Theoretische Informatik II

Einheit 6

Komplexitätstheorie

1. Grundkonzepte der Komplexitätsanalyse
2. Das $\mathcal{P}$–$\mathcal{NP}$ Problem
3. $\mathcal{NP}$-vollständige Probleme
4. Grenzen überwinden
5. Die Komplexitätsklassenhierarchie

Komplexitätstheorie

Was kann mit vertretbarem Aufwand gelöst werden?

- **Berechenbarkeitsanalyse alleine reicht nicht**
  - Klärt nur die Grundsatzfrage: berechenbar/entscheidbar oder nicht
  - Für praktische Lösbarkeit muß Berechnungsaufwand vertretbar sein
- **Typische Fragestellungen**
  - Wie schnell kann man Datensätze in großen Datenbanken finden?

    Bei 100 Milliarden Einträgen braucht mann im Schnitt 50 Milliarden Schritte

    .. aber nur 37 Schritte, wenn Sortierungsschlüssel vorhanden sind
  - Warum ist eine https-Verbindung praktisch nicht zu knacken?
    - Mit dem richtigen Schlüssel ist die Kommunikation sehr schnell
    - Aber Angriffe auf die Verschlüsselung brauchen mindestens 10^{20} Schritte
  - Kann man eine Rundreise durch alle Knoten eines großen Graphen in akzeptaber Zeit planen?
    - Bisher nicht möglich, schon ab 70 Knoten ist es so gut wie unlösbar
    - Aber wenn jemand einen Vorschlag macht, kann man diesen leicht überprüfen

Kann man den Aufwand möglichst genau analysieren?

Komplexität hängt von vielen Faktoren ab

- **Größe der Eingabe**
  - Sortieren von 100 Milliarden Einträgen dauert länger als 10000 Einträge
  - Bei kleinen Eingaben sind fast alle Lösungen eines Problems gleich gut
- **Berechnungsparadigma**
  - Deterministische sequentielle Algorithmen sind der Standard
  - Parallele, nichtdeterministische, probabilistische Algorithmen können oft erheblich effizienter sein
  - Echte parallele / nichtdeterministische Maschinen gibt es (bis jetzt) nicht
- **Berechnungsmodell (Programmiersprache / Hardware)**
  - Turingmaschinen sind ineffizienter als von-Neumann Architekturen
  - Imperative, funktionale, logische Sprachen sind unterschiedlich effizient
  - Unterschiede sind erheblich geringer als bei Berechnungsparadigmen
- **Art der betrachteten Ressource**
  - Zeitbedarf eines Algorithmus — Time
  - Speicherbedarf des Verfahrens (RAM, Harddisk) — Space
  - Netzzugriffe, Zugriff auf andere Medien, Stromverbrauch, ...

Konsequenz für sinnvolle Komplexitätsanalysen

- **Komplexität sollte Funktion der Eingabegröße sein**
  – Wie stark wächst der Ressourcenaufwand relativ zur Größe des Problems?
- **Entscheidend ist die Komplexität der Algorithmen**
  – Hardwaresteigerungen bringen erstaunlich wenig
  – Wenn Algorithmen schlecht sind, nützt die beste Hardware wenig
  Auch 100000 Cores heben den Unterschied zwischen 50 Milliarden und 37 nicht auf
- **Die Meßgröße der Komplexität muß objektiv sein**
  – Unabhängig von konkreter Hardware und Programmiersprache, Optimierungsfähigkeiten des Compilers oder Auswahl der Testdaten
- **Genaue Analysen sind weder praktikabel noch sinnvoll**
  – Zu mühsam bei nichttrivialen Algorithmen
  – Zu abhängig von Programmierdetails und Maschinenmodell

