2. Zeit und Platz

Neben experimentell beobachteten Aussagen über Zeit- und Platzbedarf eines Programms sind fundierte theoretische Vorhersagen von Interesse. Wie solche Vorhersagen zu treffen sind, wird im folgenden behandelt.

Zunächst etwas zum Platzbedarf.

Ein Zeichen nach ASCII:

7 Bit pro Zeichen. Das 8. Bit eines Byte ist Prüfbit (erlaubt das Erkennen eines falsch stehenden Bits – nicht dessen Korrektur)

$2^7 = 128$ Zeichen.

Ein Zeichen nach Unicode:

2 Byte, also $2^{16} = 2^8 \cdot 2^8 = 65.536$ Zeichen.

Der Übergang von ASCII nach Unicode ist ein schönes Beispiel der „kombinatorischen Explosion“: Die einfache Verdopplung der Anzahl Bits von 8 auf 16 bewirkt eine Explosion der Anzahl Möglichkeiten: Von 256 auf 65.536. Exponentieller Zusammenhang:

(2.3)

- 1 Bit mehr = Verdopplung ($2^{n+1} = 2 \cdot 2^n$)
- Verdopplung der Anzahl Bits
  = Quadrierung ($2^{2n} = (2^n)^2$).
- Anzahl Bits mal k
  = hoch k ($2^{k \cdot n} = (2^n)^k$)
- Quadrierung der Anzahl Bits
  = hoch Anzahl Bits
  ($2^{(n^2)} = 2^{n \cdot n} = (2^n)^n$).

Dagegen bei etwa n^c, c konstant, statt 2^n:

- Verdopplung des n
  = Mult. mit Konstante ($(2n)^c = 2^c \cdot n^c$)

Konstante

(9.4)

- Mult. von n mit k

  = Mult. mit Konstante $\left((k\cdot n)^c = k^c \cdot n^c\right)$.

- Quadrierung des n

  = Quadrierung $\left((n^2)^c = (n^c)^2\right)$.

- Addition von 1

  = Addition von a, $n \le a \le n^c$:

  $\left(n^c + n^c \le (n+1)^c \le (2n)^c\right)$

Bei linearer Abhängigkeit zusätzlich:

- Addition von 1

  = Addition einer Konstanten.

  $\left(c\cdot(n+1) = c\cdot n + c \right)$

Größere Mengen von Byte

1 Kilobyte (kB): 2^{10} = 1024 Byte.

1 Megabyte (MB): 2^{20} = 1.048.576 Byte

1 Gigabyte (GB): 2^{30} = 1.073.741.824 Byte.

1 Terabyte : 2^{40} Byte.

Eigentlich: Kilo = 1000, Mega = 1.000.000, ... Manchmal auch in der Informatik.

Beispielwerte

Seite eines Buches	Ungefähr
40 Zeilen à 80 Zeichen.	3 – 6,5 kB
Insgesamt 3200 Zeichen.	
1000 Seiten	3 – 6,5 MB

2.6

Diskette	1,4 MB
Windows Programm	5 MB
Musikstück	40 MB
Hauptspeicher	56 - 128 MB
DVD	5 GB
Festplatte	80 GB bis 100 GB

Speicherplatzbedarf von unseren Programmen:

PRIM.java 2 long

Elemente vom Typ String wie im Programm ausgegeben.

immer gleich viel Speicherplatz

ggTPrim.java — Eine Anzahl von long Variablen und im Programm wird etwas Text und im Programm.

Immer gleich viel Speicherplatz.

Partition.java — Auf jeden Fall Speicherplatz für ein array mit n Einträgen vom Typ int, also n Plätze für int.

4 Byte

Hängt von dem eingelesenen n ab!

Außerdem eine Zahl von weiteren Variablen. Auch nicht-explizite und die Länge

Diese Zahl hängt nur vom Programm ab. Nicht von der Eingabe!

Faku.java

Einige int-Variablen.

Der Laufzeitkeller,
maximal n Rahmen.

n ist eingegeben.

Größe eines Rahmens?

Eine konstante Zahl
viele Plätze. Also
insgesamt etwa

$$c + d \cdot n$$

bei Eingabe n, c, d
Konstanten programmabhängig.

Abhängig vom
Programm,
vom Übersetzer
usf. Unabhängig von
der Eingabe.

DreiFarb.java

Wir haben 2 arrays.
graph und F, graph
mit n^2 int Speicherplätzen,
F mit n int Speicherplätzen.

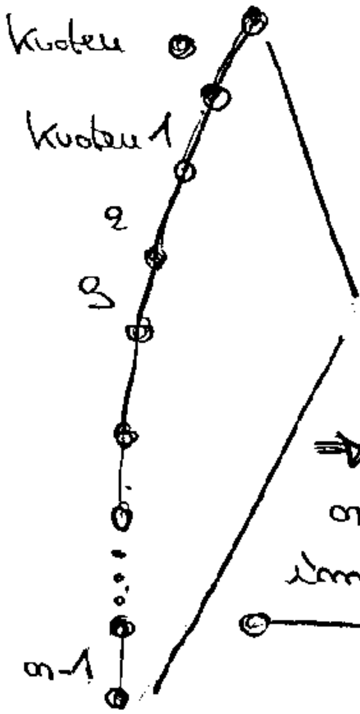
Von Eingabe
abhängig!

Konstant, programmabhängig.

