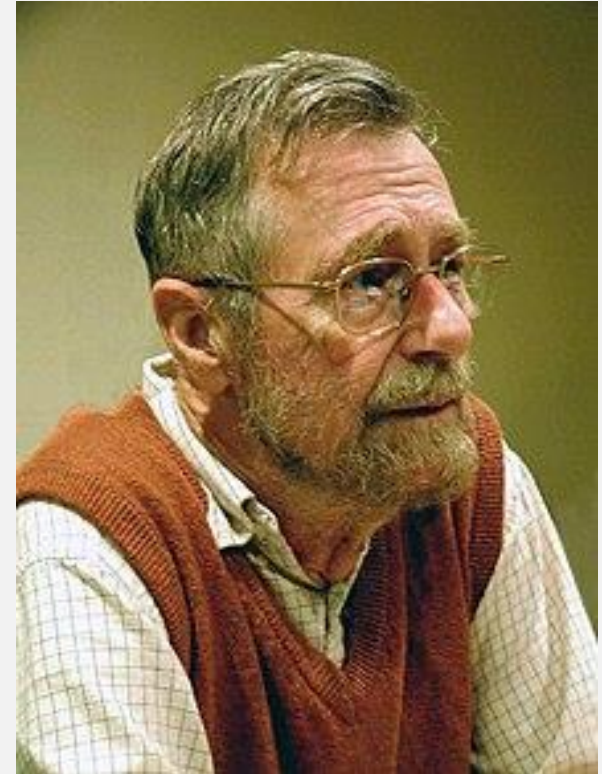


# RUND UM DIJKSTRA

„In der Informatik geht es so wenig um Computer,  
wie es in der Astronomie um Teleskope geht.“

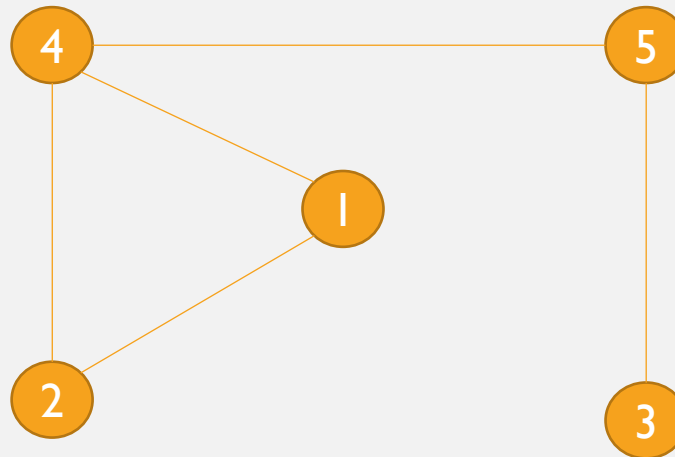
# EDSGER W. DIJKSTRA

- \* 11. Mai 1930 in Rotterdam;
- † 6. August 2002 in Nuenen
- Shunting-yard-Algorithmus
- Dijkstra Algorithmus
- Verfechter der strukturierten Programmierung
- 1972 – Turing Award



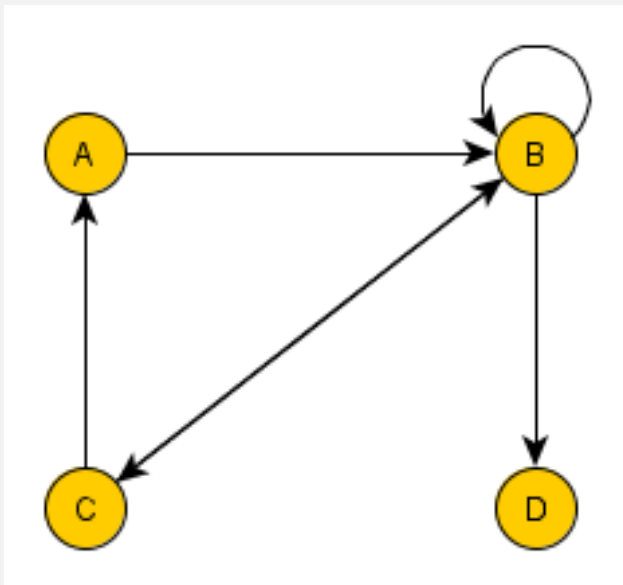
# GRAPHEN

- Ein Graph  $G$  ist ein geordnetes  $(V, E)$ , wobei  $V$  eine Menge von Knoten (englisch vertex/vertices, oft auch Ecken genannt) und  $E$  eine Menge von Kanten (engl. edge/edges) bezeichnet.



