

7.1 Entwurfsschema

7.2 Induktion und Teile-und-Herrsche

7.3 Probabilistische Algorithmen



Innensicht

Algorithmenentwurf und Algorithmen (incl. Aufwände)

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

Prozesse

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

Außensicht

Skalierbarkeit und Persistenz

- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

Autonome Dienste

- Enge/lose Kopplung
- Protokolle / Automaten
- Verteilung



| | |
|--------------------|---|
| Definitionsbereich | Auf was für Daten arbeitet der Algorithmus? |
| zeitlicher Aufwand | Wie schnell/langsam läuft er? |
| räumlicher Aufwand | Wie viel Speicherplatz benötigt er? |
| Terminierung | Wann stoppt der Algorithmus? Tut er das? |
| Korrektheit | Liefert er immer das richtige Ergebnis? |
| Algorithmenschema | Nach welchem Muster läuft er ab? <i>Kapitel 7</i> |
| Spezifikationen | Wie kann man sein Wirken exakt beschreiben? |
| Operationsweise | Funktional? Imperativ? Deklarativ? |
| Eleganz | Nachvollziehbarkeit? Verständlichkeit? |



Induktion

- schließe aus Lösung für $n-1$ auf Lösung für n

Teile und Herrsche

- teile Problem in einfache Teile, kombiniere Ergebnisse

Probabilistisch

- Lösung „zufällig“ erraten

Brute force

- alle Lösungskandidaten ausprobieren und nachprüfen

Schrittweises Verfeinern

- grobe Schritte _ feine Schritte

Reduktion

- Abbildung auf „ähnliches“ Problem und dieses lösen

Branch and bound/ Backtracking

- versuche verschiedene Lösungswege und verfolge die „guten“

Greedy (raffgierig)

- wähle unter allen möglichen Schritten den zunächst „besten“

Parallel

- führe mehrere Schritte gleichzeitig (nebenläufig, parallel) aus

Besondere

- indeterministisch, genetisch, neuronal, Vorberechnen, u.v.a.



Beispiel:

Gegeben: `int x, y`

Gesucht: `int z = ggT(x, y)`

Algorithmus:

```
for (int i = 1; i <= x; i++)  
if ( (x % i) == 0 && (y % i) == 0 ) z=i;
```

Rechnet offensichtlich viel zu lange für sehr große Werte von **x**.

Brute Force Lösungen bieten sich nur selten an:

- wenn kein „richtiger“ Algorithmus bekannt
- wenn Entwicklung des Algorithmus „teurer“ als CPU-Ressourcen



Beispiel:

Gegeben: `int x, y`

Gesucht: `int z = ggT(x, y)`

```
A=Faktorisierung(x)
B=Faktorisierung(y)
sortiere(A)
sortiere(B)
für i=|A|..1
  in = A[i] ∈ B
  falls in
    drucke A[i]
```

```
A=Faktorisierung(x)
B=Faktorisierung(y)
sortiere(A)
sortiere(B)
für i=|A|..1
  in = false
  für j=1..|B|
    wenn A[i]==B[j]
      in = true
  falls in
    drucke A[i]
```

```
für n=1..x/2
  falls x mod n =0
    füge n zu A
...
sortiere(A)
sortiere(B)
für i=|A|..1
  in = false
  für j=1..|B|
    wenn A[i]==B[j]
      in = true
  falls in
    drucke A[i]
```



Beispiel:

Gegeben: `int x[] = {7, 22, 39, 20, 9}`

Aufgabe: Packe möglichst viel in Rucksack mit Kapazität 50.

