051013 VO Theoretische Informatik

Komplexitätstheorie

Ekaterina Fokina

Letztes Mal

In der letzten VO haben wir Folgendes behandelt:

- Berechenbarkeit
  - Berechenbarkeit
  - Entscheidbarkeit
  - Halteproblem
  - Unentscheidbare Problem
  - Reduktionen

Damit kann man

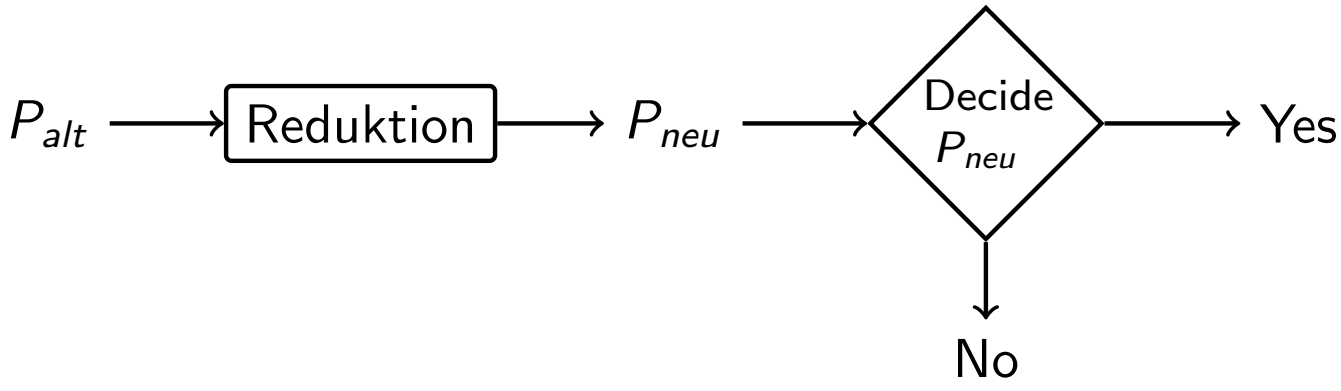
- nicht berechenbare Probleme und
- (theoretisch) berechenbare Probleme unterscheiden.

Berechenbarkeitstheorie

- **Berechenbarkeit** von Funktionen mit Hilfe einer Turing Maschine.
- **Entscheidbarkeit** von Problemen und Mengen = Berechenbarkeit von charakteristischen Funktionen.
- Unentscheidbares Problem: das spezielle Halteproblem

$$K = \{w \in \{0,1\}^* \mid M_w \text{ angewendet auf } w \text{ hält}\}.$$

- Reduktionen als Ansatz zum Beweisen der Unentscheidbarkeit:



- Eine Liste von unentscheidbaren Problemen.

Komplexität von berechenbaren Problemen

Nicht jedes theoretisch berechenbare Problem ist auch tatsächlich praktisch lösbar (aufgrund des hohen Berechnungsaufwands).

Verschiedene berechenbare Probleme haben ganz unterschiedliche Anforderungen an

- Rechenzeit
- Speicher
- ...

Wir wollen die berechenbaren Probleme anhand dieser Anforderungen weiter unterscheiden.

- **Komplexitätstheorie**
  - Komplexitätsklassen **P**, **NP**, **NPC**
  - Polynomialzeitreduktion
  - Beispiele für NP-vollständige Probleme
  - Praktische Implikationen von Komplexität

Zeitkomplexität

Prinzip: Aufwand wird als Funktion der Inputgröße angegeben (viel aussagekräftiger als Zeitmessung in sec).

Algorithmus mit polynomialem Aufwand. Für Inputs der Länge n beträgt die max. Laufzeit $O(n^k)$.

Beispiel

- **Minimum / Maximum – Suche**: n Elemente, daher $(n-1)$ Vergleiche, daher Aufwand $O(n)$.
- **Sortieren durch Minimum / Maximum – Suche**: n Elemente, daher $(n-1)+(n-2)+\ldots+1$ Vergleiche, i.e. $\sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} = O(n^2)$.

