Teil II

Eigenschaften von Algorithmen

Überblick

Berechenbarkeit und Entscheidbarkeit

- Frage: Gibt es Problemstellungen, für die kein Algorithmus zur Lösung existiert?
- Oder: Kann man alles berechnen / programmieren?
- Berechenbarkeit:

  Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt berechenbar, wenn es einen Algorithmus gibt, der für alle Eingaben $(x_1, \ldots, x_k) \in \mathbb{N}^k$ terminiert, d.h. in endlicher Zeit $f(x_1, \ldots, x_k)$ berechnet.

- These: Die Menge dieser Funktionen entspricht allen jemals mit Computern berechenbaren Funktionen (CHURCH'sche These), siehe unten.

Existenz nichtberechenbarer Funktionen

- Nicht jede Funktion ist berechenbar.
- Dies folgt aus der Unlösbarkeit der Halteproblems:

Kann man ein Programm entwickeln, das als Eingabe

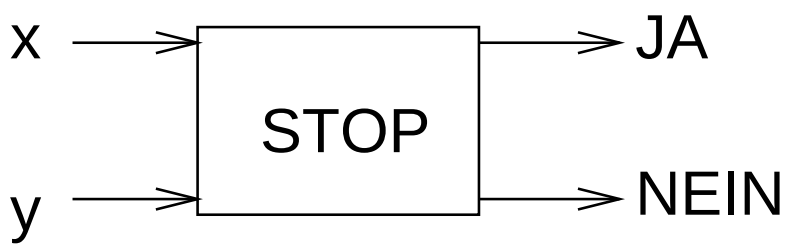
- den Quelltext eines beliebigen Programms
- sowie Eingabewerte für dieses zweite Programm

akzeptiert und dann entscheidet, ob das eingegebene Programm bei der gegebenen Eingabe terminiert?

Veranschaulichung des Halteproblems

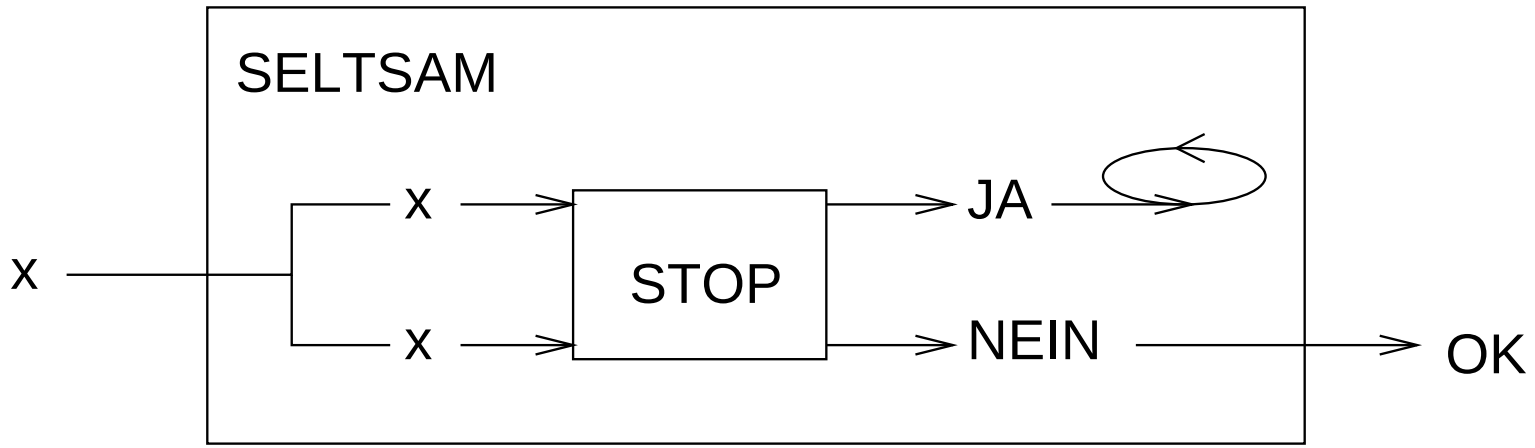
Angenommen, solch eine Maschine (=Algorithmus) `STOP` gibt es:

- Sie hat dann zwei Eingaben
  - Algorithmentext `x`
  - Eingabe `y` für `x`
- sowie zwei Ausgaben
  - `JA`: `x` stoppt bei Eingabe von `y`
  - `NEIN`: `x` stoppt nicht bei Eingabe von `y`.



