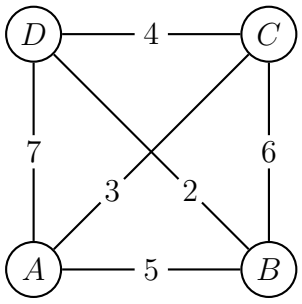
Das Travelling Salesman Problem (TSP)

Die Problemstellung

Ein Handelsvertreter soll 4 Städte besuchen und wieder in die zuerst besuchte Stadt zurückkehren. Bestimme die kürzeste Route.

Städtemodell



Distanztabelle

	A	B	C	D
A				
B				
C				
D				

Die Brute Force-Methode

Wie berechnen die Längen aller möglichen Touren und wählen die kürzeste Tour aus.

Die Menge aller Touren erhalten wir, indem wir alle möglichen Anordnungen der Städte bestimmen. Eine Anordnung von n Elementen auf n Plätzen wird *Permutation* von n Elementen genannt.

Ohne Beschränkung der Allgemeinheit können wir bei jeder Tour die Stadt A als Ausgangspunkt wählen.

Aufgabe 1

Berechne die Distanzen aller möglichen Touren. Welche ist (sind) am kürzesten?

Tour	**Länge**
$\underline{A}BCD\underline{A}$	$5+6+4+7=22$

Tour	**Länge**

Die Distanztabelle in Python

Als Datenstruktur für die Distanzen wählen wir eine Liste von Listen (LoL), wobei die Indizes 0, 1, 2, 3 für die Städte A, B, C, D stehen. In unserem Beispiel:

```
D = [[0, 5, 3, 7],
     [5, 0, 6, 2],
     [3, 6, 0, 4],
     [7, 2, 4, 0]]
```

Distanz zwischen den Städten 1 und 3:

Eine Tour in Python

Wählen wir (willkürlich) die Stadt mit dem Index 0 ans Start- und Enpunkt der Tour, so genügt es, eine Rundtour durch n Städte als Liste mit $n-1$ Elementen darzustellen:

$ACDBA$ $\Leftrightarrow$ `[0, 2, 3, 1, 0]` $\Leftrightarrow$ `T = [2, 3, 1]`

Die Tourlänge berechnen

Folgende Funktion berechnet die Länge einer Tour `T` (Liste):

```
def roundTripLength(D, T):
    '''Länge einer Rundreise T mit Start und Ende in 0'''
    length = D[0][T[0]]
    for i in range(0, len(T)-1):
        length += D[T[i]][T[i+1]]
    length += D[T[-1]][0] # Rückkehr zum Anfangspunkt:

    return length
```

Erzeugung der Permutationen

Das Generieren aller Permutationen einer Liste ist nicht ganz einfach. Deshalb verwenden wir das Python-Modul `itertools`, das alle Permutationen einer Liste `L` mit der Methode `permutations(L)` erzeugt und als sogenannten Iterator zurückgibt, der mit einer `for`-Schleife durchlaufen werden kann.

```
from itertools import permutations

for p in permutations([1,2,3]):
    print(p)
```

Über eine solche Schleife lassen sich alle Tour-Längen berechnen und die aktuell kleinste herausfiltern.

Die Laufzeitkomplexität der Brute Force-Methode

- Wählt man eine beliebige Stadt als Ausgangspunkt, so sind bei n Städten $(n-1)!$ Permutationen (Touren) zu erzeugen.
- Für jede Permutation (Tour) sind n Einzeldistanzen aus der Distamzmatrix herauszulesen und zu addieren.
- Nehmen wir stark vereinfachend an, dass für die Erzeugung einer Permutation, für das Herauslesen einer Einzeldistanz und das Addieren der Distanzen jeweils ein konstanter Zeitaufwand C nötig ist, erhalten wir folgende Laufzeitkomplexität:

Eine einfache Implementierung in Python

```
from itertools import permutations

def round_trip_length(D, T):
    '''Länge einer Rundreise T mit Start und Ende in Stadt 0.'''
    length = D[0][T[0]]
    for i in range(0, len(T)-1):
        length += D[T[i]][T[i+1]]
    length += D[T[-1]][0]
    return length


def tsp_brute_force(D):

    n = len(D)

    opt_dist = float('inf')
    opt_tour = None

    for p in permutations(range(1, n)):
        dist = round_trip_length(D, p)
        if dist < opt_dist:
            opt_dist = dist
            opt_tour = p

    return opt_dist, [0] + list(opt_tour) + [0]
```

Aufgabe 2

Schreibe ein Python-Modul `tsp_benchmark.py`, das den Zeitaufwand für die Brute Force-Lösung für $n = 4, 5, \ldots, 11, 12$ Städte mit der `time()`-Funktion aus dem gleichnamigen Modul berechnet. Der folgende Code erzeugt zufällige Distanzmatrizen.

```
from random import randint

def random_dist_matrix(n, symmetric=True, a=10, b=99):
    '''Erzeugt zufällige (n x n)-Distanzmatrix mit Werten zwischen a und b'''
    D = [[0 for i in range(n)] for j in range(n)]

    for i in range(0, n-1):
        for j in range(i+1, n):
            D[i][j] = randint(a, b)
            if symmetric:
                D[j][i] = D[i][j]
            else:
                D[j][i] = randint(a, b)

    return D
```

Das Problem

Die Lösung von Aufgabe 2 zeigt, dass der Zeitaufwand für das Testen aller möglichen Pfade mit unserer Implementierung faktoriell in Abhängigkeit der Städtezahl n wächst. Dies bedeutet, dass schon bei relativ bescheidenen Städtezahlen Wartezeiten entstehen, die nicht mehr tolerierbar sind.

Zwar gibt es Algorithmen, welche das TSP „nur“ mit exponentiellem Zeitaufwand exakt lösen können aber auch dort tritt früher oder später das Problem auf, dass die Wartezeiten inakzeptabel gross werden.

Die Nearest Neighbor-Heuristik

Der Begriff *Heuristik* bezeichnet ein Verfahren, das trotz begrenztem Wissen eine mögliche Lösung eines Problems findet. Diese Lösung kann jedoch von der optimalen Lösung abweichen.

Bei der Nearest Neighbor-Heuristik (NNH) für das TSP wählen wir einen Startknoten aus und bestimmten eine Rundtour, in dem wir, im Startknoten beginnend, jeweils zu einer der am nächsten liegenden Städte gehen und dann auf die gleiche Weise die folgenden Städte aufsuchen. Bei der Rückkehr zum Ausgangspunkt können wir nicht mehr auswählen und müssen die vorgegebene Distanz verwenden.

Diese Strategie wird in der informatik als *greedy* (gierig) bezeichnet und führt beim TSP im allgemeinen nicht zu einer optimalen Lösung.

Aufgabe 3

Berechne die Länge der Rundtouren mit der NNH für alle möglichen Startpunkte und vergleiche mit der kürzesten Tour. Welche Schlussfolgerungen lassen sich aus den Resultaten ziehen?

	A	B	C	D
A	0	1	1	1
B	1	0	4	1
C	1	4	0	7
D	1	1	7	0

Aufgabe 4

Gegeben ist folgende Distanzmatrix

	A	B	C	D
A	0	1	3	2
B	1	0	2	3
C	3	2	0	100
D	2	3	100	0

Zeichne den Graphen zur Distanzmatrix und zeige, dass man mit der Nearest-Neighbor-Heuristik für jeden Startknoten dieselbe Tourlänge erhält. Gibt es eine bessere Lösung? Wenn ja, welche?