Optimale Lösung: $22 + 20 + 7 = 49$

Greedy Algorithmus (greedy = raffgierig):

- | | | | | |
|----|----|-------------|---------------|----------------------|
| 1. | 39 | passt | \Rightarrow | 39 drin |
| 2. | 22 | passt nicht | \Rightarrow | 39 drin |
| 3. | 20 | passt nicht | \Rightarrow | 39 drin |
| 4. | 9 | passt | \Rightarrow | 48 drin |
| 5. | 7 | passt nicht | \Rightarrow | 48 drin (suboptimal) |



Beispiel:

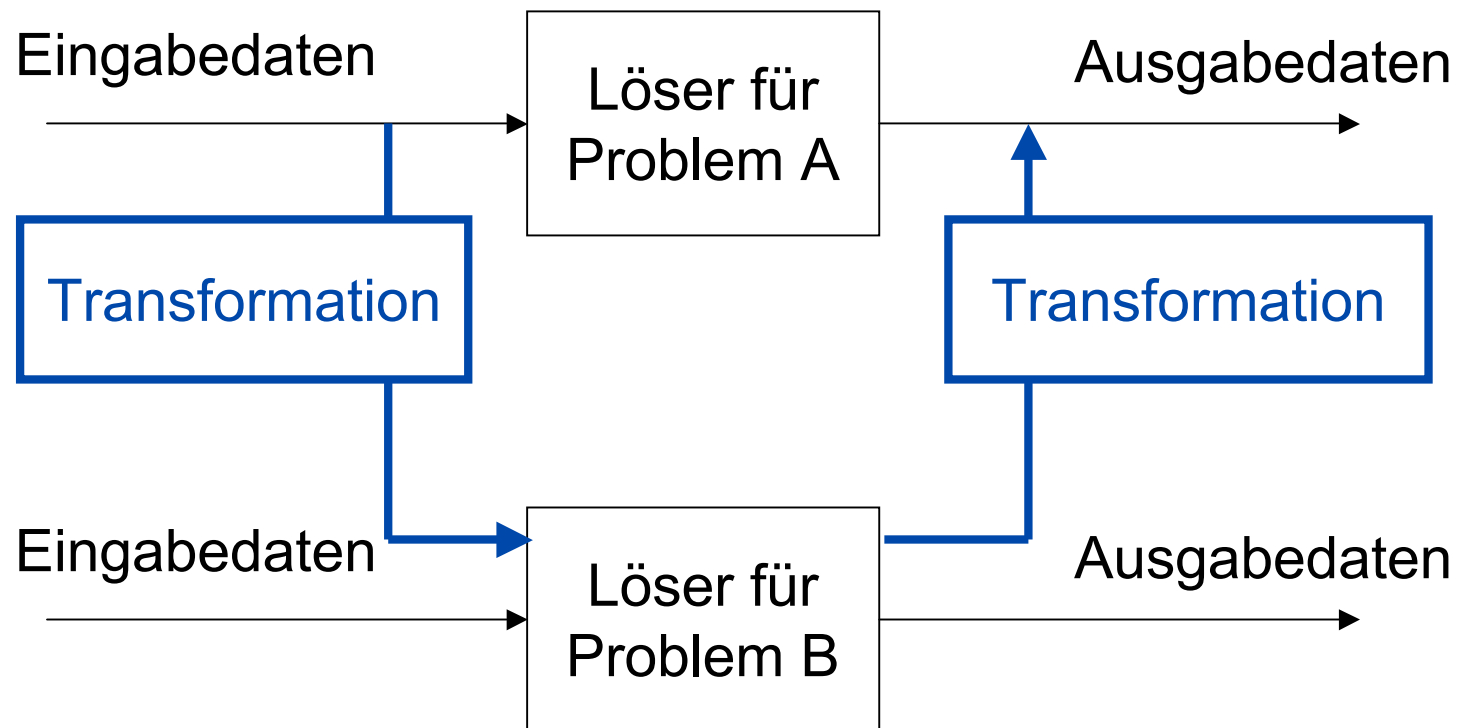
Gegeben: natürliche Zahlen $x[0], x[1], \dots, x[n]$
Gesucht: deren Summe

Algorithmus:

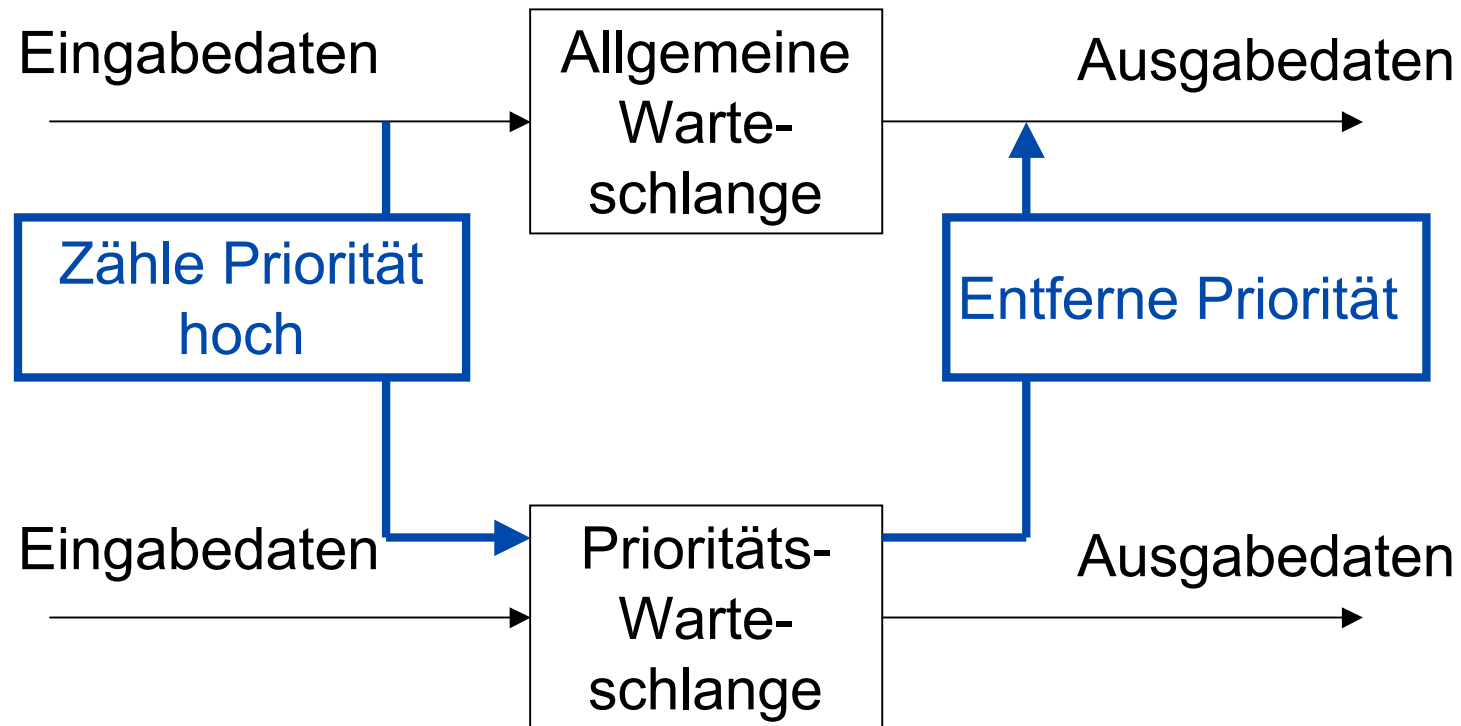
```
a = Prozessor1 (Summe (x[0] , ... , x[n/2] ) ) ;  
b = Prozessor2 (Summe (x[n/2+1] , ... , x[n] ) ) ;  
return a+b;
```

Parallele Algorithmen machen Rechenoperationen, die zeitlich nicht voneinander abhängen, „gleichzeitig“ oder „nebenläufig“.