Veranschaulichung des Halteproblems /2

- Nun konstruieren wir eine neue Maschine `SELTSAM`:



Verhalten von SELTSAM

Hält SELTSAM bei der Eingabe von SELTSAM?

1. Wenn ja, so wird der JA-Ausgang von STOP angelaufen und SELTSAM gerät in die Endlosschleife, hält also nicht. Widerspruch!
2. Wenn nein, so wird der NEIN-Ausgang von STOP angelaufen und SELTSAM stoppt mit OK. Widerspruch!

Die Church-Turing-These (Churchsche These, 1936)

Welche Probleme lassen sich also nun berechnen (und welche nicht)?

Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen.

- Die Turingmaschine ist ein formales Modell für eine Rechenmaschine (das in der Vorlesung Theoretische Informatik genauer behandelt wird).
- Diese These ist nicht beweisbar, da der Begriff “intuitiv berechenbar” nicht exakt formalisiert werden kann.
- Man bezeichnet damit alle Funktionen, die prinzipiell auch von einem Menschen ausgerechnet werden könnten.

Konsequenz der Church-Turing-These

- Die These wird dadurch verstärkt, dass viele Algorithmen-Paradigmen genau die gleiche Leistungsfähigkeit haben: Turingmaschinen, Registermaschinen, imperative Algorithmen, λ-Kalkül, ...
- Man geht in der Informatik davon aus, dass die These stimmt.
- Fazit:

Jeder noch so gute PC ist kann prinzipiell nicht leistungsfähiger sein als eine Turingmaschine.

Beispiele nichtentscheidbarer Probleme

Jede nichttriviale semantische Eigenschaft von Algorithmen ist nichtentscheidbar.

1. Ist die berechnete Funktion total? Überall undefiniert? Injektiv? Surjektiv? Bijektiv? etc. etc.
2. Berechnen zwei gegebene Algorithmen dieselbe Funktion?
3. Ist ein gegebener Algorithmus korrekt, d.h., berechnet er die gegebene (gewünschte) Funktion?
4. Die Berechenbarkeit wird ausführlicher in der Vorlesung Theoretische Informatik behandelt.

Sprung ins nächste Kapitel

- Die nächsten Themen dieses Kapitels sind wichtige Eigenschaften von Algorithmen:
  - Korrektheit
  - Laufzeitverhalten
- Sie lassen sich jedoch leichter an konkreten Beispielen diskutieren.
- Wir springen daher zunächst in das Kapitel “Ausgewählte Algorithmen” und kommen später hierher zurück.

Korrektheit

- Die Beispiele in Kapitel 1 zeigen, dass die Korrektheit eine wichtige Eigenschaft eines Algorithmus ist.
- Problem: Sie ist im Allgemeinen nicht entscheidbar.
- Daher Vorgehen in der Praxis:
  - pragmatisches Testen
  - Beweisführung im Einzelfall
- Das Testen zeigt nur die Anwesenheit, aber nicht die Abwesenheit von Fehlern.

Korrektheit von Algorithmen

- Nachweis der Korrektheit eines Algorithmus bezieht sich immer auf eine *Spezifikation* dessen, was er tun *soll* (*relative* Korrektheit)

- *Spezifikation*: eindeutige Festlegung der berechneten Funktion bzw. des Terminierungsverhaltens
- *Verifikation*: formaler Beweis der Korrektheit bezüglich einer formalen Spezifikation
- *Validation*: (nicht-formaler) Nachweis der Korrektheit bezüglich einer informellen oder formalen Spezifikation (etwa systematisches Testen)

Betrachtete Technik

Vorbedingung (= Voraussetzungen, zulässige Eingaben,...)
Programm/Anweisungsfolge
Nachbedingung (= gewünschte Ausgabe)

- Durch Nachvollziehen der Schritte des Programms leiten wir aus der Vorbedingung die Nachbedingung her.
- Die Nachbedingung eines Schrittes wird zur Vorbedingung des nächsten Schrittes.

Angabe von Vor- und Nachbedingungen

{ VOR } ANW { NACH }

- VOR und NACH sind dabei Aussagen über den Zustand vor bzw. nach Ausführung der Anweisung ANW
- Aussage bedeutet:

Gilt VOR unmittelbar vor Ausführung von ANW und terminiert ANW, so gilt NACH unmittelbar nach Ausführung von ANW.