(9.9)

Außerdem einige int Variablen, boolean Variablen usf.

Wie groß wird der Laufzeitkeller?



n^2 Kanten,
$n+1$ Knoten
⇒
$n+1$ Rahmen im Laufzeitkeller.

Pro Rahmen eine konstante Zahl von Speicherplätzen. Dann insgesamt

$$c n^2 + n + c(n+1) + d$$

also

$$n^2 + n + cn + (d+c).$$

Quadratisch. graph

F

Laufzeitkeller

Rest.

Oberhalb Baum groß (maximal 3^n) nur $n+1$ Rahmen.

Bei Zeitangaben wird das übliche dezimale System benutzt:

1 Millisekunde (ms) $: \frac{1}{10^3} = \frac{1}{1000}$ sek.

1 Mikrosekunde (μs) $: \frac{1}{10^6} = \frac{1}{1.000.000}$ sek

1 Nanosekunde (ns) $: \frac{1}{10^9} = \frac{1}{1.000.000.000}$ sek

:

1 Hertz = 1 Schritt pro Sekunde

2 Hertz = 2 Schritt pro Sekunde

1 MegaHertz = 1024 (1000) Schritt

1 Gigahertz = 1.000.000.000 Schritt pro Sekunde

$\Rightarrow$ 1 Schritt = $\frac{1}{10^9}$ Sekunden

$\Uparrow$

1 Nanosekunde.

Hertzangabe: 1 Taktschritt in 1 Nanosekunde.

Bei Taktfrequenz von 1 Gigahertz.

1 Taktschritt = 1 elementare Aktion.

1 Instruktionszyklus = Eine endliche Zahl von Taktschritten.

1. Befehl gemäß Befehlszähler holen.
2. Befehl interpretieren
3. Operanden holen
4. Befehl ausführen
5. Befehlszähler auf Adresse des nächsten Befehls.

Im Nanosekunden Bereich.

In der Informatik: Millisekunde eher lang.

Latenzzeit der Festplatte = Zeit von Anforderung bis Daten der Festplatte im Hauptspeicher sind

Millisekunden Bereich

Zeitverbrauch von unseren Programmen.

PRIM.java Schleifendurchläufe für

$d=1, d=2, \ldots$

$d=d_0$, wobei

$d_0 \leq \sqrt{q}$ aber $(d_0+1) \geq \sqrt{q}$.

Also

$\#$ Schleifendurchläufe $\leq \sqrt{q}$

Klar von der Eingabe, hier q bezeichnet, abhängig.

Was passiert in einem Schleifendurchlauf?

Wie lang dauert einer?

1.

Die while-Schleife wird folgendermaßen übersetzt:

Load d // Sp.-platz d in
// Register (Akkumulator).

mult d // Mult von Sp.-platz d
// mit Akk. im Akk.

c ist von der Eingabe.

Store e // Ergebnis in Sp-platz e
// abspeichern.

d*d ≤ c

Analog in f e-c ausrechnen

Load f // d.d-c in Register

Jgez E // Reg >= 0, dann Ende
// (Jump greater equal zero)

Bis hier Kopfzeile der while Schleife.

Jetzt der
Rumpf

c % d in Speicherplatz
m ausrechnen

Load m

Jgz F // Sprung wenn Register > 0
// (Jump greater zero)

if(c%d)
{...}

Übersetzung von
System.out.println(--)

Jp E // Unbedingter Sprung
// ans Ende.

