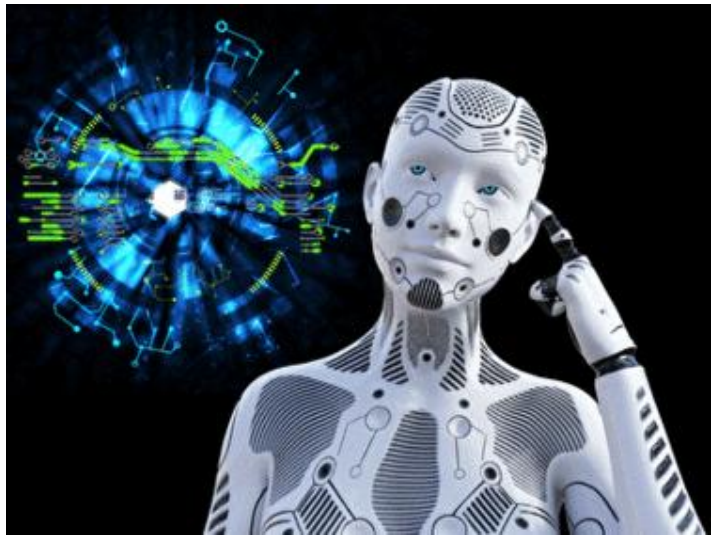
Algorithmen II

Dr. Philipp Hurni, Kantonsschule Sursee

Algorithmen II

- Komplexität von Code bestimmen
  - Analytisch
  - Mittels "Monte Carlo Simulation"
- Travelling Salesman Problem
  - Code Rekursiv Formulieren
- Übungen zu Algorithmen und Komplexität



Komplexität von Code bestimmen

```
def bubble_sort(liste):
        n = len(liste)
        for i in range(n):
                for j in range(0, n-1):
                # Tausche, wenn das Element grösser ist als das nächste
                        if liste[j] > liste[j+1]:
                                temp = liste[j]
                                liste[j] = liste[j+1]
                                liste[j+1] = temp
        return liste

zahlen = [64, 34, 25, 12, 22, 11, 90]
sortierte_zahlen = bubble_sort(zahlen)
print("Sortierte Liste:", sortierte_zahlen)
```

n ma

*n-1 mal

O(n^2

Komplexität von Code bestimmen

- Komplexität lässt sich analytisch bestimmen, und experimentell verifizieren
- Erstere Methode ist sicher vorzuziehen – aber experimentelles Vorgehen kann auch zielführend sein
- Lasst uns den oben abgebildeten Code experimentell punkto Komplexitätsklasse (Zeitkomplexität) bestimmen – was würden Sie tun?

Komplexität von Code bestimmen - Analytisch

```
def bubble_sort(liste):
        n = len(liste)
        for i in range(n):
                for j in range(0, n-1):
                # Tausche, wenn grösser als das nächste
                        if liste[j] > liste[j+1]:
                                temp = liste[j]
                                liste[j] = liste[j+1]
                                liste[j+1] = temp
        return liste

zahlen = [64, 34, 25, 12, 22, 11, 90]
sortierte_zahlen = bubble_sort(zahlen)
print("Sortierte Liste:", sortierte_zahlen)
```

n mal

(n-1) mal

"Prüfen und tauschen"

⇨ **n * (n-1) * C (irgendwas Konstantes)**

= $(n^2 - n) * C$
= $n^2 * C - n * C$

⇨ **$O(n^2)$**

"Monte Carlo Simulation"

Monte Carlo Simulation

A) **Zufallswerte erzeugen**: Es wird eine grosse Anzahl von Zufallswerten für die unsicheren Eingabevariablen erzeugt. Das Modell (oder die Formel) wird mit den generierten Zufallswerten berechnet.

B) **Modellauswertung wiederholen**: Der Vorgang Punkt A kann Tausende oder Millionen von Malen geschehen.

C) **Ergebnisse analysieren**: Nach allen Durchläufen werden die Ausgaben des Modells gesammelt und statistisch ausgewertet, um eine Verteilung der möglichen Ergebnisse zu erhalten. Diese Verteilung zeigt die Wahrscheinlichkeit verschiedener Resultate, wie z.B. das durchschnittliche Ergebnis, die Bandbreite der möglichen Ergebnisse und die Wahrscheinlichkeit, dass ein bestimmtes Ziel erreicht wird.

Auswertung mittels Monte Carlo Ansatz

- Öffne Algorithmen II – BubbleSort.py
- Betrachte die Messung der Laufzeit des Codes für unterschiedliche Listen-Grössen (LIST_SIZE) – das eigentliche n des Beispiels in "bubble_sort"
- TRIALS_PER_LISTSIZE: Wie viele Messungen mache ich für die definierte Listen-Grösse

```
20 LIST_SIZE = 1000
21 TRIALS_PER_LISTSIZE = 10
22
23 # Generiere eine Liste von Zufallszahlen
24 numbers = []
25 for num in range(LIST_SIZE):
26     numbers.append(num)
27
28 time_needed_per_trial = []
29 for i in range(TRIALS_PER_LISTSIZE):
30
31     # Randomisieren der Zahlen
32     random.shuffle(numbers)
33
34     # Starte die Zeitmessung
35     start_time = time.time()
36
37     sorted_numbers = bubble_sort(numbers)
38
39     # Beende die Zeitmessung
40     end_time = time.time()
41
42     # Berechne die benötigte Zeit: end_time - start_time
43     time_needed = end_time - start_time
44
45     # Millisekunden
46     milliseconds = time_needed * 1000
47     time_needed_per_trial.append(milliseconds)
48     print("Lauf Nr.",i, ":", milliseconds, "(Millisekunden)")
49
50
51 average = sum(time_needed_per_trial)/len(time_needed_per_trial)
52
53 print("Durchschnitt: Benötigte Zeit für die Sortierung von: ", LIST_SIZE, " Zahlen (in Millisekunden): ", average )
```

Wie oft machen wir das

Der eigentliche Aufruf des Sortier-Algorithmus

Die Auswertung

Frage: was müsste jetzt passieren, wenn ich die List-Size von 1000 auf 2000 erhöhe? Und bei Erhöhung von 1000 auf 10'000?

