OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Algorithmen und Datenstrukturen

- Grundlagen (Begriffe und Beispiele) -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 04. Oktober 2018

Zentraler Begriff: Algorithmus

- Persischer Mathematiker und Astronom (~ 825 n. Chr.) Abu Ja‘far Muhammad Ibn Musa al-Khwarizmi
  - Lehrbuch über die Rechenregeln in dem aus Indien stammenden dezimalen Stellenwertsystem

    „Kitab al jabr w‘al mugabala“

- Intuitive, informale Definition eines Algorithmus:
  - Ein Algorithmus ist eine Vorschrift zur Lösung eines Problems, die für eine Realisierung in Form eines Programms auf einem Computer geeignet ist (Taschenbuch der Informatik, 2004)
  - Ein Algorithmus ist eine präzise Handlungsvorschrift, um aus vorgegebenen Eingaben in endlich vielen Schritten eine bestimmte Ausgabe zu ermitteln (Eingabe-Verarbeitung-Ausgabe, EVA-Prinzip)

Eigenschaften von Algorithmen

- Die Abfolge der einzelnen Verarbeitungsschritte muss eindeutig aus einem Algorithmus hervorgehen
- Ein Algorithmus ist unabhängig von einer Notation (z.B. natürliche Sprache, Pseudocode, `C`, `C++`, `Java`, `C#`, ...)
- Es gibt viele Beschreibungen desselben Algorithmus
- Hauptziel beim Entwurf von Algorithmen
  - Korrekte Problemlösung (totale Korrektheit):
    - Terminiertheit: Der Algorithmus endet für jede spezifizierte Eingabe
    - Partielle Korrektheit: Der Algorithmus liefert für jede spezifizierte Eingabe das geforderte Ergebnis
- Nebenziel beim Entwurf von Algorithmen
  - Effiziente Problemlösung hinsichtlich Zeit und Platz
- Algorithmen sollten effektiv sein

Beispiele für Algorithmen

- Addieren, Subtrahieren, Multiplizieren, Dividieren, etc.
- Kochrezepte, Bastel-/Gebrauchsanleitungen, Spielregeln, …
- Sortieren, Suchen, Effizienter Umgang mit Datenstrukturen, …
- Vorsicht: Forderung der Eindeutigkeit fordert Präzision!

- Algorithmus zur Bestimmung des größten gemeinsamen Teilers (ggT) zweier natürlicher Zahlen (Euklidischer Algorithmus, ~ 300 v. Chr.):

- Eingabe: a, b $\in \mathbb{N}$ (zwei natürliche Zahlen a und b)
- Ausgabe: a = ggT(a,b) (ggT von a und b)
- Algorithmus:

```
Wiederhole
        r = Rest der ganzzahligen Division von a/b
        a = b
        b = r
Bis r = 0 ist
Gib a aus
```

Nachweis der totalen Korrektheit?

Partielle Korrektheit mittels Hoare Kalkül

Vorbedingung: a, b ∈ ℕ ; sei G größter gemeinsame Teiler von a und b, E = x + y

Nachbedingung: x = G

{ G ist ggT(a, b) ∧ a ∈ $\mathbb{N}^+$ ∧ b ∈ $\mathbb{N}^+$ }

x := a; y := b;

{ INV: G ist ggT(x, y) ∧ x ∈ $\mathbb{N}^+$ ∧ y ∈ $\mathbb{N}^+$ ∧ x + y > 0 }

solange x ≠ y wiederhole

{ INV ∧ x ≠ y ∧ x + y = k }

falls x > y :

{ G ist ggT(x, y) ∧ x > y ∧ x ∈ $\mathbb{N}^+$ ∧ y ∈ $\mathbb{N}^+$ ∧ x + y = k } ⟹

{ G ist ggT(x − y, y) ∧ x − y ∈ $\mathbb{N}^+$ ∧ y ∈ $\mathbb{N}^+$ ∧ x − y + y = k − y < k }

x := x - y

{ INV ∧ x + y < k }

sonst

{ G ist ggT(x, y) ∧ y > x ∧ x ∈ $\mathbb{N}^+$ ∧ y ∈ $\mathbb{N}^+$ ∧ x + y = k } ⟹