Partielle versus totale Korrektheit

```
PROG: var   X,Y, ...: int ;
            P,Q ...: bool ;
      input  X_1,...,X_n;
                α ;
      output  Y_1,...,Y_m
```

- PROG heißt partiell korrekt bzgl. VOR und NACH gdw. $\{VOR\}\alpha\{NACH\}$ wahr ist (d.h.: *falls* das Programm terminiert, liefert es das korrekte Ergebnis).
- PROG heißt total korrekt bzgl. VOR und NACH gdw. PROG partiell korrekt ist bzgl. VOR und NACH, und wenn α darüber hinaus immer dann terminiert, wenn vorher VOR gilt.

Beispiele für Vor- und Nachbedingungen

- $\{X = 0\} \quad X := X + 1 \quad \{X = 1\}$
  ist wahr
- $\{\texttt{true}\} \quad X := Y \quad \{X = Y\}$
  ist ebenfalls wahr
- $\{Y = a\} \quad X := Y \quad \{X = a \wedge Y = a\}$
  ist wahr für alle $a \in \mathbb{Z}$
- $\{X = a \wedge Y = b \wedge a \neq b\}$
  $X := Y;\ Y := X$
  $\{X = b \wedge Y = a\}$
  ist *falsch* für alle $a, b \in \mathbb{Z}$
- $\{X = a \wedge Y = b\}$
  $Z := X;\ X := Y; Y := Z$
  $\{X = b \wedge Y = a\}$
  ist wahr für alle $a, b \in \mathbb{Z}$

Beweise bei Sequenzen

$$\{\ \texttt{VOR}\ \}\quad \alpha;\ \beta\ \{\ \texttt{NACH}\ \}$$

- Beweisschritt in zwei Schritte aufteilen: Zwischenbedingung `MITTE` finden, dann

$$\{\texttt{VOR}\}\ \alpha\ \{\texttt{MITTE}\}$$

und

$$\{\texttt{MITTE}\}\ \beta\ \{\texttt{NACH}\}$$

zeigen.

Beweise bei Selektion

{ VOR } **if** B **then** α **else** β **fi** { NACH }

- Fallunterscheidung notwendig: Beweise

$$\{\text{VOR} \wedge B\}\ \alpha\ \{\text{NACH}\}$$

und

$$\{\text{VOR} \wedge \neg B\}\ \beta\ \{\text{NACH}\}$$

- Wichtig ist, dass die Nachbedingung gilt, egal welcher Zweig selektiert wurde.

Beweise bei Schleifen

```
{ VOR }
   while B do β od
{ NACH }
```

1. Geeignete Schleifeninvariante P finden
2. Prüfen, dass
$$\text{VOR} \implies P$$
3. Zeigen, dass der Schleifenrumpf P bewahrt:
$$\{P \wedge B\}\ \beta\ \{P\}$$
4. Zeigen, dass die Nachbedingung nach Verlassen der Schleife gilt:
$$\{P \wedge \neg B\} \implies \text{NACH}$$

(Partielle) Korrektheit von MULT I

```
MULT: var  W,X,Y,Z : int;
      input  X, Y
      Z:=0;
      W:=Y;
      while  W ≠ 0  do  Z:=Z+X;  W:=W-1 od;
      output  Z
```

Wir wollen die Korrektheit von MULT zeigen bezüglich der

- Vorbedingung $\{Y \geq 0\}$ und der
- Nachbedingung $\{Z = X \cdot Y\}$.

(Partielle) Korrektheit von MULT II

Die Anwendung der ersten beiden Schritte von MULT auf die Vorbedingung ergibt

$\{Y \geq 0\}$
`Z:=0;`
$\{Y \geq 0 \wedge Z = 0\}$
`W:=Y;`
$\{Y \geq 0 \wedge Z = 0 \wedge W = Y\}$

Wir müssen also noch zeigen:

$\{Y \geq 0 \wedge Z = 0 \wedge W = Y\}$
`while W` $\neq$ `0 do Z:=Z+X; W:=W-1 od;`
$\{Z = X \cdot Y\}$.

(Partielle) Korrektheit von MULT III

$\{Y \geq 0 \wedge Z = 0 \wedge W = Y\}$ (={VOR})
`while W ≠ 0 do Z:=Z+X; W:=W-1 od;`
$\{Z = X \cdot Y\}$ (={NACH})

zeigen wir nach dem Schema von Folie 2-18
mit $B := (W \neq 0)$ und $\beta := (Z := Z + X;\ W := W - 1)$.