⇨ **Experimentiert damit**
⇨ **passiert das, was ihr erwartet?**

Übung A: Komplexität von Code bestimmen

```
def hasDup(numbers):
    for x in range(0,len(numbers)):
        for y in range(0,len(numbers)):
            if numbers[x] == numbers[y] and x!=y:
                return True
    return False

numbers = [21,12,15,8,22,13,5,9,0,1,2,3,4,6,3,4]
result = hasDup(numbers)
print("Has Duplicates:", result)
```

- Analysiere den Algorithmus in "hasDup" auf seine Zeitkomplexität.
- Wie lange dauert es, eine beliebige Liste mit n Zahlen nach Duplikaten zu durchsuchen?
- Weise den Algorithmus einer Komplexitätsklasse zu und begründe

Übung B: Komplexität von Code bestimmen

```
def hasDupII(numbers):
    already_seen_list = []
    for i in range(max(numbers)+1):
        already_seen_list.append(False)

    for x in range(0,len(numbers)):
        num = numbers[x]
        if already_seen_list[num] == True:
            return True
        else:
            already_seen_list[num] = True

    return False

numbers = [21,12,15,8,22,13,5,9,0,1,2,3,4,6,3,4]
result = hasDupII(numbers)
print("Has Duplicates:", result)
```

- Analysiere den Algorithmus in "hasDupII" auf seine Zeitkomplexität.

  Nimm an, dass das Maximum der Zahlen in der Liste der Länge der Liste entspricht

- Wie lange dauert es, eine beliebige Liste mit n Zahlen nach Duplikaten zu durchsuchen?

- Weise den Algorithmus einer Komplexitätsklasse zu und begründe

O(n)

Übung C: Miss die Zeit experimentell

```
20 LIST_SIZE = 1000
21 TRIALS_PER_LISTSIZE = 10
22
23 # Generiere eine Liste von Zufallszahlen
24 numbers = []
25 for num in range(LIST_SIZE):
26     numbers.append(num)
27
28 time_needed_per_trial = []
29 for i in range(TRIALS_PER_LISTSIZE):
30
31     # Randomisieren der Zahlen
32     random.shuffle(numbers)
33
34     # Starte die Zeitmessung
35     start_time = time.time()
36
37     sorted_numbers = bubble_sort(numbers)
38
39     # Beende die Zeitmessung
40     end_time = time.time()
41
42     # Berechne die benötigte Zeit: end_time - start_time
43     time_needed = end_time - start_time
44
45     # Millisekunden
46     milliseconds = time_needed * 1000
47     time_needed_per_trial.append(milliseconds)
48     print("Lauf Nr.",i, ":", milliseconds, "(Millisekunden)")
49
50
51 average = sum(time_needed_per_trial)/len(time_needed_per_trial)
52
53 print("Durchschnitt: Benötigte Zeit für die Sortierung von: ", LIST_SIZE, " Zahlen (in Millisekunden): ", average )
```

- Wende dieselbe Methode wie hier an
- Was für Resultate ergeben sich für hasDup und hasDupII?
- Entspricht das den Erwartungen?

Komplexität abschätzen

Übung A: Komplexität von Code bestimmen

```
def hasDup(numbers):
    for x in range(0,len(numbers)):
        for y in range(0,len(numbers)):
            if numbers[x] == numbers[y] and x!=y:
                return True
    return False

numbers = [21,12,15,8,22,13,5,9,0,1,2,3,4,6,3,4]
result = hasDup(numbers)
print("Has Duplicates:", result)
```

n mal

n mal

⇨ **n * n * C (irgendwas Konstantes)**

= n^2 * C

⇨ **O(n^2)**

Übung B: Komplexität von Code bestimmen

```
def hasDupII(numbers):
    already_seen_list = []
    for i in range(max(numbers)+1):
        already_seen_list.append(False)

    for x in range(0,len(numbers)):
        num = numbers[x]
        if already_seen_list[num] == True:
            return True
        else:
            already_seen_list[num] = True

    return False

numbers = [21,12,15,8,22,13,5,9,0,1,2,3,4,6,3,4]
result = hasDupII(numbers)
print("Has Duplicates:", result)
```

n mal "etwas Konstantes" (C)

n mal "etwas Konstantes" (K)

⇨ **n * C + n *K**

= n * (C + K)

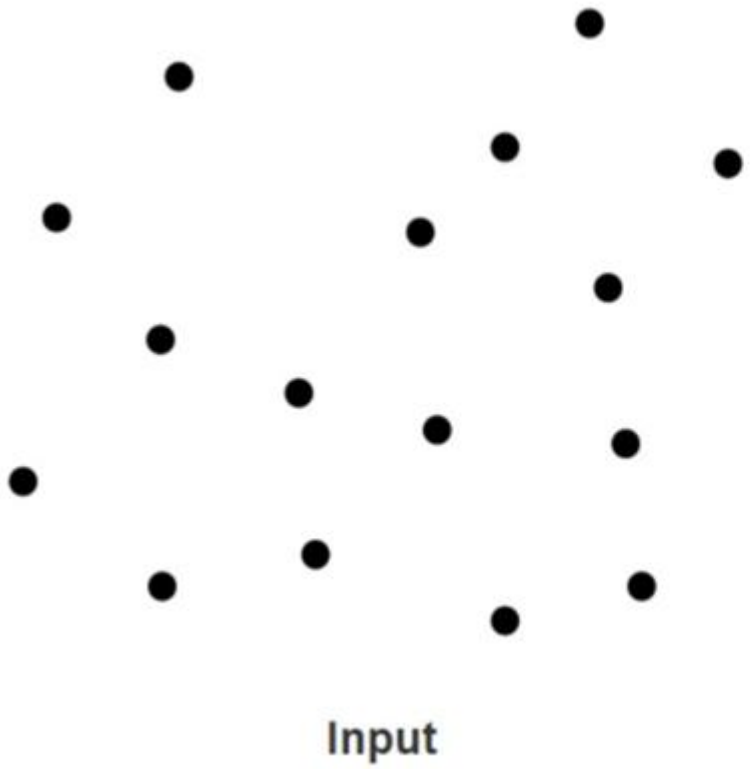
⇨ **O(n)**

"Harte" Probleme der Informatik

Travelling Salesman Problem (TSP)