Aufwandabschätzung: ein Beispiel

Sortieren durch Minimum-Suche:

i=1:	9	6	(2)	5	7	*4 Vergl.*
i=2:	2	6	9	(5)	7	*3 Vergl.*
i=3:	2	5	9	(6)	7	*2 Vergl.*
i=4:	2	5	6	9	(7)	*1 Vergl.*
	2	5	6	7	9	*fertig*

- Durchgeführte Basisoperation: Vergleich zweier Zahlen und gegebenenfalls ein Vertausch
- Anzahl der Basisoperationen:
  - bei 5 Elementen wie im Beispiel: 4+3+2+1=10
  - allgemein bei n Elementen: $\sum_{i=1}^{n-1}(n-i)$
  - Aufwand durch Inputgröße n beschreibbar

Zeitkomplexität

Definition

Eine Turingmaschine M besitzt eine Zeitkomplexität $T(n)$, wenn M bei jeder Eingabe der Länge n nach maximal $T(n)$ Schritten hält.

Wir betrachten also asymptotische worst case Zeitkomplexität.

Wichtige Klasse von Funktionen: $T(n)$ ist polynomial

$T(n)$ ist polynomial wenn $T(n) = \sum_{i=0}^{k} a_i n^i$ für ein $k < \infty$

Intuition: Probleme, die in polynomialer Zeit von deterministischen Turingmaschinen berechnet werden können, können auch auf einem typischen Computer in polynomialer Zeit berechnet werden.

Komplexitätsklassen

2 Komplexitätsklassen sind von besonderer Bedeutung[1]:

- **Klasse P (polynomial)**: Probleme, die mit einer deterministischen Turingmaschine in polynomialer Zeit gelöst werden können.
- **Klasse NP (nichtdeterministisch polynomial)**: Probleme, die mit einer nichtdeterministischen Turingmaschine in polynomialer Zeit gelöst werden können.

Beachte: Zu jeder nichtdet. TM gibt es auch eine äquivalente det. TM. Die Zeitkomplexität der det. TM kann jedoch exponentiell werden.

[1] Es gibt aber bei weitem mehr: `https://complexityzoo.uwaterloo.ca`.

Alternative Charakterisierung von **NP**

Zwei Phasen:

1. **Generate/Guess:** Es werden mögliche Lösungsvorschläge für ein Problem generiert.
2. **Check:** Ein Problem ist in der Klasse **NP**, wenn es in polynomialer Zeit möglich ist, einen Lösungsvorschlag zu verifizieren.

Verwendbar für Beweis, dass ein Problem aus **NP** ist

- SAT-Problem: Eine Variablen-Belegung kann in polynomialer Zeit verifiziert werden.

Komplexitätsklassen und Praktische Lösbarkeit

- Probleme mit Komplexität **P** können i.A. praktisch mit vertretbarem Aufwand gelöst werden, und werden auch als "traktabel" (engl. tractable) bezeichnet.
- Probleme mit Komplexität **NP** werden i.A. als praktisch nicht mehr lösbare gesehen (oder nur für kleine Instanzen).

Vorsicht: Es ist immer noch unbekannt, ob $\mathbf{P} \neq \mathbf{NP}$, mehr dazu später in der VO.

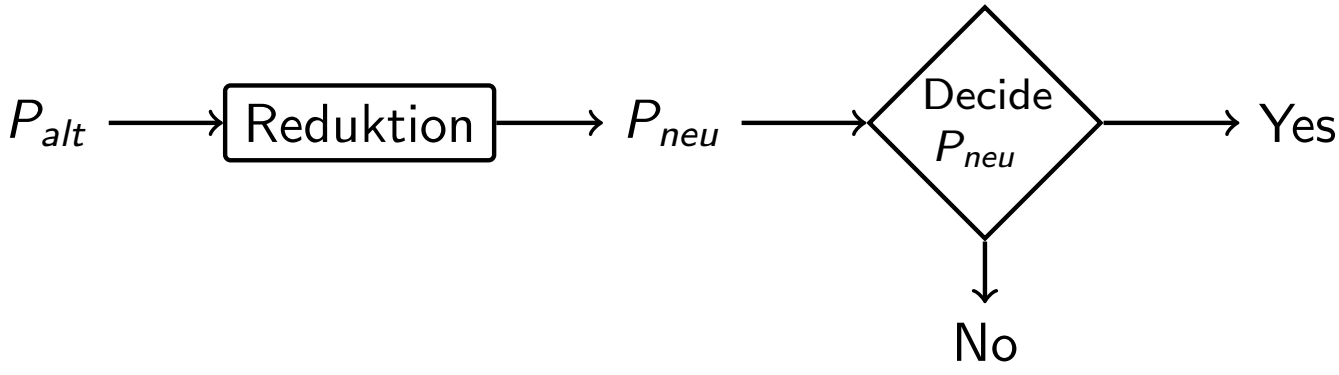
Es ist in der Praxis von großer Wichtigkeit zeigen zu können, ob ein Problem in der Klasse **P** liegt.

