14 Algorithmen

41 Backtracking / Algorithmen

41.1 Rekursion und Iteration an einfachen mathematischen Beispielen

Ein ***Algorithmus*** ist eine endliche Folge von eindeutig bestimmten Elementaranweisungen, die den Lösungsweg eines Problems exakt und vollständig beschreiben. [1]

Ein ***Programm*** ist ein Algorithmus, der in einer Sprache formuliert ist, die die Abarbeitung durch einen Computer ermöglicht.

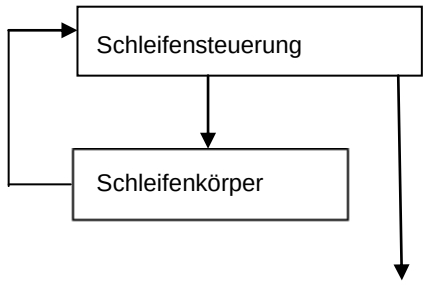
Der Begriff Algorithmus entstand durch Sprachtransformation aus dem Namen eines persisch-arabischen Gelehrten, der u.a. ein Buch „Über die indischen Zahlen“ geschrieben hat. Bis ins letzte Jahrhundert verstand man unter Algorithmen Rechenverfahren wie die schriftliche Addition, Subtraktion, aber auch:

Die Berechnung des größten gemeinsamen Teilers von zwei (natürlichen) Zahlen (**Euklidischer Algorithmus**)

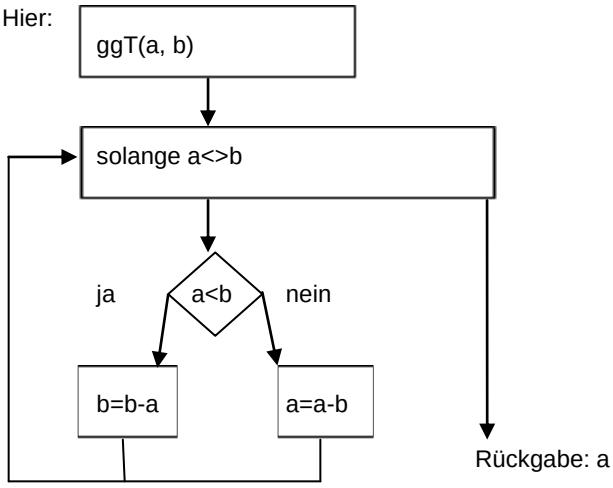
Iterative Formulierung	Rekursive Formulierung
Man nehme abwechselnd immer das Kleinere vom Größeren weg, bis die beiden Zahlen gleich sind.	Wenn die beiden Zahlen gleich sind, ist die Lösung gefunden. Sonst berechne den ggT von der kleineren Zahl und der Differenz der beiden Zahlen
FUNKTION ggt (a, b) **SOLANGE a <> b TUE** **WENN a < b** **DANN b := b – a** **SONST a := a – b** **RÜCKGABE a**	**FUNKTION ggt (a, b)** **WENN a = b DANN RÜCKGABE a** **WENN a < b** **DANN RÜCKGABE ggt (a, b – a)** **SONST RÜCKGABE ggt (b, a – b**

Veranschaulichung der Iteration:

Flussdiagramm



Hier:



Speicherbelegungsplan:

a	**b**	**a<b**
27	6	nein
21	6	nein
15	6	nein
9	6	nein
3	6	ja
3	3	„fertig“

[1] Ziegenbalg: Algorithmen von Hammurapi bis Gödel; Frankfurt 2007; S. 23 ff.

Veranschaulichung der Rekursion:

Abfolge der rekursiven Aufrufe:

ggT (27, 6), a>b, also
ggT (6, 21), a<b, also
ggT (6, 15), a<b, also
ggT(6, 9), a<b, also
ggT(6, 3), a>b, also
ggT(3, 3), a=b, also Rückgabe 3
Rückgabe 3
Rückgabe 3
Rückgabe 3
Rückgabe 3
Rückgabe 3

Die **Fakultät** einer natürlichen Zahl
kann man mit Pünktchen beschreiben: n! := 1 · 2 · 3 · … · n
oder rekursiv definieren: 1! := 1 und n! := n · (n-1)!

```
FUNKTION fakultät (n)
f := 1;
FÜR i := 1 BIS n TUE
   f := f * i
RÜCKGABE f
```

```
FUNKTION fakultät (n)
WENN n=1
DANN RÜCKGABE 1
SONST RÜCKGABE n* fakultät(n-1)
```

Weshalb benötigt man bei der iterativen Version eine lokale Variable („f“), bei der Rekursion aber nicht ?