F : Load d

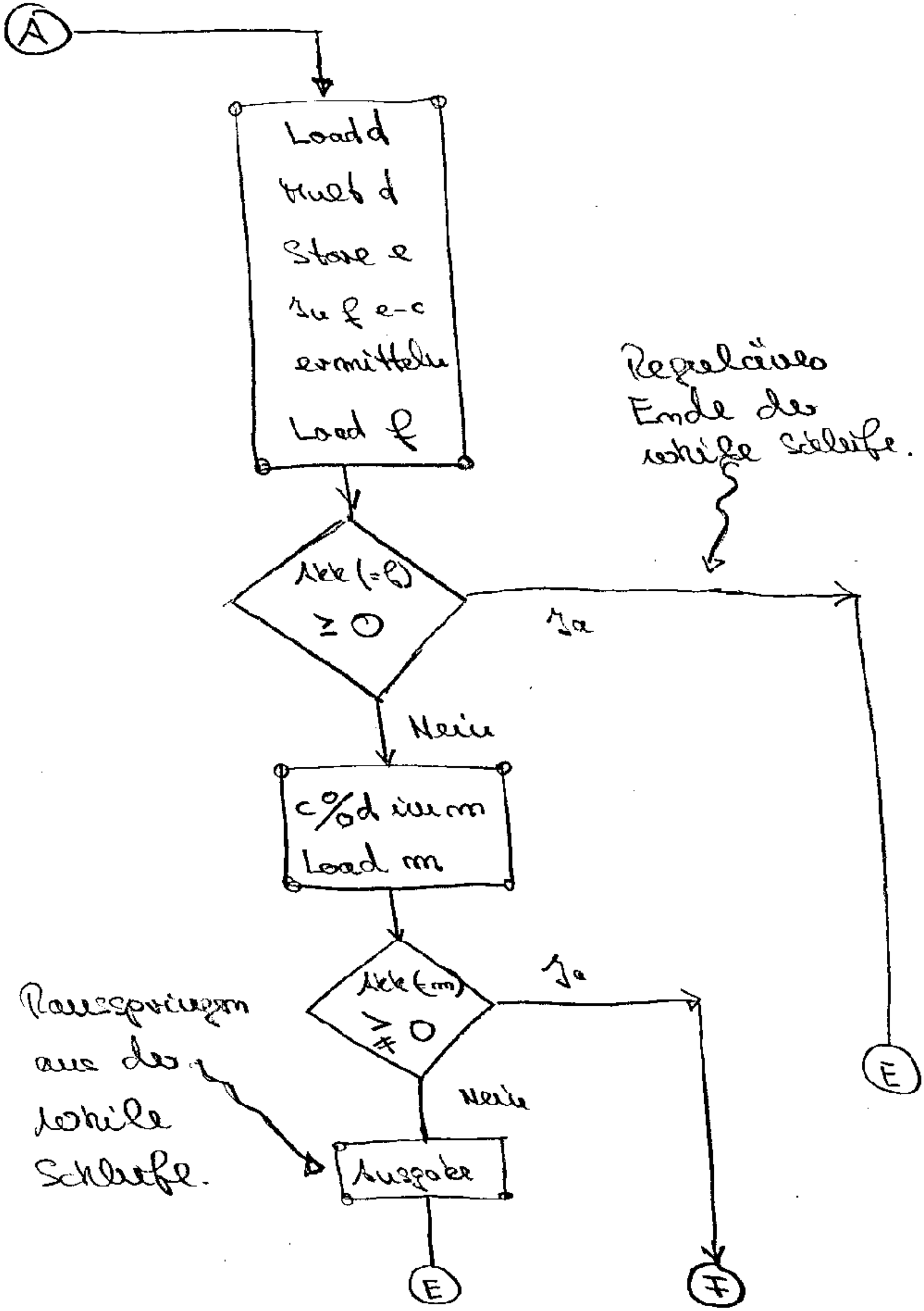
d++

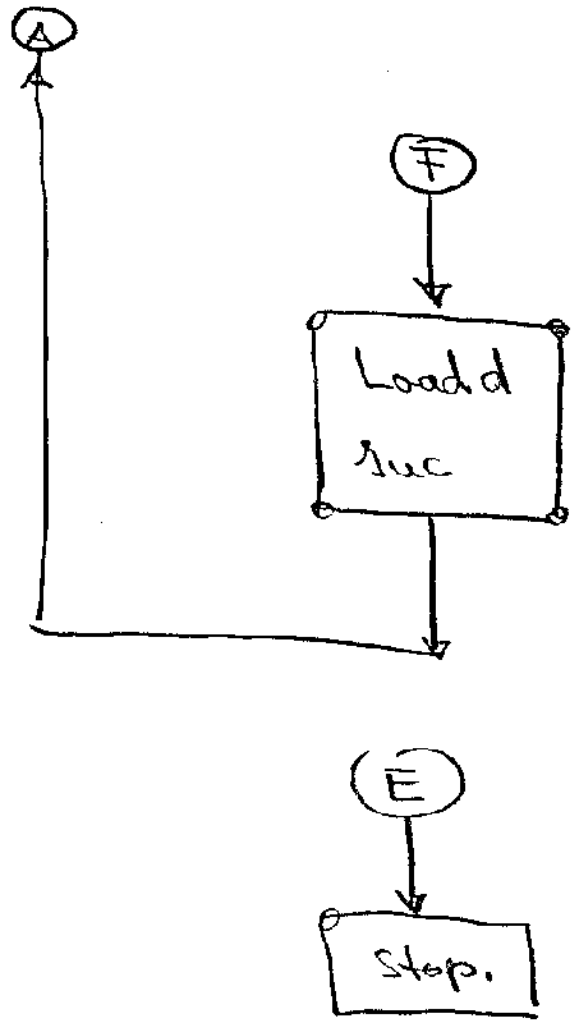
inc // Register erhöhen

Jp A // Zum Anfang springen.