1. Geeignete Schleifeninvariante P finden:
   $P := (X \cdot Y = Z + W \cdot X)$
2. Prüfen, dass VOR $\Longrightarrow$ P:
   $X \cdot Y = 0 + X \cdot Y = Z + X \cdot W$
3. Zeigen, dass $\{P \wedge B\}\ \beta\ \{P\}$ gilt:
   $X' \cdot Y' = ? = Z' + W' \cdot X'$
4. $\{P \wedge \neg B\} \Longrightarrow$ NACH zeigen:
   Einsetzen von $\neg B$ (d.h. von $W = 0$) in P ergibt $X \cdot Y = Z + 0 \cdot X$.

Terminierung von MULT

Mit der Vor-/Nachbedingungstechnik können wir nur die partielle Korrektheit nachweisen!

```
MULT: var  W,X,Y,Z : int;
      input  X, Y
      Z:=0;
      W:=Y;
      while  W ≠ 0  do  Z:=Z+X;  W:=W-1 od;
      output  Z
```

- Wegen VOR $= (Y > 0)$ und $W := Y$ gilt vor der Schleife $W > 0$.
- Wegen $W := W - 1$ muss $W = 0$ nach endlich vielen Schritten gelten.

Fazit: MULT ist total korrekt.

Korrektheitsbeweis auf Algorithmenebene

Korrektheitsbeweise für Algorithmen werden häufig weniger formal geführt:

- Konzentration auf nichttriviale Aspekte
- nur informelle Algorithmenbeschreibung
- kürzer, einfacher, fehleranfälliger
- Beweistechniken: (Schleifen-)Invariante, indirekt etc.

Bsp.: Korrektheit von InsertionSort folgt aus der Invariante:

- Zu Beginn jeder Iteration der for-Schleife befinden sich in a[0] ... a[i-1] die gleichen Elemente wie zu Beginn, aber jetzt in sortierter Reihenfolge.
- Nachweis durch die Technik für Schleifeninvarianten von Folie 2-18.

Komplexität

- Die Korrektheit eines Algorithmus ist unerlässlich.
- Wünschenswert ist außerdem: möglichst geringer Aufwand.
- Daher:
  - Abschätzung des Aufwands eines Algorithmus (Komplexität)
  - Bestimmung des Mindestaufwands für spezielle Problemklassen

Laufzeitabschätzung

- Hauptziel beim Algorithmenentwurf: Effizienz = geringe Laufzeit
- weitere Ziele: Speicherverbrauch, Stromverbrauch, Programmgröße, Kürze/Einfachheit, ...
- Experimentelle Überprüfung (Algorithmus $\rightarrow$ Programm $\rightarrow$ currentTimeMillis) hat viele Einflussfaktoren:
  - Rechner
  - Compiler
  - Compileroptimierungen
  - Implementierung (Umsetzung in Programmiersprache)
  - konkrete Eingabe

  ...und ist daher keine Bewertung “des Algorithmus”.

Asymptotische Analyse

Abstraktion von Einflussfaktoren und praktische Durchführbarkeit der Analyse auch für komplexe Algorithmen wird erreicht durch:

1. idealisiertes Rechnermodell (RAM)
2. Laufzeit als Funktion in Eingabegröße: $T(n)$
3. Einschränkung auf worst case (oder best, average)
4. Betrachten einer oberen Schranke der realen Kosten: $T(n) \geq \ldots$
5. O-Notation, d.h. Vernachlässigung konstanter Faktoren etc.

Random Access Machine (RAM)

Maschinenmodell

- Speicher potentiell unendlicher Größe sowie Register
- Speicherzellen/Register können reelle Zahlen aufnehmen
- Befehlssatz analog Assemblersprachen:
  - Lade-/Speicherbefehle
  - arithmetische, logische etc. Operationen,
  - Sprungbefehle
- beschreibt nicht: Parallelrechner, Speicherhierarchie, ...