{ G ist ggT(x, y − x) ∧ x ∈ $\mathbb{N}^+$ ∧ y − x ∈ $\mathbb{N}^+$ ∧ x + y − x = k − x < k }

y := y - x

{ INV ∧ x + y < k }

{ INV ∧ x + y < k }

{ INV ∧ x = y } ⟹ { x = G }

Zentraler Begriff: Datenstruktur

- Algorithmen verarbeiten Eingaben zu Ausgaben und benutzen dabei ggf. unstrukturierte (skalare) und strukturierte Daten
- Datentyp
  - Menge von Objekten mit darauf definierten Operationen
  - Zwei Varianten:
    - (konkreter) Datentyp (abhängig von Rechner und Implementierung)
    - abstrakter Datentyp (abstrahiert von spezieller Implementierung)
- Datentyp wird durch folgendes Tupel festgelegt:
  - Objektmenge (Werte)
  - Operationen, definiert durch
    - Signatur (Name mit Definitions- und Wertebereich, Syntax)
    - Regeln/Axiome (Wirkung der Operationen, Semantik)
- Datenstruktur (= strukturierte Daten + Operationen)
  - Menge von Datentypen, zwischen denen Beziehungen bestehen

Beispiele für Datentypen in C++ (32-Bit)

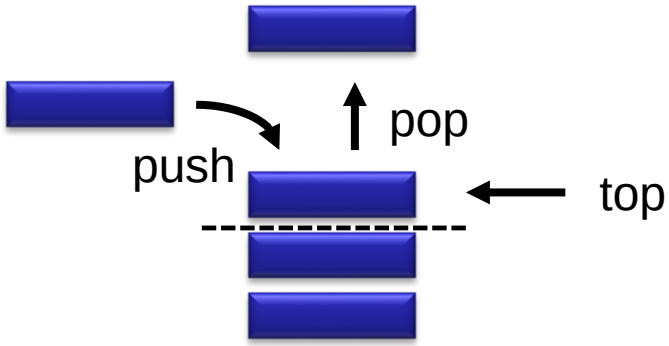
Datentyp	Bits	Wertebereich	Typische Operationen
char, signed char	8	$-128 \dots 127$	+,-,*,/,<;>,%
unsigned char	8	$0 \dots 255$	+,-,*,/,<;>,%
short, signed short	16	$-32768 \dots 32767$	+,-,*,/,<;>,%
unsigned short	16	$0 \dots 65535$	+,-,*,/,<;>,%
int, signed int	32	$-2.147.483.648 \dots 2.147.483.647$	+,-,*,/,<;>,%
unsigned,unsigned int	32	$0 \dots 4.294.967.295$	+,-,*,/,<;>,%
long, signed long	32	$-2.147.483.648 \dots 2.147.483.647$	+,-,*,/,<;>,%
unsigned long	32	$0 \dots 4.294.967.295$	+,-,*,/,<;>,%
float	32	$1{,}2 \cdot 10^{-38} \dots 3{,}4 \cdot 10^{38}$	+,-,*,/,<;>
double	64	$2{,}2 \cdot 10^{-308} \dots 1{,}8 \cdot 10^{308}$	+,-,*,/,<;>
long double	96	$3{,}4 \cdot 10^{-4932} \dots 1{,}1 \cdot 10^{4932}$	+,-,*,/,<;>

Beispiel: Abstrakter Datentyp Boolean (bool)

- Boolean = (Objekte O_B , Funktionen F_B) mit
  - O_B = $\{ w, f \}$ // oder: {wahr, falsch} = {true, false}
  - F_B = $\{\neg, \wedge, \vee\}$ // oder: {nicht, und, oder} = {not, and, or}
- Signaturen
  - w: $\rightarrow O_B$
  - f: $\rightarrow O_B$
  - $\neg$: $O_B \rightarrow O_B$ // Negation
  - $\wedge$: $O_B \times O_B \rightarrow O_B$ // Konjunktion
  - $\vee$: $O_B \times O_B \rightarrow O_B$ // Disjunktion
- Regeln/Axiome (x, y, z seien vom ADT Boolean)
  - $x \wedge y = y \wedge x$, $x \vee y = y \vee x$ (Kommutativgesetze)
  - $(x \wedge y) \wedge z = x \wedge (y \wedge z)$, $(x \vee y) \vee z = x \vee (y \vee z)$ (Assoziativgesetze)
  - $(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$, $(x \vee y) \wedge z = \ldots$ (Distributivgesetze)
  - $x \wedge w = x$, $x \vee f = x$, $x \wedge \neg x = f$, $x \vee \neg x = w$ (Neutralität, Komplement)
  - $\neg w = f$, $\neg f = w$ (Dualität)

