

# Ausgewählte Methoden der Künstlichen Intelligenz

## Vorlesung 2, Praktikum 2

Prof. Dr. Tatyana Ivanovska

`<t.ivanovska@oth-aw.de>`

24. Oktober, 25. Oktober 2023

- Die Komplexität von Algorithmen (Groß  $\mathcal{O}$ -Notation)  
(Wiederholung)
- Graphen

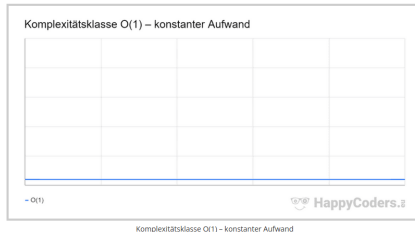
- **Zeit** (englisch: "computational time complexity"): die Ausführungszeit in Abhängigkeit der Größe der Eingabedaten
  - ▶ Um wie viel verlangsamt sich ein Algorithmus, wenn die Menge der Eingabedaten größer wird?
- **Platz** (englisch: "space complexity"): wie viel zusätzlichen Speicherplatz ein Algorithmus in Abhängigkeit von der Größe der Eingabedaten benötigt.
  - ▶ Damit ist nicht der Speicherbedarf für die Eingabedaten selbst gemeint.
- Wir werden vor allem über die Zeitkomplexität reden
- Die Komplexitätsklassen können nur verwendet werden , um Algorithmen einzuordnen, nicht aber, um deren **genaue Laufzeit oder Platzbelegung** zu berechnen
- Die Änderung der benötigten Zeit (oder Platz) in Abhängigkeit von der Änderung der Eingabegröße

<https://www.happycoders.eu/de/algorithmen/o-notation-zeitkomplexitaet/>

- Der systematische Weg, die Obergrenze der Laufzeit eines Algorithmus auszudrücken
- Es berechnet die Worst-Case-Zeitkomplexität oder die maximale Zeit, die ein Algorithmus benötigt, um die Ausführung abzuschließen.
- am häufigsten verwendete Notation

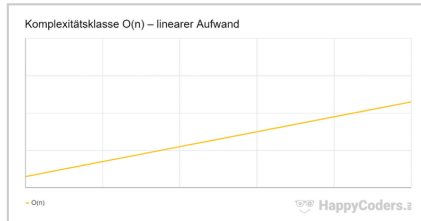
# $O(1)$ : konstanter Aufwand

- Ausgesprochen: "O von 1"
- Die Ausführungszeit ist konstant, also unabhängig von der Anzahl der Eingabeelemente  $n$ .
- Beispiele:
  - ▶ Zugriff auf einen Wert mit einem Array-Index:  $a = arr[i]$
  - ▶  $push()$  und  $pop()$  beim Stack (Stapel)



# $O(n)$ : linearer Aufwand

- Ausgesprochen: "O von n"
- Der Aufwand wächst linear mit der Anzahl der Eingabeelemente  $n$
- Verdoppelt sich  $n$ , dann verdoppelt sich auch ungefähr der Aufwand.
- Beispiel: Summieren aller Elemente eines Arrays



# $O(n)$ : linearer Aufwand

C C++ Java **Python3** C# Javascript

```
# A function to calculate the sum of the elements in an array
def list_sum(A, n):
    sum = 0
    for i in range(n):
        sum += A[i]
    return sum

# A sample array
A = [5, 6, 1, 2]

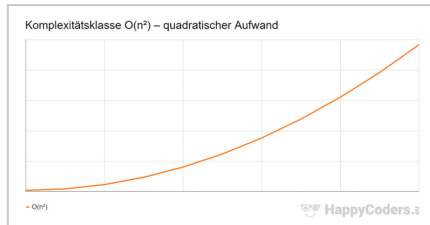
# Finding the number of elements in the array
n = len(A)

# Call the function and print the result
print(list_sum(A, n))
```