Die Antwort ergibt sich, wenn man eine Rekursionsübung in der Klasse durchspielt:

Schülerin A erhält die Aufgabe, z. B. 4! zu berechnen.

Mit Hilfe des Übungsblatts (s. u.) entscheidet er, einen Auftrag zur Berechnung von 3! an Schüler B zu erteilen.

Dieser beauftragt Schülerin C, 2! zu berechnen.

Jene fordert 1! von einem weiteren Schüler an.

Der nun liefert ein Ergebnis zurück und C kann rechnen.

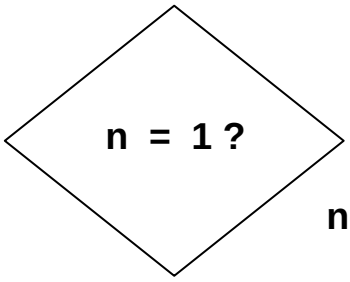
u.s.w.

Die Zwischenergebnisse werden zurückgereicht (technisch: über den Stack, der die Unterprogrammaufrufe abwickelt).

Bei der iterativen Version haben wir diesen „Service“ nicht und müssen die Zwischenergebnisse selbst verwalten.

Übungsblatt Rekursion (Fakultät)

Auftrag: Berechne Fakultät (n) für n = ☐



ja **nein**

Ergebnis : 1

Erteile den Auftrag

Berechne Fakultät von ☐

n - 1

Warte auf die Rückgabe

Rechne

Ergebnis = ☐ ***** ☐

n Rückgabe

= ☐

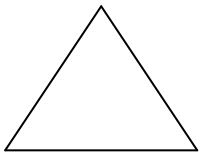
Gib Dein Ergebnis zurück an den Auftraggeber !

Die **Binomialkoeffizienten** entstehen beim Ausmultiplizieren der Terme $(a + b)^n$ mit wachsendem n. Sie lassen sich im Pascalschen Dreieck darstellen und berechnen

n \ k	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

usw.

Die Rekursionsformel lautet: $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$, d. h. jede Zahl ist die Summe aus den beiden schräg darüber stehenden (wenn man das Pascalsche Dreieck so aufschreibt: Hier sind es die links und direkt darüber stehenden Zahlen.



Eine iterative Lösung wird auf die Definition $\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \left(= \frac{n(n-1)(n-2)...(n-k+1)}{k!} \right)$ zurückgreifen.

```
FUNKTION binomial (n, k)
f := 1;
FÜR i := n ABWÄRTSBIS n-k+1 TUE
   f := f * i
RÜCKGABE f div Fakultät(k)
```

```
FUNKTION binomial (n, k)
WENN n=0 oder k=0 oder k=n
DANN RÜCKGABE 1
SONST RÜCKGABE
   binomial(n-1, k) + binomial(n-1, k-1)
```

Die berühmte Folge über die Vermehrung von Hasen, benannt nach **Fibonacci** (Leonardo von Pisa), lässt sich rekursiv elegant angeben:

$$Fib(n) = \begin{cases} 0 \;\; für\, n = 0 \\ 1 \;\; für\, n = 1 \\ Fib(n-1) + Fib(n-2) \;\; sonst \end{cases}$$

Bei der iterativen Version muss man Zwischenergebnisse wieder explizit speichern.

```
FUNKTION Fib (n)
WENN n=0 oder n=1
DANN RÜCKGABE n
SONST
  f0 := 0; f1 := 1; f2 := f0+f1;
  FÜR i := 2 BIS n TUE
    f0 := f1; f1 := f2; f2 := f0+f1;
  RÜCKGABE f1
```

```
FUNKTION Fib (n)
WENN n=0 oder n=1
DANN RÜCKGABE n
SONST RÜCKGABE Fib(n-1) + Fib(n-2)
```