Beispiel: Datenstruktur Stapel (Keller, Stack)

- Ein Stapel ist eine Datenstruktur, die eine begrenzte Menge von Objekten eines Datentyps aufnehmen kann und folgende Operationen unterstützt (Last-In-First-Out, LIFO-Prinzip):
  - create: Erzeugt einen leeren Keller
  - push: Legt ein Objekt auf den Stapel
  - pop: Holt u. entfernt das oberste Objekt vom Stapel
  - top: Holt das oberste Objekt vom Stapel (ohne entfernen)
  - empty: Liefert wahr, wenn der Stapel leer ist, sonst falsch



- Ein Stapel kann mit Hilfe eines Feldes implementiert werden

Implementierung eines Stapels in C++ (I)

```
typedef int object; // O.B.d.A. seien die Objekte vom Typ int
class stack1 {
      private:
            object *o; // Zeiger auf ein dynamisches Feld
            int size;  // Groesse des Stapels
            int tp;    // Oberstes Element
      public:
            stack1(int n);// Erzeugt leeren Stapel, n Objekte
            ~stack1(); // Gibt einen Stapel wieder frei
            void push(object o);// Legt Objekt auf den Stapel
            object pop(); // Holt u. entfernt oberstes Objekt
            object top();  // Liefert das oberste Objekt
            bool IsEmpty(); // Liefert, ob der Stapel leer ist
            bool IsFull();  // Liefert, ob der Stapel voll ist
};
```

Implementierung eines Stapels in C++ (II)

```
stack1::stack1(int n) {
    size=n;
    tp=-1;
    o=new object[size]; }
```

```
stack1::~stack1() {
    delete[] o; }
```

Was ändert sich, wenn hier **tp=0** steht?

```
void stack1::push(object o) {
    if (!IsFull()) this->o[++tp]=o; }
```

```
object stack1::pop() {
    if (!IsEmpty()) return(o[tp--]);
    else … Fehlerbehandlung … }
```

```
object stack1::top() {
    if (!IsEmpty()) return(o[tp]);
    else … Fehlerbehandlung … }
```

Welchen **Aufwand** haben die Operationen?

```
bool stack1::IsEmpty() {
    return (tp > -1 ? false : true); }
```

```
bool stack1::IsFull() {
    return (tp ≥ size-1 ? true : false);
}
```

Analyse und Design von Algorithmen

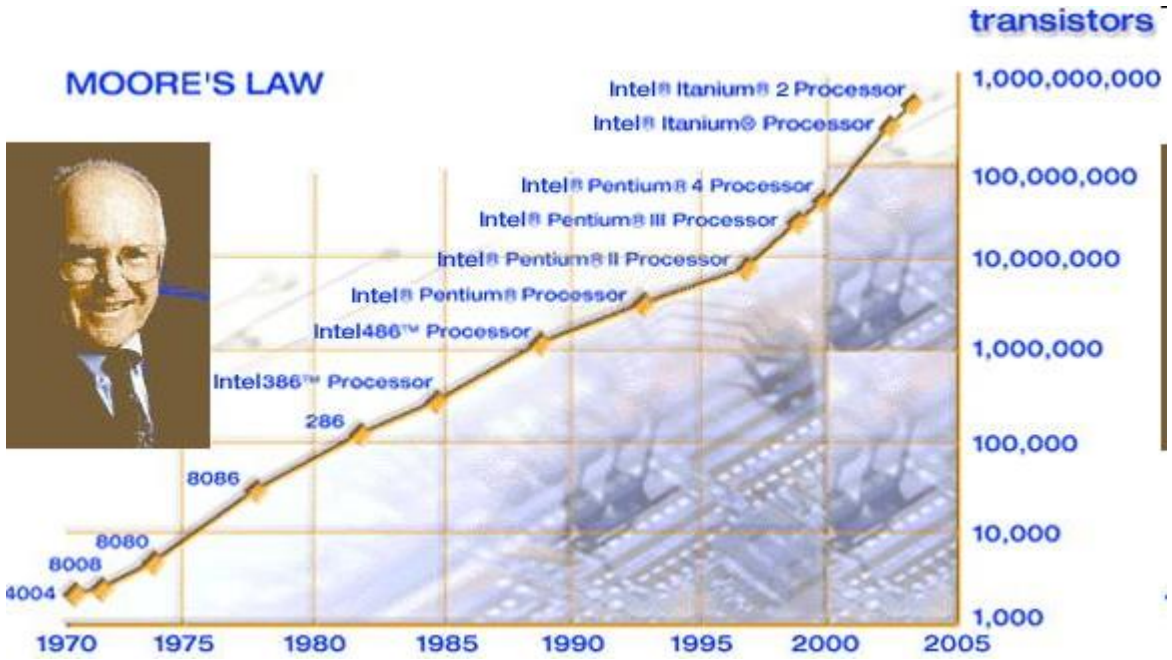
- Algorithmen stehen im Mittelpunkt der Informatik
- Hauptziel beim Entwurf von Algorithmen
  - Korrekte Problemlösung (totale Korrektheit)
    - Terminiertheit: Der Algorithmus endet für jede spezifizierte Eingabe
    - Partielle Korrektheit: Der Algorithmus liefert für jede spezifizierte Eingabe das geforderte Ergebnis
- Nebenziel beim Entwurf von Algorithmen
  - Effiziente Problemlösung hinsichtlich Zeit und Platz
- Interessant sind meist nur effiziente Algorithmen
- Wesentliche Effizienzmaße
  - Rechenzeitbedarf
    - Zeitkomplexität: Zählen der atomaren Schritte
  - Speicherplatzbedarf
    - Platzkomplexität: Zählen der verwendeten Speicherzellen