Komplexitätsmaße sollten von unwichtigen Details abstrahieren

Fragestellungen der Komplexitätstheorie

- **Wie analysiert man die Komplexität von Problemen?**
  - Formalismus zur Abschätzung oberer und unterer Schranken
  - Analysen auf der Basis abstrakter Algorithmen
- **Komplexitätsklassen**
  - Welche Probleme haben (in etwa) den gleichen Schwierigkeitsgrad?
  - Kann man effiziente Lösungen wiederverwenden? (Problemreduktion)
  - Was ist der Zusammenhang zwischen Platz- und Zeitbedarf?
- **Welche Probleme sind handhabbar?**
  - Wo liegt die Grenze?
  - Gibt es Probleme, die nicht effizient lösbar sind?
  - Lassen sich nichtdeterministische Lösungen effizient umsetzen?
  - Welche Verbesserung können unkonventionelle Ansätze erreichen? (approximierende, probabilistische Verfahren)

Theoretische Informatik II

Einheit 6.1

Grundkonzepte der Komplexitätsanalyse

1. Komplexitätsmodelle
2. Komplexitätsklassen und -hierarchien
3. Analyse konkreter Probleme
4. Handhabbarkeit

Abschätzung der Komplexität von Algorithmen

- **Formales Komplexitätsmodell basiert auf Turingmaschinen**
  - Bestimme Aufwand relativ zur Größe der Eingabe
    - $\boldsymbol{T_M(n)} = \max\{t_M(w) \mid |w|{=}n\}$ (worst-case)
    - $\boldsymbol{S_M(n)} = \max\{s_M(w) \mid |w|{=}n\}$ Einheit 5.1, Folie 7
  - Jedes andere (sequentielle) Maschinenmodell wäre genauso gut geeignet
  - Average-case Analysen würden zusätzlich statistische Modelle benötigen
- **Konkrete Analysen verwenden ein Einheitskostenmodell**
  - Vereinfachte (modellunabhängige) Zählung von Elementaroperationen
  - Operationen wie +, −, ∗, /, ... gelten (bei fester Zahlengröße) als ein Schritt
  - Konstanter Expansionsfaktor bei Übersetzung in Turingmaschinen
- **Wichtig ist asymptotisches Verhalten für große Eingaben**
  - Komplexität wird als Größenordnung relativ zur Eingabegröße angegeben (logarithmisch, linear, quadratisch, exponentiell, ...)
  - Analyse kann additive Konstanten und konstante Faktoren ignorieren (lassen sich durch Hardwaresteigerungen ausgleichen)

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ ("f wächst asymptotisch schneller als g")**
  - Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

    Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

- **(Größen-)Ordnung einer Funktion**
  - f als obere Schranke: $\mathcal{O}(f) = \{g{:}\mathbb{N}\to\mathbb{R}^+ \mid \exists c>0.\ g \leq_a c*f\}$
  - f als echte obere Schranke: $o(f) = \{g{:}\mathbb{N}\to\mathbb{R}^+ \mid \forall c>0.\ g <_a c*f\}$
  - f als untere Schranke: $\Omega(f) = \{g{:}\mathbb{N}\to\mathbb{R}^+ \mid \exists c>0.\ c*f \leq_a g\}$
  - f als exakte Schranke: $\Theta(f) = \{g{:}\mathbb{N}\to\mathbb{R}^+ \mid \exists c, c'>0.\ c*f \leq_a g \leq_a c'*f\}$

  Schreibweisen auch: $\mathcal{O}(1) \mathrel{\hat{=}} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n^2)\ldots$

- **Beispiele für Ordnung konkreter Funktionen**
  - Konstante Funktion: $g_1(n)=k$ für alle n — $g_1 \in \mathcal{O}(1)$
  - Polynome: $g_2(n)=5n^3+2n^2+47$ — $g_2 \in \mathcal{O}(n^3)$

    Allgemein: $c_0+c_1*n+..+c_m*n^m \in \mathcal{O}(n^m)$
  - Logarithmenfunktionen: $g_3(n)=log_{10}\,n$ — $g_3 \in \mathcal{O}(log_2 n)$
  - Fakultätsfunktion: $g_4(n)=n!=1*2*..*n$ — $g_4 \in \mathcal{O}(n^n)$