<https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/>

# $O(n^2)$ : quadratischer Aufwand

- Ausgesprochen: "O von n Quadrat"
- Der Aufwand wächst linear zum Quadrat der Anzahl der Eingabeelemente
- Beispiele:
  - ▶ Bubble Sort
  - ▶ Simple checking of duplicates in der Liste (2 Nested *for* Schleifen)

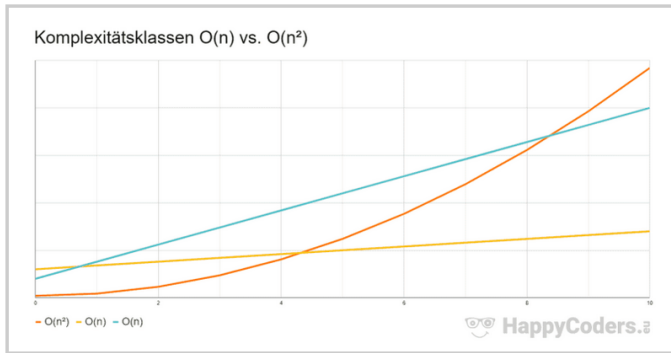




# $\mathcal{O}(n)$ vs $\mathcal{O}(n^2)$ : Faktoren sind auch wichtig!

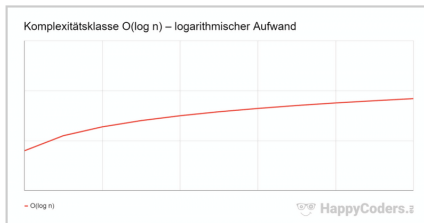
Im folgenden Beispiel-Diagramm werden drei fiktive Algorithmen gegenübergestellt: einer mit der Komplexitätsklasse  $\mathcal{O}(n^2)$  und zwei mit  $\mathcal{O}(n)$ , wobei einer davon schneller ist als der andere. Es ist gut zu sehen, wie bis zu  $n = 4$  der orangene  $\mathcal{O}(n^2)$ -Algorithmus weniger Zeit benötigt als der gelbe  $\mathcal{O}(n)$ -Algorithmus. Und sogar bis  $n = 8$  weniger Zeit als der türkise  $\mathcal{O}(n)$ -Algorithmus.

Ab hinreichend großem  $n$  – also ab  $n = 9$  – ist und bleibt  $\mathcal{O}(n^2)$  der langsamste Algorithmus.



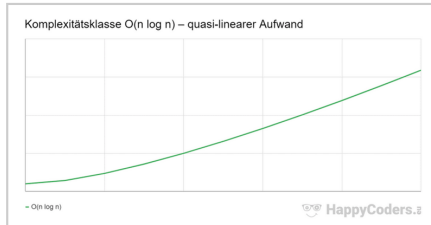
# $O(\log(n))$ : logarithmischer Aufwand

- Ausgesprochen: "O von log n"
- Der Aufwand wächst ungefähr um einen konstanten Betrag, wenn sich die Anzahl der Eingabeelemente verdoppelt.
- Beispiel: Binäre Suche (wir besprechen's heute)

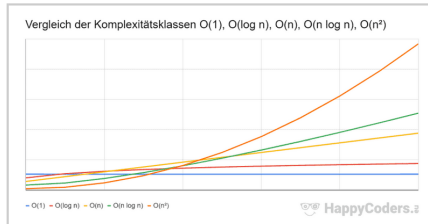


# $O(n \log(n))$ : quasi-linearer Aufwand

- Ausgesprochen: "O von n log n"
- Der Aufwand wächst etwas stärker als linear, da die lineare Komponente mit einer logarithmischen multipliziert wird
- Beispiele: Quick Sort, Merge Sort, Heap Sort