Aufgaben

1) Eine Zählschleife (for i := 1 to n do Ausgabe(i*i)) rekursiv
2) Fibonacci iterativ und rekursiv
3) Die folgenden Zahlenreihen iterativ und rekursiv berechnen lassen:

1	-2	32	479001600
2	5	67	239500800
3	12	494	159667200
4	19	1313	119750400
5	26	2524	95800320
6	33	4127	79833600
7	40	6122	68428800
8	47	8509	59875200
9	54	11288	53222400
10	61	14459	47900160
11	68	18022	43545600
12	75	21977	39916800

Effizienzbetrachtungen:

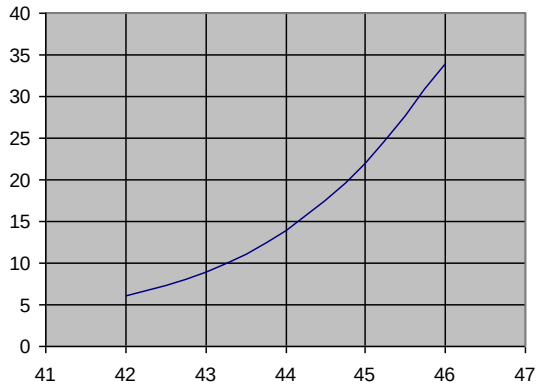
Bei einfacher Rekursion (Fakultät) wird Speicherplatz auf dem Stack benötigt (proportional zu n; jeweils der Wert von n, das Zwischenergebnis und die Rücksprungadresse), wohingegen die iterative Version nur zwei Speicherplätze für i und f braucht – unabhängig von n. Auch bei der Rechenzeit wird es einen Vorteil für die Iteration geben, weil das Handling mit den Rücksprüngen und der Stack-Verwaltung entfällt und ggf. Optimierungen greifen(i in einem Register des Prozessors).

Bei **mehrfacher Rekursion** wird der Unterschied in nahezu dramatischer Weise deutlich:

Bei Fibonacci liefert die iterative Funktion bis n=46 das Ergebnis in nicht messbarer Zeit; ab 47 ist der Zahlbereich Longint (Integer bei Delphi7) überschritten. Mit Real als Zahlbereich ist der Zeitaufwand auch für n=100 noch nicht messbar.

Die Rekursion liefert auf einem älteren Notebook (Athlon XP 2800+) folgende (gestoppte) Zeiten:

n	Zeit in sec	Zeit in sec (Real)
42	6	26
43	9	31
44	14	66
45	22	107
46	34	173
47	54	279



Woher kommt dieser exponentiellen Anstieg ?

In der nebenstehenden Grafik ist die Folge der Aufrufe von „Fib“ protokolliert, und zwar für **n=7**.
Um Fib(7) zu berechnen, wird Fib(6) berechnet (2. Zeile) und Fib(5) (weiter unten).
Für Fib(6) brauchen wir Fib(5) (3. Zeile) und Fib(4).

Die Abfolge, um Fib(5) zu berechnen, steht **zweimal** im Protokoll (ab der 3. Zeile und weiter unten), bei fib(4) sind es schon dreimal, Fib(3) fünfmal ...

Der kurze, elegante rekursive Fibonacci – Algorithmus ist also so dumm, Zahlen, die schon berechnet worden sind, erneut zu berechnen. Diese Wiederholungen von Bekanntem wächst mit n in der oben gesehene dramatischen Weise.