- Finde den kürzesten Weg, welcher alle definierten Orte besucht
- Besuche nie einen Ort zweimal
- Am Ende bin ich wieder dort wo ich gestartet bin

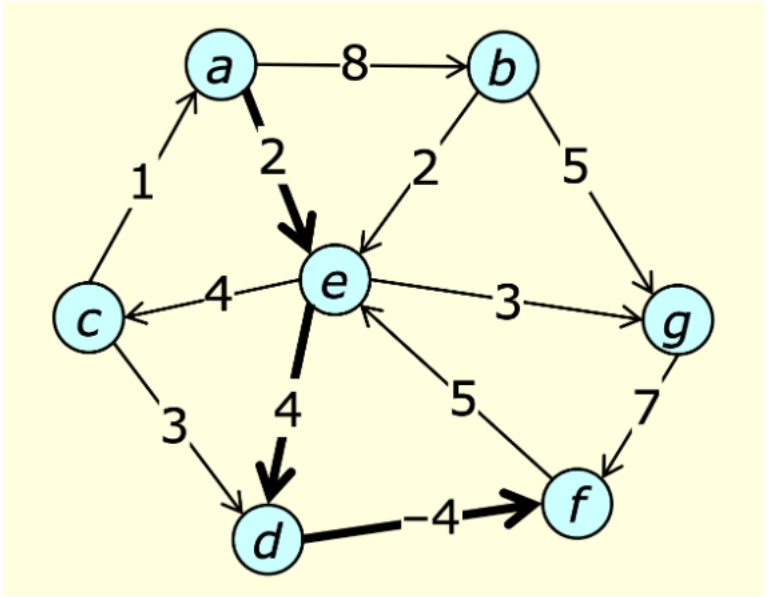
Veranschaulichung des Problems



Input

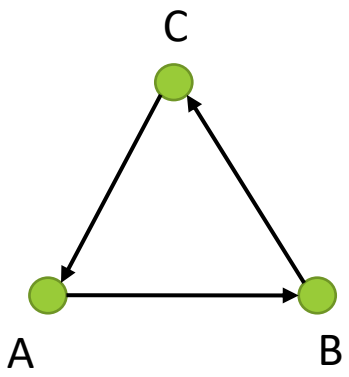
n Knoten

Haben wir das nicht schon mit Dijkstra gelöst?



Nein! Dijkstra löst das Problem des kürzesten Wegs zwischen 2 Knoten

Liste alle Pfade auf, welche A, B und C besuchen und wieder am Ursprung enden




```
cities = ["A", "B", "C"]

# gehe mit der Variable city_I durch die ganze Liste cities
for city_I in cities:
    cities_I = cities.copy()
    cities_I.remove(city_I)

    # gehe mit der Variable city_III durch die ganze Liste cities,
    # aber city_III darf nicht city_I oder city_II sein (weil ist ja schon gewählt)
    for city_II in cities_I:
        cities_II = cities_I.copy()
        cities_II.remove(city_II)

        # gehe mit der Variable city_III durch die ganze Liste cities,
        # aber city_III darf nicht city_I oder city_II sein (weil ist ja schon gewählt)
        for city_III in cities_II:
            cities_III = cities_II.copy()
            cities_III.remove(city_III)
            print(city_I,",", city_II,",", city_III,",", city_I)
```

```
A , B , C , A
A , C , B , A
B , A , C , B
B , C , A , B
C , A , B , C
C , B , A , C
```

Algorithmen II - Travelling Salesman A.py

Rekursion – wir formulieren das Problem rekursiv (eleganter)