Komplexität von Algorithmen

- Es gibt unendlich viele Algorithmen zur Lösung von Problemen (in Wissenschaft, Technik, Wirtschaft, …)
- Funktional gleichwertige Algorithmen können sich erheblich in der Effizienz/Komplexität unterscheiden
- Ein Algorithmus ist umso effizienter, je weniger er von den beiden Ressourcen Rechenzeit und Speicherplatz verwendet
- Komplexität hängt u.a. von der Eingabe für das Programm ab
- Faktoren zur Bestimmung der Komplexität
  - Messungen auf einer konkreten Maschine
  - Aufwandsermittlung in einem idealisierten Rechnermodell (z.B. RAM)
  - Asymptotische Komplexitätsabschätzung durch ein abstraktes Komplexitätsmaß in Abhängigkeit der Problem-/Eingabegröße

Moore's Gesetz

- Dr. Gordon E. Moore: Mitgründer der Firma Intel



- Verdoppelung der Prozessorleistung alle ~ 18-24 Monate
- Computer werden immer kleiner, schneller und billiger

Idealisierter Rechner: Registermaschine

- Registermaschine (engl. Random Access Machine, RAM)
  - Zentrale Recheneinheit inkl. Rechen- und Steuerwerk
    - Akkumulator (Adresse 0 im Datenspeicher)
    - Befehlsregister
    - Befehlszähler (Programm Counter, PC)
  - Programmspeicher (Program Memory, PM, Adressen 1, 2, …)
  - Datenspeicher (Data Memory, DM, Adressen 1, 2, …)
  - Vereinfachte Ein-/Ausgabe durch direkte Befehle
- Inhalt des Datenspeichers sei beschrieben durch f: $IN_0 \rightarrow IN_0$
  - f(0) ist der Inhalt des Akkumulators
  - f(*adresse*) ist der Inhalt des Datenspeichers an der Stelle *adresse*
- Befehlsstruktur
  - <Adresse PM> <Befehl> <Adresse PM/DM>

8 Bit 8 Bit

Befehle der Registermaschine I

- Verarbeitungs-, Transport- und Ein-/Ausgabebefehle:

Codierung	Syntax	Semantik	Beschreibung
0x01	ADD *adresse*	f(0) = f(0) + f(*adresse*)	Addieren
0x02	SUB *adresse*	f(0) = f(0) – f(*adresse*)	Subtrahieren
0x03	MUL *adresse*	f(0) = f0) * f(*adresse*)	Mutliplizieren
0x04	DIV *adresse*	f(0) = f(0) / f(*adresse*)	Dividieren
0x05	LDA *adresse*	f(0) = f(*adresse*)	Laden
0x06	LDK *zahl*	f(0) = *zahl*	Konstante Laden
0x07	STA *adresse*	f(*adresse*) = f(0)	Speichern
0x08	INP *adresse*	f(*adresse*) = <Eingabe>	Eingeben
0x09	OUT *adresse*	<Ausgabe> = f(*adresse*)	Ausgeben
0x0A	HLT 99		Programmende

Warum ist hier eine Zahl sinnvoll?