Lösungsmuster



Beispiel:



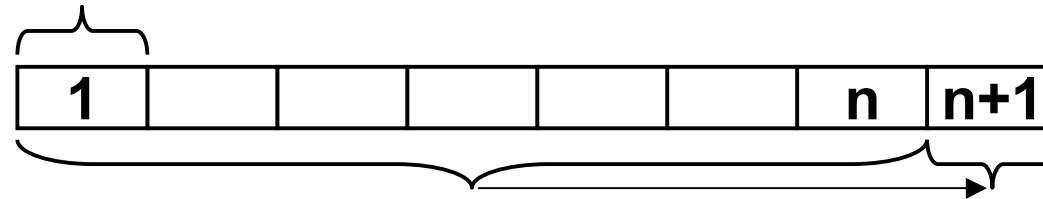
Beide Methoden haben gemeinsam:

- Problem ist gelöst (z.B. vordefiniert) für triviale Fälle
- Jeder nichttriviale Fall kann dadurch gelöst werden, dass seine Lösung eine „einfache“ Folgerung aus der Lösung eines einfacheren Falls ist.
- Jeder nichttriviale Fall wird schließlich bis zu den trivialen Fällen „heruntergebrochen“.



Induktion

Induktionsanfang (triviale Lösung)



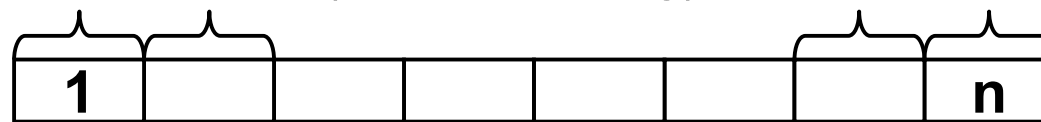
Induktionsvoraussetzung

Induktionsschritt

↳ Gesamtlösung

Teile-und-Herrsche

n Basisfälle (triviale Lösung)



Teillösungen

Lösung 1.Hälfte

Lösung 2. Hälfte

Gesamtlösung

Beispiel:

Gegeben: natürliche Zahlen $x[0], x[1], \dots, x[n]$

Gesucht: deren Maximum

Naheliegender Algorithmus:

```
static long max () {  
    long max = 0;  
    for (int i = 0; i <= n; i++)  
        if ( x[i] > max ) max = x[i];  
    return max;  
}
```



Beispiel:

Gegeben: natürliche Zahlen $x[0], x[1], \dots, x[n]$

Gesucht: deren Maximum

Induktiver Algorithmus (Aufruf mit `max(n)`):

```
static long max(int p) {  
    if ( p==0 ) return x[0];  
  
    long m = max(p-1);  
  
    if ( m > x[p] )  
        return m  
    else  
        return x[p];  
}
```



Beispiel:

Gegeben: natürliche Zahlen $x[0], x[1], \dots, x[n]$

Gesucht: deren Maximum

Teile-und-Herrsche-Algorithmus (Aufruf mit `max(0, n)`):

```
static long max(int l, int r) {  
    if ( l==r ) return x[l];  
  
    long m1 = max (l, (l+r)/2);  
    long m2 = max ((l+r)/2+1, r);  
  
    if (m1 > m2) return m1;  
    else return m2;  
}
```



Beispiel:

Gegeben: natürliche Zahl n

Gesucht: $n! = 1 * 2 * \dots * n$

Behauptung: $n! = n \cdot (n-1)!$

Beweis durch vollständige Induktion:

- 1. Induktionsanfang $n = 0, n = 1$:

$$0! = 1$$

$$1! = 1 \cdot (1-1)! = 1 \cdot 1 = 1$$

- 2. Induktionsschritt $n \rightarrow n+1$:

$$(n-1)! = 1 \cdot 2 \cdot \dots \cdot (n-1)$$

$$n \cdot (n-1)! = (n-1)! \cdot n = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = n!$$

- 3. Folgerung: weil die Behauptung für $n=1$ gilt, und weil aus der Gültigkeit für $n-1$ die für n folgt, gilt die Behauptung für alle n



Entwurf des Algorithmus :

```
static long fakultaet ( long n ) {  
    if ( n <= 1 )  
        // Induktionsanfang  
        return 1;  
    else  
        // Induktionsschritt  
        return n * fakultaet (n-1) ;  
}
```



Beispiel:

Gegeben: Zahlen $x[0], x[1], \dots, x[n]$
Gesucht: irgendeine Zahl aus der oberen Hälfte
der sortierten Folge

Mögliche Lösung:

- bestimme Maximum (erfordert $n-1$ Vergleiche)

Weitere Lösung:

- bestimme Maximum m von $n/2+1$ Zahlen (erfordert $n/2$ Vergleiche)
⇒ garantiert, dass m in oberer Hälfte liegt



Idee:

- Verwendung zufällig gewählter Daten, um die Laufzeit eines Algorithmus zu senken

Einige Aufgaben sind heute überhaupt nur auf diese Weise effizient zu lösen.