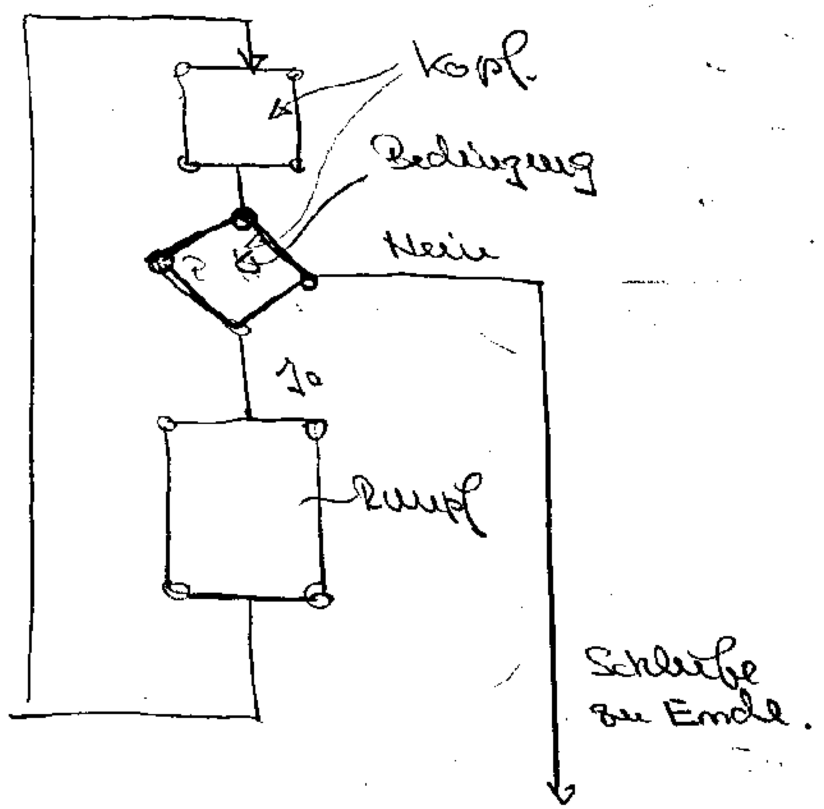
E : Stop

Als Flußdiagramm:

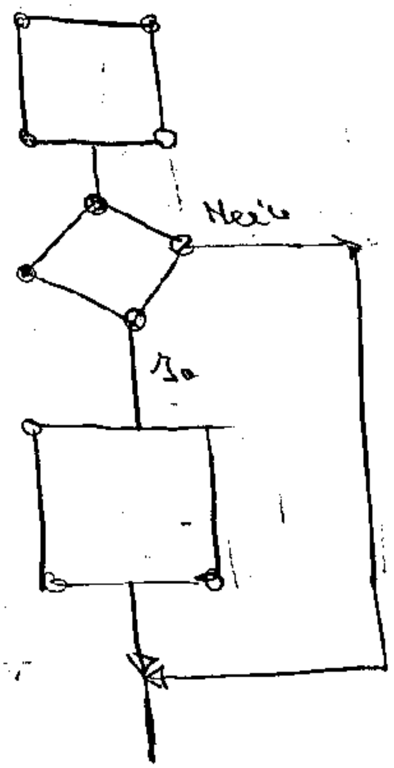




while Schleife



if - then



Beobachtung: Jede einzelne Zeile des Java Programms führt zur Abarbeitung einer konstanten (d.h. unabhängig von c) Anzahl von Maschinenbefehlen.

Damit gilt für unser Programm

\# Ausführungen von Maschinenbefehlen bei Eingabe von q

$$\leq \underbrace{d \cdot \sqrt{q}}_{\text{Schleife}} + \underbrace{f}_{\text{Anfang und Ende.}}$$

Ist etwa q durch 3 teilbar sind wir schneller fertig, also $\leq$.

Also auch

Zeit bei Eingabe von q:

$$\leq \left(d \cdot \sqrt{q} + f\right) \cdot \text{Zeit für einen Befehl}$$

Nanosekunden

$$= \left(d \cdot \sqrt{q} + f\right) \cdot z \text{ ns}$$

Und auch

\# durchlaufene elementare Programmzeilen

$$\leq \underbrace{d' \cdot \sqrt{q}}_{\text{Schleife}} + \underbrace{f'}_{\text{Rest}}.$$

Fazit: Laufzeit entspricht der # ausgeführten elementaren Programmzeilen. Ob in ms oder in Maschinenbefehlen nur Unterschied um einen konstanten programmabhängigen, eingabeunabhängigen Faktor.

Je komplizierter die Zeilen umso größer.

Definition ($\mathcal{O}$ - Notation)

Ist $f, g : \mathbb{N} \to \mathbb{R}$ (oder auch $f, g : \mathbb{R} \to \mathbb{R}$) so sagen wir

$f(n)$ ist von $\mathcal{O}(g(n))$

($f(n)$ hat Wachstums (Größen-) ordnung höchstens $g(n)$)

gdw. gilt:

Es gibt eine Konstante $c \geq 0$, ein $n_0 \in \mathbb{N}$ so daß

$$f(n) \leq c \cdot g(n) \quad \text{für alle } n > n_0$$

Ist $f(n) < 0$, so ist $|f(n)|$ gemeint.

Ab n_0 sozusagen

□

2.21

Die Laufzeit unseres Primzahl-programms ist also $O(\sqrt{n})$, wenn $n = q$ die Eingabe ist.

Es ist

$$d \cdot \sqrt{n} + f \le \overbrace{(d+1)\sqrt{n}}^{O(\sqrt{n})}$$

für $\sqrt{n} > f$, also $n_0 > f^2$ gilt.

Jetzt einmal das Programm Beisp. java. Ganzzahlige Wurzel nach dem Muster der binären Suche.

Zeit bis zum Beginn der Schleife:

konstant $\leq d$, man sagt $O(1)$.