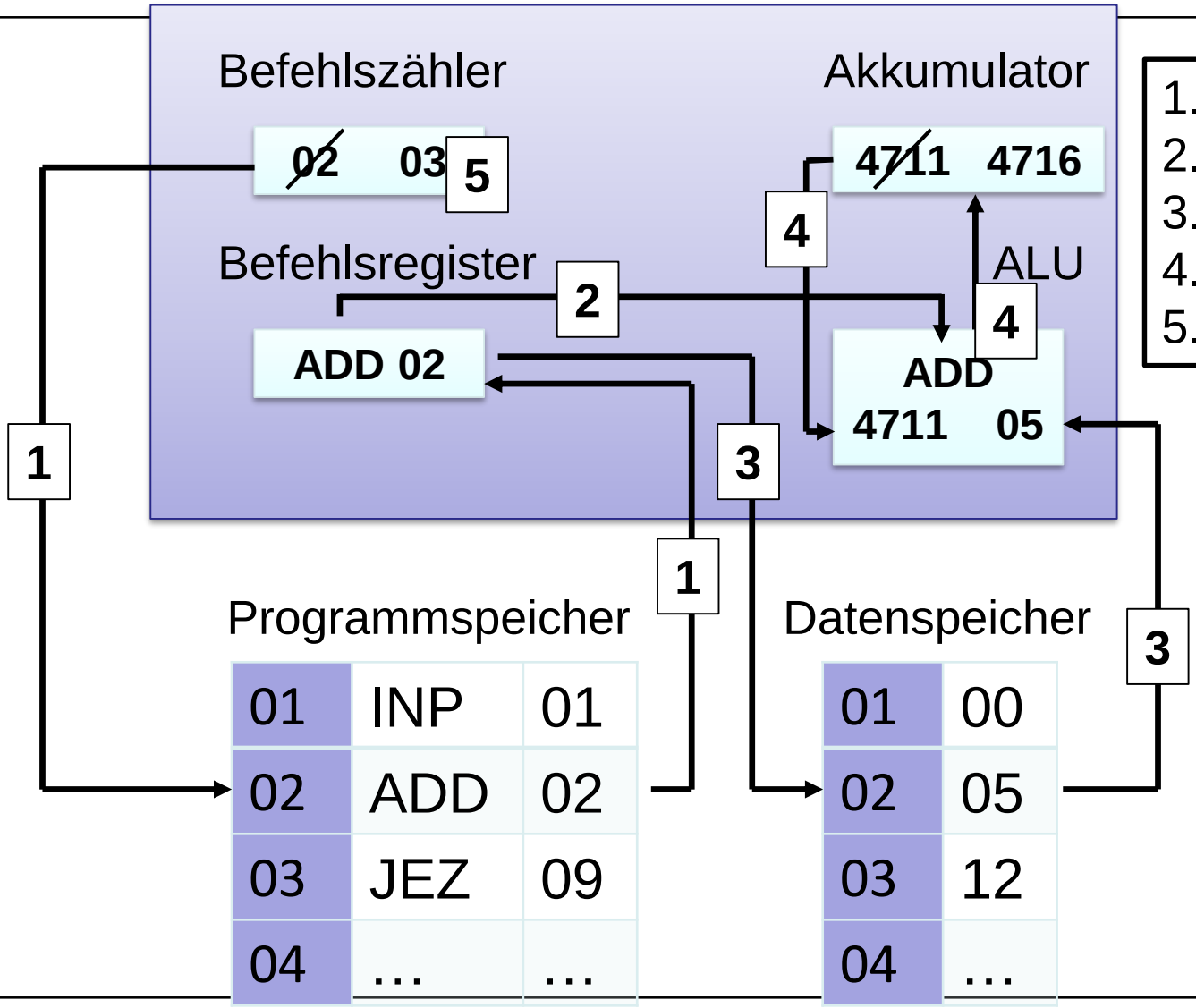
Befehle der Registermaschine II

- Sprungbefehle:

Codierung	Syntax	Semantik	Beschreibung
0x0B	JMP *adresse*	PC = *adresse*	Verzweigen
0x0C	JEZ *adresse*	Falls f(0) = 0, dann PC = *adresse*	
0x0D	JNE *adresse*	Falls f(0) $\neq$ 0, dann PC = *adresse*	
0x0E	JLZ *adresse*	Falls f(0) < 0, dann PC = *adresse*	
0x0F	JLE *adresse*	Falls f(0) $\leq$ 0, dann PC = *adresse*	
0x10	JGZ *adresse*	Falls f(0) > 0, dann PC = *adresse*	
0x11	JGE *adresse*	Falls f(0) $\geq$ 0, dann PC = *adresse*	

- Nach jedem Befehl, wenn es keine Verzweigung gab:
  - PC = PC +1

Befehlszyklus der Registermaschine



1. Befehl holen
2. Befehl dekodieren
3. Operand holen
4. Operation ausführen
5. Befehlszähler ändern

Beispielprogramm

- Was tut folgendes Programm?
  - 06000702080105010C09010207020B0309020A99

01	06	00
02	07	02
03	08	01
04	05	01
05	0C	09
06	01	02
07	07	02
08	0B	03
09	09	02
10	0A	99

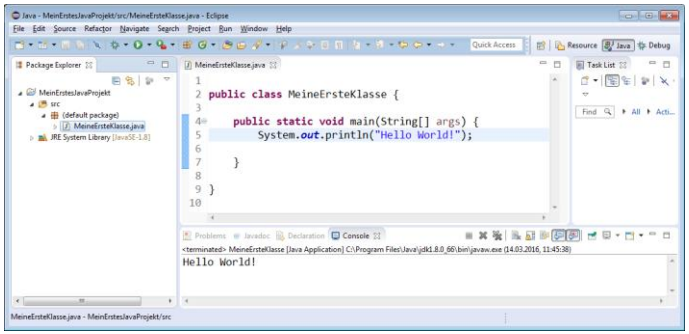
01	LDK	00	Akku auf 0
02	STA	02	Akku → Adr. 2
03	INP	01	Eingabe → Adr. 1
04	LDA	01	Adr. 1 → Akku
05	JEZ	09	Akku = 0 ? Ja: 09
06	ADD	02	Akku + Adr. 2
07	STA	02	Akku → Adr. 2
08	JMP	03	Weiter bei 03
09	OUT	02	Ausgabe Adr. 2
10	HLT	99	Programmende

Church'sche These

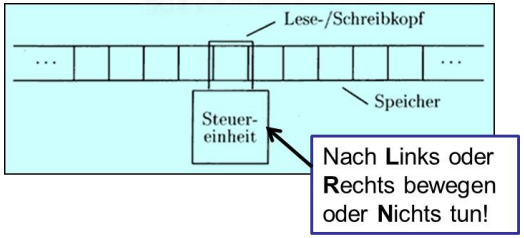
(1936, auch Church-Turing-These)

- These: Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen
- Alternative Formulierungen/Folgerungen
  - Jede Funktion, die überhaupt in irgendeiner Weise berechenbar ist, kann durch eine Turingmaschine berechnet werden
  - Jedes Problem, das überhaupt maschinell lösbar ist, kann von einer Turingmaschine gelöst werden
- Anmerkungen
  - Church'sche These ist nicht beweisbar, da „intuitiv berechenbare Funktionen" nicht formalisiert werden können
  - Anerkanntes Rechenmodell: Von-Neumann-Rechner
  - Idealisierte Von-Neumann-Rechner: Registermaschinen
  - Registermaschinen sind äquivalent zu Turingmaschinen
- Church'sche These gilt als allgemein akzeptiert

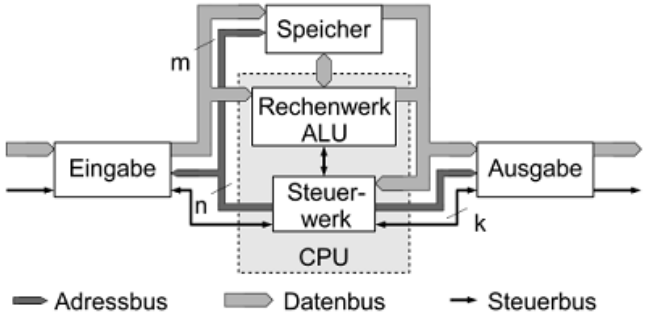
Church'sche These (grafische Interpretation)



