

Informatik 2

Komplexität (Aufwand) von Algorithmen

Irene Rothe

Zi. B 241

irene.rothe@h-brs.de

Instagram: irenerothesdesign



Informatik: 2 Semester für Ingenieure

Informatik = Lösen von Problemen mit dem Rechner

✓ Zum Lösen von Problemen mit dem Rechner braucht man **Programmierfähigkeiten** (nur mit Übung möglich): Was ist Programmieren?

✓ Was ist ein Flussdiagramm?

→ **Programmiersprache C:**

- ✓ Elementare Datentypen
- ✓ Deklaration/Initialisierung
- ✓ Kontrollstrukturen: if/else, while, for
- ✓ Funktionen
- ✓ Felder (Strings)
- ✓ Zeiger
- ✓ struct
- Speicheranforderung: malloc
- Listen
- Bitmanipulation

✓ Wie löst der Rechner unsere Probleme? → mit **Dualdarstellung** von Zeichen und Zahlen und mit Hilfe von **Algorithmen**





→ Ein Beispiel für ein Problem: **Kryptografie**

→ Sind Rechner auch Menschen? → **Künstliche Intelligenz**

→ Für alle Probleme gibt es viele Algorithmen. Welcher ist der Beste? → **Aufwand** von Algorithmen



Design der Folien

-  hinterlegt sind alle Übungsaufgaben. Sie sind teilweise sehr schwer, bitte absolut nicht entmutigen lassen! Wir können diese in Präsenz besprechen oder über Fragen im Forum.
-  hinterlegte Informationen und grüne Smileys sind wichtig und klausurrelevant.
- Alles hinter „**Achtung**“ unbedingt beachten!
-  verwende ich, wenn überraschende Probleme auftreten können. Wenn Sie schon programmiererfahrend sind, können das eventuell besonders große Überraschungen für Sie sein, wenn Sie eine andere Sprache als C kennen.
- „Tipp“ benutze ich, um Ihnen einen Weg zu zeigen, wie ich damit umgehen würde.
- „Bemerkung“ in Folien beziehen sich meist auf Sonderfälle, die nicht unbedingt klausurrelevant sind, aber für Sie beim Programmieren eine Bedeutung haben könnten
-  hinter diesem Symbol ist ein Link fürs Anhören bzw. Gucken weiterer Infos

Aufbau der Folien:

- Am Anfang motiviere ich gerne mit einem Beispiel, das eventuell schwer verständlich ist. Wem das nicht zusagt, dem empfehle ich, diese Folien zu überspringen.
- Weiter arbeite ich mit vielen Beispielen, die oftmals immer wieder das Gleiche erklären nur auf unterschiedliche Arten. Hat man einen Sachverhalt einmal verstanden, braucht man eventuell diese Beispiele nicht.
- Folien, die mit **Einschub** beginnen, beinhalten Zusatzinformationen, die nicht nötig für das Verständnis des Themas sind.
- Grün hinterlegte Informationen sind das, was Sie aus der Vorlesung rausnehmen sollen, alles andere sind vertiefende Informationen und Motivation.

Aufwand: Motivation

Ich habe zwei Sortieralgorithmen (z.B. Bubble und Quick).
Welchen nehme ich?

- **Berechenbarkeit:** Frage danach, welche Probleme überhaupt berechenbar sind oder nicht
- **Aufwand:** Frage, ob berechenbare Probleme effizient (also praktikabel) berechenbar sind oder nicht

Aufwand: Fragen zum Thema

- Wie kann man 2 Algorithmen, die das gleiche tun, vergleichen, ohne dass man einen speziellen Rechner im Auge hat?
- Wie berechnet man die Laufzeit von Algorithmen?
- Was ist eine Aufwandfunktion?
- Was sind nicht praktisch lösbare Algorithmen?
- Was ist das große offene Problem in der Informatik?

Warum nicht nur Programmierung?

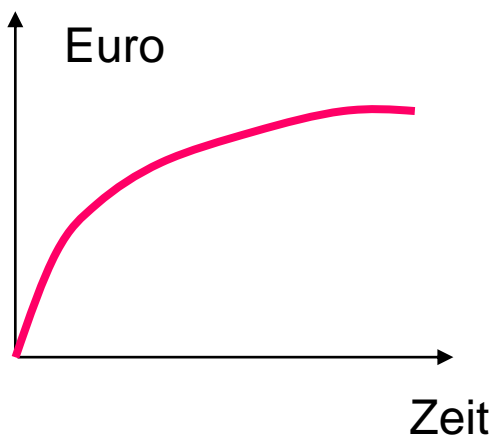
Mathematik \neq Rechnen

Informatik \neq Programmieren

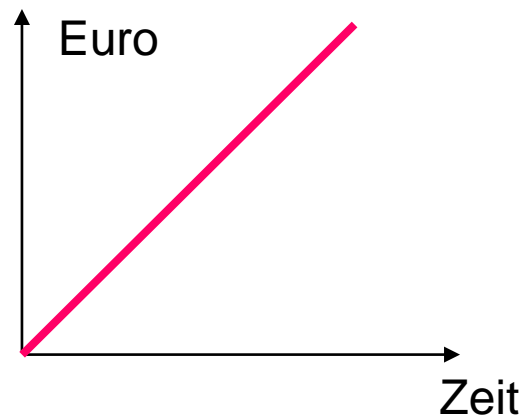


Beispiel - Geld

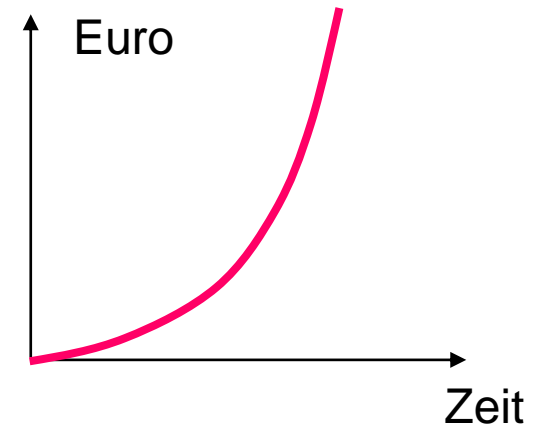
Welcher der folgenden Investments A, B oder C macht über eine Zeit hinweg das meiste Geld?



A



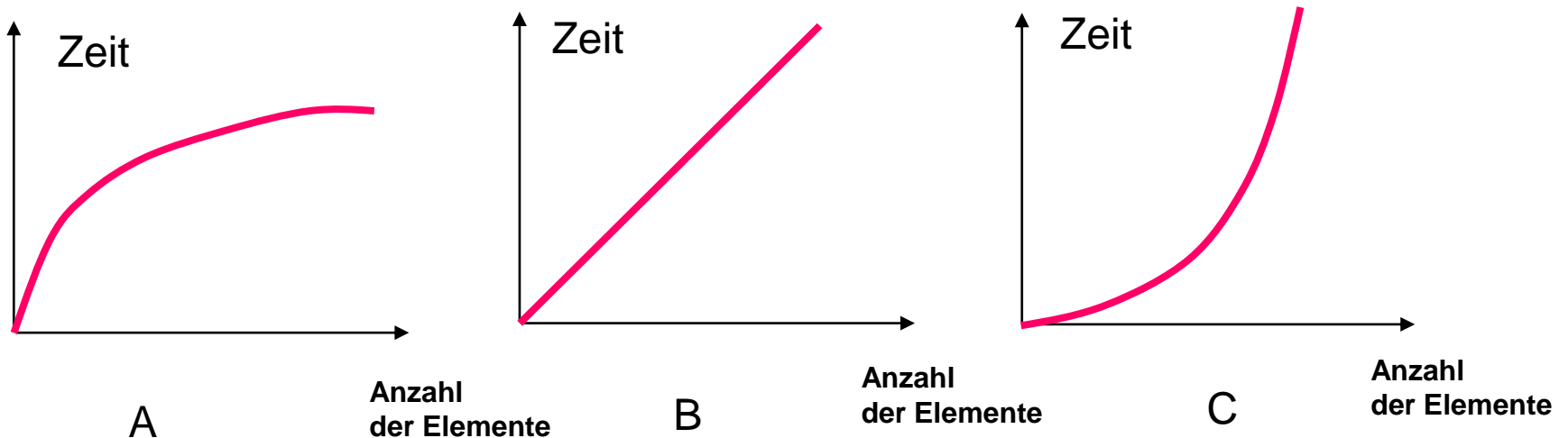
B



C

Beispiel: Sortieralgorithmus

Welchen Sortieralgorithmus würden Sie wählen?



<https://www.youtube.com/watch?v=5ObeH3tVvk4>

Warum interessieren uns effiziente Algorithmen?

Bei vielen Problemlösungen ist die Zeit wichtig, das heißt, umso schneller das Problem gelöst wird, umso zufriedener ist der Kunde. Zum Beispiel für:

- Internetsuchmaschinen
- Berechnung von Bahnverbindungen
- Datenkompression
- Computerspiele

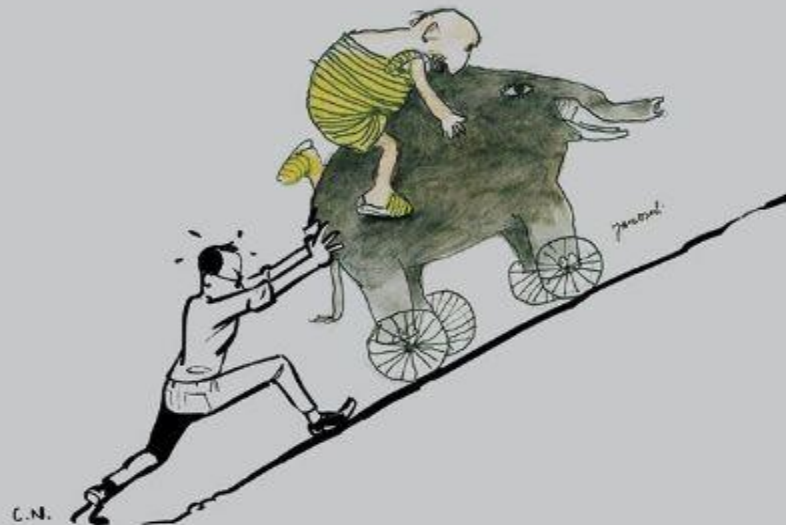


Was sind harte (ineffiziente) Probleme?

Die Sonne wird ein kalter Brocken sein, bevor das Problem gelöst ist.

Was ist schwer?

