🟦 Minimaler Spannbaum (Minimum Spanning Tree, MST)

1️⃣ Grundidee

Ein **minimaler Spannbaum (MST)** ist ein Teilgraph eines **ungerichteten, gewichteten Graphen**, der:

- **alle Knoten verbindet**
- **keine Zyklen enthält**
- die **Summe der Kantengewichte minimiert**

➡️ Ein Spannbaum mit minimalen Gesamtkosten.

2️⃣ Voraussetzungen

- **Ungerichteter Graph**
- **Gewichtete Kanten**
- Der Graph muss **zusammenhängend** sein (sonst erhält man einen minimalen Spannwald)

3️⃣ Eigenschaften & Laufzeiten (Übersicht)

Eigenschaft	Wert
Anzahl Kanten	V − 1
Zyklen	keine
Gewicht	minimal
Existenz	eindeutig, wenn alle Gewichte verschieden sind

4️⃣ Typische Algorithmen für MST

🔹 Prim-Algorithmus

- Greedy-Algorithmus
- Start bei einem Knoten
- wächst den Baum schrittweise

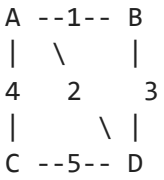
🔹 Kruskal-Algorithmus

- Greedy-Algorithmus
- sortiert alle Kanten
- fügt Kanten hinzu, wenn kein Zyklus entsteht

➡️ Beide liefern **denselben minimalen Spannbaum** (bei gleichen Gewichten evtl. andere Struktur).

5️⃣ Beispiel

Graph:



Minimaler Spannbaum:

- A–B (1)
- B–D (3)
- A–C (4)

Gesamtgewicht = **8**

6️⃣ Besonderheiten / Prüfungsrelevante Hinweise

- MST ist **nicht** dasselbe wie kürzester Weg
- MST minimiert **Gesamtgewicht**, nicht einzelne Pfade
- Nur für **ungerichtete Graphen**
- Sehr häufige Prüfungsfrage:
  - *Unterschied MST vs. Dijkstra?*

7️⃣ Vor- und Nachteile

Vorteile

- optimale Verbindung aller Knoten
- nützlich für Netzwerke (Kabel, Straßen, Leitungen)

Nachteile

- keine Pfadinformationen
- nicht anwendbar auf gerichtete Graphen

🧠 Merksatz für die Prüfung

Ein minimaler Spannbaum verbindet alle Knoten eines ungerichteten Graphen ohne Zyklen und mit minimalem Gesamtgewicht.

🎱 Python-Implementierung (Prim-Algorithmus)

In [ ]:
```python
import heapq

def prim(graph, start):
    visited = set()
    mst = []
    pq = [(0, start, None)]  # (gewicht, knoten, vorgänger)

    while pq:
        weight, node, parent = heapq.heappop(pq)
        if node in visited:
            continue

        visited.add(node)
        if parent is not None:
            mst.append((parent, node, weight))

        for neighbor, w in graph[node]:
            if neighbor not in visited:
                heapq.heappush(pq, (w, neighbor, node))

    return mst


# Beispiel
graph = {
    "A": [("B", 1), ("C", 4)],
    "B": [("A", 1), ("D", 3), ("C", 2)],
    "C": [("A", 4), ("B", 2), ("D", 5)],
    "D": [("B", 3), ("C", 5)]
}

print(prim(graph, "A"))
```