- Falls in **P**: polynomialen Algorithmus angeben
- Falls nicht in **P**: Beweis mit Hilfe einer polynomialen Reduktion.

Polynomiale Reduktion

Kann man ein Problem P_{alt}, das nicht in **P** ist, auf ein P_{neu} polynomial reduzieren, dann ist auch P_{neu} nicht in **P**.

Polynomiale Reduktion und Widerspruch



Eine **polynomiale Reduktion** ist eine Funktion $R(.)$.

1. Jede Instanz I von P_{alt} wird auf eine Instanz $R(I)$ von P_{neu} abgebildet
2. I yes/no Instanz von $P_{alt} \Leftrightarrow R(I)$ yes/no Instanz von P_{neu}
3. $R(.)$ ist in polynomial Zeit berechenbar

Gebe es einen polynomialen Algorithmus für P_{neu} dann auch für P_{alt}.

NP-vollständige Probleme **NPC** (NP-complete)

- Kann man ein Problem A polynomial auf ein Problem B reduzieren, schreiben wir $A \preceq B$ und sagen B ist schwerer als A.
- Für gleich schwere Probleme gilt dann $A \preceq B$ und $B \preceq A$.

Definition

Ein Problem L heißt **NP**-vollständig, falls

1. $L \in \mathbf{NP}$, und
2. Für jedes Problem $L' \in \mathbf{NP}$ gilt $L' \preceq L$, d.h. es existiert eine polynomiale Reduktion von L' nach L.

Beispiel **NPC**: **SAT-Problem**

SAT-Problem: Erfüllbarkeit aussagenlogischer Ausdrücke

- SAT für "formula satisfiability"
- Frage: ist ein gegebener aussagenlogischer Ausdruck erfüllbar oder nicht.
- Wichtiges Problem: erstes Problem von dem gezeigt wurde, dass es **NP**-vollständig ist.

Ein aussagenlogischer Ausdruck besteht aus:

1. n Booleschen Variablen $v_1, v_2, \ldots, v_n$.
2. Boolesche Operatoren: $\wedge$ (Konjunktion), $\vee$ (Disjunktion), $\neg$ (Negation), $\rightarrow$ (Implikation) und $\leftrightarrow$ (Äquivalenz)
3. Klammern

Bespiele:

$$F_1 : \neg(B \rightarrow C) \wedge (D \vee (A \rightarrow \neg B))$$

$$F_2 : \neg((A \wedge B) \rightarrow C) \wedge (C \vee (A \rightarrow \neg B))$$

Beispiel **NPC**: **SAT-Problem** (2)

SAT-Problem:

- **Gegeben:** Aussagenlogischer Ausdruck F mit n Variablen
- **Gesucht:** Ist F erfüllbar oder nicht.

Algorithmus

- Bilde die Menge aller möglicher Variablenbelegungen (n-Tupel) und teste auf Erfüllbarkeit.

```
function SAT-SOLVER (IN: formula F with n variables)
    // build all 2^n possible assignments
    A := {(true_1, true_2, ..., true_n), (false_1, true_2, ..., true_n), ...,
          (false_1, false_2, ..., false_n)};
    for each tuple T in A do
       if F is satisfied by T
          return true;
    end for
    return false;
end
```

Beispiel **NPC**: **SAT-Problem** (3)

Aufwand von SAT-SOLVER:

- Dieser einfache Algorithmus hat eine worst-case Laufzeit von $O(2^n)$ und hat somit **exponentielles** Wachstum.
- Beispiel: 100 Variablen: $2^{100} \approx 10^{30}$.
- 1 TB $= 10^{12}$ Byte, d.h., Menge A ist nicht speicherbar unabhängig von der Codierung der n-Tupel, d.h., n-Tupel müssen in der for-Schleife generiert werden.
  Anm.: Weltweite Speichervolumina werden üblicherweise in Exabyte (10^{18}) angegeben!
- Bei Nichterfüllbarkeit ist eine Terminierung nicht absehbar.