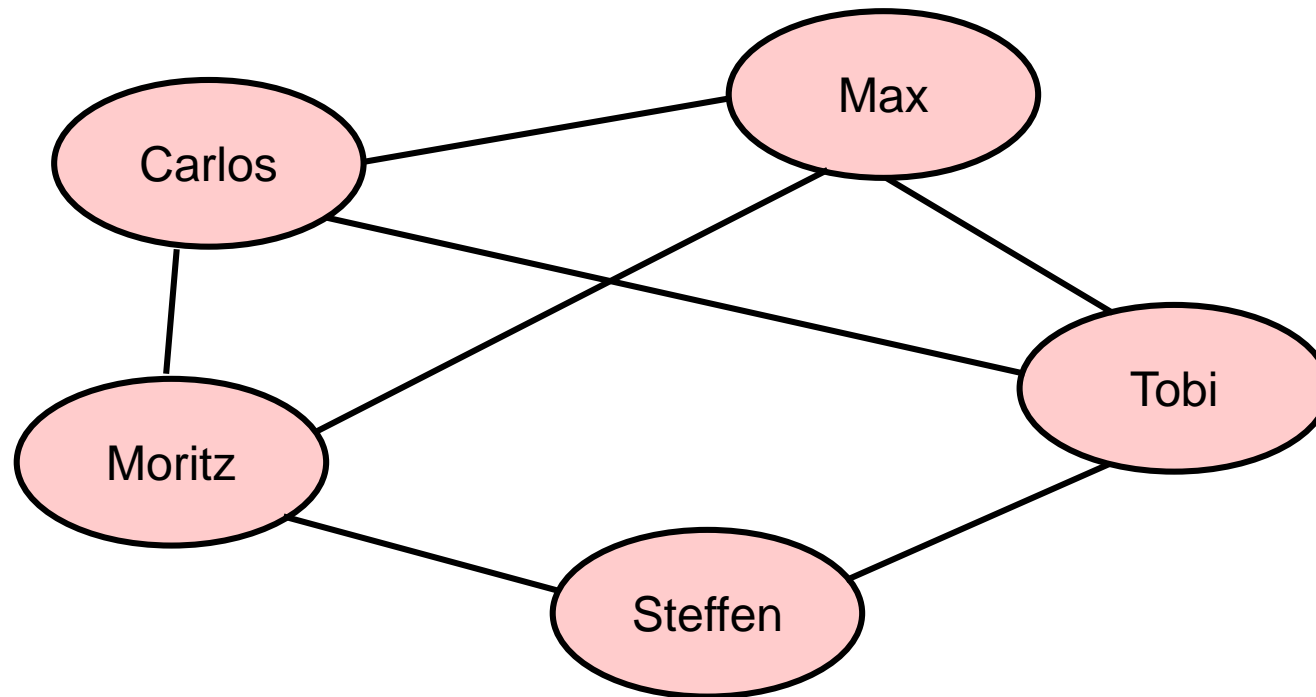
Herr Janosch,
was ist eigentlich schwer?



»Einen Elefanten einen Berg hochzuschieben
ist schwer. Außer wenn man Hilfe von
Wondrak bekommt. Er setzt sich obendrauf
und macht sich ganz leicht.«

Beispiel: 2-Färbbarkeit eines Graphens

Zimmeraufteilung bei einer Klassenfahrt



Reichen zwei Zimmer für eine erholsame Zeit des Klassenlehrers aus?

Beispiel: Algorithmus für Zweifärbbarkeit

Reichen **zwei** Zimmer für eine Klasse aus?

1. Wähle einen beliebigen Schüler (Knoten) und stecke ihn in Zimmer 1 (färbe ihn rot).
2. Stecke alle Graph-Nachbarn dieses Schülers in Zimmer 2 (färbe alle Nachbarknoten blau).
3. Stecke nun alle Schüler, die einen Graph-Nachbarn in Zimmer 1 haben, in Zimmer 2 (färbe alle Nachbarn eines roten Knotens blau).
4. Stecke alle Schüler, die einen Graph-Nachbarn in Zimmer 2 haben, in Zimmer 1 (färbe alle Nachbarn eines blauen Knotens rot).
5. Wiederhole 3. und 4. bis alle Schüler einem Zimmer zugeordnet sind (färbe alle Knoten ein).
6. Gibt es nun zwei Schüler, die Graph-Nachbarn sind und dennoch dem gleichen Zimmer zugeordnet wurden, ist obige Frage beantwortet:

Zwei Zimmer reichen NICHT aus!

Man sieht: die Antwort für obige Frage ist schnell gefunden.



Beispiel: Algorithmus für Dreifärbbarkeit

Reichen **drei** Zimmer für eine Klasse aus?

→ Bei einer Klasse mit 32 Schülern durchläuft der naivste Algorithmus im schlechtesten Fall $3 \cdot 32$ Anweisungen.

Für 32 Schüler mag so ein Algorithmus noch in vernünftiger Zeit ein Ergebnis liefern.

Beispiel:

Lagerung von Chemikalien, wobei gewisse Paare nicht nebeneinander aufbewahrt werden dürfen.

Bei 1000 Chemikalien sind $2 \cdot 1000$ schon 10^{308}

Bedeutend schnellere Algorithmen sind bis heute nicht bekannt, das heißt, bis heute wurde kein *effizienter* Algorithmus gefunden.

Nützlich z.B. für die Verschlüsselung: Man erzeugt einen Graphen, der dreifärbbar ist (was sehr einfach ist) und löscht dann die Einfärbungen, so dass nur der Graph selbst übrig bleibt. Nur wer eine Einfärbung der Knoten kennt (und zwar nur der Erzeuger) kann die Tür öffnen oder sich in einen Rechner einloggen.....



Beispiel: Fibonacci-Algorithmen

Fibonacci-rekursiv:

```
double fibonacci(int n){
    if (n==0) return 0;
    if (n==1) return 1;

    return fibonacci(n-2)+fibonacci(n-1);
}
```

Fibonacci mit
Zwischenspeicher:

```
double fibonacci(int n){
    int i=0;
    int f_i=0;
    int f_i_1=1;
    int f_i_2=0;
    if(n==0) return 0;
    if(n==1) return 1;
    for(i=2;i<n+1;i++){
        f_i=f_i_1+f_i_2;
        f_i_2=f_i_1;
        f_i_1=f_i;
    }
    return f_i;
}
```

Fibonacci-iterativ:

```
double fibonacci(int n){
    return 1/sqrt(5)*(pow((1+sqrt(5))/2,n)+pow((1-sqrt(5))/2,n));
}
```

Welcher Algorithmus ist der beste?



Begriffe

- Problemumfang: Größe der Eingabe
- Zeitaufwand: Abstraktion der Dauer eines Algorithmus
- Platzbedarf : Speicherbedarfs des Algorithmus
- bester, durchschnittlicher und schlechter Fall der Effizienz eines Algorithmus
- Komplexität eines Problems: minimale Komplexität eines Algorithmus zur Lösung des Problems

Beurteilung von Algorithmen nach...

- Programmieraufwand?
- elegante Lösung?
- Speicherverbrauch?
- Zeit (Zeitintensive Operationen: Lesen und Schreiben von und in Speicher)?

Verschiedene Algorithmen zur Lösung eines Problems

→ Beurteilung der Algorithmen nach ...

- schneller/langsamer
- mehr Speicherverbrauch/weniger

→ Komplexität eines Algorithmus (Zeit: Laufzeit/Raum: Speicherplatz)

Algorithmen sind auch abhängig von der Umgebung:

- Student, der Algorithmus mit Papier und Bleistift berechnet
- Compiler, der Programmtext in Maschinenprogramm umwandelt
- Betriebssystem, das das Maschinenprogramm ausführt
- Hardware (Hardwarechips), auf der das Betriebssystem läuft

Solche Zeitmessungen liefern Erkenntnisse über die Schnelligkeit von Algorithmen in einer **konkreten** Umgebung. Umgebungen haben die Eigenschaften, sehr zahlreich und vergänglich zu sein.

Deshalb ist man interessiert, Laufzeitverhalten von Algorithmen **unabhängig von Umgebungen** zu erhalten, die auch noch in ein paar Jahren gelten.



1. → Überlegungen zu Algorithmen
2. Laufzeiten von Algorithmen
3. Aufwand eines Algorithmus
4. Laufzeitberechnung - Regeln
5. Komplexität (Aufwand) – Beispiele
6. Aufteilung der Probleme unserer Welt
7. Berühmtes offenes Problem



1. Überlegungen zu Algorithmen

→ Effizienzüberlegungen

Man vergleicht Algorithmen, die das gleiche tun,

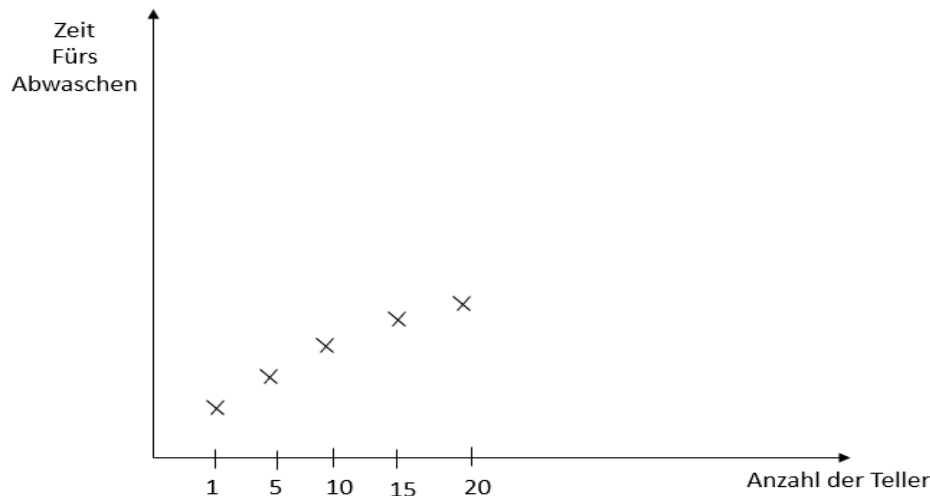
- nach Laufzeit, ohne einen ganz bestimmten Rechner im Sinn zu haben und
- nach Speicherplatzbedarf.

Einteilung von algorithmisch lösbaren Problemen (für die also ein Algorithmus existiert) nach Klassen (Komplexitätsklassen).

Achtung: Im Folgenden beschäftigen wir uns nur noch mit dem Laufzeitverhalten von Algorithmen.

2. Laufzeit: Tellerabwaschen mit der Hand

Stellen wir uns vor, man wäscht viele Teller mit der Hand ab. Die Telleranzahl ist die Eingabegröße. Weiter misst man die Zeit, die man braucht, um einen Teller abzuwaschen, um zwei Teller abzuwaschen, drei und immer so weiter. Dies trägt man in ein Koordinatensystem ein, also an der x-Achse ist die Anzahl der Teller verzeichnet und an der y-Achse ist die Zeit eingetragen, die man für den Abwasch der entsprechenden Zahl von Tellern gebraucht hat.



Misst man die Zeit für jede mögliche Anzahl von Tellern und trägt sie in das Koordinatensystem ein, erhält man eine Aufwandsfunktion f fürs Tellerabwaschen mit der Hand.

2. Laufzeit: Sequentielle Suche

Gegeben: n beliebige ganze Zahlen in einem Feld zahlen[n]

Gesucht: Ist x unter diesen n Zahlen?

```
int n=10;
int x,i;
int zahlen[]={1,2,5,6,8,9,18,19,22,27};
printf("Bitte geben Sie eine Zahl ein:");
scanf("%i",&x);
i=0;
while (x != zahlen[i] && i < n)
{
    i=i+1;
}
printf("An folgender Stelle steht die gesuchte Zahl: %i",i+1);
```

Bei erfolgreicher Suche: Ausgabe i (< 11)

Bei erfolgloser Suche: Ausgabe 11

2. Laufzeit: Suchalgorithmus

Wie lange der Algorithmus rechnet, hängt von der **Eingabe** ab.