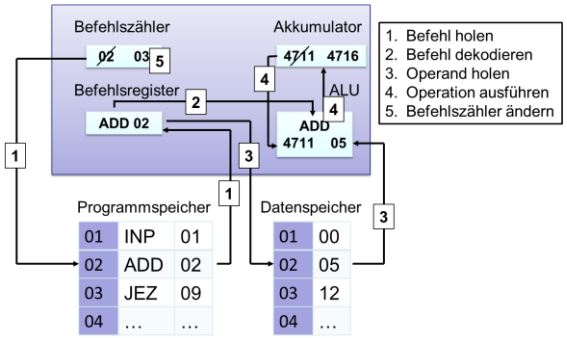
C, C++, C#, Java Programm



Turingmaschine (TM)



Von-Neumann-Rechner



Registermaschine (RAM)

Algorithmus-Begriff

- Interpretation der Church-Turing-These:
  - Bisher und in Zukunft vorgenommene „vernünftige“ Definitionen von Algorithmus sind gleichwertig und haben die gleiche Bedeutung wie die bisher bekannten Definitionen!
- Algorithmus = Programm für eine TM
  = Programm für eine RAM
  = Programm für andere Modelle
  (Goto, While, μ-Rekursion)
  = Programm in C/C++
  Pascal
  Java
  C#, …

Komplexität (Zeit/Platz)

- Uniforme Komplexität (Einheitsmaß)
  - Jeder Befehl, der ausgeführt wird, entspricht einem Schritt (konstant)
  - Sei A ein Algorithmus für eine RAM mit Eingabe x
    - $T_A(x)$: Anzahl der Schritte, die A bei Eingabe von x durchführt
    - $S_A(x)$: Anzahl der Speicherzellen, die A bei Eingabe von x benutzt
- Nachteil der uniformen Komplexität
  - Befehle mit unterschiedlicher Bit-Komplexität werden gleich bewertet
- Logarithmische Komplexität (Logarithmisches Maß)
  - Jeder Befehl, der ausgeführt wird, wird mit der Bit-Länge der Operanden des Befehls gewichtet (entspricht der Bit-Komplexität)
  - Anstelle einer Speicherzelle wird die Bit-Länge der in einer Speicherzelle abgespeicherten größten Zahl verwendet
  - Kennzeichnung analog zu oben: ${T_A}^{\log}(x)$ bzw. ${S_A}^{\log}(x)$
- Beispiel: Addition und Multiplikation von zwei n-Bit Zahlen

Worst Case, Average Case, Best Case I

- Komplexität wird über alle Eingaben unterschieden nach
  - Worst Case: Laufzeit im schlechtesten Fall (Wesentlich!)
  - Average Case: Laufzeit im Mittel (Mittelwert)
  - Best Case: Laufzeit im besten Fall (Eher uninteressant)
- Beispiel: Addition um eins im Binärsystem
  - Eingabe: $\text{bin}(a) = (a_{n-1}, \ldots, a_0)$, $a \in \mathbb{N}$, $a_i \in \{0,1\}$
    (Binärdarstellung einer natürlichen Zahl)
  - Ausgabe: $\text{bin}(a) + 1$
  - Algorithmus:
    Falls $(a_{n-1}, \ldots, a_0) = (1, \ldots, 1)$
    Gib $(1,0 \ldots, 0)$ aus // n Nullen
    Sonst
    Ermittle i mit $a_i = 0$ und $a_j \neq 0$ für alle $j < i$
    Gib $(a_{n-1}, \ldots, a_{i+1}, 1,0, \ldots, 0)$ aus // i Nullen
  - Laufzeit: Worst Case: $n + 1$ Schritte // Wann?
    Best Case: 1 Schritt // Wann?