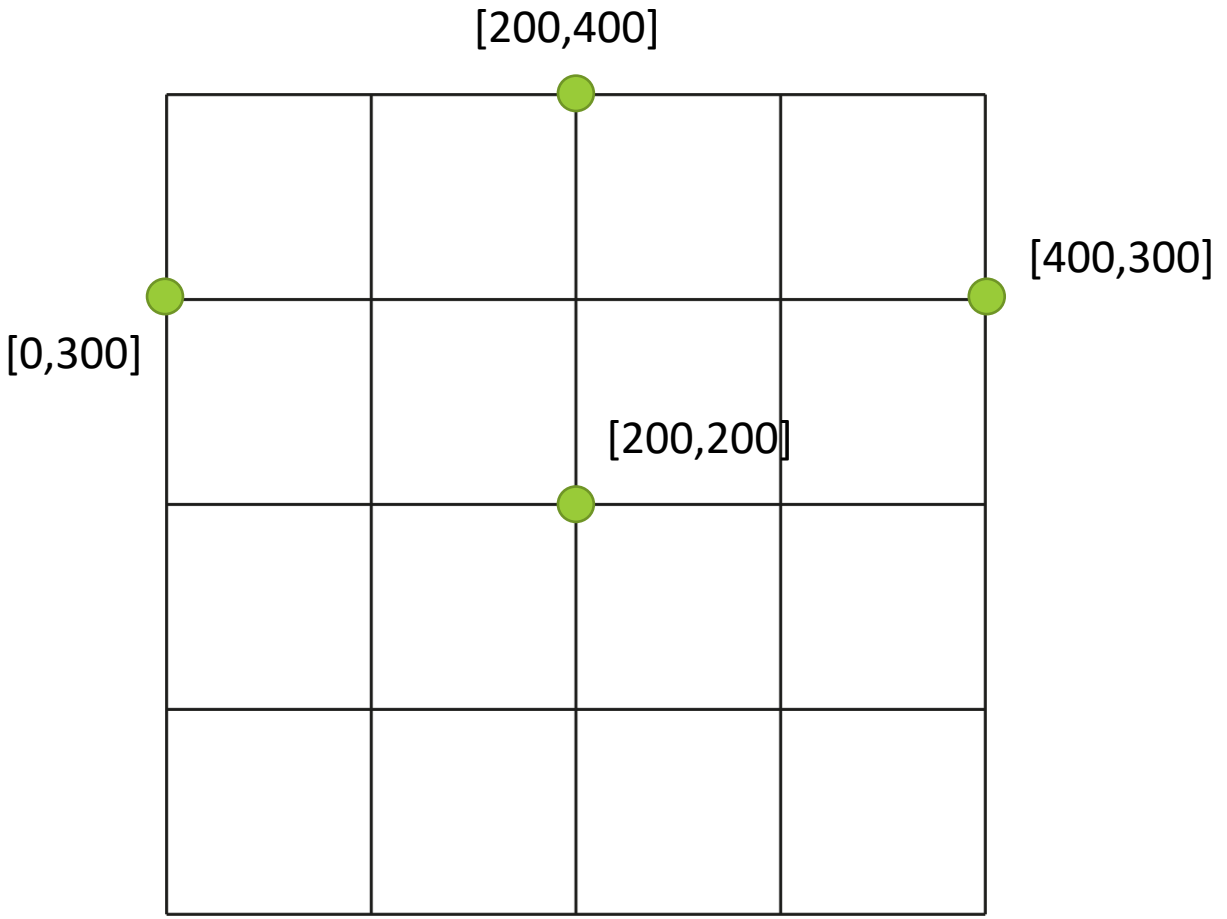
```
cities = ["A", "B", "C"]

def nextCity(current_path, cities_remaining):
    if len(cities_remaining) == 0:
        print(current_path)
    else:
        for i in range(len(cities_remaining)):
            new_current = current_path.copy()
            new_current.append(cities_remaining[i])
            new_cities_remaining = cities_remaining.copy()
            new_cities_remaining.pop(i)
            nextCity(new_current, new_cities_remaining)


nextCity([], cities)
```