# PROBLEM

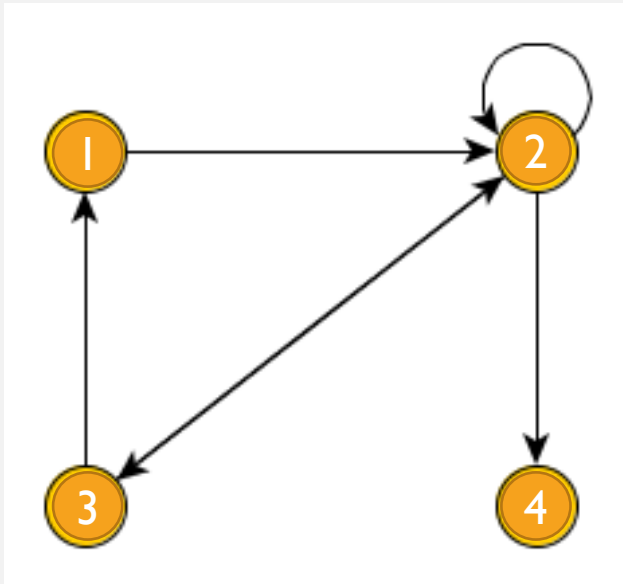
Gegeben ist der folgenden Graph:



Wie könnte man diesen Graphen  
im Computer speichern?  
Wie könnte man die  
Informationen dieses Graphen im  
Computer darstellen?



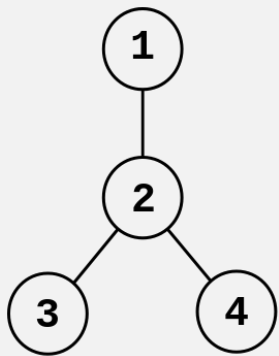
# VARIANTE I



$$V = \{1, 2, 3, 4\}$$

$$E = \{(1, 2), (2, 2), (2, 4), (3, 2), (2, 3), (3, 1)\}$$

# ADJAZENZMATRIX



	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	0
4	0	1	0	0

Knoten

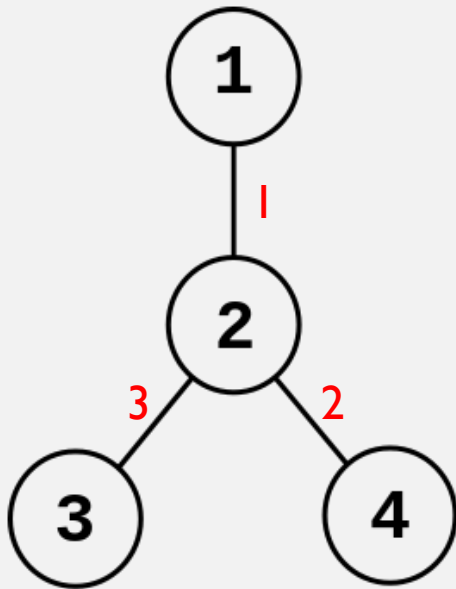
Knoten

1, wenn Kante  
0, wenn keine Kante

Eine Adjazenzmatrix (manchmal auch Nachbarschaftsmatrix) eines Graphen ist eine **Matrix**, die speichert, welche Knoten des Graphen durch eine Kante verbunden sind.

# INZIDENZMATRIX

- Bei einer Inzidenzmatrix wird jeder Knoten durch eine Zeile und jede Kante durch eine Spalte beschrieben.