Komplexität von Maschinen und Sprachen

- **Komplexität einer Maschine M**
  - M hat Zeitkomplexität $\mathcal{O}(f)$, falls $T_M \in \mathcal{O}(f)$
  - M hat Platzkomplexität $\mathcal{O}(f)$, falls $S_M \in \mathcal{O}(f)$
- **Komplexität einer Sprache (“eines Problems”) L**
  - L hat (deterministische) Zeitkomplexität $\mathcal{O}(f)$, falls es eine DTM M mit $T_M \in \mathcal{O}(f)$ und $L = L(M)$ gibt
  - L hat nichtdeterministische Zeitkomplexität $\mathcal{O}(f)$, falls es eine NTM M mit $T_M \in \mathcal{O}(f)$ und $L = L(M)$ gibt
  - L hat (nicht-)deterministische Platzkomplexität $\mathcal{O}(f)$, falls $L = L(M)$ für eine DTM (bzw. NTM) M mit $S_M \in \mathcal{O}(f)$
  - Statt “Sprache L” wird oft auch “Problem P” oder “Menge M” benutzt
- **Komplexitätsklassen**

  TIME(f) = $\{L \mid L$ hat deterministische Zeitkomplexität $\mathcal{O}(f)\}$
  NTIME(f) = $\{L \mid L$ hat nichtdeterministische Zeitkomplexität $\mathcal{O}(f)\}$
  SPACE(f) = $\{L \mid L$ hat deterministische Platzkomplexität $\mathcal{O}(f)\}$
  NSPACE(f) = $\{L \mid L$ hat nichtdeterministische Platzkomplexität $\mathcal{O}(f)\}$

Wichtige Komplexitätsklassen

- $\mathit{LOGSPACE} = \bigcup_k \mathit{SPACE}(\log_2^k n)$

  Klasse der mit logarithmischem (Berechnungs-)Platz lösbaren Probleme
- $\mathcal{P} = \bigcup_k \mathit{TIME}(n^k)$

  Klasse der in polynomieller Zeit lösbaren Probleme
- $\mathcal{NP} = \bigcup_k \mathit{NTIME}(n^k)$

  Nichtdeterministisch in polynomieller Zeit lösbare Probleme
- $\mathit{PSPACE} = \bigcup_k \mathit{SPACE}(n^k)$

  Klasse der in polynomieller Platz lösbaren Probleme
- $\mathit{NPSPACE} = \bigcup_k \mathit{NSPACE}(n^k)$

  Nichtdeterministisch in polynomieller Platz lösbare Probleme
- $\mathit{EXPTIME} = \bigcup_k \mathit{TIME}(2^(n^k))$

  Klasse der in exponentieller Zeit lösbaren Probleme

Wie stehen diese Klassen zueinander?

Zeit- vs. Platzkomplexität

- $DTIME(f) \subseteq NTIME(f)$
  - NTMs sind mindestens genauso effizient wie DTMs
- $DSPACE(f) \subseteq NSPACE(f)$
  - Gleiches Argument
- $NTIME(f) \subseteq DSPACE(f)$
  - In Zeit $\mathcal{O}(f)$ kann man maximal $\mathcal{O}(f)$ Speicherzellen verwenden
- $NSPACE(f) \subseteq DTIME(2^f)$
  - Eine terminierende Maschine, die maximal $f(n)$ Bandzellen aufsucht, terminiert nach maximal $|\Gamma|^{f(n)} * |Q| * f(n)$ Schritten
    (Zahl der Konfigurationen = Bandinhalte $*$ Zustände $*$ Kopfpositionen)
- **Hierarchie der wichtigsten Komplexitätsklassen**
  - $LOGSPACE \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq PSPACE \subseteq EXPTIME \subseteq \ldots$
  - Es wird vermutet, daß alle Inklusionen echt sind

Komplexitätsanalyse konkreter Probleme

Wie effizient ist ein Problem lösbar?