Lösung der Beispiele:

- F_1 ist erfüllbar mit (false,true,false)
- F_2 ist nicht erfüllbar

NP-vollständige Probleme **NPC** (NP-complete)

Für **NP**-vollständige Probleme gilt:

- Einerseits: bis dato wurden keine Algorithmen mit polynomialem Aufwand gefunden.
- Andererseits: bis dato konnte nicht ausgeschlossen werden, dass Algorithmen mit polynomialem Aufwand existieren könnten.

Per Definition können alle **NP**-vollständige Probleme gegenseitig aufeinander polynomial reduziert werden.

$\hookrightarrow$ Entweder können alle in **P** gelöst werden oder keines.

Methode

Für P_{neu}, zeigt man das P_{neu} mind. so schwer ist wie **NPC**.
$\hookrightarrow$ Dazu reduziert man ein $P_{alt} \in$ **NPC** polynomial auf P_{neu}.
Wenn noch zusätzlich $P_{neu} \in$ **NP**, dann $P_{neu} \in$ **NPC**.

Polynomiale Reduktion - Beispiel

SAT: Testen ob eine aussagenlogische Formel erfüllbar ist.

CNFSAT: Testen ob eine aussagenlogische Formel in KNF erfüllbar ist.

Wir wollen aus $SAT \in$ **NPC** auf $CNFSAT \in$ **NPC** schließen.

1.Ansatz: Wir kennen einen Algorithmus der jede aussagenlogische Formel in eine semantisch äquivalente in KNF umwandelt.

- Bedingung 1 & 2 sind also erfüllt ✓
- aber der Algorithmus ist im worst case exponentiell. 🗲

2.Ansatz: Es gibt einen polynomialen Algorithmus der jede Formel in eine erfüllbarkeitsäquivalente in KNF umwandelt, d.h. die Formeln sind entweder beide erfüllbar oder beide unerfüllbar.

- Bedingung 1 - 3 sind erfüllt ✓

Zusätzlich müssen wir noch zeigen, dass $CNFSAT \in$ **NP** ✓

Ähnliche Probleme unterschiedl. Komplexität

1. **Shortest-Path-Problem vs. Longest-Path-Problem.**
   - Kürzester Pfad zwischen 2 Konten eines Graphen ist in polynomieller Zeit lösbar (siehe Dijkstra's Algorithmus).
   - Längster einfacher (d.h., ohne Schleifen) Pfad in einem Graphen ist ein **NP**-vollständiges Problem.
2. **Euler-Kreis vs. Hamiltonscher Kreis**.
   - Euler-Kreis: Ein Pfad durch einen Graphen G, der alle Kanten genau einmal entält und wieder zum Anfangspunkt zurückkehrt.
     Polynomielles Problem.
   - Hamiltonscher Kreis: Ein Pfad durch einen Graphen G, der jeden Knoten genau einmal besucht und wider zum Anfangspunkt zurückkehrt.
     NP-vollständiges Problem.

Weitere **NP**-vollständige Probleme

1. **Traveling Salesman Problem.**
   Gegeben ist ein Graph mit Knoten, die Städten entsprechen, und Kanten, welche die Kosten angeben, um von einer Stadt in die andere zu gelangen. Gesucht ist ein Weg mit minimalen Kosten, der alle Knoten besucht und zum Ausgangspunkt zurückkehrt.
2. **Subset-Sum Problem.**
   - **Gegeben:** Menge natürlicher Zahlen $S \subseteq \mathbb{N}$. Zahl $t \in \mathbb{N}$.
   - **Gesucht:** Gibt es eine Teilmenge $S' \subseteq S$ deren Elemente aufsummiert t ergeben.