Fragen:

1. Wie viele Durchläufe der while-Schleife werden ausgeführt im **schlechtesten Fall**?
2. Wie viele Durchläufe der while-Schleife werden ausgeführt im **besten Fall**?
3. Wie viele Durchläufe der while-Schleife werden ausgeführt **im Mittel**?

Antworten:

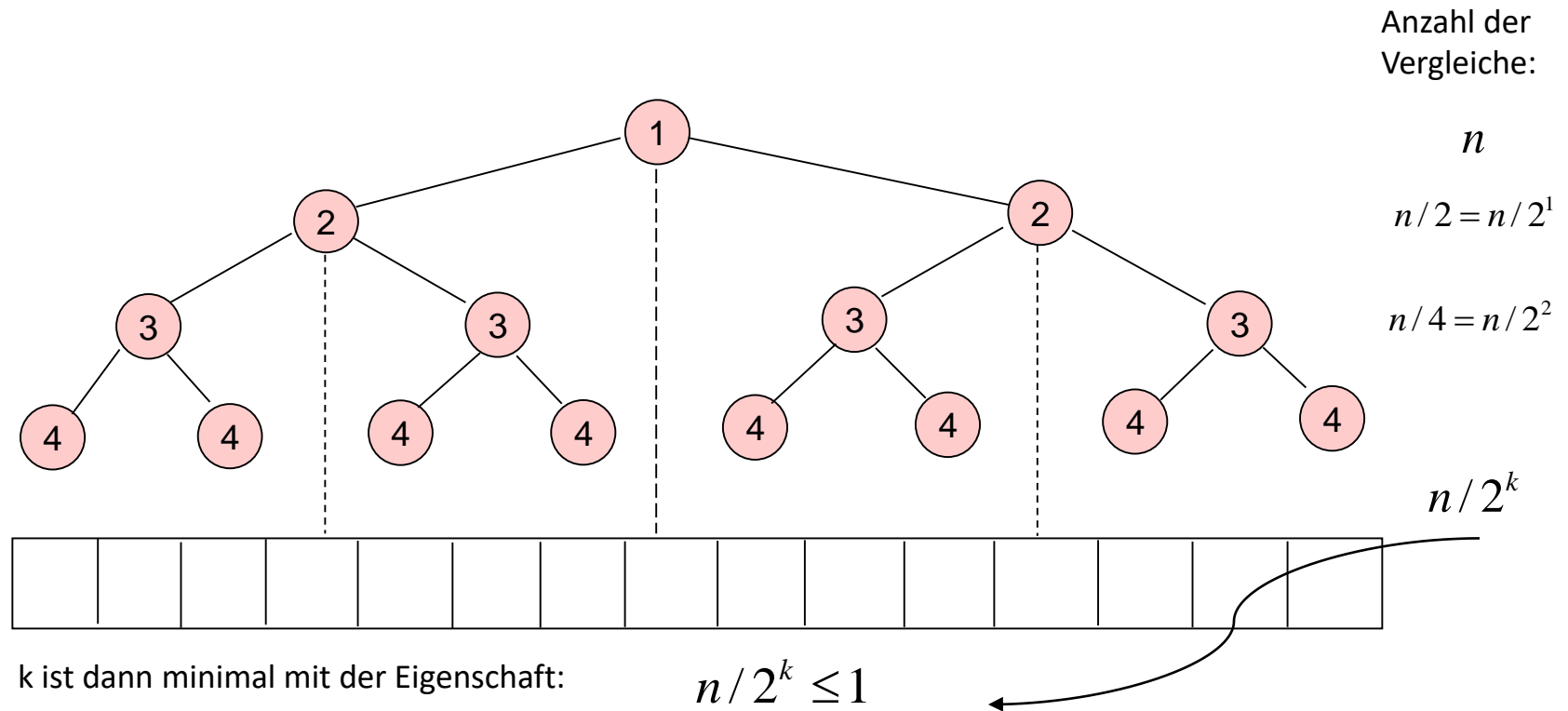
1. Im schlechtesten Fall (die gesuchte Zahl kommt gar nicht vor): $n+1$
2. Im besten Fall (die gesuchte Zahl ist die erste): 1
3. Im Mittel (viele Durchläufe, wobei die gesuchte Zahl an jeder Stelle gleichhäufig stehen kann): $(n+1)/2$



2. Laufzeit: Binären Suche

Aufwand entspricht Anzahl der Vergleiche bei einer Liste der Länge n

→ Sukzessive Halbierung führt zum Logarithmus (zur Basis 2)



Es gilt: $2^k \geq n > 2^{k-1} \Rightarrow k \geq \log n > k-1 \Rightarrow k = \lceil \log n \rceil$



2. Laufzeiten von Algorithmen

- Einen schnellen Algorithmus zu haben, ist wichtig, wenn der Algorithmus viel benutzt wird (z.B. Suche und Sortieren).
 - Laufzeitbetrachtungen kann man dann in Stunden, Minuten und Sekunden umrechnen für einen konkreten Rechner.
 - Die Laufzeit hängt von der Länge der Eingabewerte ab, d.h. von der Größe des Problems.
 - Interessant ist folgendes:
 - **mittlere** Laufzeit
 - Laufzeit im **schlechtesten** Fall (Laufzeiten im besten Fall sind ein Glück und damit uninteressant)
- Ein Algorithmus ist **effizient**, wenn er sehr gute Laufzeiten hat.
- **Komplexitätstheorie** beschäftigt sich mit der Aufteilung aller Algorithmen in Klassen bzgl. ihrer Effizienz.

4. Laufzeitberechnung - Regeln

Tipp: Eine gute Pi-mal-Daumen-Regel für die Berechnung der Aufwandsfunktion eines Algorithmus ist es, alle Semikolons zu zählen, die im Programmcode vorkommen. Eine Anweisung, beendet durch ein Semikolon, ist in der Regel eine Zuweisung, also das Schreiben eines Wertes in den Speicher des Rechners. Wie oft diese Zeilen mit Semikolons abgearbeitet werden, hat mit der Größe der Eingabe zu tun. Die Eingabegröße bestimmt in der Regel, welche if oder else-Verzweigungen ausgeführt werden oder wie oft Schleifen durchlaufen werden. Es gelten folgende Regeln für die Berechnung der Aufwandsfunktion eines Algorithmus:

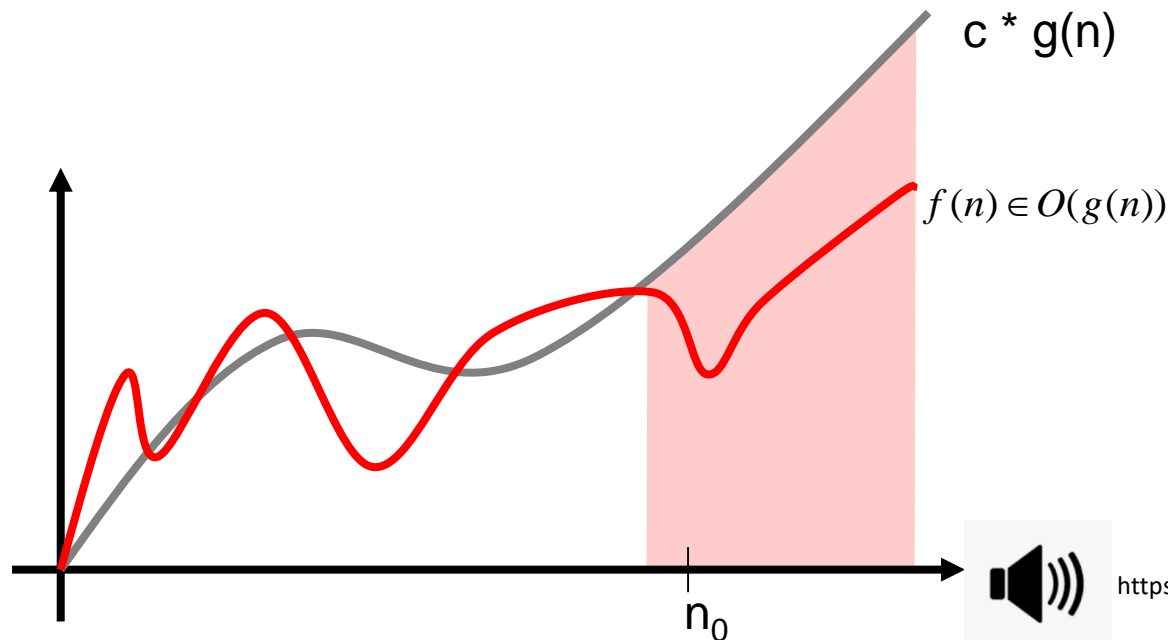
	Laufzeit
jegliche Anweisungen (z.B. Zuweisung eines Wertes)	konstanter Aufwand, unabhängig von der Eingabe (später $O(1)$ genannt)
Sequenz von Anweisungen	Summe der Laufzeiten der einzelnen Anweisungen
if/else Anweisung	Summe der Laufzeit für die Auswertung der Bedingung (oft $O(1)$) und der Laufzeit für den/die bedingt auszuführenden Schritte
Schleifen	Summe der Laufzeiten über alle Durchläufe und der innerhalb der Schleife auszuführenden Schritte



3. Aufwand $O(g(n))$ - Definition

Eine *Aufwandsfunktion* $f(n)$ ist die Berechnung der Zeit, die der Algorithmus braucht, um auf ein Ergebnis zu kommen, wobei n die Größe der Eingabe ist.

Kann man die Aufwandsfunktion $f(n)$ durch eine *einfache* Vergleichsfunktion $g(n)$ abschätzen, und es gilt, dass alle Funktionswerte von f ab einer bestimmten Eingabegröße n_0 , immer unter den Funktionswerten von $g(n)$ liegen (das heißt $f(n) \leq c \cdot g(n)$ oder g ist eine asymptotische obere Schranke), dann kann man dies wie folgt beschreiben: $f(n) \in O(g(n))$. O von g beschreibt dann den Aufwand.



Das heißt: ab n_0 wächst f nicht stärker als g .



<https://www.youtube.com/watch?v=EJ4BV6Z9ZY>

3. Aufwand eines Algorithmus

→ Mehr zu: $f(n) \in O(g(n))$

Der Algorithmus mit der Aufwandsfunktion f kann durch g abgeschätzt werden. Man sagt dazu, dass der Algorithmus einen „**Aufwand** O von g “ hat. Dies wird auch **O-Notation** genannt und ist ein Weg in der Informatik, die Komplexität eines Algorithmus (z.B. bzgl. seiner Laufzeit) zu charakterisieren.

g sollte so klein wie möglich sein, um den Algorithmus so attraktiv wie möglich zu machen. g schätzt dann sozusagen den Mindestaufwand des Algorithmus ab.

4. Aufwand - Bestimmung

Die O-Notation ist also eine obere Grenze für eine Klasse von Funktionen. Aus der Aufwandsfunktion f ermittelt man die einfache Funktion g wie folgt:

- von der Aufwandsfunktion behält man nur die höchste Potenz (in Abhängigkeit von der Eingabe n) und
- lässt alle anderen additiven Konstanten und Terme mit kleiner Potenz abhängig von der Eingabegröße weg,
- sowie alle konstanten Faktoren vor der höchsten Potenz.