Kostenmodell

- Ausführung jedes Befehls benötigt eine Zeiteinheit (Einheitskostenmaß)
- Alternative wäre: nur “wichtige” Operationen berücksichtigen, z.B. Anzahl Vergleichsoperationen in Sortieralgorithmen, Anzahl Speicherzugriffe

Abhängigkeit von der Eingabegröße

- Laufzeit ist Funktion in Abhängigkeit von Eingabegröße: $T(n)$
  - z.B. Sortieren von n Zahlen, Umwandlung Dezimalzahl in Binärzahl mit n Stellen, auch: Sortieren von n Zahlen mit m Duplikaten
- meist worst-case Analyse (schlechtester Fall):

  $T(n)$ = maximale Laufzeit für Problem der Größe n

  d.h. wir betrachten für jedes n die jeweils ungünstigste Eingabe.
- manchmal best-case Analyse (bester Fall):

  $T(n)$ = minimale Laufzeit für Problem der Größe n

  d.h. wir betrachten für jedes n die jeweils günstigste Eingabe.
- manchmal average-case Analyse (durchschnittlicher Fall):
  - erfordert Annahmen über die Häufigkeit des Auftretens mehr oder weniger günstiger Fälle (ist also schwierig).

Motivierendes Beispiel

- sequenzielle Suche in Folgen
- gegeben:
  - eine Zahl $n \geq 0$,
  - n Zahlen $a_1, \ldots, n$, alle verschieden
  - eine Zahl b
- gesucht:
  - Index $i = 1, 2, \ldots, n$, so dass $b = a_i$ falls Index existiert
  - sonst $i = n + 1$
- Lösung

$$i := 1;\ \textbf{while}\ i \leq n \ \wedge\ b \neq a_i\ \textbf{do}\ i := i + 1\ \textbf{od}$$

Aufwand der Suche

- Ergebnis hängt von der Eingabe ab, d.h. von $n, a_1, \ldots, a_n$ und b:
  1. *erfolgreiche* Suche (wenn $b = a_i$): $S = i$ Schritte
  2. *erfolglose* Suche: $S = n + 1$ Schritte

Aufwand der Suche /2

- Ziel: globalere Aussagen, die nur von *einer* einfachen Größe abhängen, z.B. von der Länge n der Folge
- Fragen:
  - A: Wie groß ist S für gegebenes n *im schlechtesten Fall?*
  - B: Wie groß ist S für gegebenes n *im Mittel?*

Analyse für erfolgreiche Suche (A)

- im schlechtesten Fall:
  b wird erst im letzten Schritt gefunden: $b = a_n$

$S = n$ im schlechtesten Fall

Analyse für erfolgreiche Suche (B)

- im Mittel:
  - wiederholte Anwendung mit verschiedenen Eingaben
  - Annahme über Häufigkeit
    Wie oft wird b an erster, zweiter, . . . , letzter Stelle gefunden?
  - Gleichverteilung

Läuft der Algorithmus N-mal, $N \gg 1$, so wird n gleich häufig an erster, zweiter . . . , letzter Stelle gefunden, also jeweils $\frac{N}{n}$-mal.

Analyse für erfolgreiche Suche (B) /2

- Insgesamt für N Suchvorgänge:

$$\begin{aligned} M &= \frac{N}{n} \cdot 1 + \frac{N}{n} \cdot 2 + \ldots + \frac{N}{n} \cdot n \\ &= \frac{N}{n}(1 + 2 + \ldots + n) = \frac{N}{n} \cdot \frac{n(n+1)}{2} \\ &= N \cdot \frac{n+1}{2} \end{aligned}$$

- für **eine** Suche: $S = \frac{M}{N}$ Schritte, also

$S = \frac{n+1}{2}$ im Mittel (bei Gleichverteilung)

Asymptotische Analyse

- Analyse der Komplexität
  $\rightsquigarrow$ Angabe einer Funktion $T\colon\ \mathbb{N} \to \mathbb{N}$ als Maß für den Aufwand
- $T(n) = a$ bedeutet:
  bei einem Problem der Größe n beträgt der Aufwand a.

- *Problemgröße:* Umfang der Eingabe, wie z.B. Anzahl der zu sortierenden oder zu durchsuchenden Elemente
- *Aufwand:* Rechenzeit (Abschätzung als Anzahl der Operationen wie z.B. Vergleiche), Speicherplatz

Aufwand für Schleifen

- Wie oft wird die Wertzuweisung „$x := x + 1$“ in folgenden Anweisungen ausgeführt?