Beziehung **P**, **NP**, **NPC**

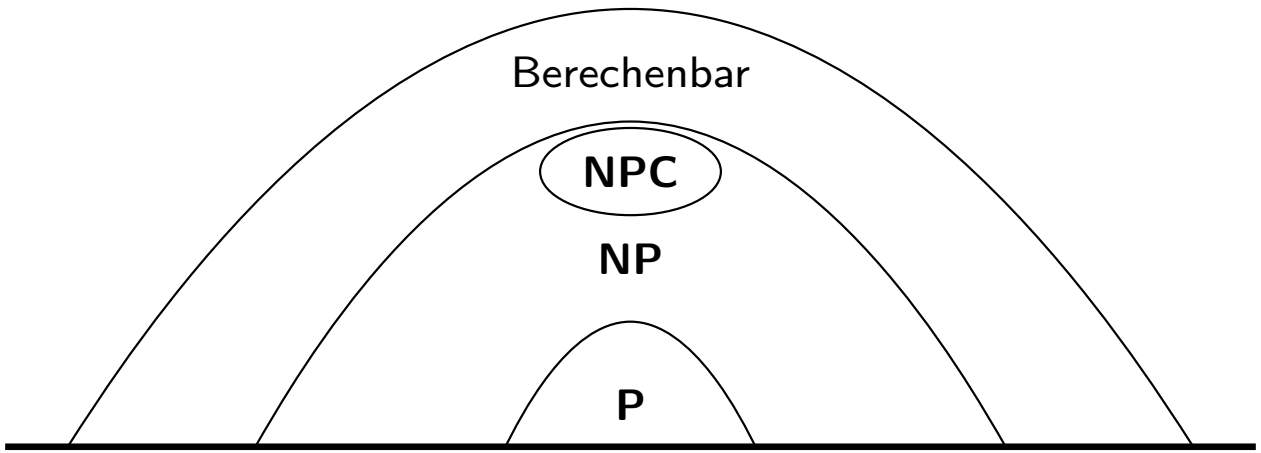
Wir wissen, dass

- $\mathbf{P} \subseteq \mathbf{NP}$: jedes polynomiale Problem ist auch nichtdeterministisch polynomial lösbar.
- $\mathbf{NPC} \subseteq \mathbf{NP}$: jedes **NP**-vollständige Problem ist in **NP**