- Beispiel: Primzahleigenschaft großer Zahlen

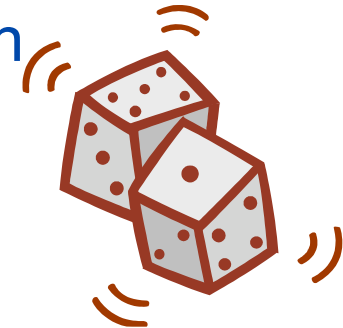
Zentrale Eigenschaft zufallsgesteuerter Algorithmen

- Nichtdeterministisch

Zentrales „Werkzeug“:

- Zufallszahlen
- Beispiel in Java

- `static double random()` der Klasse `Math` realisiert eine im Intervall $[0, 1[$ gleichverteilte Zufallsvariable



Prinzip

Zufallsgesteuerter Algorithmus

=

Deterministischer Algorithmus + Zufallsexperimente



Oft:

- garantiert korrekte Lösung nicht erforderlich
- optimale Lösung nicht erforderlich (Näherungslösung)

Wenn korrekte Lösung nicht garantiert sein muss:

$$p(\text{irgend eine Zahl } x_i \text{ ist in unterer Hälfte}) = 1/2$$

$$p(\text{irgendwelche } x_i \text{ und } x_j \text{ in unterer Hälfte}) = 1/4$$

$$p(\max(x_i, x_j) \text{ ist in unterer Hälfte}) = 1/4$$

$$p(\max(x_{i_1}, x_{i_2}, \dots, x_{i_k}) \text{ in unterer Hälfte}) = 2^{-k}$$

$$p(\max(x_{i_1}, x_{i_2}, \dots, x_{i_k}) \text{ in oberer Hälfte}) = 1 - 2^{-k}$$

Die Wahrscheinlichkeit steigt für große k asymptotisch gegen 1

- z.B. $k = 20 \Rightarrow p > 0.999999$.

Aufwand ist konstant unabhängig von n .



Monte Carlo (**mc** = **mostly correct**)

- Die Wahrscheinlichkeit $0 < p < 1$ für korrektes Ergebnis kann vorgegeben werden (p -Korrektheit).
 - terminiert daher stets
- Hat kürzere Laufzeit als der beste deterministische Algorithmus
 - p kann erhöht werden um den Preis höherer Laufzeit.
- s -faches Wiederholen des Aufrufs setzt die Fehlerwahrscheinlichkeit auf $(1-p)^s$ herab.
 - Gesamtergebnis: das häufigste Ergebnis der s Aufrufe (ggfs. Mittelung)

Las Vegas

- Gibt immer korrektes Ergebnis aus
- Versucht es zu erraten
 - Laufzeit ist nicht garantiert kürzer als deterministisch (muss nicht terminieren)
 - Durchschnittliche Laufzeit ist kürzer



Für probabilistische Algorithmen werden Zufallszahlen benötigt.
Jeder Programmablauf ist deterministisch
⇒ keine echten Zufallszahlen.

Daher in der Praxis: Pseudozufallszahlen.