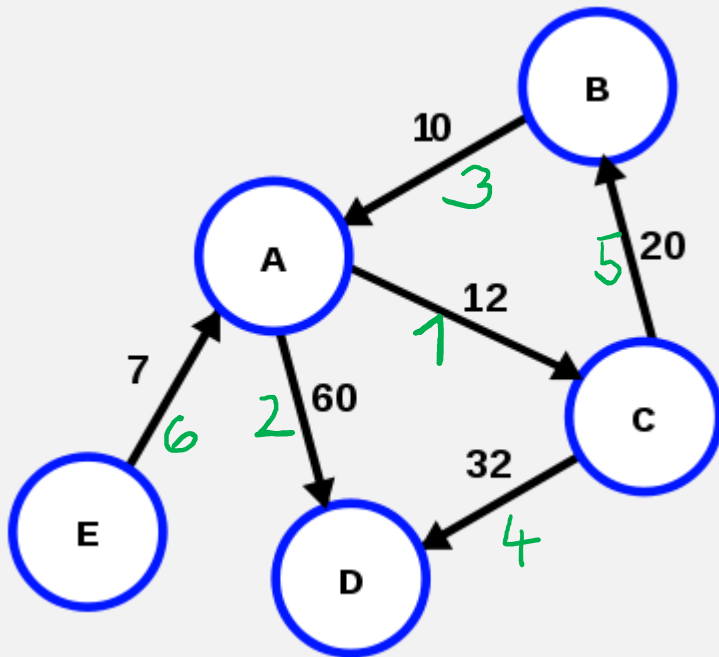


		Knoten			
		1	2	3	4
Kanten	1	1	1	0	0
	2	0	1	0	1
	3	0	1	1	0

1, wenn Knoten mit Kante inzidiert  
0, Knoten nicht mit Kante inzidiert

# ÜBUNG

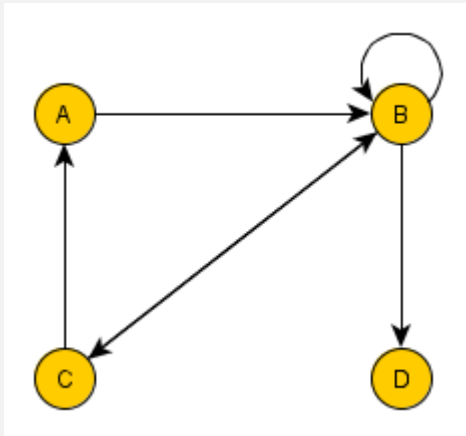
Gegeben sei der folgende Graph. Stellen Sie die Adjazenz- und die Inzidenzmatrix auf. Recherchieren Sie das Problem des gewichteten Graphen und der Inzidenzmatrix.



$$A = \begin{pmatrix} 0 & 0 & 12 & 60 & 0 \\ 10 & 0 & 0 & 0 & 0 \\ 0 & 20 & 0 & 32 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 \end{pmatrix}$$



# DATENSTRUKTUREN



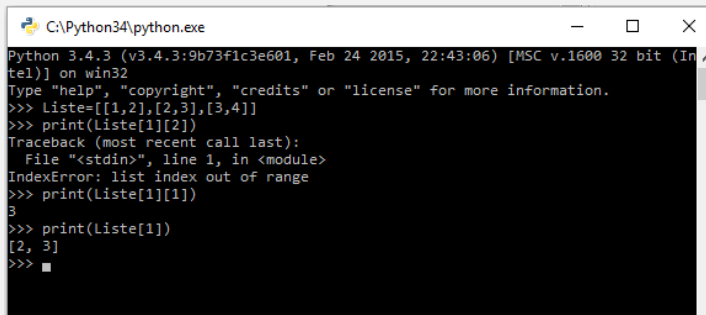
```
knotenliste = ['A', 'B', 'C', 'D']  
adjazenzmatrix = [[0, 1, 0, 0],  
[0, 1, 1, 1], [1, 1, 0, 0],  
[0, 0, 0, 0]]
```

# WIEDERHOLUNG

- Zugriff auf zweidimensionale Liste:

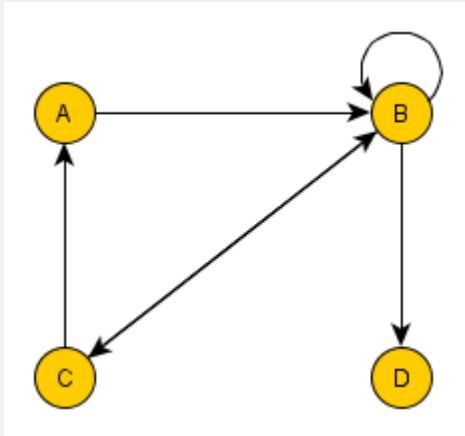
```
Liste = [[1,2], [2,3], [3,4]]  
print(Liste[0][1])  
print(Liste[1])
```

- in Python Command Shell testen



```
C:\Python34\python.exe  
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license()" for more information.  
>>> Liste=[[1,2],[2,3],[3,4]]  
>>> print(Liste[1][2])  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: list index out of range  
>>> print(Liste[1][1])  
3  
>>> print(Liste[1])  
[2, 3]  
>>> ■
```

# AUFGABE I (LEICHT)



```
knotenliste = ['A', 'B', 'C', 'D']  
adjazenzmatrix = [[0, 1, 0, 0],  
[0, 1, 1, 1], [1, 1, 0, 0],  
[0, 0, 0, 0]]
```

Entwickle eine Funktion `existiertKnoten(nameKnoten)`, die überprüft, ob ein vorgegebener Knoten im Graph vorkommt.

(Ausgabe: ja/nein oder Rückgabe True/False)



# LÖSUNG AUFGABE I

```
def existiertKnoten(Knoten):  
    if Knoten in knotenliste:  
        print("Ja")  
    else:  
        print("Nein")
```

Python arbeitet toll mit  
Listen!



# AUFGABE 2 (MITTEL)

- Entwickle eine Funktion `existiertKante(nameStartKnoten, nameZielKnoten)`, die überprüft, ob es eine Kante zwischen den übergebenen Knoten gibt.  
Aufruf: `existiertKante("A", "D")`
- Tipp: Mit `knotenliste.index('A')` kann man den Index („Stelle“) des übergebenen Bezeichners in der Knotenliste bestimmen. (An welcher Stelle steht ein A)
- Tipp 2: Wie kann man diese Information aus der Adjazenzliste „per Hand“ bestimmen?

# LÖSUNG AUFGABE 2

```
def existiertKante(Startknoten, Endknoten):  
    indexstart=knotenliste.index(Startknoten)  
    indexende=knotenliste.index(Endknoten)  
    if adjazenzmatrix[indexstart][indexende]==1:  
        print("Kante existiert")  
    else:  
        print("Kante existiert nicht")
```

Eigentlich fehlt noch ein Test,  
ob Knoten existieren.  
Wie könnte man das  
umsetzen?

# AUFGABE 3

- Entwickle eine Funktion `getAlleNachbarn(nameKnoten)`, die sämtliche Nachbarn eines vorgegebenen Knotens ermittelt und zurückgibt.  
Aufruf: `getAlleNachbar ("B")`
- Hinweis: um Zahlen in Zeichen umzuwandeln, nutze die Funktion `chr()`