4. Aufwand $O(g(n))$: Beispiele

Beispiele:

$$\begin{array}{llll} f(n) = 3n+5 & \rightarrow O(n) & \text{mit} & g(n)=n \\ f(n) = 4n^3 + 2n^2 + n - 7 & \rightarrow O(n^3) & \text{mit} & g(n)=n^3 \end{array}$$

Beispiele für typische Funktionen, die als Aufwand geeignet sind:

$1, \log(n), n \log(n), \log(\log(n)), n^2, n^3, n^{\log(n)}, 2^n, n!$

Aufwand: Übung

Wie viele Rechenoperationen sind notwendig?

$$1^2 + 2^2 + 3^2 + 4^2 + \dots + (n-1)^2 + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Additionen: ??

Multiplikationen: ??

Divisionen: ??

f(n)= ??

Aufwand: ??

Additionen: ???

Multiplikationen: ??

Divisionen: ??

f(n)= ??

Aufwand: ??

Achtung: Auf der nächsten Folie ist die Lösung!



Aufwand: Lösung

Wie viele Rechenoperationen sind notwendig?

$$1^2 + 2^2 + 3^2 + 4^2 + \dots + (n-1)^2 + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Additionen: $n-1$

Multiplikationen: n

f(n)= $2n-1$

Aufwand: $O(n)$

Additionen: 2

Multiplikationen: 3

Divisionen: 1

f(n)= 6

Aufwand: $O(1)$



4. Aufwand - Bemerkung

In der Regel ist es sehr schwer, die Funktion g zu finden oder besser gesagt, die Funktion g mit dem *geringsten* Wachstum zu finden, die den Aufwand des Algorithmus nachweisbar abbildet. Die Beweisführung dafür gehört zum Aufgabengebiet von Komplexitätsforschern. Ein Algorithmus ist umso besser und teurer, je schneller er ist. Das muss man aber beweisen können, sonst ist es nur eine Behauptung und keine Garantie. Also sind Wissenschaftler bestrebt, die bestmögliche Komplexität O von g für ihren Algorithmus zu finden und diese zu beweisen, um den Algorithmus dann so teuer wie möglich verkaufen zu können.

5. Aufwand (Komplexität) – wichtige Beispiele

Konstanter Aufwand (Komplexität) $O(1)$: Laufzeit ist nicht von der Eingabe abhängig (z.B. Erhöhung einer ganzen Zahl um Eins).

Lineare Komplexität (Aufwand) $O(n)$: Laufzeit wächst proportional mit der Eingabe (z.B. Fibonacci mit Zwischenspeicher, sequentielle Suche, einfache Schleifen).

Quadratische Komplexität $O(n^2)$: doppelt so große Eingabe vervierfacht die Laufzeit (z.B. Bubblesort, verschachtelte Schleifen).

Kubische Komplexität $O(n^3)$: z.B. Matrixmultiplikation

Logarithmische Laufzeit $O(n \log n)$: z.B. Quicksort

Polynomiale Laufzeit $O(n^k)$: **Eingabegröße** geht in die **Basis** ein.

Exponentielle Komplexität $O(2^n)$: **Eingabegröße** geht in den **Exponenten** ein (z.B. kürzesten Weg finden für den Handelsreisenden).

Zunahme gemäß Fakultät $O(n!)$: z.B. Bildung aller Permutationen



5. Komplexität - Wachstum

$g(n)$	$n = 10$	$n = 50$	$n = 100$	$n = 1000$
$\log n$	1	1,69	2	3
n	10	50	100	1000
n^2	100	2500	10000	1000000
2^n	1024	10^{15}	10^{30}	10^{308}
$n!$	3628800	10^{64}	riesig	riesig

Achtung: Das Universum existiert ca. 15 Milliarden Jahre = 10^{10} Jahre

= $3 * 10^{17}$ Sekunden!



<https://www.youtube.com/watch?v=JqzSPZ00gbM>

5. Alternative Big O Notationen

$O(1) = O(\text{yeah})$

$O(n \log n) = O(\text{nice})$

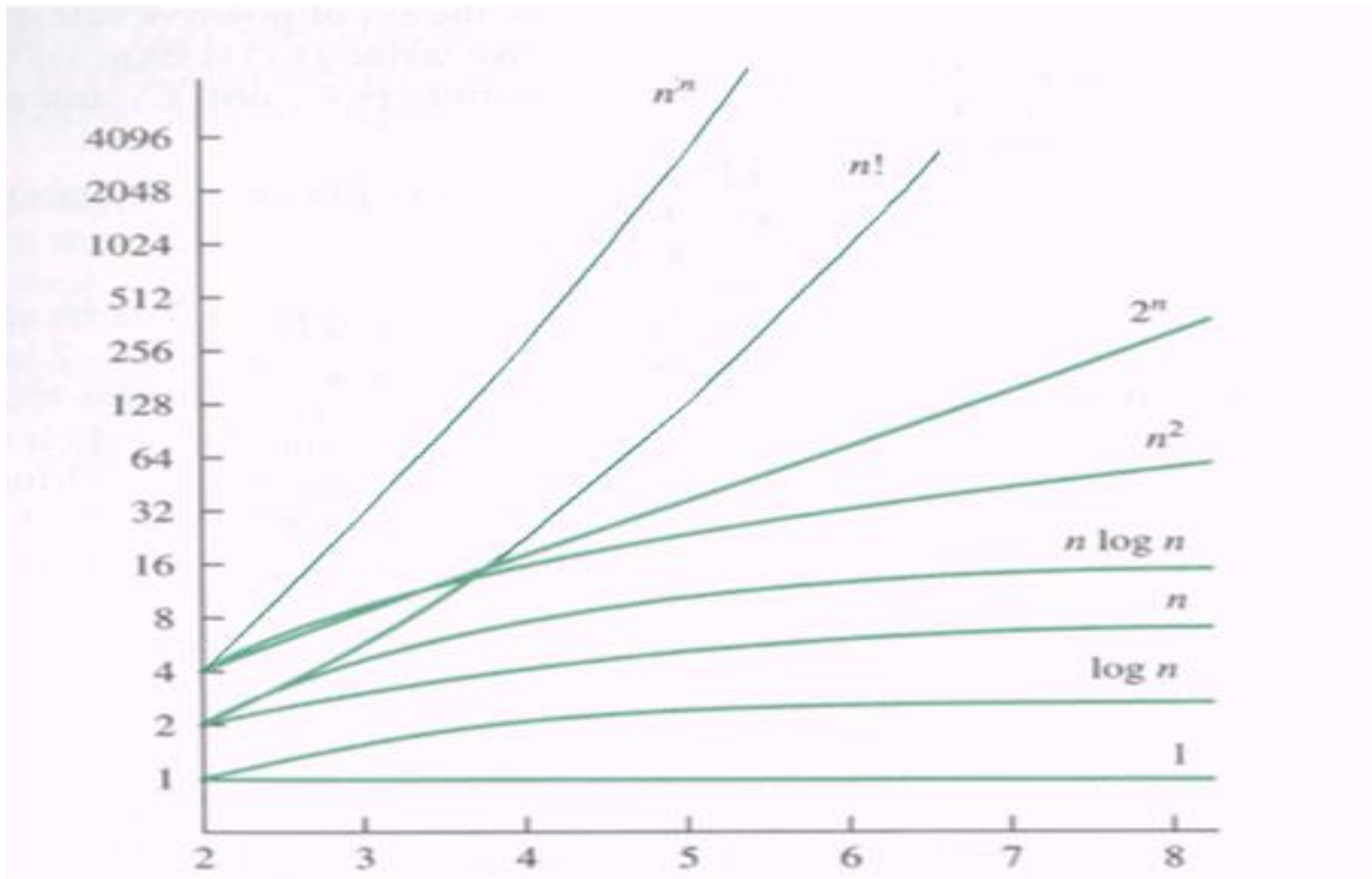
$O(n) = O(\text{ok})$

$O(n^2) = O(\text{my})$

$O(2^n) = O(\text{no})$

$O(n!) = O(\text{mg!})$

Funktionen



5. Aufwand: Vergleich der Sortieralgorithmen

	Insertion	Selection	Bubble	Quick
Laufzeit im Mittel	$\frac{n^2 + n - 2}{2}$	$\frac{n^2 + n}{2}$	$O(n^2)$	$O(n \log n)$

Der Bubblesort ist somit langsamer als der Quicksort, weil die Funktion $n \log n$ langsamer wächst mit der Zeit als n hoch 2.

5. Aufwand: Nichteffizienz

Ein Algorithmus ist *nicht effizient*, wenn die Eingabegröße in den Exponenten des Aufwandes wandert.

Was ist Informatik?

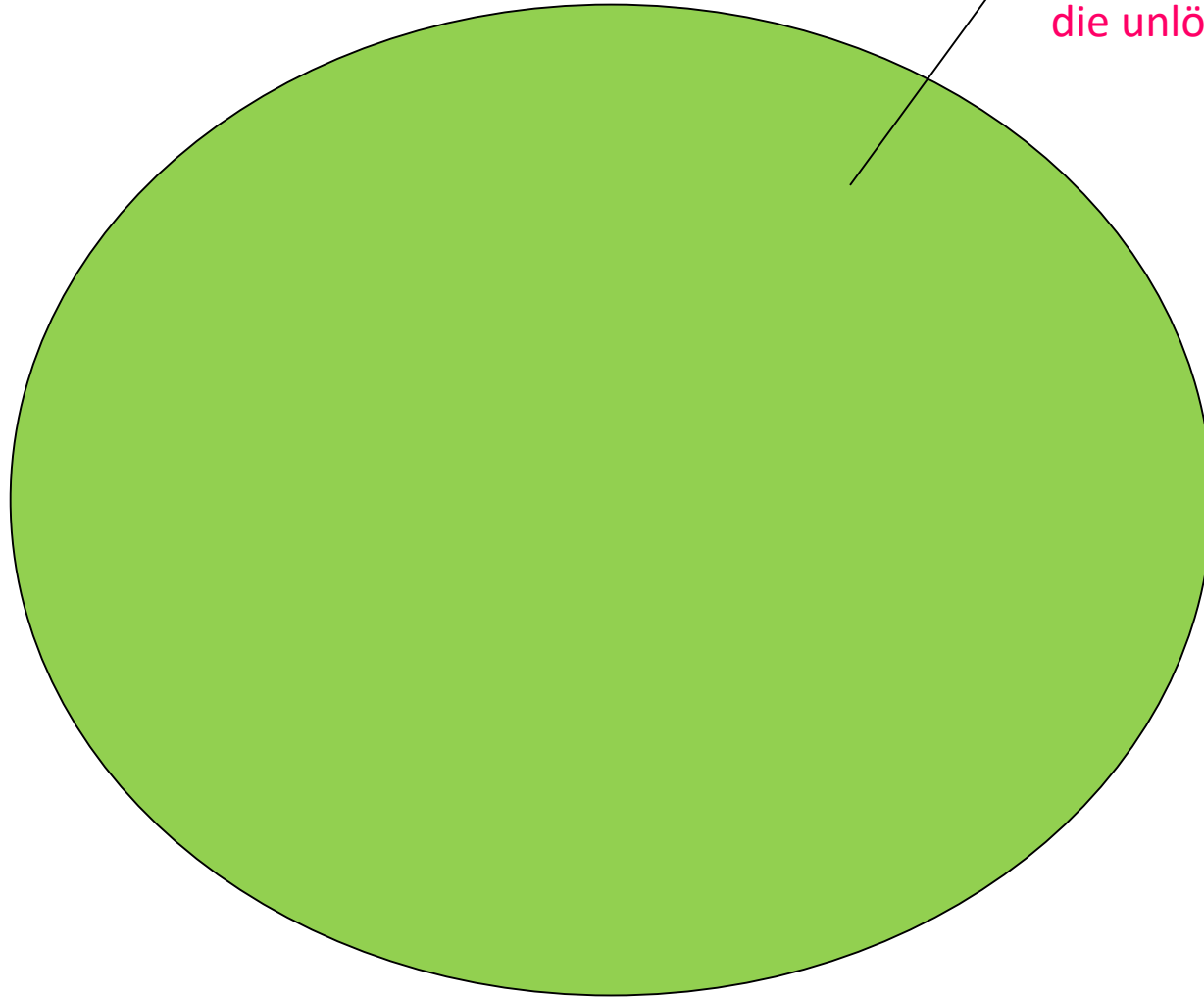
Lösen von Problemen mit dem Rechner
=
Suche nach Algorithmus für Probleme