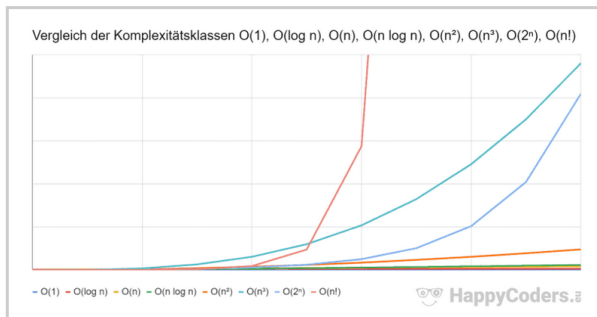


1.  $\mathcal{O}(1)$  - konstanter Aufwand
2.  $\mathcal{O}(\log(n))$  - logarithmischer Aufwand
3.  $\mathcal{O}(n)$  - linearer Aufwand
4.  $\mathcal{O}(n \log(n))$  - quasi-linearer Aufwand
5.  $\mathcal{O}(n^2)$  - quadratischer Aufwand



# Was gibt's noch?

1.  $\mathcal{O}(n^m)$  - polynomieller Aufwand
2.  $\mathcal{O}(2^n)$  - exponentieller Aufwand
3.  $\mathcal{O}(n!)$  - faktorieller Aufwand