Wieviele Läufe maximal durch die while Schleife? Suchintervall

$[0, n+1)$, n Eingabe,

$[0, n/2)$, $[n/2, n+1)$,

$[0, \frac{n}{4})$ $[\frac{n}{4}, \frac{n}{2})$ $[\frac{n}{2}, \frac{n}{2} + \frac{n+1-n/2}{2})$

$[\quad , n+1)$

Ist $n+1 = 9$, dann

9 Elemente im Suchintervall.

≤ 5 Elemente

≤ 3 Elemente

≤ 2 Elemente

1 Element, Ende.

(9.23)

Beobachtung: Hat ein Suchintervall m Elemente, also $[q, q+m)$, dann haben die nächsten Suchintervalle

$$\left\lfloor \frac{m}{2} \right\rfloor \text{ und } \left\lceil \frac{m}{2} \right\rceil \leq \frac{m}{2} + 1$$

Elemente. Also bekommen wir für unsere Situation:

$m = n+1$ Elemente

$$\leq \frac{m}{2} + 1 \qquad \text{"}$$

$$\leq \frac{\frac{m}{2} + 1}{2} + 1 = \frac{m}{4} + \frac{1}{2} + 1$$

$$\leq \frac{m}{8} + \frac{1}{4} + \frac{1}{2} + 1$$

$$\vdots$$

$$\leq 1 + \frac{1}{2^{\ell}} + \frac{1}{2^{\ell-1}} + \dots + \frac{1}{2} + 1$$

wobei $l+1 \ge \lceil \log m \rceil$ ist.

Nun ist

$$\sum_{i=1}^{l} \left(\frac{1}{2}\right)^i \lneq \frac{1}{2} \cdot \frac{1}{1-\frac{1}{2}} = 1$$

(geometrische Reihe). Also ist die Anzahl von Elementen ≤ 2 und wir machen noch einen Lauf durch die Schleife.

Damit ist die Laufzeit von Binf. java

Pro Lauf konstant viel.

$$f + d \cdot \underbrace{\left(\log(n+1) + 1\right)}_{\#\,\text{Läufe}}$$

Rest, Anfang, Ende.

(2.25)

Das ist $O(\log n)$. Denn

es ist

Konstante

$\rightarrow f \quad O(\log n)$, da $\log n \to \infty$

Weiter ist

$\log(n+1)$

$\leq$ Für $n \geq 1$ ist $n+1 \leq 2n$.

$\log(2n)$

$=$

$(\log n) + 1$

$\leq$ Für $n \geq 2$ ist $\log n \geq 1$.

$2 \cdot (\log n)$,

also $\log(n+1)$ ist $O(\log n)$.

Schließlich ist

$$d \quad O(\log n).$$

Also haben wir etwas von der Form

$$\underset{\overset{\cup}{\ell}}{O(\log n)} + \underset{\overset{\cup}{d \cdot \log(n+1)}}{O(\log n)} + O(\log n)$$

Das ist aber

$$3 \cdot O(\log n),$$

also

$$O(\log n).$$

Das Programm Eukl. java,

Euklidische Algorithmus mit

modulo Operation.

Eingabe $g \geq 2$, $h \geq 2$.

Wird im Laufe der while

Schleife $g = 1$ oder $h = 1$

kommt es noch zu einem Lauf

der Schleife ; d.h. $O(1)$ Zeit,

und der ggT ist 1.

Schleife:

while ($g>0$ & $h>0$)

if ($g>h$)

$g = g \% h$; // Danach $g \lneq h$.

if ($h>g$)

$h = h \% g$; // Danach $h \lneq g$.

Für die $g\%h$, $h\%g$ oben

gilt $g\%h \leq g/2$, $h\%g \leq h/2$.

Denn da $g>h$, ist

$$g = \underbrace{\underbrace{g/h}_{\geq 1} \cdot h}_{> g\%h} + \underbrace{g\%h}_{<h} > 2 \cdot g\%h$$

Ebenso für $h\%g$.

Wir bekommen folgende Werte:

$$g_0 > h_0$$

$$\leq g_0/2 \quad \leq h_0/2$$

$$\leq g_0/4 \quad \leq \frac{h_0}{4}$$

$$\vdots$$

Also nach dem l-ten Lauf ist der Wert auf $g \leq \frac{g_0}{2^l}$.

Nun ist $\frac{g_0}{2^l} \leq 1 \Leftrightarrow \log_2 g_0 \leq l$.

Also Laufzeit $O(\log_2 g)$!

Bei Eingabe $g \geq h > 0$ und % mit konstant vielen Masch-befehlen.

Hanoi.java

Methodenaufruf, welche Zeit?

Laufzeitstelle verwalten

+ Parameter übergeben + Rückgabewert übg.

+ Zeit im Rumpf

Laufzeitstelle: Rahmen einrichten,

Rahmen löschen: $O(1)$

Parameter übergeben: $O(1)$

(Kein Kopieren von Feldern,

Instanzen von Klassen!)

Rückgabewert: $O(1)$

Alle Konstanten natürlich

programmabhängig.

Hanoi(A, B, C, n)

H(.., n-1)   Umsetzen   H(.., n-1)

T_n = # Knoten des Aufrufbaums von H(... n).