Ziegenbalg stellt diese Aufrufe auf S. 171 in Form eines Baumes dar:

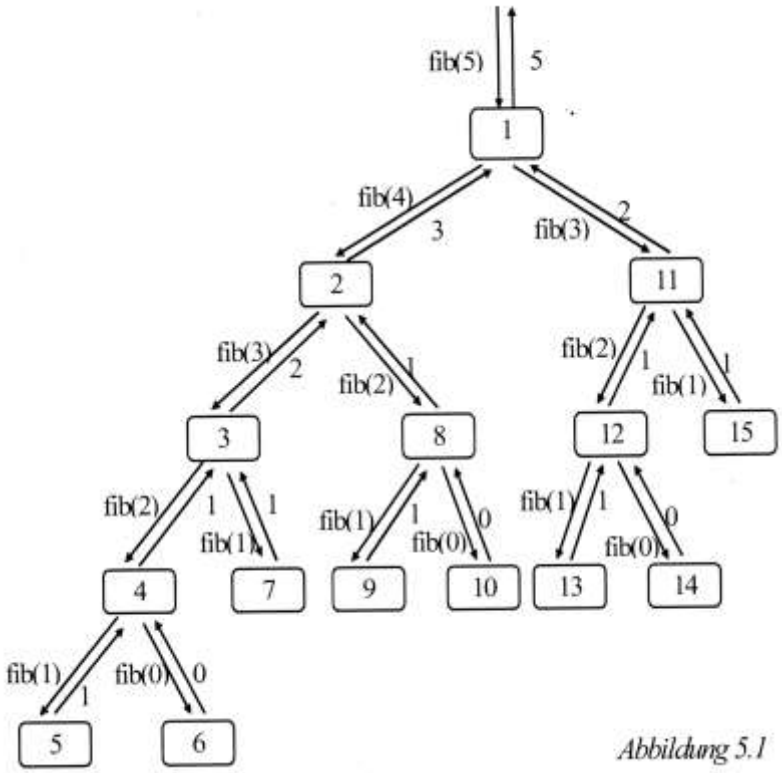
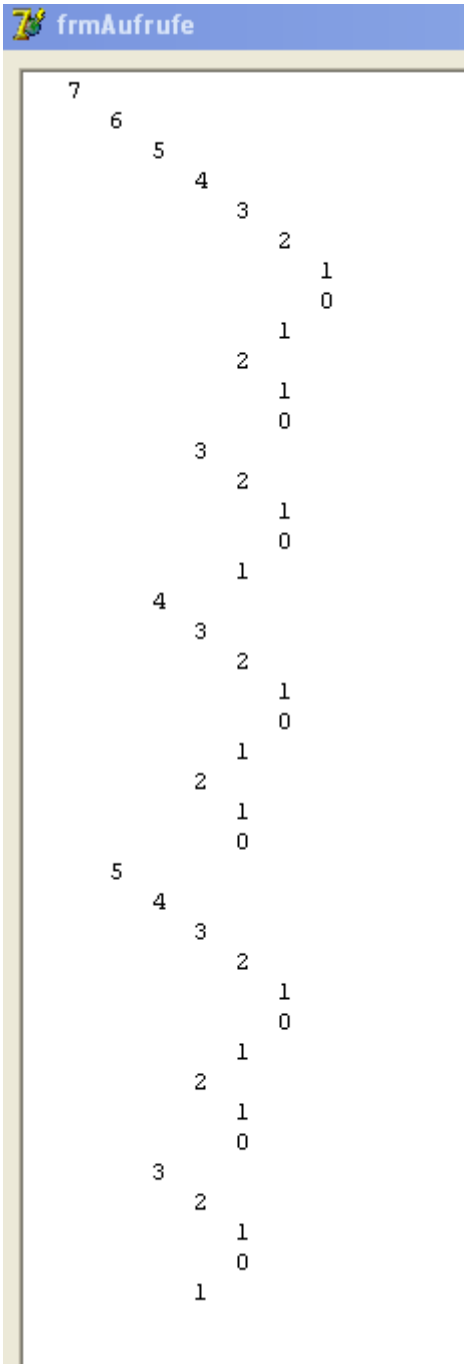


Abbildung 5.1



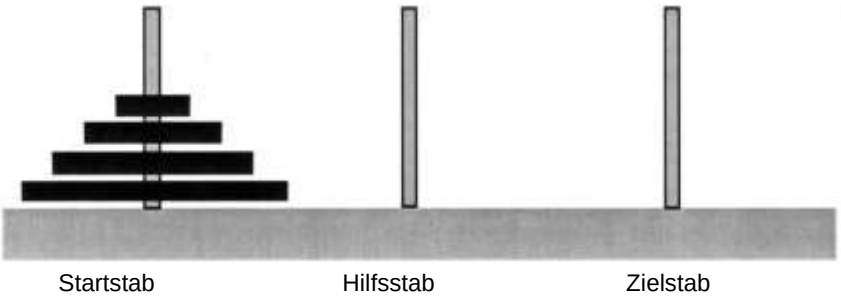
Wieso macht die iterative Version das besser?