# Beispiel $\mathcal{O}(2^n)$ : Fibonacci

The Fibonacci series is a great way to demonstrate exponential time complexity. Given below is a code snippet that calculates and returns the nth Fibonacci number:

```
long long int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

<https://www.crio.do/blog/time-complexity-explained/>

$$\mathcal{O}(1) < \mathcal{O}(\log(n)) < \mathcal{O}(\sqrt{n}) < \mathcal{O}(n) < \mathcal{O}(n \log(n))$$

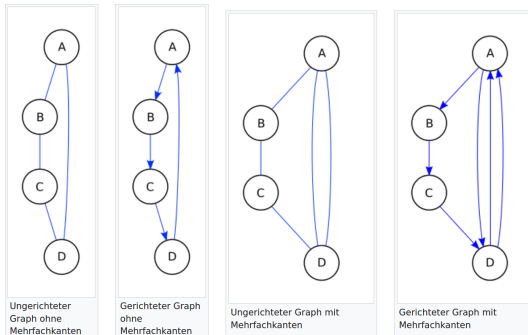
$$\mathcal{O}(n \log(n)) < \mathcal{O}(n^2) < \mathcal{O}(n^3) < \dots < \mathcal{O}(2^n) < \dots < \mathcal{O}(n!) < \mathcal{O}(n^n)$$

- Ein Graph ist eine abstrakte Struktur, die eine Menge von Objekten zusammen mit den zwischen diesen Objekten bestehenden Verbindungen repräsentiert.
- Die Objekte sind **Knoten**
- Die Verbindungen sind **Kanten**



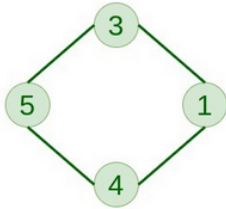


- Graph  $G = (V, E)$
- $V$  sind die Knoten (vertices)
- $E$  sind die Kanten (edges)

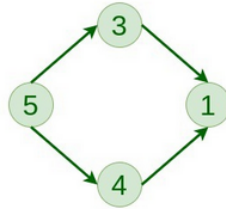


Quelle: [https://de.wikipedia.org/wiki/Graph\\_\(Graphentheorie\)](https://de.wikipedia.org/wiki/Graph_(Graphentheorie))

- Ungerichtete Graphen (Die Kanten haben keine Richtung)
- Gerichtete Graphen (Die Kanten haben eine Richtung)
- Gewichtete Graphen (Kanten sind gewichtet)
- Beispiel: eine Landkarte



Undirected Graph

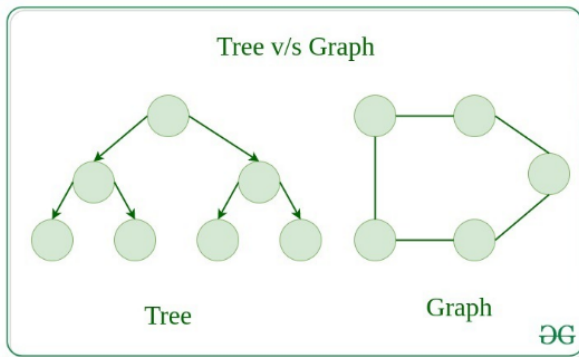


Directed Graph



<https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/>

# Baum ist ein spezieller Graph



<https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/>

# Beispiel: Landkarte von Rumänien

Welcher Graph ist das?

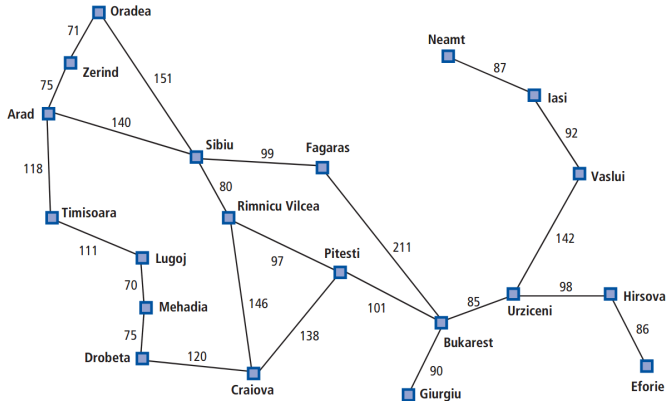
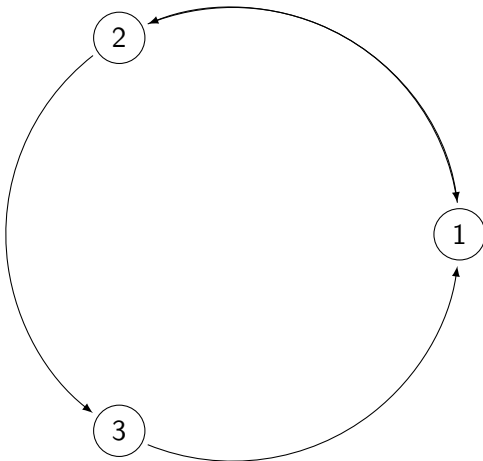


Abbildung 3.2: Eine vereinfachte Straßenkarte eines Teiles von Rumänien.

Source: Russel Norvig, KI Buch

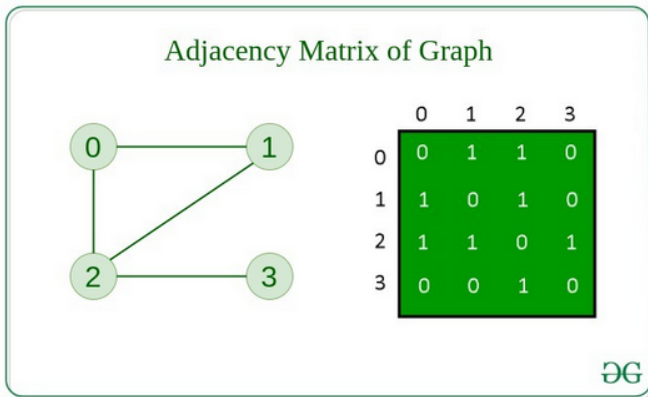
# Noch ein Beispiel: Twitter Followers

1: Alice, 2: Bob, 3: Chuck



# Wie kann man Graphen beschreiben?

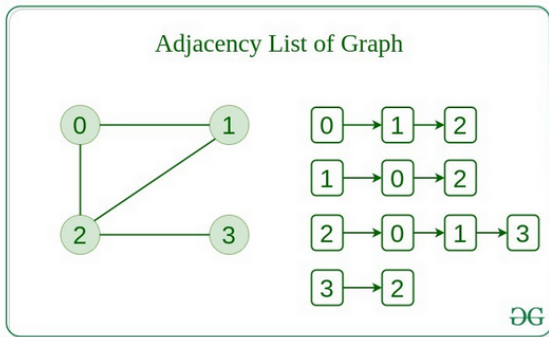
## Adjacency Matrix



<https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/>

# Wie kann man Graphen beschreiben?

## Adjacency List



<https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/>



# Adjacency Matrix vs List

Action	Adjacency Matrix	Adjacency List
Adding Edge	$O(1)$	$O(1)$
Removing an edge	$O(1)$	$O(N)$
Initializing	$O(N*N)$	$O(N)$

<https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/>

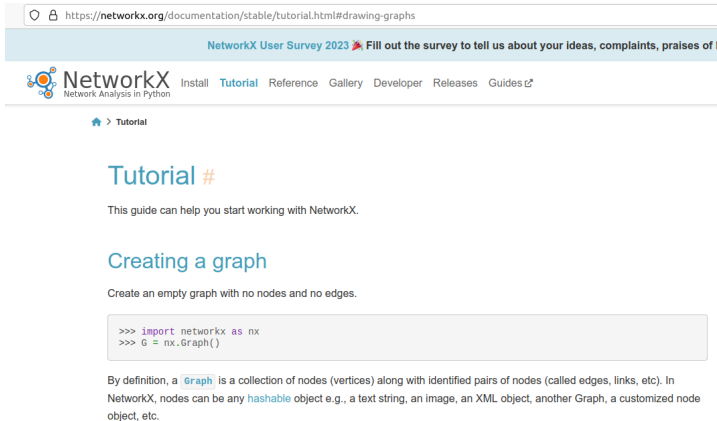
# Wie kann man einen Graph implementieren?

- Was brauchen wir denn?

# Wie kann man einen Graph implementieren?

- Was brauchen wir denn?
- Knoten
- Kanten
- Gewichte

## Über pip installieren, Version ab 3.0



The screenshot shows the NetworkX documentation page for the tutorial. The browser address bar displays the URL `https://networkx.org/documentation/stable/tutorial.html#drawing-graphs`. A banner at the top encourages users to fill out a survey. The navigation menu includes links for Install, Tutorial (highlighted), Reference, Gallery, Developer, Releases, and Guides. The page title is "Tutorial #". The introductory text states: "This guide can help you start working with NetworkX." The section "Creating a graph" instructs users to create an empty graph with no nodes and no edges. A code block shows the following Python code:

```
>>> import networkx as nx
>>> G = nx.Graph()
```

By definition, a **Graph** is a collection of nodes (vertices) along with identified pairs of nodes (called edges, links, etc). In NetworkX, nodes can be any **hashable** object e.g., a text string, an image, an XML object, another Graph, a customized node object, etc.

# Beispiel mit Bob, Alice und Chuck

```
import networkx as nx

G = nx.Graph()
G.add_node('Alice')
G.add_node('Bob')
G.add_node('Chuck')

print (G.nodes())

G.add_edge('Alice', 'Bob')
G.add_edge('Bob', 'Chuck')

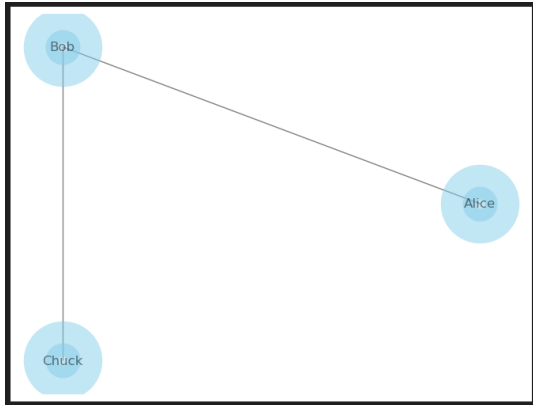
print (G.edges())

nx.draw_circular(G, with_labels=True, node_size=1000, node_color="skyblue", node_shape="o", alpha=0.5, linewidths=40)
```