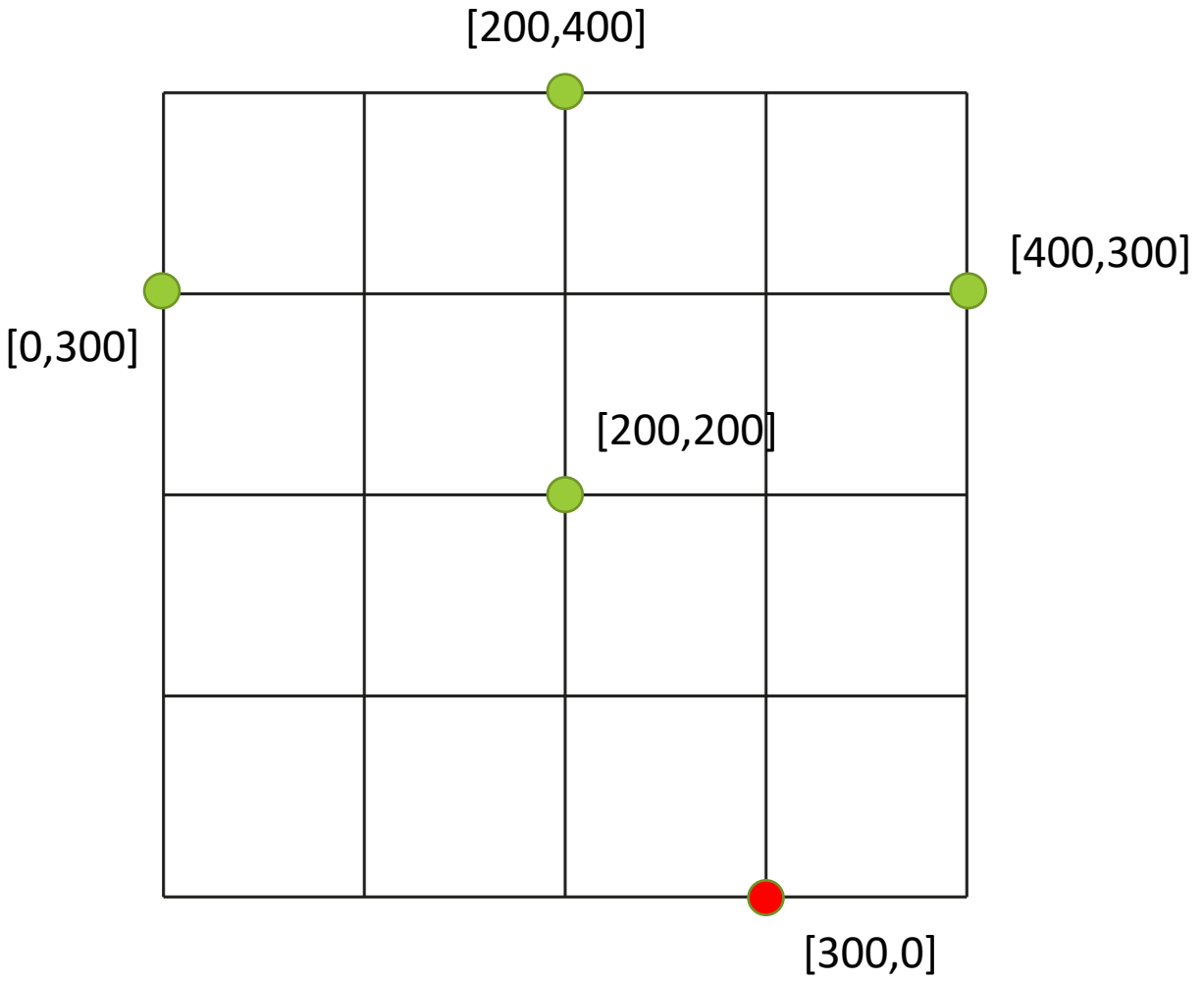
Pfad mit 4 Städten

- Finde den kürzesten Weg um das Travelling Salesman Problem zu lösen – von Hand
- Schaue dir das Python Programm "Travelling Salesman C" an – löse die Aufgabe dieser Folie mit diesem Programm.
- Miss, wie lange das Programm auf deinem Computer braucht (im Programm eingeblendet)



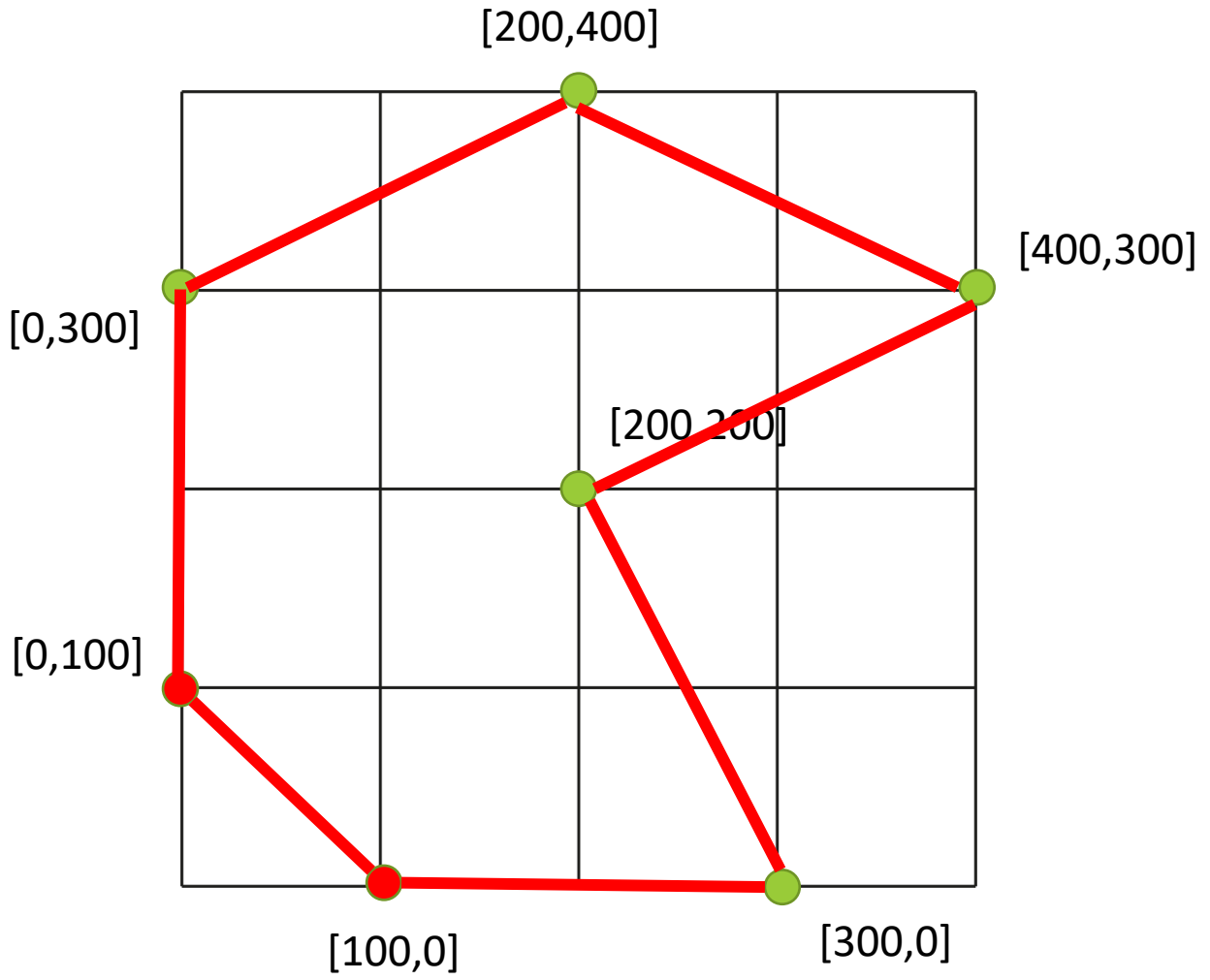
5 Städte

- Es kommt eine neue Stadt hinzu [300,0]. Versuche den schnellsten Weg zu finden
- Miss die Zeit, und vergleiche zur vorherigen Aufgabenstellung.