1. ✓ Überlegungen zu Algorithmen
2. ✓ Laufzeiten von Algorithmen
3. ✓ Aufwand eines Algorithmus
4. ✓ Regeln zur Laufzeitberechnung
5. ✓ Aufwandsfunktionen
6. → Aufteilung der Probleme unserer Welt
7. Berühmtes offenes Problem



6. Aufteilung von Problemen

Alle Probleme
dieser Welt (auch
die unlösbaren).

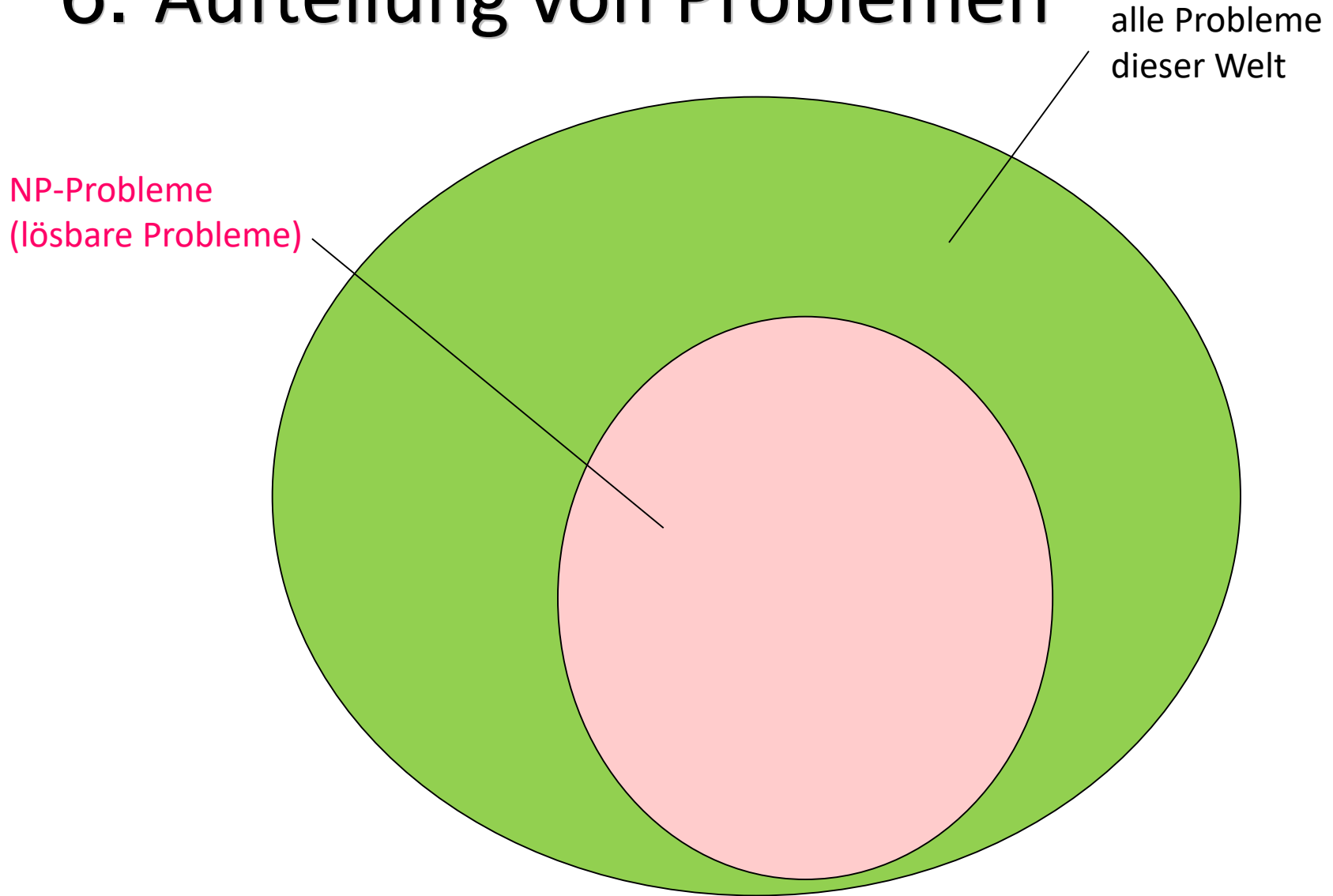


6. Probleme: Definition

Ob ein Problem **algorithmisch lösbar** ist oder nicht, hat mit dem Aufwand der **Entwicklung** des Algorithmus zu tun.

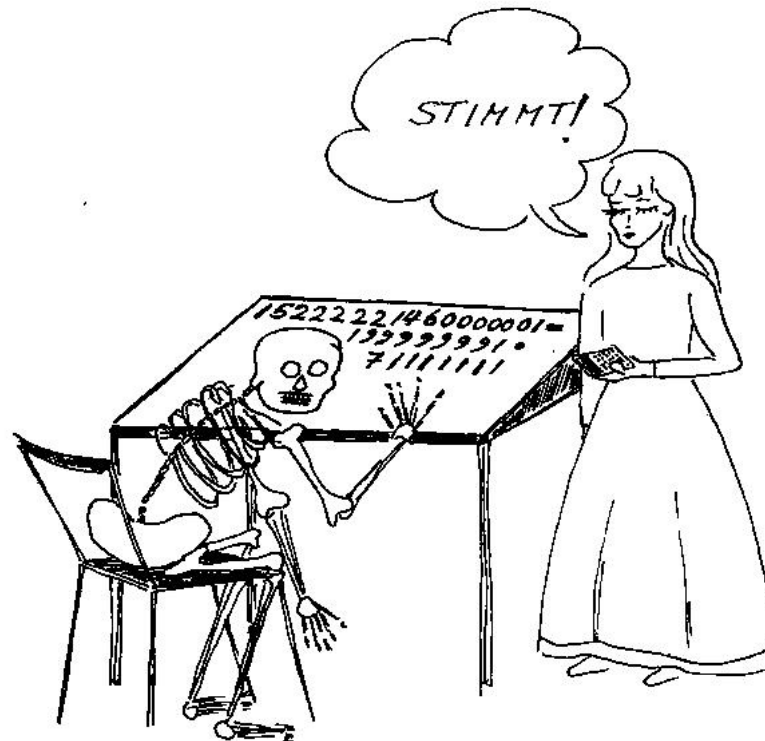
Ob ein Problem **praktisch lösbar** ist oder nicht, hat mit dem Aufwand der **Ausführung** des besten bekannten Lösungsalgorithmus zu tun.

6. Aufteilung von Problemen



6. NP-Probleme

Hat man eine Lösung gegeben, kann man effizient überprüfen, ob sie wirklich das Problem löst.



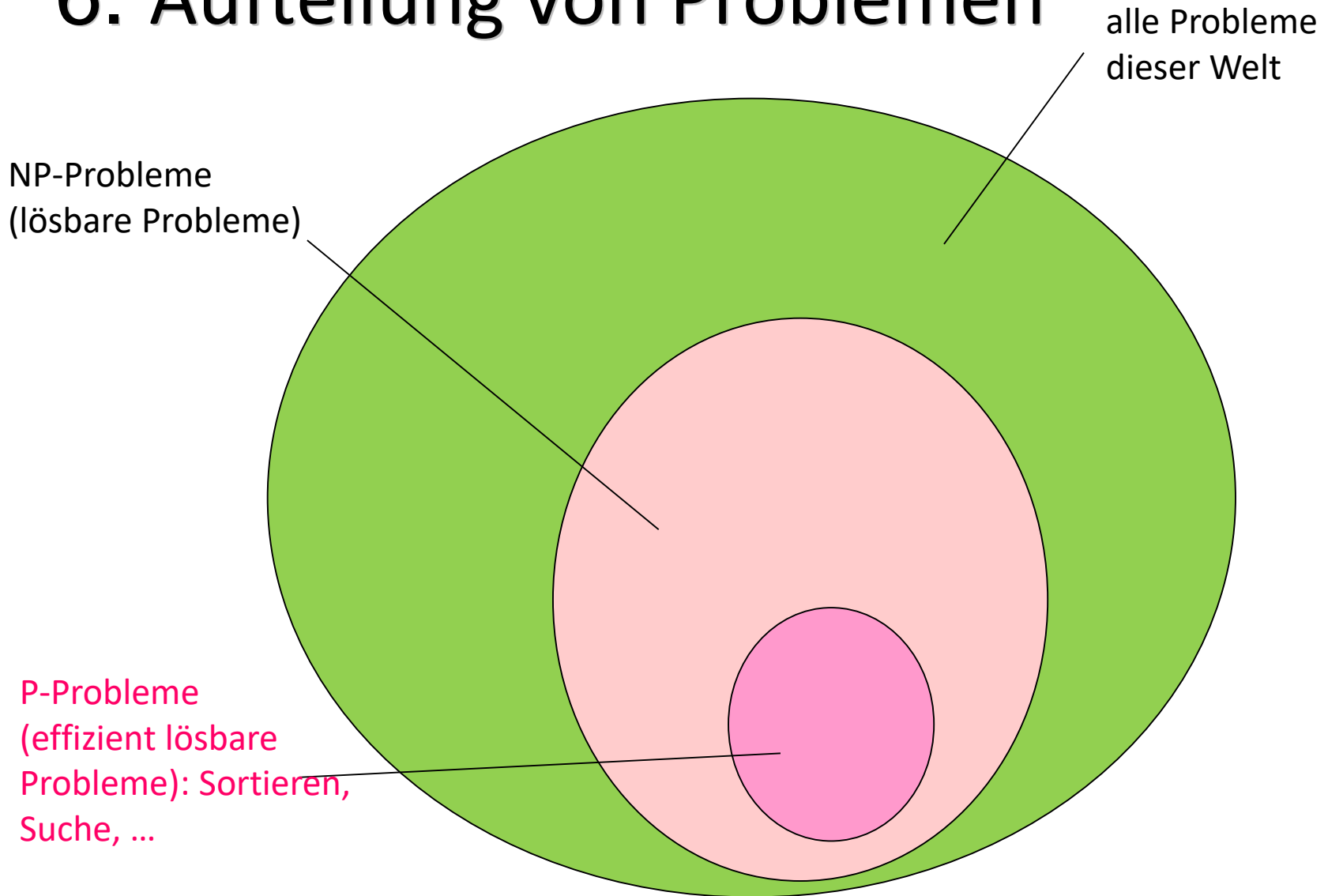
6. Einteilung von lösbaren Problemen

Für die Einteilung in praktisch lösbare und praktisch nicht lösbare Probleme existiert folgende Definition:

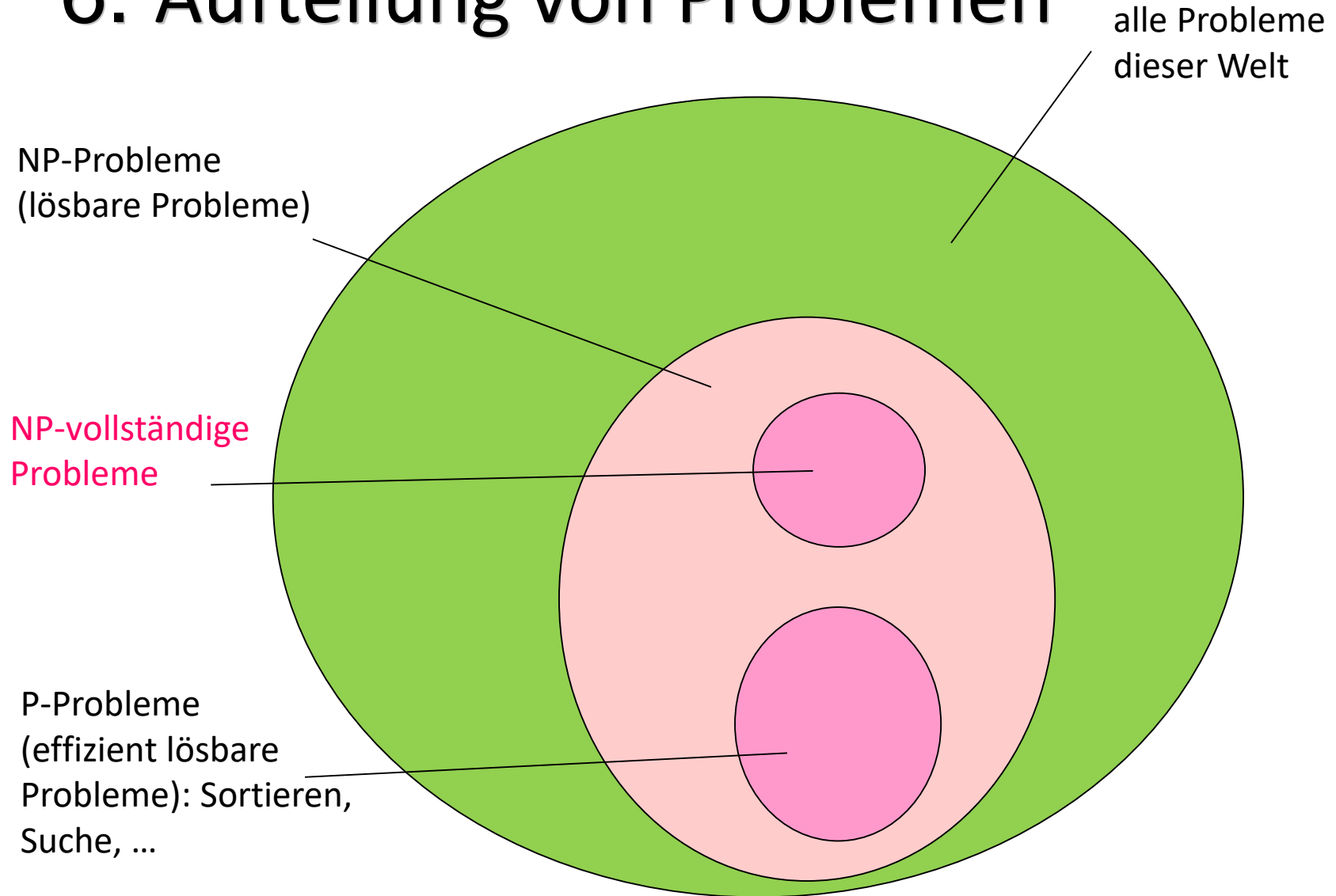
- **Praktisch nicht lösbare Probleme:** Es existieren bis jetzt nur Algorithmen mit *exponentieller* Laufzeit. Das heißt, die **Eingabegröße** geht in der Aufwandsfunktion in den **Exponenten** ein.
- **Praktisch lösbare Probleme:** Es existieren Algorithmen mit polynomialer Laufzeit. Das heißt, **Eingabegröße** geht in der Aufwandsfunktion in die **Basis** ein. Man nennt sie P-Probleme.

Die Grenze zwischen praktischen lösbaren und praktisch nicht lösbaren Problemen ist nicht *scharf*.

6. Aufteilung von Problemen



6. Aufteilung von Problemen



6. Praktisch nicht lösbare Probleme

Problem des Handelsreisenden:

Gesucht wird ein Algorithmus mit:

- **Eingabe:** Graph mit Städten als Knoten und Straßen als Kanten. Zu jeder Straße existiert eine positive ganze Zahl, die die Entfernungen der Städte von einander angibt.
- **Ausgabe:** kürzeste Rundreise, wobei man jede Stadt (außer der ersten Stadt) nur einmal besucht.

Rucksackproblem:

Gesucht wird ein Algorithmus für einen Dieb in einem Kaufhaus mit:

- **Eingabe:** Liste von Gegenständen mit Größe und Wert aus dem Kaufhaus, Größe des Rucksackes des Diebes
- **Ausgabe:** Liste der Gegenstände, die der Dieb in seinen Rucksack packen muss, um maximalen Wert aus dem Kaufhaus rausschleppen zu können.

Aufteilungsproblem:

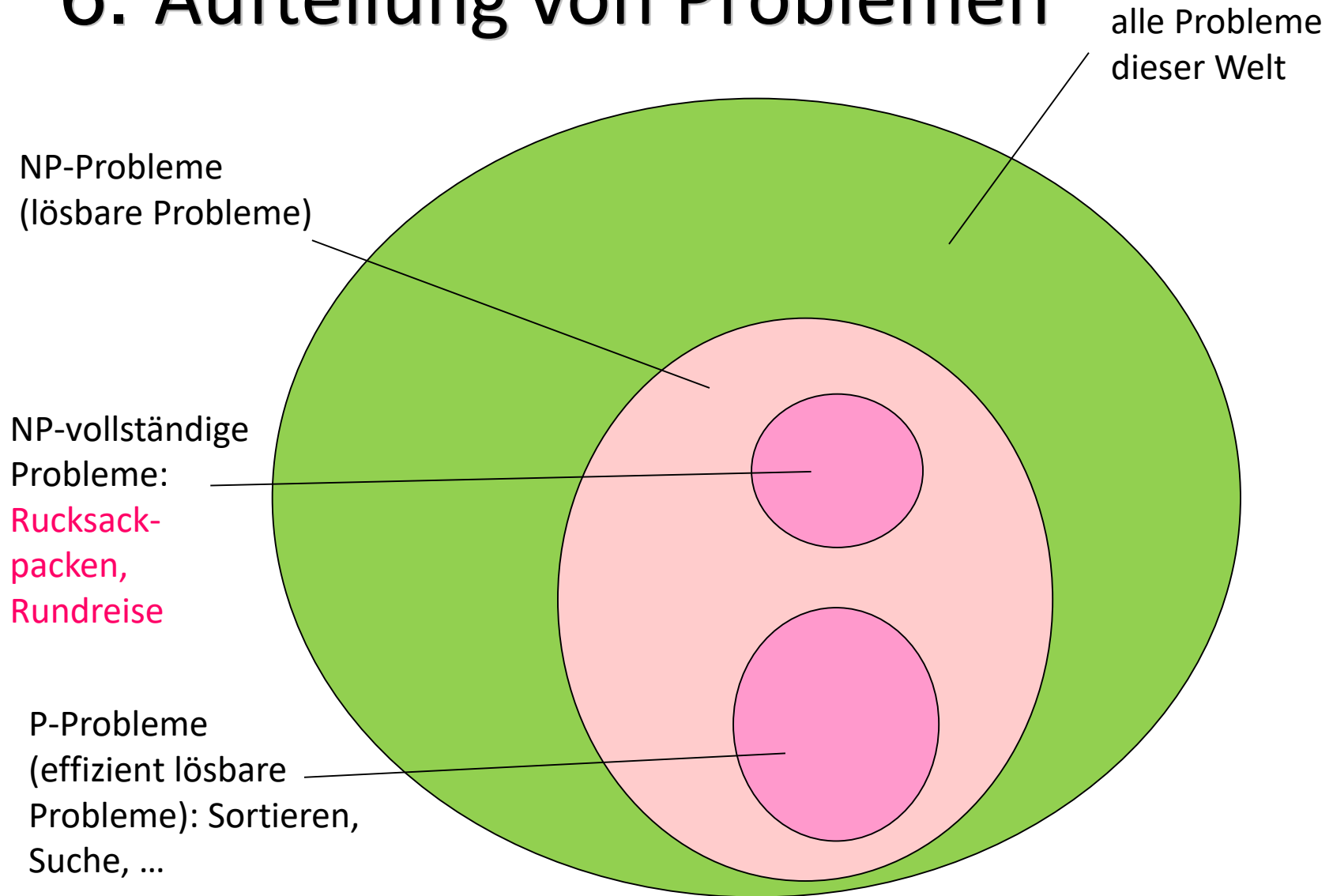
Gesucht wird eine Aufteilung des Erbes unter 2 Schwestern:

- **Eingabe:** Liste von Gegenständen mit Werten aus dem Erbe
- **Ausgabe:** günstigste Aufteilung, so dass sich keine der Schwestern materiell benachteiligt fühlt.

