

Aufwandsschätzung

Programm A (berechnet ggt(n,m))

```
for (i=1; i<=n; i++)
    if (m%i==0 && n%i ==0) ggt=i;
```

Programm B (berechnet ggt(n,m))

```
while (n!=m)
    if (n<m) n=m%n else m=n%m;
ggt=m;
```

Welches ist besser?

Aufwandsschätzung

Programm C (sortiert natürliche Zahlen $x[1..n]$)

```
for (max=0, i=1; i<=n; i++)
    if (x[i]> max) max=x[i];
s = new int[max];
for (i=0; i<=max; i++) s[i]=0;
for (i=1; i<n; i++) s[x[i]]++;
for (i=0, j=0; i<=max; i++)
    while (s[i]>0) x[j++]=i;
```

} finde Maximum (n Durchläufe)
} neues Feld der Länge max initialisieren
} zählen wie oft welche Zahl in x
} x neu auffüllen

Insgesamt $3n+\max$ Schleifendurchläufe zum Sortieren von n Zahlen $\in \mathbb{N}$
Taugt das Programm was? Unter welchen Umständen ist es sinnvoll?

Wozu Aufwandsschätzung

- Bewertung der Qualität von Algorithmen (zumindest der Qualitätsdimension Laufzeit)
- Laufzeitoptimierung:
Feststellen, wo gegebene Algorithmen ihre Schwachstellen haben, Verschnellern der Programmausführung
- Bewertung der Schwierigkeit von Problemen



Aufwände

Speicherplatz

Platz für das Programm selbst

```
for (...); if (...); while (...) { ... }
```

Platz für seine statischen Daten (Teil des Programms)

```
int x[100]; String h = "Hallo World";
```

Platz für seine dynamischen Daten (entstehen zur Laufzeit)

```
x = new float[max]; x = (char*)malloc(100);
```

Laufzeit

Ausführungszeit des Programms selbst

in der Praxis auch: Zeit für I/O, Zeit für System

Größenordnungen

- Exakte Zahl (#Rechenschritte, Zeit in μs) ist sowieso kaum genau berechenbar.
 - Linearer Faktor interessiert nicht so sehr (in der Theorie). Verschiedene Rechner sind „um Faktor“ unterschiedlich schnell.
 - Es gibt best-case, mittlere Zeit und worst-case. Wir interessieren uns v.a. für den worst-case.
 - Laufzeit hängt i.d.R. von der Größe der Eingabe ab. Wir interessieren uns v.a. für Eingabe gegen unendlich groß.
- ⇒ Exakte Anzahl Schritte i.d.R. uninteressant, wichtiger ist ungefähre Größenordnung

Maschinenmodelle

Offensichtlich:

Laufzeit hängt vom Rechnermodell ab (nicht nur linearer Faktor)

z.B. Turing-Maschine (kann nicht jederzeit auf gesamten Speicher zugreifen) vs. Random-Access-Machine (kann das).

Daher für Theorie: ein Standard-Maschinenmodell

Annahmen:

- eine Zeiteinheit je elementarer Operation (egal welche)
- Arithmetische Operation, Daten auslesen oder schreiben
- beliebig große Zahlen in einer Operation verwendbar
- Random Access Machine (RAM) mit einem Prozessor

Laufzeitberechnung

Beispiel: gegeben sortiertes Feld $a[0] \dots a[n-1]$ von Integers,
 gesucht: 1. Position der Zahl x (oder n wenn nicht da)

1. Programm: (sei k die gesuchte Position)

```
i=0;
while (a[i] != x) i=i+1;
print i
```

Laufzeit: 1 Schritt für $i=0$;
 $(k+1) \cdot ($ 1 Schritt für $a[i]$
 1 Schritt für $!=$)
 $k \cdot ($ 1 Schritt für $i+1$
 1 Schritt für $i=\dots;$)
 1 Schritt für $print i$

} insgesamt
 $1+2k+2(k+1)+1$
 $= 4k+4$

Wie viele Schleifendurchläufe?