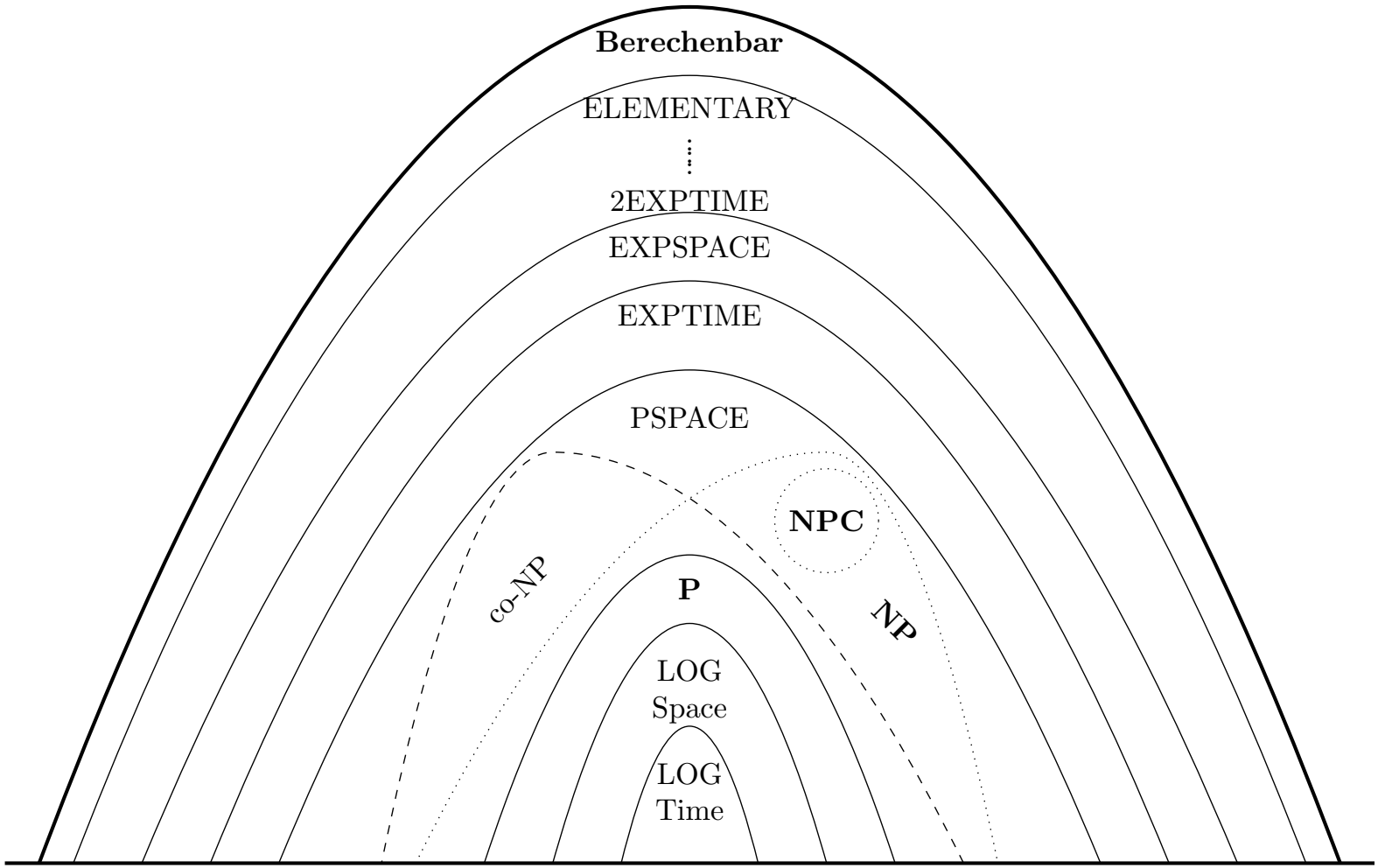
P-NP Problem

Ob $\mathbf{P} \neq \mathbf{NP}$ (oder $\mathbf{P} = \mathbf{NP}$) ist eines der wichtigsten offenen Probleme der theoretischen Informatik.

Die starke Vermutung ist, dass $\mathbf{P} \neq \mathbf{NP}$ und daher $\mathbf{P} \cap \mathbf{NPC} = \emptyset$.



Mehr Komplexitätsklassen [2]



[2] Bild basiert auf Sebastian Sardina: `http://www.texample.net/tikz/examples/complexity-classes/`

Optimierungsprobleme und Entscheidungsprobleme

- Theorie zur **NP**-Vollständigkeit basiert auf Entscheidungsproblemen.
- **Shortest-path. Optimierungsproblem**
  - **Gegeben:** u, v.
  - **Gesucht:** Pfad von u nach v mit min. Anzahl der Kanten k.
- Ein Optimierungsproblem kann in ein Entscheidungsproblem umgewandelt werden.
- **Path. Entscheidungsproblem**
  - **Gegeben:** u, v, k.
  - **Frage:** Existiert ein Pfad von u nach v mit max. k Kanten?

Lösung **NP**-vollständiger Probleme

Approximationsverfahren.

- Viele **NP**-vollständige Probleme haben praktische Bedeutung und Lösungen sind gesucht.
- Optimale Lösung: **NP**-vollständig und nicht möglich
- Gesucht: Gute Lösung (nahe dem Optimum) in polynomialer Zeit

- **Approximative Algorithmen:**
  Es werden Algorithmen entwickelt, die mit polynomialem Aufwand z.B. unter Verwendung von Heuristiken (suboptimale) Lösungen finden.

Optimale vs. Näherungslösung (1)

Bereits bekannt:
Subset-Sum Problem. (Entscheidungsproblem, **NP**-vollständig)

- Geg.: Menge $S \subseteq \mathbb{N}$ mit $|S| = n$ und Zahl $t \in \mathbb{N}$.
- Frage: Gibt es eine Teilmenge $S' \subseteq S$ deren Elemente aufsummiert t ergeben?

Neu:
Optimierungsproblem zu diesem Entscheidungsproblem:
Gesucht ist jene Teilmenge S' mit max. Summe $\leq t$
(**Rucksackproblem**).
Algorithmus zur optimaler Lösung des Rucksackproblems:

1. Bilde alle 2^n Teilmengen und berechne die Summen s_i
2. Wähle als opt. Lösung die Summe $s_i \leq t$ mit $t - s_i =$ minimal

exponentieller Aufwand! (wie bereits gesagt: $2^{100} \approx 10^{30}$)

Optimale vs. Näherungslösung (2)

Algorithmus Rucksackproblem.