- **Obere Schranke: Analyse von Lösungsalgorithmen**
  - Problem ist maximal so schwer wie der beste bekannte Algorithmus
  - Mathematische Analyse spezifischer Algorithmen ist aufwendig
  - Häufig reicht eine Abschätzung auf Basis der Algorithmenidee
- **Untere Schranke: Mindestzahl notwendiger Operationen**
  - Zeigt an, ob ein Lösungsalgorithmus optimal ist

    Weitere Optimierungen können sich auf Details konzentrieren
  - Analyse ausfwendig, da selten klar ist was wirklich notwendig ist
- **Oft gibt es Lücken zwischen oberer und unterer Schranke**
  - Es ist unklar, ob der beste bekannte Algorithmus optimal ist
  - Häufig: Algorithmus muß gezielt nach Lösung des Problems suchen

    und Suche nach Lösung ist aufwendiger als ihre Konstruktion

Analyse konkreter Algorithmen: Binärsuche

Liste L *muß sortiert sein*

```
function search_bin(x,L) ≡
   let function search_b(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_b(x,L,left,mid-1)
               elseif x>L[mid] then search_b(x,L,mid+1,right)
               else return true
            fi;
   return search_b(x,L,1,length(L))
```

- Anzahl von Operationen pro Aufruf von $\texttt{search}_b$ ist eine Konstante k
  Anzahl der Aufrufe von $\texttt{search}_b$ ist entscheidend für Komplexität
- Abstand von `left` und `right` halbiert sich pro Aufruf (mit Abrundung)
- Abstand zu Beginn ist $n{-}1$ (n ist die Größe der Liste L)
- $\texttt{search}_b$ terminiert bei Erfolg oder wenn Abstand Null ist
  Lösung der Gleichung $time(n) = k{+}time(\lfloor n/2 \rfloor)$ ist $time(n) = k{*}\log_2 n$

$$\Downarrow$$

Binäre Suche ist in $\mathcal{O}(\log_2 n)$

Komplexitätsabschätzung: Sortierverfahren

Abstrakte Algorithmenskizze reicht für Laufzeitanalyse

- **Identifiziere Läufe (geordnete Teilfolgen) in der Liste**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Verschmelzen liegt in $\mathcal{O}(n)$ (Liste wird jeweils komplett durchlaufen)
  – Verschmelzen halbiert Anzahl der Läufe (je zwei werden gemischt)

- **Wiederhole bis Liste geordnet ist**

7	8	9	1	2	5	6
1	2	5	6	7	8	9

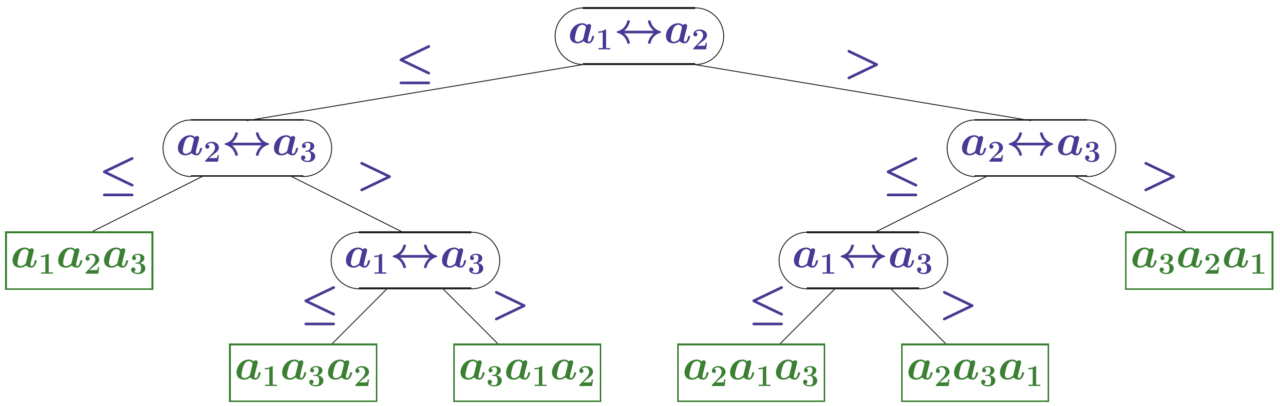
  – Am Ende ist ein einziger Lauf übrig, d.h. die Liste ist sortiert
  – Man braucht maximal $\log_2 n$ Verschmelzungszyklen