# LÖSUNG AUFGABE 3

```
def getAlleNachbar(Knoten) :  
    index=knotenliste.index(Knoten)  
    nachbarn=[]  
    liste=adjazenzmatrix[index]  
    count=0  
    for ele in liste:  
        if ele==1:  
            nachbarn.append(chr(count + 65))  
            count=count+1  
    print(nachbarn)
```



```

import sys

class Vertex:
    def __init__(self, node):
        self.id = node
        self.adjacent = {}
        # Set distance to infinity for all nodes
        self.distance = sys.maxint
        # Mark all nodes unvisited
        self.visited = False
        # Predecessor
        self.previous = None

    def add_neighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def get_connections(self):
        return self.adjacent.keys()

    def get_id(self):
        return self.id

    def get_weight(self, neighbor):
        return self.adjacent[neighbor]

    def set_distance(self, dist):
        self.distance = dist

    def get_distance(self):
        return self.distance

    def set_previous(self, prev):
        self.previous = prev

    def set_visited(self):
        self.visited = True

    def __str__(self):
        return str(self.id) + ' adjacent: ' + str([x.id for x in self.adjacent])

class Graph:
    def __init__(self):
        self.vert_dict = {}
        self.num_vertices = 0

    def __iter__(self):
        return iter(self.vert_dict.values())

    def add_vertex(self, node):
        self.num_vertices = self.num_vertices + 1
        new_vertex = Vertex(node)
        self.vert_dict[node] = new_vertex
        return new_vertex

    def get_vertex(self, n):
        if n in self.vert_dict:
            return self.vert_dict[n]
        else:
            return None

    def add_edge(self, frm, to, cost = 0):
        if frm not in self.vert_dict:
            self.add_vertex(frm)
        if to not in self.vert_dict:
            self.add_vertex(to)

        self.vert_dict[frm].add_neighbor(self.vert_dict[to], cost)
        self.vert_dict[to].add_neighbor(self.vert_dict[frm], cost)

    def get_vertices(self):
        return self.vert_dict.keys()

    def set_previous(self, current):
        self.previous = current

    def get_previous(self, current):
        return self.previous

```



```

def shortest(v, path):
    ''' make shortest path from v.previous'''
    if v.previous:
        path.append(v.previous.get_id())
        shortest(v.previous, path)
    return

import heapq

def dijkstra(aGraph, start, target):
    print '''Dijkstra's shortest path'''
    # Set the distance for the start node to zero
    start.set_distance(0)

    # Put tuple pair into the priority queue
    unvisited_queue = [(v.get_distance(),v) for v in aGraph]
    heapq.heapify(unvisited_queue)

    while len(unvisited_queue):
        # Pops a vertex with the smallest distance
        uv = heapq.heappop(unvisited_queue)
        current = uv[1]
        current.set_visited()

        #for next in v.adjacent:
        for next in current.adjacent:
            # if visited, skip
            if next.visited:
                continue
            new_dist = current.get_distance() + current.get_weight(next)

            if new_dist < next.get_distance():
                next.set_distance(new_dist)
                next.set_previous(current)
                print 'updated : current = %s next = %s new_dist = %s' \
                    %(current.get_id(), next.get_id(), next.get_distance())
            else:
                print 'not updated : current = %s next = %s new_dist = %s' \
                    %(current.get_id(), next.get_id(), next.get_distance())

        # Rebuild heap
        # 1. Pop every item
        while len(unvisited_queue):
            heapq.heappop(unvisited_queue)
        # 2. Put all vertices not visited into the queue
        unvisited_queue = [(v.get_distance(),v) for v in aGraph if not v.visited]
        heapq.heapify(unvisited_queue)

if __name__ == '__main__':

    g = Graph()

    g.add_vertex('a')
    g.add_vertex('b')
    g.add_vertex('c')
    g.add_vertex('d')
    g.add_vertex('e')
    g.add_vertex('f')

    g.add_edge('a', 'b', 7)
    g.add_edge('a', 'c', 9)
    g.add_edge('a', 'f', 14)
    g.add_edge('b', 'c', 10)
    g.add_edge('b', 'd', 15)
    g.add_edge('c', 'd', 11)
    g.add_edge('c', 'f', 2)
    g.add_edge('d', 'e', 6)
    g.add_edge('e', 'f', 9)

    print 'Graph data:'
    for v in g:
        for w in v.get_connections():
            vid = v.get_id()
            wid = w.get_id()
            print '( %s, %s, %3d)' % ( vid, wid, v.get_weight(w))

    dijkstra(g, g.get_vertex('a'), g.get_vertex('e'))

    target = g.get_vertex('e')
    path = [target.get_id()]
    shortest(target, path)
    print 'The shortest path : %s' %(path[::-1])

```