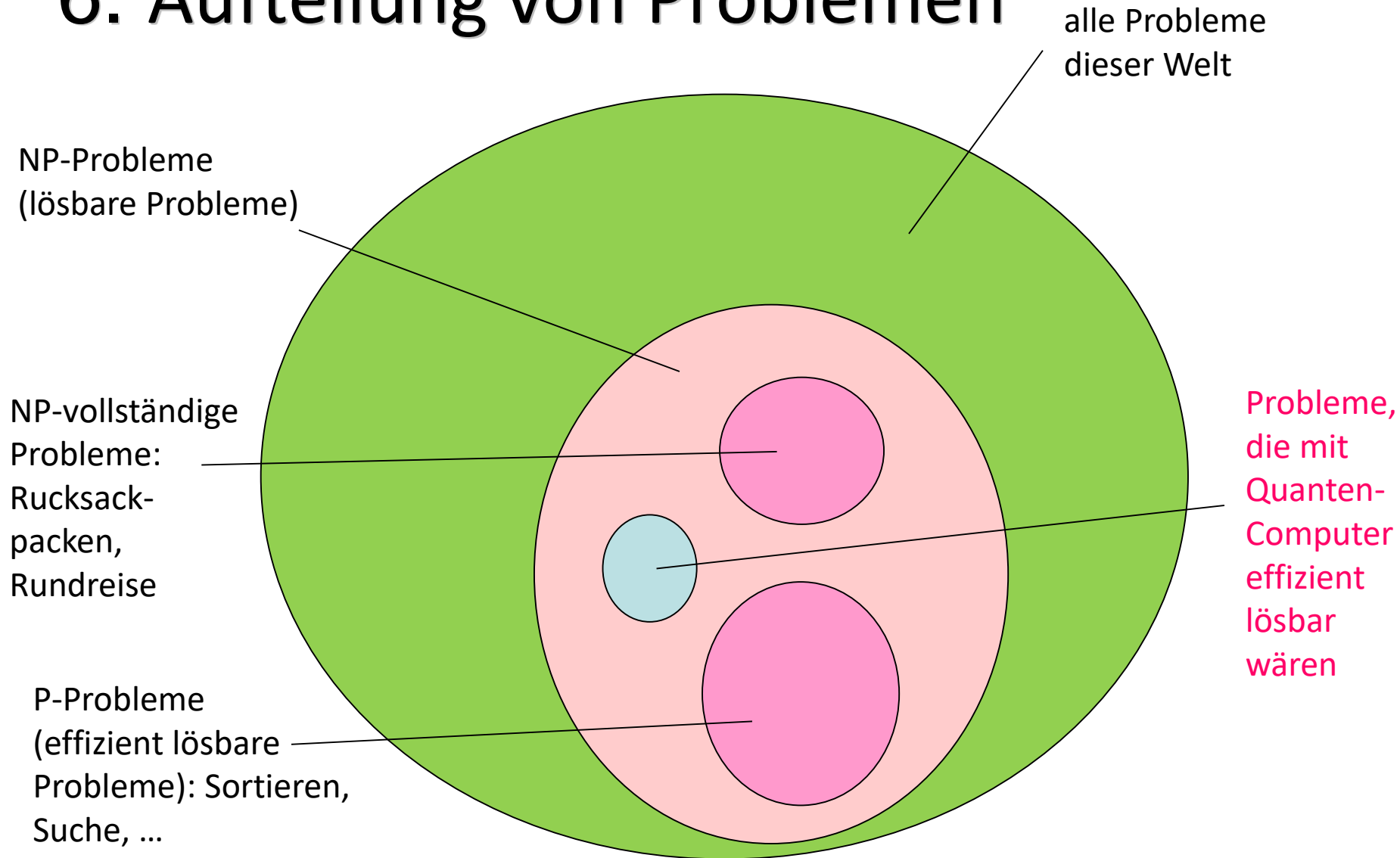
Geheimnummernverschlüsselung: $N=p*q$



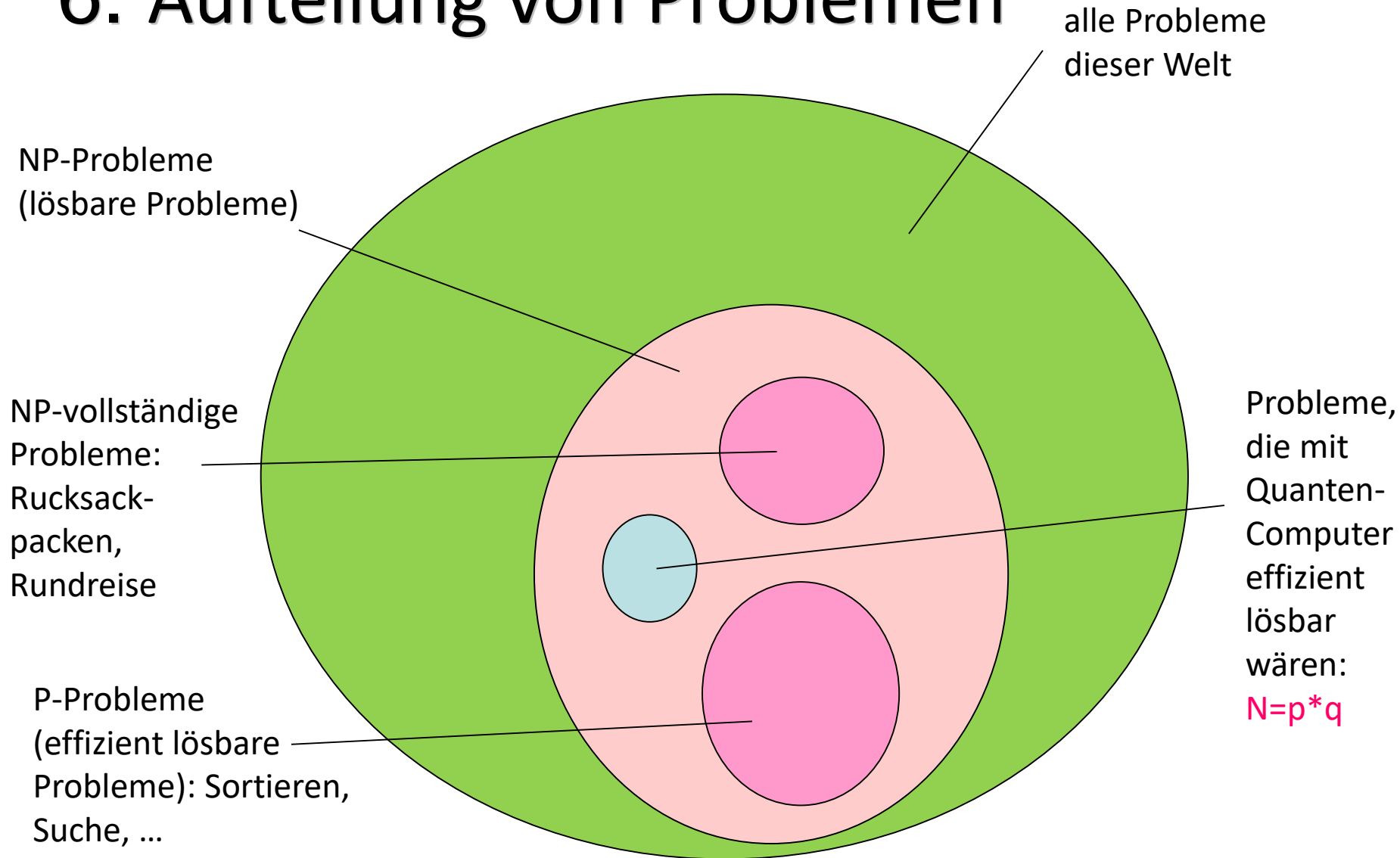
6. Aufteilung von Problemen



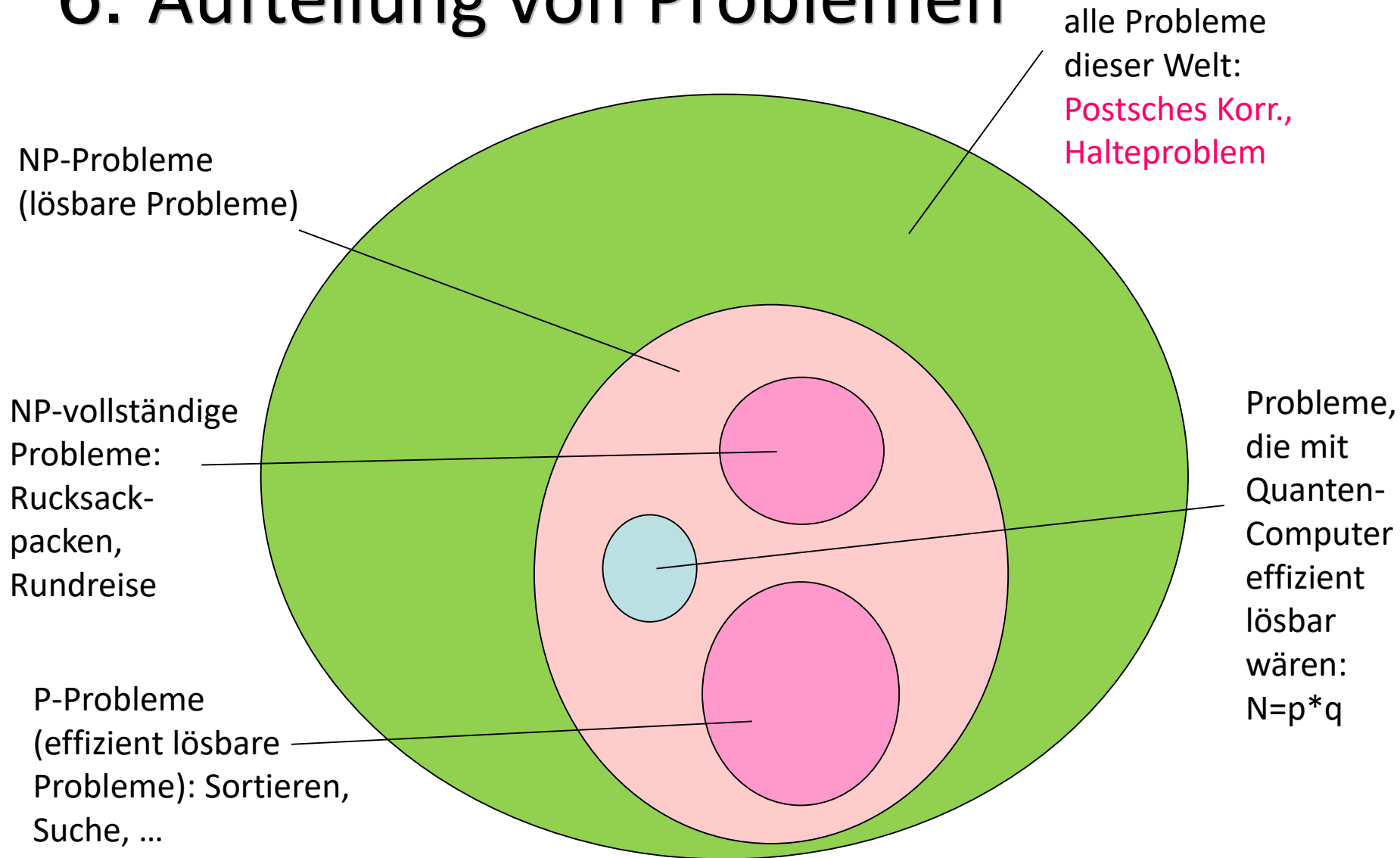
6. Aufteilung von Problemen



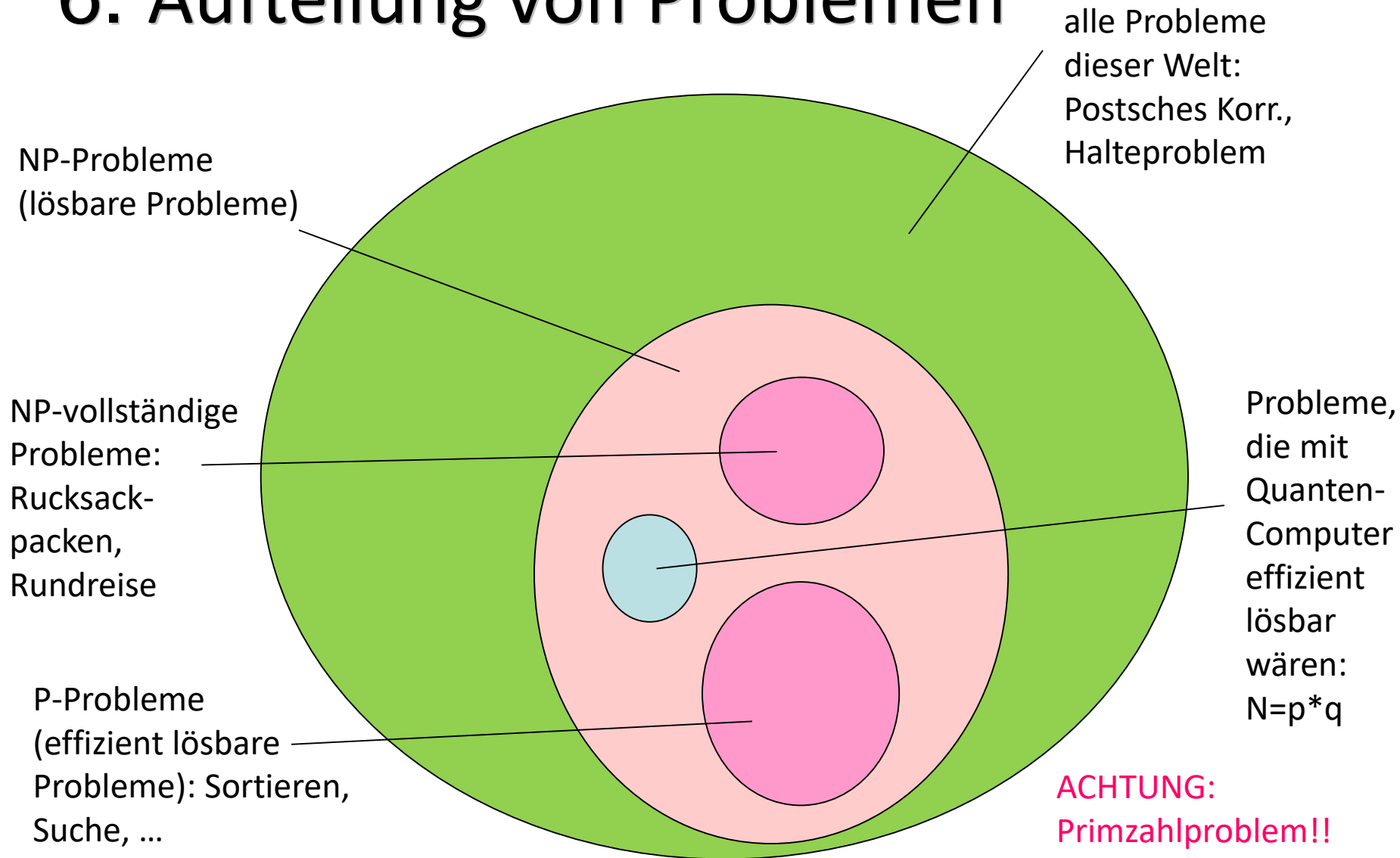
6. Aufteilung von Problemen



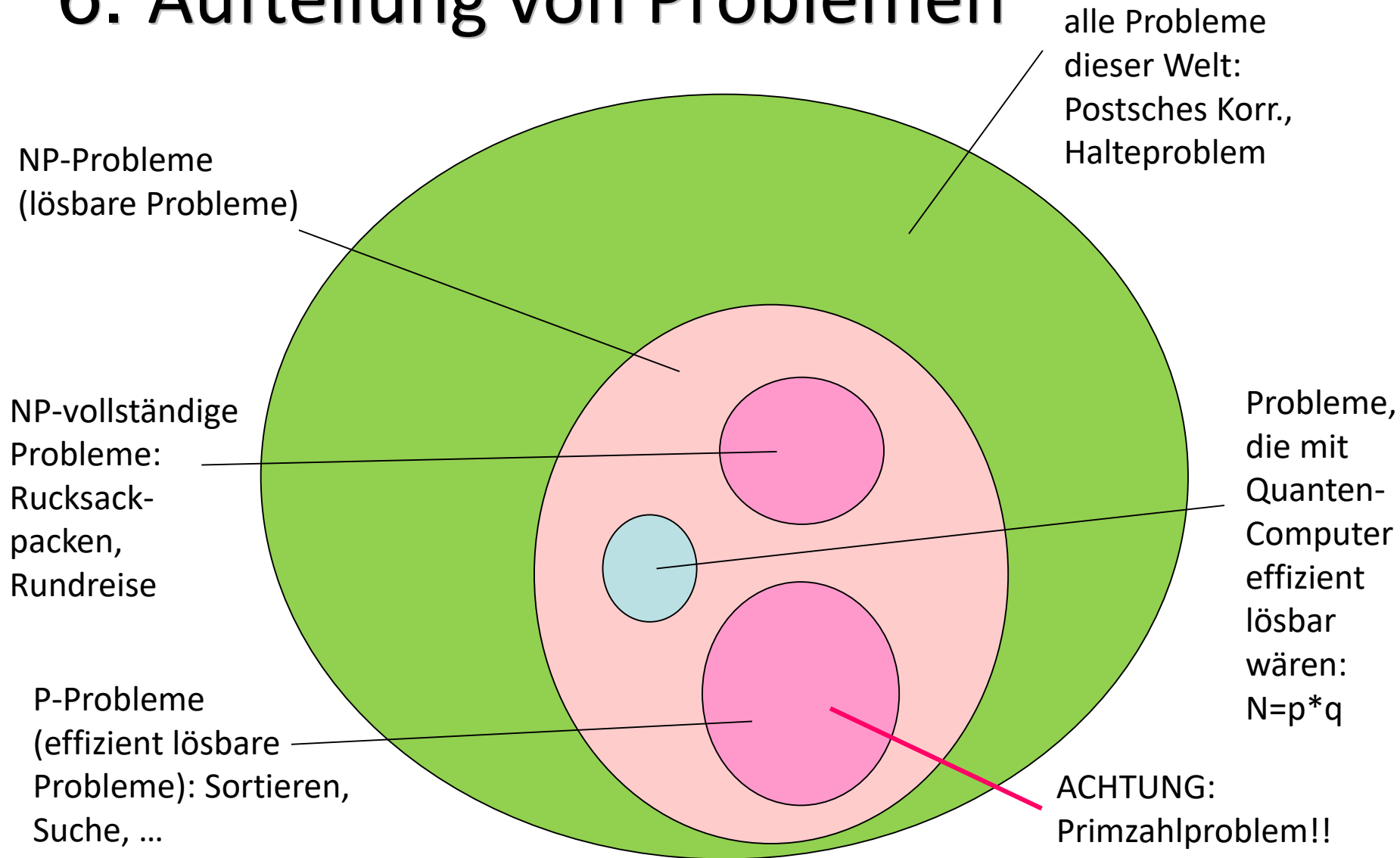
6. Aufteilung von Problemen



6. Aufteilung von Problemen



6. Aufteilung von Problemen



Ungelöstes Rätsel der theoretischen Informatik



Aus: Homer hoch 3 (https://www.google.com/search?client=firefox-b-d&q=Simpson+Homer+hoch+3&tbm=isch&sa=X&ved=2ahUKEwjBxNechs__AhUNxQIHhUnkBo4Q0pQJegQIDBAB&biw=1920&bih=955&dpr=1#imgrc=U2SJzQ9muyFv7M)



7. Berühmtes offenes Problem der Informatik

Es ist ein offenes Problem, ob sich die in dieser Vorlesung vorgestellten, nur mit exponentiellen Aufwand zu lösende Probleme nicht doch mit polynomialem Aufwand lösen lassen, also

$$NP = P ?$$