```
[ 'Alice', 'Bob', 'Chuck' ]
[ ('Alice', 'Bob'), ('Bob', 'Chuck') ]
```

# Beispiel mit Bob, Alice und Chuck

Was für ein Graph ist das?



# Beispiel mit Bob, Alice und Chuck

```
import networkx as nx

G = nx.Graph()
G.add_node('Alice')
G.add_node('Bob')
G.add_node('Chuck')

print (G.nodes())

G.add_edge('Alice', 'Bob')
G.add_edge('Bob', 'Chuck')

print (G.edges())

nx.draw_circular(G, with_labels=True, node_size=1000, node_color="skyblue", node_shape="o", alpha=0.5, linewidths=40)
```

```
[ 'Alice', 'Bob', 'Chuck' ]
[ ('Alice', 'Bob'), ('Bob', 'Chuck') ]
```

# Noch ein Beispiel mit Bob, Alice und Chuck

Was für ein Graph ist das?

```
G1 = nx.DiGraph()
G1.add_node('Alice')
G1.add_node('Bob')
G1.add_node('Chuck')

print (G1.nodes())

G1.add_edge('Alice', 'Bob')
G1.add_edge('Bob', 'Alice')
G1.add_edge('Bob', 'Chuck')

print (G1.edges())

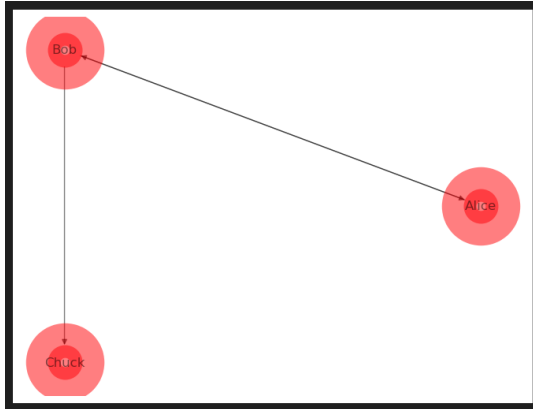
nx.draw_circular(G1, with_labels=True, node_size=1000, node_color="red", node_shape="o", alpha=0.5, linewidths=40)

['Alice', 'Bob', 'Chuck']
[('Alice', 'Bob'), ('Bob', 'Alice'), ('Bob', 'Chuck')]
```



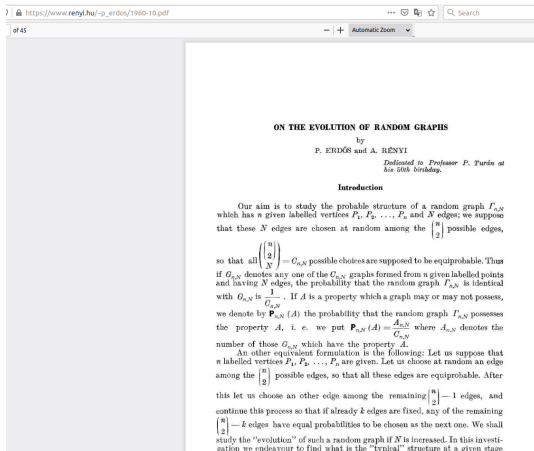
# Noch ein Beispiel mit Bob, Alice und Chuck

Was für ein Graph ist das?



# Zufallsgraphen: Erdős-Rényi (ER) Model (1960)

Dieses Modell wurde in den 1960er Jahren von den Mathematikern Paul Erdős und Alfréd Rényi vorgestellt.



- $n$  ist die Anzahl von Knoten
- $p$  ist die Wahrscheinlichkeit, dass eine Kante zwischen zwei Knoten gibt.
- In dieser Variante wird ein Netzwerk mit  $n$  Knoten erzeugt, wobei jede mögliche Kante zwischen zwei Knoten mit Wahrscheinlichkeit  $p$  vorhanden ist, unabhängig von den anderen Kanten.

# Einige charakteristische Eigenschaften des Erdős-Renyi-Modells

- Eine scharfe Schwelle ( $p^*$ ) in Bezug auf die Entstehung einer Riesenkompone (einer zusammenhängenden Komponente, die einen signifikanten Teil des Netzwerks umfasst).
- Wenn die Wahrscheinlichkeit  $p > p^*$ , entsteht plötzlich eine Riesenkompone im Netzwerk
- $p^* = \frac{\ln(n)}{n}$

Wir werden uns mit einer eigenen Nachimplementierung beschäftigen

## NetworkX

<https://www.geeksforgeeks.org/erdos-renyi-model-generating-random-graphs/>

Python