Sortieren durch Verschmelzen ist in $\mathcal{O}(n * \log_2 n)$

Analyse unteren Schranken: Sortieren

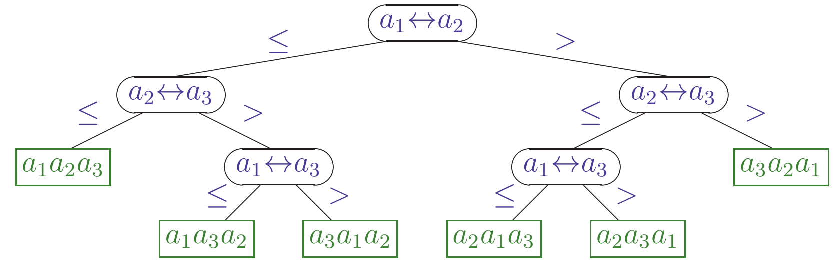
Geht sortieren schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**



  - Innere Knoten entsprechen den durchgeführten Vergleichen
  - Kanten markiert mit Vergleichergebnis ($\leq$,$>$)
  - Blätter sind resultierende Anordnung der Elemente

Analyse unteren Schranken: Sortieren (II)



- **Jeder Algorithmus entspricht einem Entscheidungsbaum**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter
  - Ein binärer Baum der Tiefe k hat maximal 2^k Blätter
  - Jeder Entscheidungsbaum hat mindestens Tiefe $\log_2 n!$
  - $\log_2 n! = \log_2(\Pi_{i=1}^n i) = \Sigma_{i=1}^n \log_2 i \geq \Sigma_{i=n/2}^n \log_2(n/2) = n/2*(\log_2 n - 1)$

Sortieren ist in $\Omega(n * \log_2 n)$

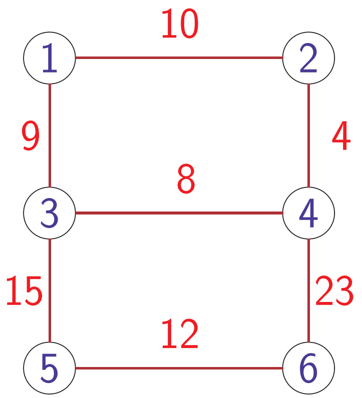
Komplexität häufiger Problemstellungen

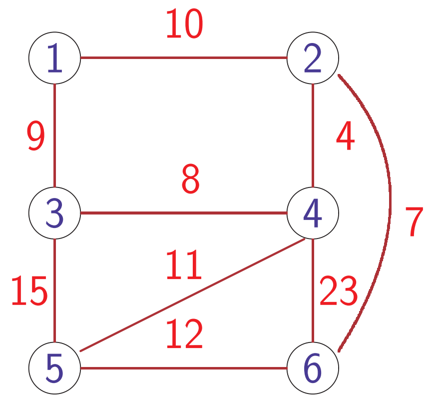
- **Arithmetik auf großen (k-stelligen) Zahlen**
  - Addition: Einstellig von rechts nach links mit Übertrag $\mathcal{O}(k)$
  - Multiplikation: Jede Stelle muß mit jeder multipliziert werden $\mathcal{O}(k^2)$
  - Division: Schriftliche Division von links nach rechts $\mathcal{O}(k^2)$
- **Matrixmultiplikation $n{\times}n$-Matrizen** $\mathcal{O}(n^3)$
- **Berechnung von $n!$** $\Theta(n^2{*}(\log_2 n)^2)$
  - Obergrenze: n-fache Multiplikation von n und $n!$: $n * \log_2 n * \log_2(n^n)$
  - Untergrenze: $n/2$-fach $n/2{*}(n/2)!$: $n/2 * \log_2(n/2) * n/4{*}(\log_2 n - 2)$
- **Primzahltest bei k-stelligen Zahlen** $\mathcal{O}(k^{12})$
  - AKS Algorithmus auf Basis tiefer mathematischer Einsichten (2002)
  - Alle früheren Verfahren waren exponentiell
  - Alle bekannten Faktorisierungsverfahren sind exponentiell
  - Ergebnis gut für offene kryptographische Systeme (wähle $k > 500$)

Das Problem des Handlungsreisenden

Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B. Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