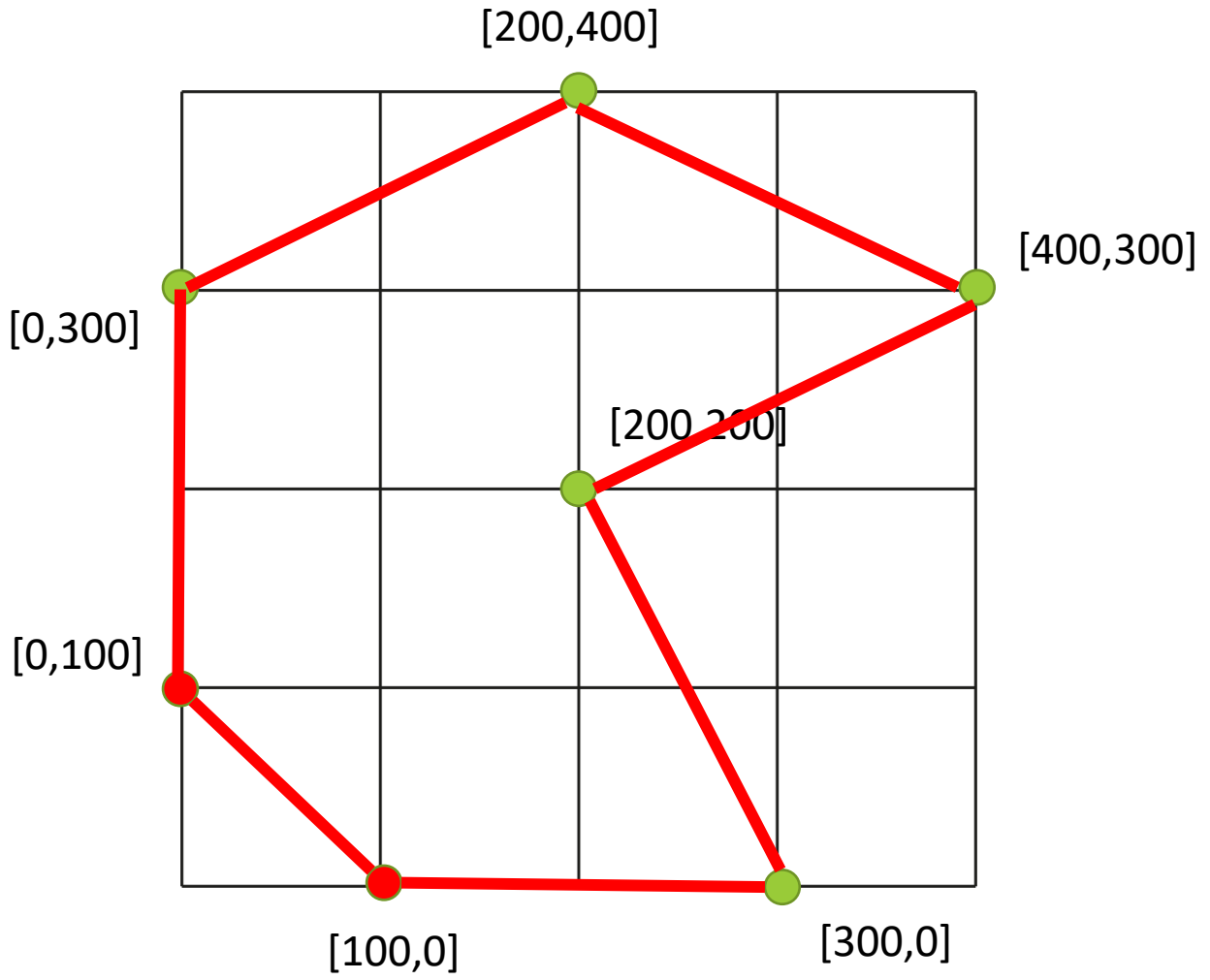
7 Städte

- Es kommen 2 neue Städte hinzu
- Führe das Programm "Travelling Salesman D" aus – in der Zwischenzeit versuche den kürzesten Weg von Hand zu berechnen.
- Was fällt dir auf bezüglich der Dauer des Programms?



8 Städte

- Füge noch eine weitere Stadt hinzu…
- Schätze, wie stark es zunimmt



Welche Komplexität hat unsere Lösung hier?

n = Anzahl Städte

```
cities = ["A", "B", "C"]

def nextCity(current_path, cities_remaining):
    if len(cities_remaining) == 0:
        print(current_path)
    else:
        for i in range(len(cities_remaining)):
            new_current = current_path.copy()
            new_current.append(cities_remaining[i])
            new_cities_remaining = cities_remaining.copy()
            new_cities_remaining.pop(i)

            nextCity(new_current, new_cities_remaining)


nextCity([], cities)
```

n * (n-1) * (n-2) * … 1

= O(n!)

Welche Komplexität hat unsere Lösung hier?