```
>>>H= nx.erdos_renyi_graph(10,0.5)  
>>> nx.draw(H, with_labels=True)  
>>> plt.show()
```

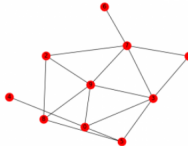
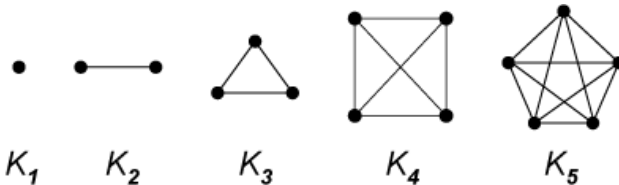


Figure 4: For  $n=10$ ,  $p=0.5$

This algorithm runs in  $O(n^2)$  time. For sparse graphs (that is, for small values of  $p$ ), `fast_gnp_random_graph()` is a faster algorithm. Thus the above examples clearly define the use of erdos renyi model to make random graphs

# Ein vollständiger Graph

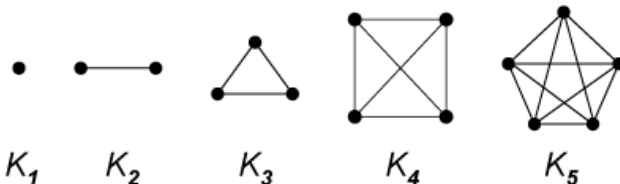
- Jeder Knoten ist mit jedem anderen Knoten durch eine Kante verbunden
- Was ist die Anzahl von Kanten in einem solchen Graphen mit  $n$  Knoten?
- Welche Bedingungen müssen beachtet werden?



<https://de.wikipedia.org/>

# Ein vollständiger Graph

- Jeder Knoten ist mit jedem anderen Knoten durch eine Kante verbunden
- Was ist die Anzahl von Kanten in einem solchen Graphen mit  $n$  Knoten?
- Welche Bedingungen müssen beachtet werden?
- $\frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$



<https://de.wikipedia.org/>

# Ein vollständiger Graph

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.complete_graph(15)

nx.draw_circular(G, with_labels=True, font_weight='bold')
plt.show()
```

<https://de.wikipedia.org/>

Wie kann man das selbst implementieren?

**10 Minuten** für die Implementierung in Python (oder Pseudocode)



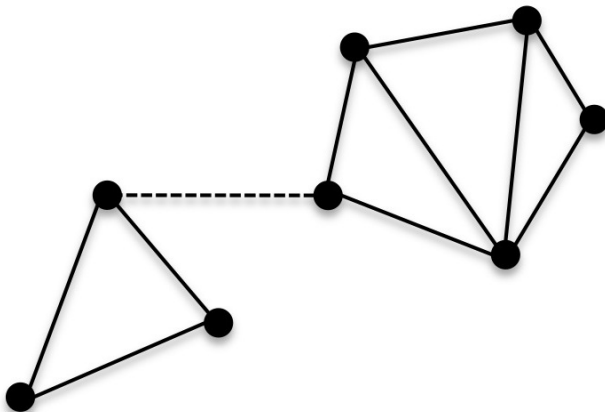
# Wie kann man überprüfen, ob ein Graph vollständig ist?