Geg.: $S = \{x_1, x_2, \ldots, x_n\}, sum \leq t$.
Kunstruktion der Teilmengen: *iterativer Anzatz*

$$
\begin{array}{ll}
L_0 = \{0\} & 1 = 2^0 \\
L_1 = L_0 \cup (L_0 + x_1) = \{0, 0 + x_1\} & 2 = 2^1 \\
L_2 = L_1 \cup (L_1 + x_2) = \{0, 0 + x_1, 0 + x_2, 0 + x_1 + x_2\} & 4 = 2^2 \\
\vdots & \\
L_n &
\end{array}
$$

```
function optimal_rucksack (S,t)
    n := |S|; L_0 = {0};
    for i := 1 to n
       L_i = L_{i-1} ∪ (L_{i-1} + x_i);
        L_i = L_i − {all elements>t};
    end for
return largest number in L_n;
end
```

Optimale vs. Näherungslösung (3)

Polynomiale Näherungslösung.

Problem: Mengen werden zu groß und müssen reduziert werden
Idee: Wenn zwei Werte nahe beisammen liegen, müssen für eine Approximation nicht beide Werte weiterverfolgt werden.

Funktion `trim`: es kann ein Element y gelöscht werden, wenn ein Element z existiert, so dass:

$$\frac{y}{1+\epsilon} \leq z \leq y$$

mit $0 < \epsilon < 1$ wählbar;
bei guter Wahl von ϵ polynomiell (siehe T. H. Cormen, C.E. Leiserson, R. L. Rivest und C. Stein: Introduction to Algorithms, 3rd ed., MIT Press, 2009)

```
function approx_rucksack (S,t,ε)
    n := |S|; L_0 = {0};
    for i := 1 to n
        L_i = L_{i-1} ∪ (L_{i-1} + x_i);
        L_i = trim(L_i, ε);
        L_i = L_i − {all elements>t};
    end for
return largest number in L_n; // nahe opt. Lösung?
end
```

Top 500 Supercomputer – www.top500.org

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
5	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, HPE Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461

Top 500 Supercomputer – www.top500.org

Supercomputer El Capitan

- **Lawrence Livermore National Laboratory, USA**
- Anm.: FLOP/s = floating point operations per second
  Giga= 10^9, Tera= 10^{12},
  Peta= 10^{15}, Exa = 10^{18},
  Zetta = 10^{21},
  $\mathbf{2^{100} \approx 10^{30}}$

$\sim 2,7$ Exaflop/s theoret.max. und $\sim 1,7$ Exaflop/s für Benchmark-Tests
Cores: 11 039 616
Power: $\sim 29,6$ MWatt

Häufig auftretende Funktionen für Laufzeit (1)

Gegeben sei die Inputgröße n:

- c: konstanter Faktor unabhängig von n.
- n: lineares Wachstum (für Problemgröße n eine gute Situation).
- $n \log n$: logarithmisches Wachstum mit linearem Faktor, auch für große n noch anwendbar.
- n^2: quadratisches Wachstum, z.B. doppelt verschachtelte Schleife.
- n^3: kubisches Wachstum, z.B. dreifach verschachtelte Schleife, für große n problematisch.
- 2^n: exponentielles Wachstum, in der Praxis nicht mehr berechenbar.

Häufig auftretende Funktionen für Laufzeit (2)

In der Praxis fallen viele Algorithmen in 2 Klassen:

- Algorithmen mit **polynomieller Laufzeit** $O(n^k)$ oder mit **exponentieller Laufzeit** $O(2^n)$.

  Gibt es auch Funktionen, die zwischen n^k und 2^n liegen?
- *ja*: $n^{\log n}$ wächst schneller als n^k, aber langsamer als 2^n.

Einwand: $O(n^{100})$ ist doch auch nicht traktabel und ähnlich einem exponentiellem Aufwand?

- Polynome solch hohen Grades treten in der Praxis kaum/nicht auf.
- Algorithmen mit hoch-gradigem polynomiellen Aufwand können üblicherweise durch effizientere Algorithmen mit niedrigerem Grad ersetzt werden.

Beispiel Laufzeiten (1)

Beispiel 1:

Um die Größenordnungen der verschiedenen Funktionen zu veranschaulichen, berechnen wir einige Werte.