Dann gilt

$$T_1 = 1$$

$$T_{n+1} = 1 + 2 \cdot T_n$$

Dann gilt induktiv

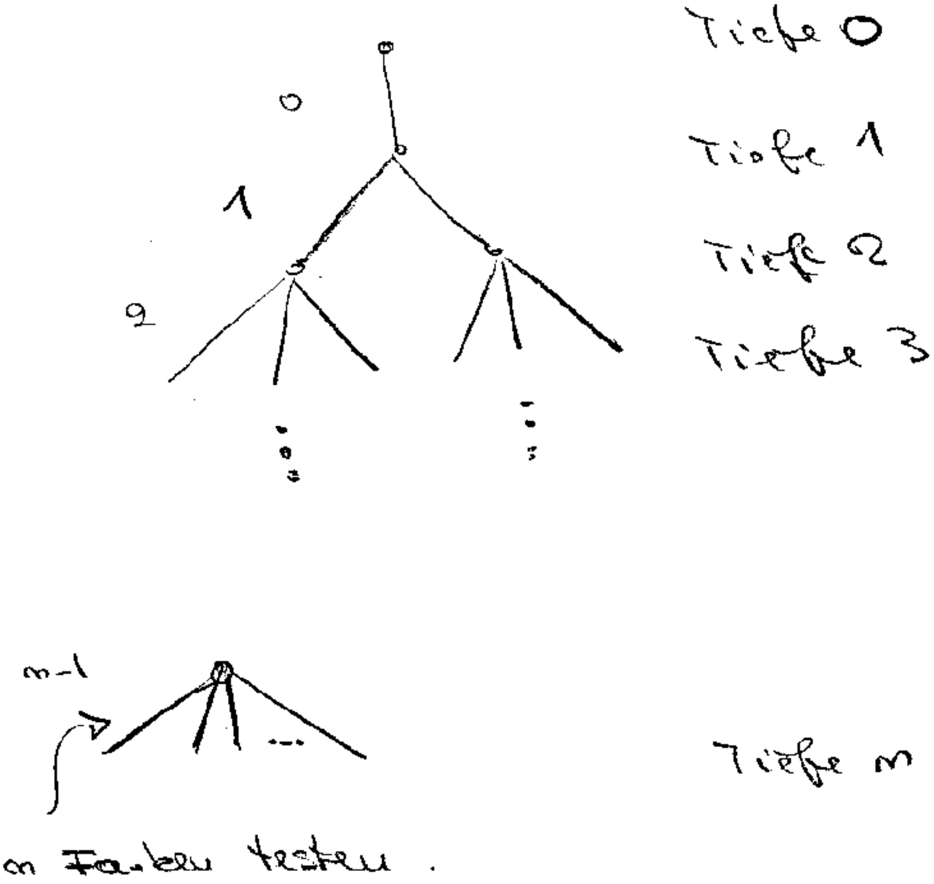
$$T_n = 2^n - 1$$

Pro Knoten $O(1)$ Zeit,

also Zeit $O(2^n)$.

Schließlich Farb.java

Aufrufbaum, schlimmstenfalls:



(1.33)

Wieviele Knoten hat der Baum?

Der Baum hat ...

$\#$ Knoten in Tiefe $l = l!$

$l=0$ ✓, $l=1$ ✓, $l=2$ ✓

$l>2$, dann

$l \cdot \#$ Knoten in Tiefe $l-1$

$=$ Ind.-Vor.

$l \cdot (l-1)!$

$= l!$ Knoten in Tiefe l.

$\sum$

Insgesamt

$$\sum_{l=0}^{m} l!$$

$$= m! + \underbrace{(m-1)! + (m-2)! + \dots + 3! + 2! + 0!}_{\le m \cdot (m-1)!}$$

$\le 2(m!)$ viele Knoten.

Dann Zeit von $O(m!)$

ist hinreichend.

Bei allen Farben mit m Knoten

sind wir sicher in $O(n!)$ fertig.

Worst-case Zeitaufwand.

Wir betrachten folgenden Aspekt einer Laufzeitfunktion $f(n)$.

Sei eine Zeit x gegeben. Ist

$$f(n_0) = x$$

bedeutet das, wir können Eingaben bis zur Größe n_0 (setzen $f(n)$ monoton) in Zeit x lösen.

Nehmen wir jetzt an, daß sich die Rechnergeschwindigkeit verdoppelt. Die Zeit ist also $\frac{1}{2} \cdot f(n)$.

Wir fragen wieder nach dem n_1 mit

$$\frac{1}{2} f(n_1) = x \text{ oder } f(n_1) = 2x.$$

3.36

$f(n)$	n_0	n_1
$\log_2 n$	2^x	$2^{2x} = (2^x)^2$
$c \cdot n$	x/c	$2 \cdot (x/c)$
n^k	$\sqrt[k]{x}$	$\underbrace{\sqrt[k]{2}}_{\text{Konstant} > 1.} \cdot \sqrt[k]{x}$
2^n	$\log_2 x$	$(\log_2 x) + 1$
$n^{\log\log n}$	$x^{\frac{1}{\log\log x}}$	$\underbrace{2^{\overbrace{\frac{1}{\log\log x}}^{\to 0}}}_{\to 1} \cdot x^{\frac{1}{\log\log x}}$ $2^0 = 1$
$n!$	kaum Vergrößerung feststellbar.	

Dagegen etwa: Besserer

Algorithmus

$$x = 2^{u_0} \quad , \quad x = 2^{\frac{1}{2} u_1} \quad ,$$

dann

$$m_0 = \log_2 x \quad , \quad m_1 = 2 \cdot \log_2 x$$

Verdopplung.

Ebenso

$$x = m_0^4 \qquad x = m_1^2$$

dann

$$m_0 = \sqrt[4]{x} \qquad m_1 = \sqrt[2]{x} = \sqrt[4]{x} \cdot \sqrt[4]{x}$$

Quadrierung.

Also: Besserer Rechner

und bessere Algorithmen!

9.38.