- Wie kann man das selbst implementieren?
- Bei welchen Bedingungen ist es relativ einfach?

**10 Minuten** für die Implementierung in Python (oder Pseudocode)

# Ein zusammenhängender Graph (Connected Graph)

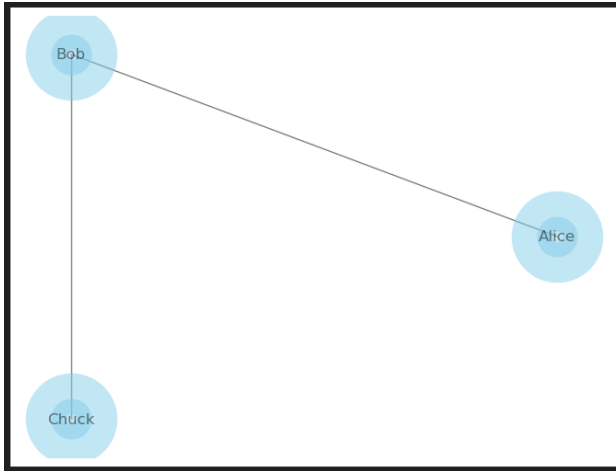
- Zwischen jedem Paar von Knoten existiert ein Pfad, der sie verbindet.



[https://en.wikipedia.org/wiki/Connectivity\\_\(graph\\_theory\)](https://en.wikipedia.org/wiki/Connectivity_(graph_theory))

# Wie kann man überprüfen, ob ein Graph connected wäre?

- Ideen? Z.B. wir starten mit Alice. Was muss man machen?



## Connectivity

To check whether a graph is connected, we'll start by finding all nodes that can be reached, starting with a given node:

```
: def reachable_nodes(G, start):  
    seen = set()  
    stack = [start]  
    while stack:  
        node = stack.pop()  
        if node not in seen:  
            seen.add(node)  
            stack.extend(G.neighbors(node))  
    return seen
```

In the complete graph, starting from node 0, we can reach all nodes:

```
: reachable_nodes(complete, 0)
```

Quelle: A. Downey Think Complexity Buch

# Was ist die Zeit Complexity von dieser Funktion?

- *pop*: eine konstante Zeit
- *in seen?*: eine konstante Zeit
- *seen.add*: eine konstante Zeit bei einem Knoten (Insgesamt  $\mathcal{O}(n)$  für  $n$  Knoten)
- *extend*:  $\mathcal{O}(nbh)$ , wobei  $nbh$  ist die Anzahl von den Nachbarn (Insgesamt  $\mathcal{O}(m)$  für  $m$  Kanten)
- Komplexität:  $\mathcal{O}(n + m)$ ,  $n$  ist die Anzahl von Knoten,  $m$  ist die Anzahl von Kanten
- Was ist es für einen kompletten Graph? (Wir wissen ja die Verbindung zwischen  $n$  und  $m$  in diesem Fall)

1. Implementieren Sie den Connectivity Test
2. Implementieren Sie die Generierung von ER Graphen (Die Numpy Funktion `np.random.random()` wird nützlich sein)
3. Wie schnell ist Ihre Implementierung? Messen Sie die Zeit und vergleichen sie mit der NetworkX Implementierung.
4. Generieren Sie 20 ER Graphen mit  $n = 10$  Knoten und  $p = 0.1$ . Wie viele Graphen sind zusammenhängend?
5. Wie ändert sich die Anzahl von connected ER Graphen, wenn  $p = 0.3$ ,  $p = 0.5$ ,  $p = 0.8$ ?
6. Nutzen Sie *matplotlib* um diese Verbindungen zu visualisieren

# Danke für die Aufmerksamkeit!