Dazu wollen wir annehmen, dass ein Rechenschritt eine Millisekunde (ms) benötigt. Die Definition eines Rechenschritts ist vom Algorithmus abhängig und muss eine sinnvolle Bewertung eines Algorithmus ermöglichen; z.B. bei einem Sortieralgorithmus könnte ein Rechenschritt aus dem Laden zweier Datensätze und einem Vergleich der Schlüsselwerte bestehen.

Tab. 1 gibt den Zeitbedarf für Algorithmen mit unterschiedlichen Aufwand für Problemgrößen zwischen 10 und einer Million an; d.h. z.B. bei einer Problemgröße von 10 benötigt man $10, 33, 10^2, 10^3, 2^{10}$ Rechenschritte.

Beispiel Laufzeiten (2)

Tab. 1:

Pbmgröße	n	$n \log n$	n^2	n^3	2^n
10	10 ms	33 ms	100 ms	1 s	1,02 s
100	100 ms	0,7 s	10 s	16,6 min	$4E+19$ a
1 000	1 s	10 s	16,6 min	11,6 d	$3,3E+290$ a
10 000	10 s	2,2 min	27,7 h	31,7 a	$\#NUM!$
100 000	1,6 min	27,7 min	3,9 m	31709 a	$\#NUM!$
1 000 000	16,6 min	5,5 h	31,7 a	$31E+06$ a	$\#NUM!$

Legende:
ms... 10^{-3}s, s... Sekunde, min... Minute, d... Tage, m... Monat,
a... Jahr
$\#NUM!$... Zahl für Tabellenkalkulator zu gross
Beachte: Zahl der Sekunden seit dem Urknall wird auf 10^{18} geschätzt.

Beispiel Laufzeiten (3)

Beispiel 2:

Nun wollen wir die umgekehrte Fragestellung beantworten. Welche Problemgrößen kann man bei den unterschiedlichen Laufzeitverhalten innerhalb einer Sekunde/Minute/Stude lösen. In *Tab. 2* nehmen wir wieder 10^{-3} s für einen Rechenschritt an. In *Tab. 3* werden die Problemgrößen für eine 10-mal so schnellen Rechner angegeben, d.h., der Rechner benötigt für einen Rechenschritt 10^{-4} s. *Tab. 4* gibt die Steigerung zwischen Tab. 2 und Tab. 3 an.

Tab. 4 zeigt, dass trotz 10-facher Steigerung der Rechenleistung kaum die Problemgröße vergrößert werden konnte.

Beispiel Laufzeiten (4)

Tab. 2: Rechenschritt 10^{-3}

	n	$n \log n$	n^2	n^3	2^n
1 sec	1000	140	32	10	10
1 min	60000	4895	245	39	16
1 h	3600000	204000	1897	153	22

Tab. 3: Rechenschritt 10^{-4}

	n	$n \log n$	n^2	n^3	2^n
1 sec	10000	1003	100	22	13
1 min	600000	39300	775	84	19
1 h	36000000	1737000	6000	330	25

Tab. 4: Problemgrößensteigerung von Tab. 2 auf Tab. 3

	n	$n \log n$	n^2	n^3	2^n	
1 sec	10	7,2	3,2	2,2	1,3	(+3)
1 min	10	8,0	3,2	2,2	1,2	(+3)
1 h	10	8,5	3,2	2,2	1,2	(+3)

Laufzeitberechnung

- Exakte Laufzeitberechnungen sehr aufwendig und üblicherweise nicht wert des Aufwandes.
- Abstraktionen helfen die Berechnung der Laufzeit zu vereinfachen, jedoch die Aussagekraft nicht zu schmälern: z.B. für Laufzeit $an^2 + bn + c$ sind niedere Terme für große n nicht significant und kann durch n^2 approximiert werden.
- Man unterscheidet zwischen worst-case, average-case und best-case Laufzeit. Üblicherweise wird die worst-case Laufzeit betrachtet.
- Interessant ist die Laufzeit von Algorithmen für große Inputs (asymptotische Laufzeit). Dafür muss das asymptotische Wachstum von Funktionen betrachtet werden.

Zusammenfassung & Ausblick

Heute haben wir Folgendes behandelt:

- Komplexität
  - Komplexitätsklassen **P**, **NP**, **NPC**
  - Polynomiale Reduktionen
- Beispiele für NP-vollständige Probleme
- Praktische Implikationen von Komplexität (und Auswege)