Schwieriger Fall:

```
i=123; j=i+1;
while (i<j)
{
    i = i - j%7 + j%3;
    j = j - i%5 + i%11;
}
```

Erinnerung: man kann nicht alles berechnen

Laufzeitberechnung

1. Programm: Suche (sei k die gesuchte Position, bzw. n wenn nicht da)

```
i=0;
while (a[i] != x) i=i+1;
print i
```

benötigt
 $4k+4$ Schritte

Minimale Laufzeit ist also: (für $k=0$) 4 Schritte
 Maximale Laufzeit ist: ($k=n$) $4n+4$ Schritte

Erwartungswert für Laufzeit: ($k=n/2$) $2n+4$ Schritte

Die maximale Laufzeit ist in etwa proportional zur Länge der Eingabe
 \Rightarrow wir sagen: das Programm hat „**linearen Aufwand**“

Laufzeitberechnung

2. Programm: Suche (sei k die gesuchte Position, bzw. n wenn nicht da)

```
l=0; r=n-1;
while (l<=r)
    m=(l+r)/2;
    if (a[m]==x) { print m; return; }
    if (a[m]<x) l=m+1; else r=m-1;
```

3 Schritte für $l=0; r=n-1$;
 $k \cdot ($ 1 Schritt für $l<=r$
 3 Schritte für $m=(l+r)/2$;
 3 Schritte für $if (a[m]==x)$
 3 Schritte für $if (a[m]<x)$
 2 Schritte für $l=m+1$; oder $r=m-1$;)
 $1+3+3+2$ Schritte für letzten Schleifendurchlauf

} insgesamt
 $12k+12$
 Schritte bei
 k kompletten
 Schleifen-
 durchläufen

Laufzeitberechnung

2. Programm: Suche (sei k die gesuchte Position, bzw. n wenn nicht da)

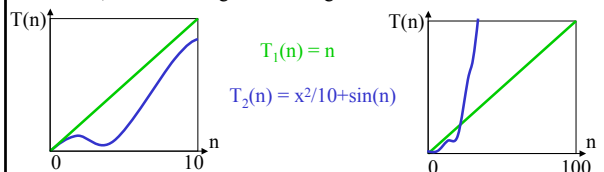
```
l=0; r=n-1;
while (l<=r)
    m=(l+r)/2;
    if (a[m]==x) { print m; return; }
    if (a[m]<x) l=m+1; else r=m-1;
```

$12k+11$ Schritte, aber wie viele Schleifendurchläufe?
 mindestens 0 mal \Rightarrow mindestens 12 Schritte
 höchstens $\lceil \log_2 n \rceil$ mal $\Rightarrow 12+12 \cdot \lceil \log_2 n \rceil$ Schritte

Die maximale Laufzeit ist in etwa proportional zum Logarithmus
 der Länge der Eingabe
 \Rightarrow wir sagen: das Programm hat „**logarithmischen Aufwand**“

Asymptotische Zeitkomplexität

sei $T(n)$ die Zahl der Schritte, die ein Programm bis zur Terminierung
 ausführt, wenn die Eingabe die Länge/Größe n hat.



kleine Wackler (wie $\sin n$) interessieren nicht, es gilt schließlich

$$\lim_{n \rightarrow \infty} \frac{n^2/10 + \sin n}{n^2/10} = 1$$

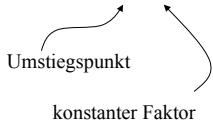
i.d.R. interessieren wir uns für das
 Verhalten für sehr große n , entspricht
 etwa der Asymptote von $T(n)$

Obere Schranken

wir definieren zu einer Funktion $f(n)$ eine Menge $O(f(n))$ von Funktionen (Ordnung von $f(n)$) für die gilt:

g ist in $O(f(n))$ wenn

$$\exists N > 0, k > 0: \forall n \geq N: g(n) \leq k \cdot f(n)$$



Ab Umstiegspunkt ist $g(n)$ kleiner als $f(n)$ multipliziert mit einer festgelegten Konstante.

O-Notation

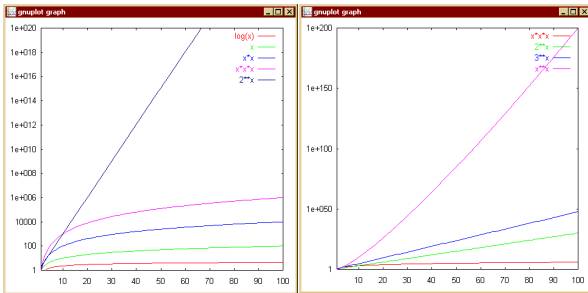
Beispiele:

Funktion ist in $O(\quad)$ Schrankenfunktion

| | |
|----------------------|------------|
| n^2+n | n^2 |
| n^2+n | $2n^2$ |
| $7n^3-2n^2+13n+201$ | n^3 |
| $2\log_7(n+1)$ | $\log(n)$ |
| $\sqrt{n} + \log(n)$ | \sqrt{n} |
| 3^n+2^n | 3^n |