1.	$x := x + 1$	1mal
2.	**for** $i := 1$ **to** n **do** $x := x + 1$ **od**	n-mal
3.	**for** $i := 1$ **to** n **do** **for** $j := 1$ **to** n **do** $x := x + 1$ **od od**	n^2-mal

Aufwandsfunktion

- Aufwandsfunktion $T: \mathbb{N} \to \mathbb{N}$ meist nicht exakt bestimmbar
- daher
  - Abschätzung des *Aufwandes im schlechtesten Fall*
  - Abschätzung des *Aufwandes im Mittel*
- und „ungefähres Rechnen in Größenordnungen“

O-Notation

- Angabe der asymptotischen oberen Schranke für Aufwandsfunktion $\rightsquigarrow$ Wachstumsgeschwindigkeit bzw. Größenordnung
- Asymptote: Gerade, der sich eine Kurve bei immer größer werdender Entfernung vom Koordinatenursprung unbegrenzt nähert
- einfache Vergleichsfunktion $g : \mathbb{N} \to \mathbb{N}$ für Aufwandsfunktion T mit

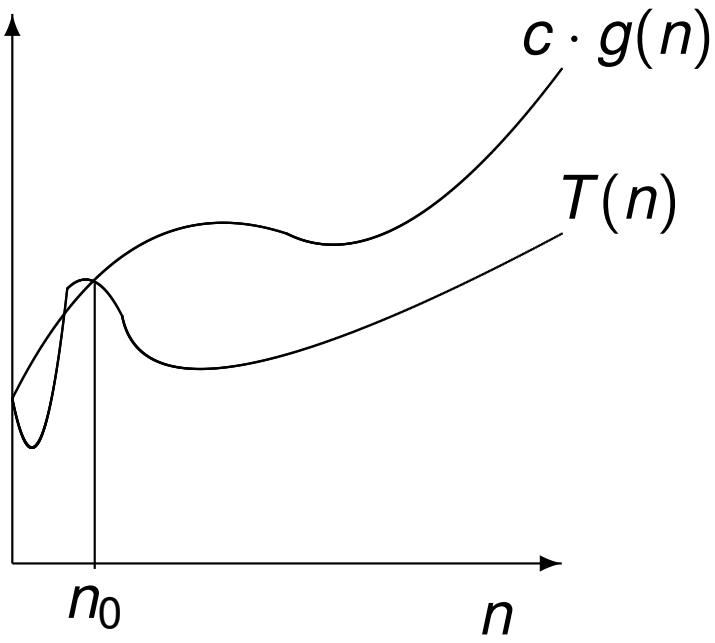
$$T(n) = O(g(n))$$

O-Notation /2

$$O(g(n)) := \{f : \mathbb{N} \to \mathbb{N} \mid \exists c, n_0 \, \forall n \geq n_0 : \; f(n) \leq c \cdot g(n)\}$$

- Es hat sich die Notation $T(n) = O(g(n))$ statt $T(n) \in O(g(n))$ durchgesetzt.
- Für $T(n) = O(g(n))$ gilt also:
  - $\frac{T(n)}{g(n)}$ ist für genügend große n durch eine Konstante c beschränkt,
  - d.h., T wächst nicht schneller als g.

O-Notation /3



Rechnen in Größenordnungen

- Weglassen von multiplikativen Konstanten

$$2 \cdot n = O(n)$$

- Basis des Logarithmus ist unerheblich

$$\log_2 n = O(\log n)$$

  - Basiswechsel entspricht Multiplikation von Konstanten: $\log_b n = \log_a n \cdot \log_b a$

- Beschränkung auf höchsten Exponenten

$$n^2 + 3n - 3 = O(n^2)$$

  - $n^2 + 3n - 3 \leq c \cdot n^2 \rightsquigarrow 1 + \frac{3}{n} - \frac{3}{n^2} \leq c$ (z.B. für $c = 2$ und $n = 1$)

Komplexitätsklassen

$O(1)$	konstanter Aufwand
$O(\log n)$	logarithmischer Aufwand
$O(n)$	linearer Aufwand
$O(n \cdot \log n)$	
$O(n^2)$	quadratischer Aufwand
$O(n^k)$ für ein $k \geq 0$	polynomialer Aufwand
$O(2^n)$	exponentieller Aufwand

Motivation für die Vernachlässigung konstanter Faktoren

- konstantengenaue Analyse ist aufwendig
- Programmiersprache, Compiler etc. beeinflussen Konstante
- nächste Rechnergeneration ist um Konstante schneller
- für große n ist Wachstum in n viel wichtiger als Konstante (s. nächste Folie)