n = Anzahl Städte

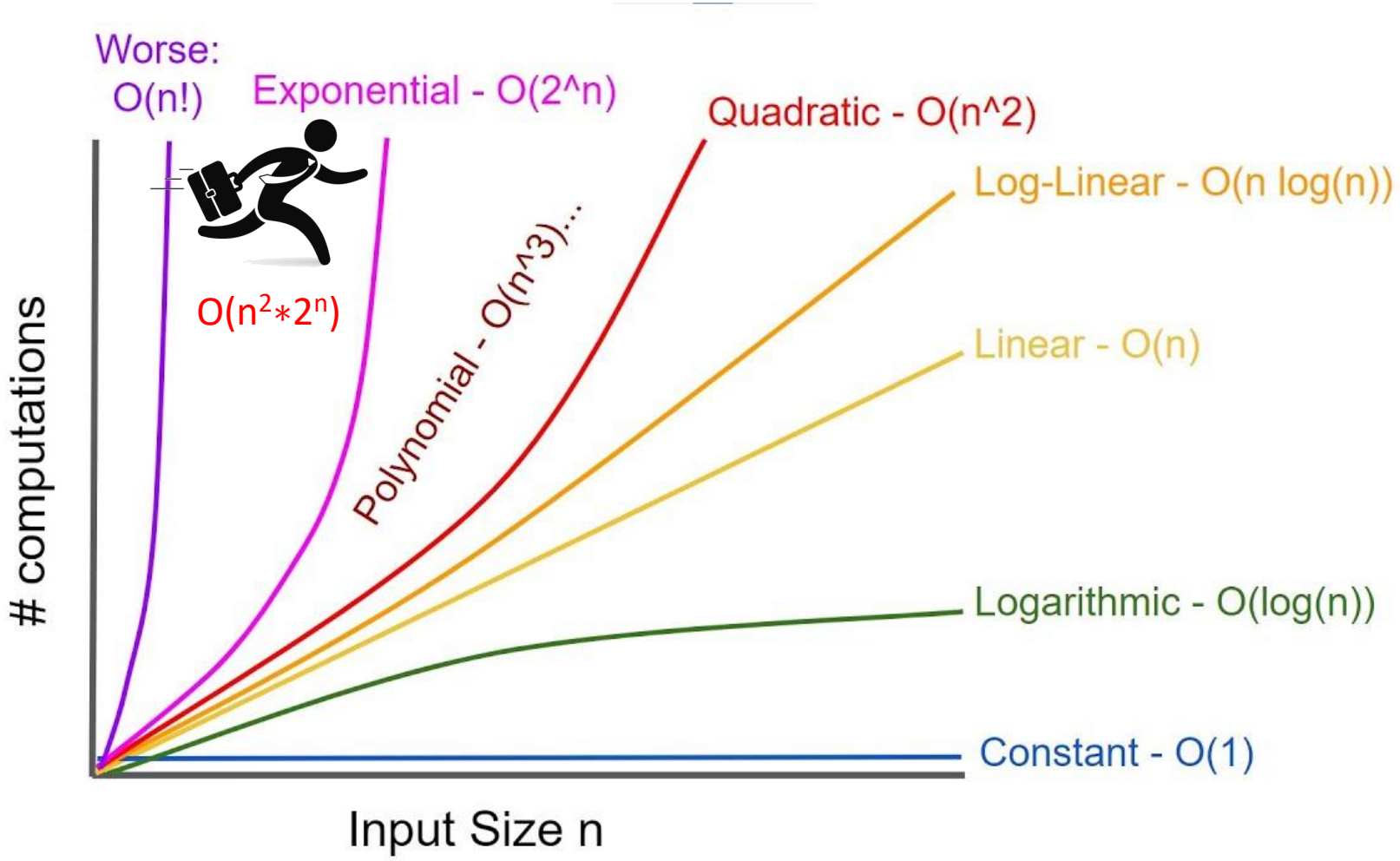
```
cities = ["A", "B", "C"]

def nextCity(current_path, cities_remaining):
    if len(cities_remaining) == 0:
        print(current_path)
    else:
        for i in range(len(cities_remaining)):
            new_current = current_path.copy()
            new_current.append(cities_remaining[i])
            new_cities_remaining = cities_remaining.copy()
            new_cities_remaining.pop(i)

            nextCity(new_current, new_cities_remaining)


nextCity([], cities)
```