$k(N)$	Bezeichnung	10	100	1000	10^4	10^5	10^6
1	konstant	1	1	1	1	1	1
$\log(N)$	logarithmisch	3	7	10	13	17	20
$\log^2(N)$		10	50	100	170	300	400
$\sqrt{N}$		3	10	30	100	300	1000
N	linear	10	100	1000	10^4	10^5	10^6
$N \cdot \log(N)$	log-linear	30	700	10^4	10^5	$2 \cdot 10^6$	$2 \cdot 10^7$
$N^{3/2}$		30	1000	$3 \cdot 10^4$	10^6	$3 \cdot 10^7$	10^9
N^2	quadratisch	100	10^4	10^6	10^8	10^{10}	10^{12}
N^3	kubisch	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
2^N	exponentiell	1000	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{300000}

Abb. 4-4: Häufig auftretende Komplexitäten und ungefähre Werte für C=1.

k konstant.

(9.33)

Polynomial n^k. Beschleunigung des Rechners um konstanten Faktor c

$\Rightarrow$ Vergrößerung der Eingaben in gegebener Zeit um konstanten Faktor ($\sqrt[k]{c}$).

Exponentiell $2^{\varepsilon n}$, $\varepsilon > 0$; c^n, $c > 1$.

oder auch $n^{\varepsilon \frac{n}{\log n}}$, $\varepsilon > 0$.

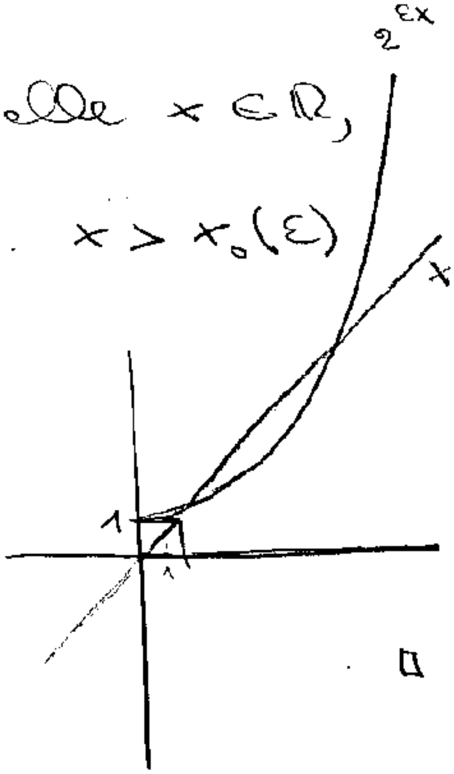
Analog wie oben: Nur Vergrößerung des Eingaben um konstanten Summanden.

Tatsächlich ist exponentiell immer größer als polynomial (für n hinreichend groß).

Satz

Sei $\varepsilon > 0$ fest. Für alle $x \in \mathbb{R}$, x hinreichend groß, d.h. $x > x_0(\varepsilon)$ für ein $x_0(\varepsilon)$, gilt

$$2^{\varepsilon \cdot x} > x.$$



□

Daraus folgt das ganze Formelkram „exponentiell größer polynomial":

Für alle $\varepsilon > 0$ konstant, alle $z > 0$ konstant, alle $x > x_0(\varepsilon, z)$

$$(1) \quad 2^{\varepsilon \cdot x} > x^z.$$

Interessanter Trick, einen Exponenten loszuwerden.

Folgt, da gilt $2^{\frac{1}{z}\varepsilon \cdot x} > x$ indem der letzte Satz mit $\frac{1}{2} \cdot \varepsilon > 0$ statt ε verwendet wird. Die Operation „hoch 2" ist monoton, erhält also die Ungleichung. Dann insbesondere $2^{\varepsilon \cdot x}$ nicht $\Theta(x^2)$.

Unter den Voraussetzungen wie vorher gilt dann auch

$$x^{\varepsilon} > (\log x)^2 . \qquad (2)$$

Das folgt, da auch $\log_2 x \to \infty$ geht und deshalb mit (1) für hinreichend großes x,

(groß genug, daß $\log_2 x$ groß genug)

gilt

$$x^{\varepsilon} = 2^{\varepsilon \log_2 x} > (\log_2 x)^{7}.$$

Dann ist insbesondere

$$x^{\varepsilon} > c \cdot \log_2 x$$

für jede Konstante c, x groß genug.

Daraus ergibt sich wegen der

Monotonie von „2 hoch"

$$2^{(x^{\varepsilon})} > 2^{\log_2 x \cdot c} = x^{c} \qquad (3)$$

für alle $\varepsilon > 0$, c.

Beweis des Satzes von S. 9.40

Mit Analysis zeigen wir:

$$2^{\varepsilon\cdot x} - x \to \infty .$$

Dazu zeigen wir, daß $2^{\varepsilon\cdot x} - x$ streng monoton steigend ist, dazu, daß die Ableitung > 0 ist für x groß genug. Es ist

$$2^{\varepsilon\cdot x} = e^{\overbrace{\ln 2}^{<1}\cdot\varepsilon\cdot x}$$

und die Ableitung ist (Kettenregel)

$$\ln 2\cdot\varepsilon\cdot e^{\ln 2\cdot\varepsilon\cdot x} = (\ln 2)\cdot\varepsilon\cdot 2^{\varepsilon\cdot x} \to \infty$$

Die Ableitung von $-x$ ist -1.
Die Behauptung gilt.