Wachstum

$f(n)$	$n = 2$	$2^4 = 16$	$2^8 = 256$	$2^{10} = 1024$	$2^{20} = 1048576$
$\mathrm{ld} n$	1	4	8	10	20
n	2	16	256	1024	1048576
$n \cdot \mathrm{ld} n$	2	64	2048	10240	20971520
n^2	4	256	65536	1048576	$\approx 10^{12}$
n^3	8	4096	16777200	$\approx 10^{9}$	$\approx 10^{18}$
2^n	4	65536	$\approx 10^{77}$	$\approx 10^{308}$	$\approx 10^{315653}$

Zeitaufwand

G	T = 1 Min.	1 Std.	1 Tag	1 Woche	1 Jahr
n	$6 \cdot 10^7$	$3,6 \cdot 10^9$	$8,6 \cdot 10^{10}$	$6 \cdot 10^{11}$	$3 \cdot 10^{13}$
n^2	7750	$6 \cdot 10^4$	$2.9 \cdot 10^5$	$7.8 \cdot 10^5$	$5,6 \cdot 10^6$
n^3	391	1530	4420	8450	31600
2^n	25	31	36	39	44

1 Schritt = 1 Mikrosekunde

Typische Problemklassen

Aufwand	Problemklasse
$O(1)$	einige Suchverfahren für Tabellen (Hashing)
$O(\log n)$	allgemeine Suchverfahren für Tabellen (Baum-Suchverfahren)
$O(n)$	sequenzielle Suche, Suche in Texten, syntaktische Analyse von Programmen (bei „guter“ Grammatik)
$O(n \cdot \log n)$	Sortieren

Typische Problemklassen /2

Aufwand	Problemklasse
$O(n^2)$	einige dynamische Optimierungsverfahren (z.B. optimale Suchbäume), Multiplikation Matrix-Vektor (einfach)
$O(n^3)$	Matrizen-Multiplikation (einfach)
$O(2^n)$	viele Optimierungsprobleme (z.B. optimale Schaltwerke), automatisches Beweisen (im Prädikatenkalkül 1. Stufe)

Bemerkungen zur O-Analyse

- Egal, ob Formulierung des Algorithmus in Programmiersprache oder Pseudocode: die RAM-Laufzeit kann bis auf konstanten Faktor genau abgeschätzt werden.
- nur grobe Abschätzung der Laufzeit
- ist nur für große n aussagekräftig, da “asymptotische” Analyse
- verwandte Fragestellung: Wie schnell lässt sich ein gegebenes Problem lösen?
  - untere Schranke $T(n) = \Omega(n^2)$, d.h., $T(n) \geq cn^2$
  - Wenn obere und untere Schranke übereinstimmen: $T(n) = \Theta(n^2)$

P- vs. NP-Probleme

- Grenze zwischen **effizient** lösbaren (polynomialer Aufwand) und **nicht effizient** lösbaren (exponentieller Aufwand) Problemen
- **Problemklasse P**: Menge aller Probleme, die mithilfe deterministischer Algorithmen in polynomialer Zeit gelöst werden können
- **Problemklasse NP**: Menge aller Probleme, die nur mithilfe nichtdeterministischer Algorithmen in polynomialer Zeit gelöst werden können
  - Nichtdeterminismus: „Raten“ der richtigen Variante bei mehreren Lösungen
  - Umsetzung mit deterministischen Algorithmen: exponentieller Aufwand

Erfüllbarkeitsproblem

- Problem der Klasse NP
  - Gegeben: aussagenlogischer Ausdruck mit Variablen *a*, *b*, *c* etc., logischen Operatoren $\wedge$, $\vee$, $\neg$ und Klammern
  - Gesucht: Algorithmus der prüft, ob Formel erfüllbar ist, d.h., ob Belegung der Variablen mit booleschen Werten **`true`** und **`false`** existiert, so dass die Formel **`true`** liefert
- Lösungsidee: Testen aller Belegungen (nicht effizient).