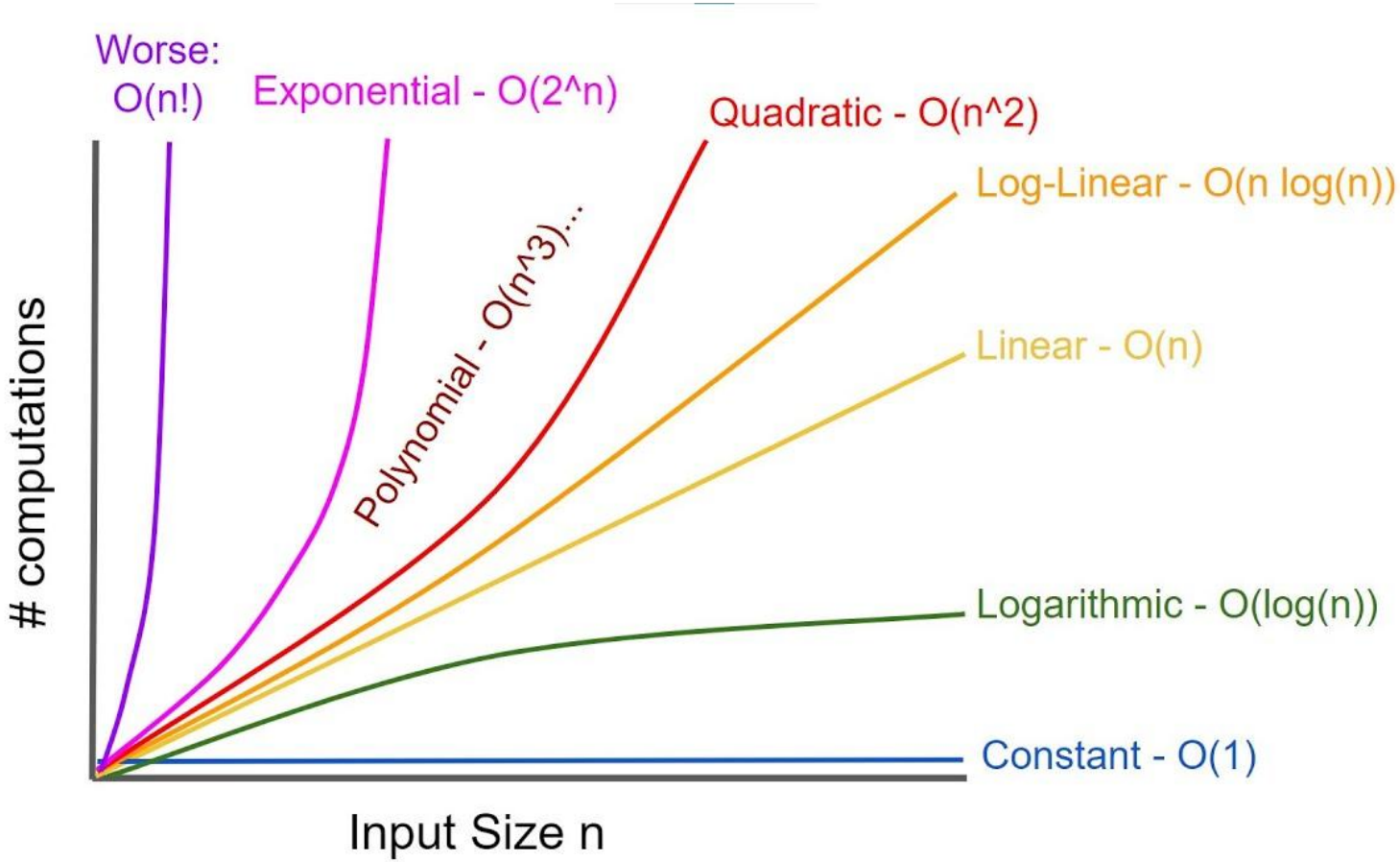
Travelling Salesman Problem



Effizienz eines Algorithmus

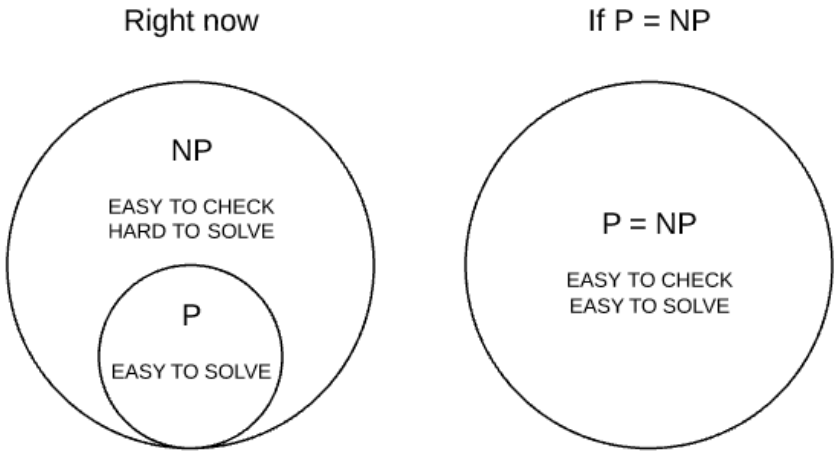
- Sei **n** die Bezeichnung für die Grösse des Inputs in einem Algorithmus
- Algorithmen mit einer **polynomialen Laufzeit** von $\mathbf{n^2}$ oder allgemein $\mathbf{n^k}$ (k endlich) skalieren bereits schlecht. Das bedeutet, dass sie bei einem grösseren Input stark mehr Zeit in Anspruch nehmen. Sie gelten aber immer noch als **effizient**.
- Algorithmen mit **exponentieller Laufzeit**, also 2^n oder k^n werden nicht mehr als effizient bezeichnet (merke: n strebt gegen ∞).

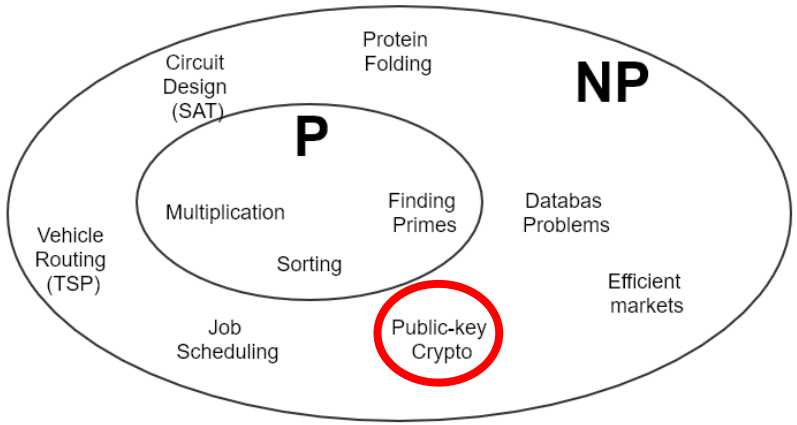
Wo ist die "Zone der Sünde" in unserer Darstellung?



Problemkategorien P und NP

- P: Polynomiale ($y=ax^3 + bx^2 + cx^1 + d$)
  Probleme können in polynomialer Laufzeit gelöst werden
- NP: Nichtdeterministisch-Polynomial:
  Prüfen ist polynomial, lösen ist aber
  (bis jetzt jedenfalls) aufwändiger – man weiss das aber nicht mit Sicherheit





Turing Award

- Jedes Jahr einer oder mehrere Preisträger
- «Computer Science equivalent of Nobel Prize»
- 1 Mio. USD
- 1 Schweizer Preisträger 1984: Niklaus Wirth, Erfinder von PASCAL

Übungen (Hausaufgabe!)

Den Mittelwert finden

- Schreibe ein Python Programm, welches für eine Liste L mit Zahlen drin den Mittelwert berechnet
- Du darfst keine Funktionen verwenden, nur Schlaufen, +,-,*, /, und die Längenfunktion len()
- Analysiere, in welche Komplexitätsklasse dein Programm gehört. Es sei n die Anzahl Elemente der Liste

Lösung: Den Mittelwert finden

```
L = [10, 20, 30, 40, 50, 70, 90, 110, 23, 12, 34, 45, 56, 67]
gesamtsumme = 0

for zahl in L:
    gesamtsumme = gesamtsumme + zahl
anzahl_elemente = len(L)

mittelwert = gesamtsumme / anzahl_elemente
print("Der Mittelwert ist:", mittelwert)
```

n Schritte

1 Schritt (Division)

⇨ n + 1 Schritte

⇨ O(n)

Einen PIN Knacken

- Du willst einen PIN knacken. Du weisst, er hat n Ziffern [0-9].
- Schreibe ein Programm, um den PIN zu knacken
- Du weisst, dass im Passwort mindestens 3 mal die Ziffer 1 vorkommt
- Was ist die Komplexitätsklasse deines Algorithmus?