Worst Case, Average Case, Best Case II

- Average Case bei Addition um 1 im Binärsystem:
  - Wie viele mögliche Eingaben für a gibt es? 2^n
  - Wie viele Eingaben gibt es für den Worst Case (1. Fall)? 1
    - Laufzeit in diesem Fall: $n+1$ Schritte
  - Wie viele Eingaben gibt es mit $a_i = 0$ und $a_j \neq 0$ für alle $j < i$? 2^{n-i-1}
    - Laufzeit in diesen Fällen: $(i+1)$ Schritte
  - Insgesamt ergibt sich für die durchschnittliche Laufzeit:

$$\frac{\sum_{i=0}^{n-1}\left(2^{n-i-1}\cdot(i+1)\right)+n+1}{2^n} \text{ Schritte}$$

  - Es gilt:

$$\sum_{i=0}^{n-1} 2^{n-i-1}\cdot(i+1) = \sum_{i=1}^{n} 2^{n-i}\cdot i$$

Worst Case, Average Case, Best Case III

- Weiter gilt:

$$\sum_{i=1}^{n} 2^{n-i} \cdot i = 2^{n-1} \cdot 1 + 2^{n-2} \cdot 2 + \ \ldots + 2^0 \cdot n$$

$$= 2^{n-1} + 2^{n-2} + \cdots + 2^0 \quad \text{(Zeile 1)}$$
$$+ 2^{n-2} + \cdots + 2^0 \quad \text{(Zeile 2)}$$
$$\ddots \qquad \vdots \qquad \vdots$$
$$+ 2^0 \quad \text{(Zeile n)}$$

- Erinnerung Geometrische Reihe:

$$\sum_{k=0}^{m} q^k = \frac{q^{m+1} - 1}{q - 1} \text{ für } q \neq 1$$

- Damit folgt:

$$\sum_{i=1}^{n} 2^{n-i} \cdot i = \frac{2^{n-1+1} - 1}{2 - 1} + \frac{2^{n-2+1} - 1}{2 - 1} + \ \ldots + \frac{2^{n-n+1} - 1}{2 - 1}$$

Worst Case, Average Case, Best Case IV

- Es folgt schließlich:

$$\sum_{i=1}^{n} 2^{n-i} \cdot i = (2^n - 1) + (2^{n-1} - 1) + \ ... + (2^1 - 1)$$

$$= \sum_{i=1}^{n} 2^i - n = 2^{n+1} - 2 - n$$

- Für die durchschnittliche Laufzeit folgt:

$$\frac{(2^{n+1} - 2 - n) + (n+1)}{2^n} = 2 - \frac{1}{2^n} \text{ Schritte}$$

- Zusammenfassend:
  - Worst Case: $n + 1$ Schritte
  - Average Case: 2 Schritte (asymptotisch)
  - Best Case: 1 Schritt
- Positiv: Durchschnitt liegt nah am Best Case (nicht immer so!)

Worst Case, Average Case, Best Case V

- Sei A ein Algorithmus für eine RAM mit Eingabe x über einem Alphabet Σ
- Laufzeit im schlechtesten Fall (Worst Case)

$$T_A{}^{wc}(n) = \max_{x \in (\Sigma)^n} \{T_A(x)\}$$

- Laufzeit im besten Fall (Best Case)

$$T_A{}^{bc}(n) = \min_{x \in (\Sigma)^n} \{T_A(x)\}$$

- Laufzeit im Durchschnitt (Average Case)

$$T_A{}^{ac}(n) = \frac{1}{|\Sigma|^n} \sum_{x \in (\Sigma)^n} T_A(x)$$

Zeitkomplexität von C, C++, C#, Java - Programmen

- Alle ausgeführten Anweisungen werden „mit 1“ gezählt
- Anweisungen sind
  - Zuweisungen
  - Auswahl-Anweisungen (if-then-else, switch-case)
  - Iterationen (for, while-do, do-while)
  - Sprünge (break, continue, goto, return)
  - Funktionsaufrufe/Methodenaufrufe
- Bei Iterationen und Funktionsaufrufen/Methodenaufrufen
  - Alle im Rahmen der Iteration oder der Funktion/Methode ausgeführten Anweisungen werden natürlich auch gezählt (iterativ oder rekursiv)
- Vorsicht:
  - Funktions-/Methodenaufrufe in höheren Programmiersprachen sind nicht sofort ersichtlich (Beispiele: Konstruktor, Destruktor, Boolesche Abfragen, …)
- Einfache und komplexe Zuweisungen werden nicht unterschieden
  - Ziel: Ermittlung des asymptotischen Laufzeitverhaltens