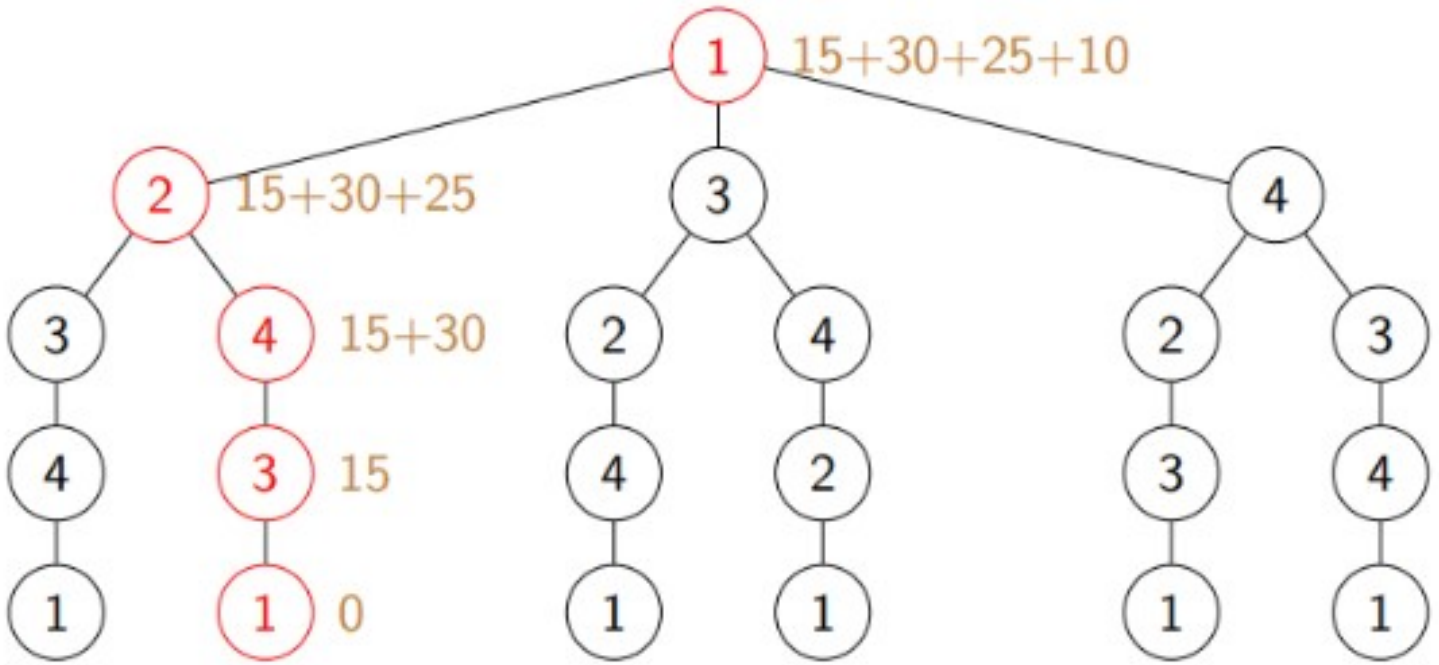
costo (1, {2, 3, 4}, 1) = min [dist[1,2]+Costo(2,{3,4},1) = 10+70 = 80

dist[1,3]+Costo(3,{2,4},1) = 15+65 = 80

dist[1,4]+Costo(4,{2,3},1) = 20+75 = 95] = 80

Entonces la solución óptima sería **1-2-4-3-1**





Pseudocódigo

```
Algorithm: Traveling-Salesman-Problem

Cost (1, {}, 1) = 0

for s = 2 to n do
       for all subsets S belongs to {1, 2, 3, … , n} of size s
              Cost (s, S, 1) = Infinity for all i Є S and i ≠ 1 Cost (i, S,
j) = min {Cost (i, S – {i}, j) +        dist(i, j) for j Є S and i ≠ j}

Return min(i) Cost (i, {1, 2, 3, …, n}, j) + d(j, i)
```

Implementacion en Python

```
from sys import maxsize
from itertools, import permutations
 V = 4
def tsp(graph, s):
          vertex = []
          for i in range(V):
          if i != s:
                    vertex.append(i)
           min_cost = maxsize
          next_permutation=permutations(vertex)
          for i in next_permutation:
                    current_cost = 0
                    k = s
                    for j in i:
                              current_cost += graph[k][j]
                              k = j
          current_cost += graph[k][s]
          min_cost = min(min_cost, current_cost)
          return min_cost
graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30,
0]]
s = 0
print(tsp(graph, s))
```

Salida:

```
80
```

Ejemplo QuickSort

```
C:\Users\Luis Cabrera Benito\Desktop
λ python quicksort.py
Antes de ordenarlo:
[5, 1, 2, 1, 1, 5, 5, 1, 5, 1, 99, 251, 234, 12, 121, 312, 123, 123, 12, 312,
321, 312, 31, 23, 12, 3123, 123]
Después de ordenarlo:
[1, 1, 1, 1, 1, 2, 5, 5, 5, 12, 12, 12, 23, 31, 99, 121, 123, 123, 123, 231
, 234, 312, 312, 312, 321, 3123]

C:\Users\Luis Cabrera Benito\Desktop
λ
```

```python
def quicksort(arreglo, izquierda, derecha):
    if izquierda < derecha:
        indiceParticion = particion(arreglo, izquierda, derecha)
        quicksort(arreglo, izquierda, indiceParticion)
        quicksort(arreglo, indiceParticion + 1, derecha)
```

Ejercicio

miLista = [34,93,19,58,12,28,95,37,23,80,57,82,55,48,21,39,53,65,30,32,84,64,44,68,36]