Es geht auch ohne Analysis:

Dazu zeigen wir, daß für alle

$m \in \mathbb{N}$, hinreichend groß

$$2^{\varepsilon \cdot m} > m+1 \qquad (4)$$

ist. Dann folgt für $x \in \mathbb{R}$, groß genug,

Abrunden nach unten.

$$2^{\varepsilon x} \geq 2^{\varepsilon \lfloor x \rfloor} > \lfloor x \rfloor + 1 \geq x,$$

Wegen Obigem.

was die Behauptung des Satzes ist.

Die Behauptung (4) folgt aus

$$2^{m} > c \cdot m + 1, \qquad (5)$$

für alle $c \in \mathbb{R}$ und m groß genug, denn setzen wir $c = 1/\varepsilon$ gilt

$$2^n > 1/\varepsilon \cdot n + 1$$

Ein wichtiger Trick: Der Substitutionstrick!

für alle n groß genug. Da:

$\varepsilon n \to \infty$ können wir substituieren

εn für n

$$2^{\varepsilon n} > \underbrace{n}_{= 1/\varepsilon \cdot \varepsilon n} + 1.$$

(5) ist nun auch einfach: Für

$c \geq 5$ ist $2^c > c^2 + 1$: gilt für

$c = 5$ und $(c+1)^2 + 1 = c^2 + 2c + 1 + 1$

Es ist $2^c \underset{\text{Ind.-Vor.}}{>} c^2 + 1 \underset{c \geq 5}{>} 2c + 1$. Also

$$2^{c+1} > (c+1)^2 + 1$$

und für alle $c \geq 5$ $2^c > c^2 + 1$.

Nun eine Induktion über n.

Ind.-Anf. $n = c$, dann

$$2^c > c^2 + 1,$$

Hier sieht man: n muß hinreichend groß sein

wie eben gezeigt. Nun zum

Ind.-Schluß: Sei $n \geq c$ fest.

$$2^n > cn + 1$$

nach Ind.-Vor. Dann

Für $k>1$ ist deutlich: $n^k > c \cdot n$. Das impliziert $n > c^\varepsilon n^\varepsilon$, impliziert $\delta n > n^\varepsilon$ für $\varepsilon < 1$

$$\begin{aligned} 2^{n+1} &> 2 \cdot 2^n \\ &\overset{\text{Ind.-Vor}}{>} (cn+1) + \underbrace{(cn+1)}_{\geq c} \qquad n \geq 1 \\ &> cn + 1 + c \\ &= c(n+1) + 1. \end{aligned}$$

□

Einige kombinatorische Formeln:

$n! = n(n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$ =

Bijektive Abbildungen zwischen
2 n-elementigen Mengen, =
Permutationen auf $1, \ldots, n$.

Induktion über n.

Keine Bijektionen
oder so etwas.

k^n =

Abbildungen von M nach N
mit $|M| = n$, $|N| = k$.

Induktion über n. Für jedes
weitere Element k neue
Möglichkeiten.

2^n.

\# Abbildungen von M mit $|M| = n$
nach $N = \{0,1\}$.

\# Teilmengen von M, $|M| = n$.

$i \in M$	$f(i) \in \{0,1\}$
1	0
2	1
3	1
$\vdots$	0
$\vdots$	$\vdots$
$\vdots$	1
n	1

Dann entspricht $f: M \to \{0,1\}$
der Teilmenge
$$T = \{ i \in M \mid f(i) = 1 \}.$$

Für $k > n$ ist

$$\binom{n}{k} = 0$$

Für $k \leq n$ ist

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\cdot\ldots\cdot(n-k+1)}{k!}.$$

$\#$ Teilmengen mit genau k Elementen

von M mit $|M| = n$.

Induktion mit

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Pa-

Pascalsches Dreieck

$n \backslash k$

$$0 \qquad \binom{0}{0} = 1 \qquad 0$$

$$\binom{1}{0} = 1 \qquad \binom{1}{1} = 1$$

$$\binom{2}{0} = 1 \qquad \binom{2}{1} = 2 \qquad \binom{2}{2} = 1$$

$$\binom{3}{0} = 1 \qquad \binom{3}{1} = 3 \qquad \binom{3}{2} = 3 \qquad \binom{3}{3} = 1$$

$$\binom{4}{0} = 1 \qquad \binom{4}{1} = 4 \qquad \binom{4}{2} = 6 \qquad \binom{4}{3} = 4 \qquad \binom{4}{4} = 1$$

$$\vdots$$

Geometrische Reihe:

$$\sum_{i=0}^{n} q^i = \frac{q - q^{n+1}}{1-q} \quad \text{für } q \neq 1$$

$$\sum_{i=0}^{n} q^i = n+1 \quad \text{für } q = 1.$$

Für $|q| < 1$ ist $q^{n+1} \to 0$ in n.

Also dann

$$\sum_{i=0}^{\infty} q^i = \lim_{n\to\infty} \sum_{i=0}^{n} q^i$$

$$= \lim_{n\to\infty} \frac{1-\overbrace{q^{n+1}}^{\to 0}}{1-q} = \frac{1}{1-q} \quad !$$

$q = 1/2$, dann

$$\sum_{i=0}^{\infty} q^i = \frac{1}{1-\frac{1}{2}} = 2$$

$= 1 + \frac{1}{2} + \frac{1}{4} \cdots$

Daher gilt

$$\sum_{i=1}^{\infty} q^i = \frac{1}{2}\left(\frac{1}{1-\frac{1}{2}}\right) = 1 .$$