- deterministisch bestimmt
- Zusammenhang zwischen erzeugten Zufallszahlen nicht erkennbar
- erfüllen bestimmte Verteilungen (gleichverteilt/normalverteilt)

Minimal Standard Random Number Generator nach Park/Miller

r_i berechnet sich aus r_{i-1} gemäß

$$r_i = (a * r_{i-1}) \bmod m$$

Die erste Zufallszahl r_0 wird Saat (seed) genannt

- wird echt zufällig gewählt (z.B. aktuelle Zeit)
- wird gleich gewählt (\Rightarrow Programmablauf reproduzierbar)

r_i ist immer zwischen 0 und $m-1$

a und m sind Konstanten, die sorgfältig gewählt werden sollten
(z.B. $a=16807$, $m=2^{31}-1$) :

- Es kann getestet werden, ob die erzeugten Zahlen sinnvoll verteilt sind. Wenn nicht, können andere Werte für a und m gewählt werden.



Beispiel:

Gegeben: ungerade Zahl n .

Aufgabe: Stelle fest, ob n Primzahl ist.

Exaktes Vorgehen:

- Prüfe für alle Primzahlen z , $2 < z \leq \sqrt{n}$, ob n durch z teilbar ist.

Problem:

- Die Zahlen z sind (teilweise) unbekannt.
- Daher „Gewaltlösung“: Prüfe mit allen ungeraden Zahlen $2 < z \leq \sqrt{n}$.
- Erfordert \sqrt{n} (langsame) Divisionen.



Primzahltest nach Miller-Rabin

Konstruiere Monte Carlo Algorithmus **primzahl**, der sich bei der Aussage "**n** ist Primzahl" mit Wahrscheinlichkeit $p < 2^{-s}$ irren kann:

- Konstruiere Prozedur **zeuge (a, n)**, die mit Hilfe zufällig gewählter (**random()**) Zahl $a \in J_n = \{1, \dots, n-1\}$
 - das Ergebnis **true** liefert, falls **n** Produkt zweier Zahlen und
 - **false**, falls der Nachweis auf Produkt mittels a (!) nicht erbracht werden konnte.
- Setze **zeuge (a, n)** wiederholt mit verschiedenen **a** ein:
 - Falls ein Zeuge **true** liefert, ist **n** keine Primzahl.
 - Andernfalls ist **n** mit einer gewissen Wahrscheinlichkeit Primzahl.



Die Implementierung von **zeuge** (**a**, **n**) kann aus zahlentheoretischen Überlegungen heraus konstruiert werden:

```
bool zeuge ( int a, int n ) {  
    // Implementierung z.B. nach kleinem Fermatschem Satz:  
    // n Primzahl  $\Rightarrow \forall a \in J_n: a^{n-1} \bmod n = 1$   
    // Also:  $a^{n-1} \bmod n \neq 1 \Rightarrow$  Ergebnis ist true, sonst false  
}
```



http://en.wikipedia.org/wiki/Miller-Rabin_primality_test

Die Implementierung von **random** (**i**, **j**) liefert eine Pseudozufallszahl im Intervall [**i**, **j**]:

```
int random ( int i, int j ) {  
    // Impl. des Pseudozufallszahlengenerators  
}
```



```
boolean primzahl (int n, int s) {  
    for (int j = 1; j <= s; j++) {  
        a = random (1, n-1);  
        if zeuge (a, n) == true {  
            return false;           // keine Primzahl  
                                     // 100% sicher  
        }  
    }  
    return true;                    // Primzahl  
                                     // „Fast“ sicher  
}
```



Wahrscheinlichkeit für den Fehlerfall, dass **n** Produkt ist, dies aber nicht erkannt wird:

- Sei $H = \{a \mid \text{zeuge}(a, n) = \text{false}\} \subseteq J_n$.
- Bei zufälliger gleichverteilter Wahl von a : $p(a \in H) = |H|/(n-1)$.
- Falls $p \leq 1/2$, ist die Wahrscheinlichkeit
 $p^s = p(\text{s Wiederholungen von } \text{zeuge} \text{ mit } \text{false}) \leq 2^{-s}$.
- Also: Algorithmus irrt sich mit Wahrscheinlichkeit $p^s \leq 2^{-s}$, wenn n ein Produkt ist.
- Mit s groß genug kann p^s unter jede gewünschte Schranke gedrückt werden.
 - Beispiel: für $s=10$ ist $p^s < 10^{-3}$, für $s=20$ ist $p^s < 10^{-6}$.