Achtung: Für das *Primzahlproblem* wurde 2002 ein Algorithmus mit polynomialer Laufzeit gefunden.

Das **P/NP** Problem gehört zu den **sieben** wichtigsten Problemen der Mathematik und Informatik. Für die Lösung des P/NP Problems kann man eine Prämie von **1.000.000 \$** gewinnen!

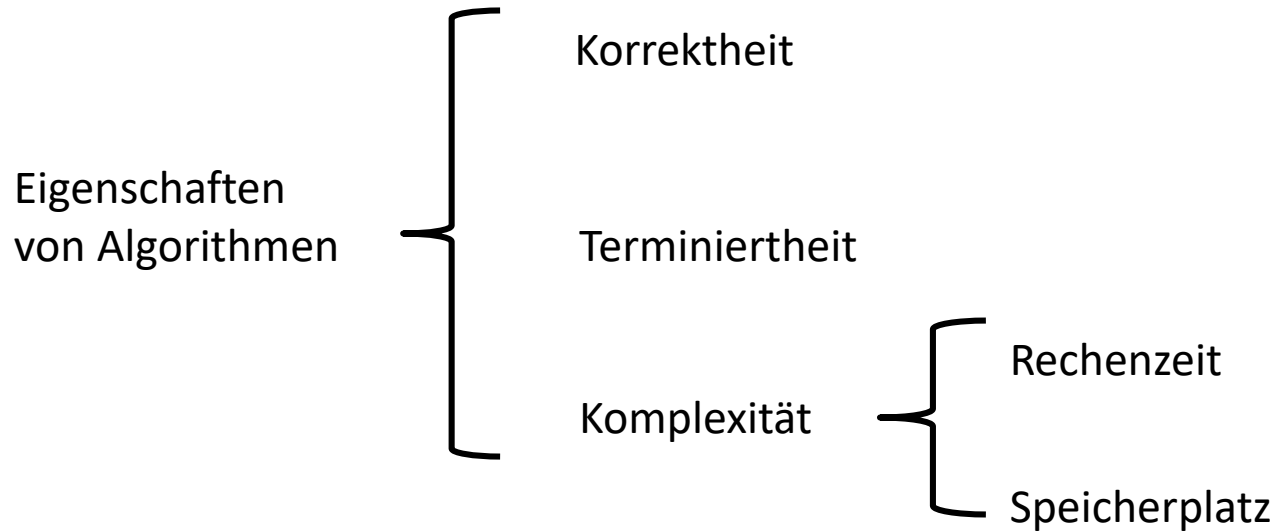
Siehe: Clay Mathematics Institute (<https://www.claymath.org/millennium-problems>)



Aufwand: Zusammenfassung

- Es gibt Probleme, die nicht von Rechnern gelöst werden können.
- Es gibt Probleme, die nur sehr aufwändig von Rechnern gelöst werden können.
- Aufwändig lösbare Probleme spielen eine große Rolle für die Kryptographie. Es gibt allerdings keinen Beweis, dass diese Probleme **nur** aufwändig lösbar sind. Es könnte sich herausstellen, dass es doch sehr schnelle Algorithmen gibt.

Aufwand: Einordnung



Literatur:

- Uwe Wegner: „Komplexitätstheorie“
- Jörg Rothe: „Complexity Theory and Cryptology“
- Simon Singh: „Homers letzter Satz“