Wir schreiben statt „ $g(n)$ ist in $O(f(n))$ “ einfach $g = O(f(n))$

Ein paar Komplexitätsklassen



Verschiedene Zeitkomplexitäten

Superrechner mit 1 Giga-Operationen pro Sekunde

| Aufwand | max. Problem in 1 Sekunde | in 1 Tag | in 1 Jahr | Beispielproblem |
|--------------|---------------------------|-------------------|-------------------|---------------------|
| $\log_2 n$ | ca. $10^{1000000000}$ | $10^{1000000000}$ | $10^{1000000000}$ | binäre Suche |
| n | 10^9 | 10^{14} | $3 \cdot 10^{16}$ | n Zahlen addieren |
| $n \log_2 n$ | $6 \cdot 10^8$ | $5 \cdot 10^{13}$ | $2 \cdot 10^{16}$ | sortieren |
| n^2 | 32000 | $3 \cdot 10^9$ | 10^{12} | Vektorprodukt |
| 2^n | 30 | 46 | 55 | Hanoi-Türme |
| $n!$ | 12 | 16 | 18 | Handlungsreisender |

Problem vs. Prozessor

Wie viel größer können Probleme werden, wenn die Rechner 10 mal schneller werden?

Ursprüngliche Problemgröße = k

| | |
|--------------|---------------------------------------|
| $\log_2 n$ | $1000 \cdot k$ |
| n | $10 \cdot k$ |
| $n \log_2 n$ | $9 \cdot 10 \cdot k$ (für große k) |
| n^2 | $3 \cdot k$ |
| 2^n | $k+3$ |
| $n!$ | k (für $k \gg 10$) |

O-Kalkül

Ein paar Rechenoperationen

$f(n) = O(r(n))$ und $g(n) = O(s(n))$ dann ist $f(n)+g(n) = O(r(n)+s(n))$
 $f(n) = O(r(n))$ und $g(n) = O(s(n))$ dann ist $f(n) \cdot g(n) = O(r(n) \cdot s(n))$

aber **nicht**:

$f(n) = O(r(n))$ und $g(n) = O(s(n))$ dann $f(n)-g(n) = O(r(n) - s(n))$

auch **nicht**:

$f(n) = O(r(n))$ und $g(n) = O(s(n))$ dann $f(n) / g(n) = O(r(n) / s(n))$

o-Notation

genaue obere Schranken mit $o(f(n))$, sprich „klein-Oh“:

$o(f(n))$ definiert eine Menge von Funktionen für die gilt:

g ist in $o(f(n))$, wenn

$$\forall k > 0: \exists N > 0: \forall n \geq N: g(n) \leq k \cdot f(n)$$

Unterschied zu „groß-Oh“: jetzt nicht „es gibt irgend eine Konstante“ sondern „egal für welche Konstante“.

z.B. ist $2n^2$ in $O(n^2)$ aber nicht in $o(n^2)$, dafür in $o(n^3)$

Es gilt auch: $g(n)$ in $o(f(n)) \Leftrightarrow \lim_{n \rightarrow \infty} g(n)/f(n) = 0$

Untere Schranke, die Ω -Notation

$\Omega(f(n))$ definiert eine Menge von Funktionen, für die gilt:

g ist in $\Omega(f(n))$, wenn

$$\exists N > 0, k > 0: \forall n \geq N: k \cdot f(n) \leq g(n)$$

z.B. ist n^2 in $\Omega(n)$ und auch in $\Omega(n^2-1)$

entsprechend zu „groß-Oh / klein-Oh“ definieren wir auch ω :

wir sagen $f(n) = \omega(g(n))$ wenn $g(n) = o(f(n))$

Die Θ -Notation

wenn gilt $g(n) = O(f(n))$ und $g(n) = \Omega(f(n))$

sagen wir $g(n) = \Theta(f(n))$

d.h. $\Theta(f(n))$ definiert eine Menge von Funktionen, für die gilt

g ist in $\Theta(f(n))$, wenn

$$\exists k > 0, l > 0, N > 0: \forall n \geq N: k \cdot f(n) \leq g(n) \leq l \cdot f(n)$$

z.B. gilt $a \cdot n^2 = \Theta(n^2)$ aber nicht $a \cdot n^2 + b \cdot n = \Theta(n^2)$