NP-Vollständigkeit

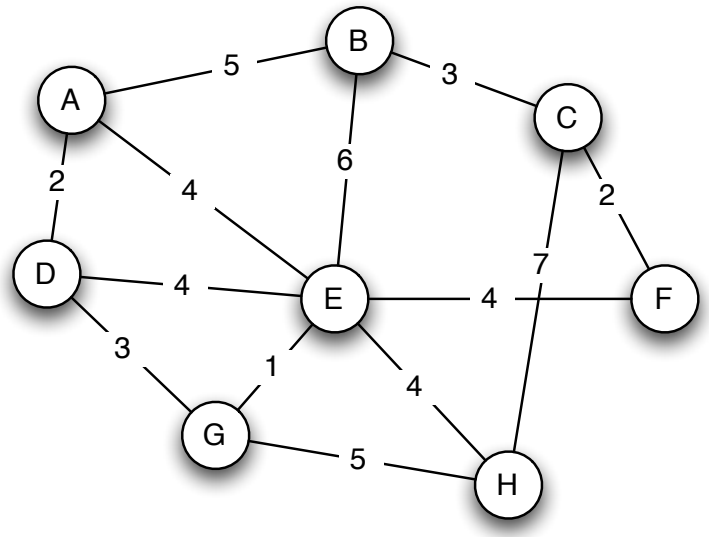
- Es gilt $P \subseteq NP$
- Unbewiesene Vermutung: $P \neq NP$
- Falls dies nicht gilt, wären u.a. alle Public-Key-Verschlüsselungsverfahren auf einen Schlag unbrauchbar!
- Beweisbar: Es existiert eine Klasse von verwandten Problemen aus NP mit folgender Eigenschaft

  Falls eines dieser Probleme in polynomialer Zeit mit einem deterministischen Algorithmus gelöst werden könnte, so ist dies für alle Probleme aus NP möglich.

  $\rightsquigarrow$ **NP-vollständige** Probleme

Problem des Handlungsreisenden

- Problem der Graphentheorie (auch *traveling salesman*)
- Gegeben: Graph mit n Knoten (Städten) und m Kanten (Straßen); Straßen sind mit Entfernung versehen



- Gesucht: kürzeste Route, die alle Städte besucht und an den Startpunkt zurückkehrt

Rucksackproblem

- Wanderer will seinen Rucksack mit verschiedenen Gegenständen packen (auch *knapsack problem*)
- Gegeben: Rucksack mit Kapazität C; n Gegenstände mit jeweils Gewicht g_i und Wert w_i,
- Gesucht: Auswahl $I \subseteq \{1, \ldots, n\}$ der Gegenstände, so dass das Gesamtgewicht die Kapazität nicht überschreitet

$$\sum_{i \in I} g_i \leq C$$

und die Summe der Werte maximal ist

$$\sum_{i \in I} w_i \text{ ist maximal}$$

Analyse von Algorithmen

- Bestimmung der Laufzeitkomplexität von Algorithmen in O-Notation
- Abschätzung der Größenordnung durch einfache Regeln

`for`-Schleifen

- Aufwand

  $$\textit{Laufzeit} := \textit{max. Laufzeit der inneren Anweisung} * \textit{Anzahl der Iterationen}$$

- Beispiel

```
for i := 1 to n do
    a[i] := 0;
od
```

- wenn innere Operation $O(1)$, dann $T(n) = n \cdot O(1) = O(n)$

Geschachtelte `for`-Schleifen

- Aufwand

  Laufzeit := *Laufzeit der inneren Anweisung* $*$
  Produkt der Größen aller Schleifen

- Beispiel

```
for i := 1 to n do
    for j := 1 to n do
        k := k + 1;
    od
od
```

$T(n) = n \cdot n \cdot O(1) = O(n^2)$

Sequenz

- Beispiel

```
for i := 1 to n do
    a[i] := 0;
od;

for i := 1 to n do
    for j := 1 to n do
        a[i] := a[i] + a[j] + i + j;
    od
od
```

$$T(n) = O(n) + O(n^2) = O(n^2)$$

`if-else`-Bedingungen

- Aufwand

$$\begin{aligned} \textit{Laufzeit} := & \textit{Aufwand für Test} + \\ & \max(\textit{Aufwand für } A_1, \textit{Aufwand für } A_2) \end{aligned}$$

- Beispiel

```
if x > 100 then
    y := x;
else
    for i := 1 to n do
        if a[i] > y then y := a[i] fi
    od
fi
```

$$T(n) = O(n)$$

Zusammenfassung

- Berechenbarkeit und Entscheidbarkeit, Halteproblem
- Korrektheit von Algorithmen
  - Spezifikation gewünschter Eigenschaften
  - Korrektheitsbeweise über Vor- und Nachbedingungen
- Komplexität
  - *O*-Notation
  - Komplexitätsklassen
  - P- vs. NP-Probleme
- Literatur: Saake/Sattler: *Algorithmen und Datenstrukturen*, Kap. 7