- **Formulierung als Graphenproblem** $\mapsto$ Handout Graphentheorie
  - Ein Hamiltonscher Kreis im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.
  - **TSP**: Finde einen Hamiltonschen Kreis mit maximalen Kosten B





Nur eine Rundreise: [1,3,5,6,4,2]

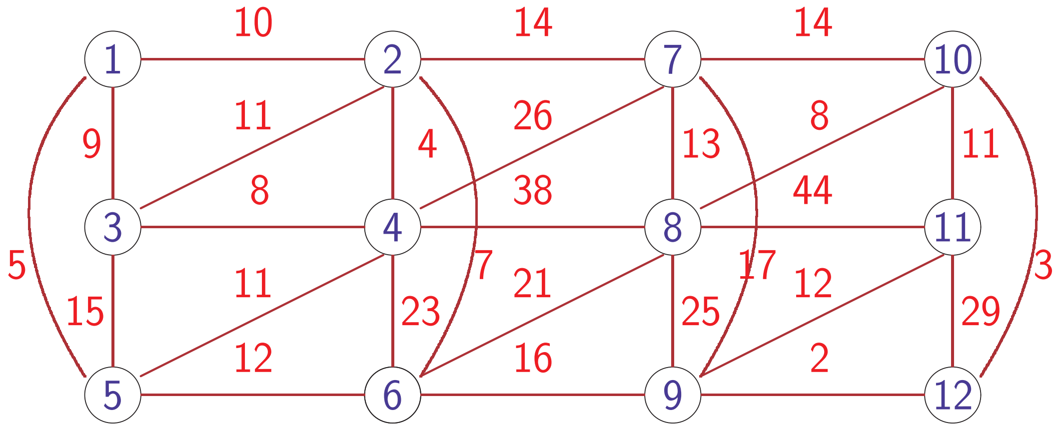
Kosten: 73

Mehrere Rundreisen möglich

Kosten von [1,2,6,5,4,3] sind 57

Was ist die Komplexität des TSP-Problems?

- **Graphen können sehr komplex sein**



- **Einziger allgemeiner Algorithmus ist "Generate & Test"**
  - Test ist linear, aber es gibt exponentiell viele Möglichkeiten
  - Es ist kein besserer Lösungsalgorithmus bekannt

Ist das TSP noch effizient handhabbar?

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor										
Größe n	10	20	30	40	50	60	…	1000	1 Mio	100 Mrd
Wachstum										
$\log_2 n$	1ns					2ns		3ns	6ns	12ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs	30s
$n{*}\log_2 n$	10ns	25ns	45ns	65ns	85ns	110ns		3μs	6ms	18m
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s	950y
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y	
2^n	300ns	300μs	300ms	300s	83.3h	9.5y				
3^n	17.8μs	1.1s	17.3h	116y	2.5Mrd y					

**Wieviel mehr kann man in der gleichen Zeit berechnen,
wenn Computer um den Faktor 1000 schneller werden?**

	$\log_2$n	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300}-fach	1000-fach	31-fach	10-fach	plus 10	plus 6

Konsequenz für Handhabbarkeit von Problemen

- **Investitionen in Hardware helfen nur beschränkt**
  - Wenn ein Algorithmus nicht in polynomieller Zeit läuft, nutzt auch ein Hochleistungsrechner nichts
  - Es lohnt sich, in die Verbesserung von Algorithmen zu investieren
    Geeignetere Datenstrukturen, bessere Suchstrukturen,
    Grundsätzlich neue Zugänge (out-of-the-box Denken)
  - Lohnenswert sind Verbesserungen um mindestens eine Größenordnung
    z.B. von $\mathcal{O}(n^3)$ auf $\mathcal{O}(n^2)$ auf $\mathcal{O}(n{*}\log n)$ oder von $\mathcal{O}(2^n)$ auf $\mathcal{O}(2^{\sqrt{n}})$
- **Polynomielle Lösbarkeit ist entscheidend**
  - Ein Problem gilt als handhabbar, wenn es in $\mathcal{P}$ liegt
  - Unterschiede innerhalb polynomieller Komplexität sind tolerierbar aber durchaus relevant für konkrete Implementierungen
  - Exponentieller Aufwand ist für die Praxis unakzeptabel