Weil sie in den Variablen f0, f1 und f2 die **Zwischenwerte speichert** und damit weiter rechnet.

41.2 Die Türme von Hanoi – eine Frage der Strategie

Turm-von-Hanoi-Spiel (Aufgabenstellung und Schilderung der Lösungsidee aus Ziegenbalg)

Gegeben sind drei Stäbe (ein Startstab, ein Hilfsstab und ein Zielstab) sowie eine bestimmte Anzahl (in der Abbildung: 4) verschieden großer, gelochter Scheiben.



Die Aufgabe besteht nun darin, den Stapel von Scheiben so vorn Startstab unter möglicher Verwendung des Hilfsstabes auf den Zielstab zu transportieren,

- dass immer nur eine Scheibe bewegt wird, und
- dass nie eine größere Scheibe auf einer kleineren Scheibe liegt.

Es wird erzählt, dass ein Mönch von Buddha den Auftrag bekommen habe, diese Aufgabe mit einem Stapel von 100 goldenen Scheiben durchzuführen. Wenn er fertig sei, sei das Ende der Welt gekommen. Der Mönch dachte sich: „Diese Aufgabe ist wohl zu schwer für mich allein - mein ältester Schüler soll mir dabei helfen." Er rief den Schüler zu sich und erläuterte ihm seinen Plan: „Lege Du die obersten 99 Scheiben nach der Regel vom Startstab unter Verwendung des Zielstabes (als zeitweiligem Hilfsstab) auf den Hilfsstab (als zeitweiligem Zielstab). Wenn Du damit fertig bist, lege ich die unterste Scheibe des Startstabes auf den Zielstab. Danach lege Du die 99 Scheiben vom Hilfsstab unter Verwendung des Startstabes auf den Zielstab. Wenn Du mit dem zweiten Teil Deines Auftrages fertig bist, sind insgesamt alle 100 Scheiben nach der Regel vom Startstab auf den Zielstab gebracht worden, und das Werk ist vollendet." Der älteste Schüler des Mönchs dachte sich: „Das ist eine ziemlich schwere Aufgabe". Aber er war ein sehr gelehriger Schüler und rief den zweitältesten Schüler zu sich, damit er ihm helfe ...

Der Algorithmus in Delphi – naher Form ist ganz kurz:
Das tatsächliche „Legen“ wird man als Text ausgeben oder grafisch simulieren.

```
procedure Bewege (Anzahl, von, hilf, nach)

  Bewege (Anzahl-1, von, nach, hilf)    // 99 von Start auf Hilf (mit Ziel als Hilfsstab)

  Lege eine Scheibe „von“  „nach“

  Bewege (Anzahl-1, hilf, von, nach,)   // 99 vom Hilfsstapel auf Ziel (mit Start als
                                           Hilfsstapel)
```

iterative Lösung

s. Programm

41.3 Backtracking – das Damenproblem

Auf einem Schachbrett mit n Zeilen und n Spalten sind n Damen so zu positionieren, dass nach den Schachregeln keine der Damen eine andere schlagen kann.
Folgerungen:

1. je Spalte nur eine Dame
2. je Zeile nur eine Dame
3. Diagonalen überprüfen

Um die Bedingung 1 von vorneherein zu erfüllen, setzen wir pro Spalte eine Dame und überprüfen Bedingung zwei und Bedingung drei.

		O	
O			
			O
	O		

eine der beiden Lösungen

Die Möglichkeiten, die es dabei gibt, lassen sich z.B. für **n=4** noch übersichtlich darstellen:
(aus: Ziegenbalg)

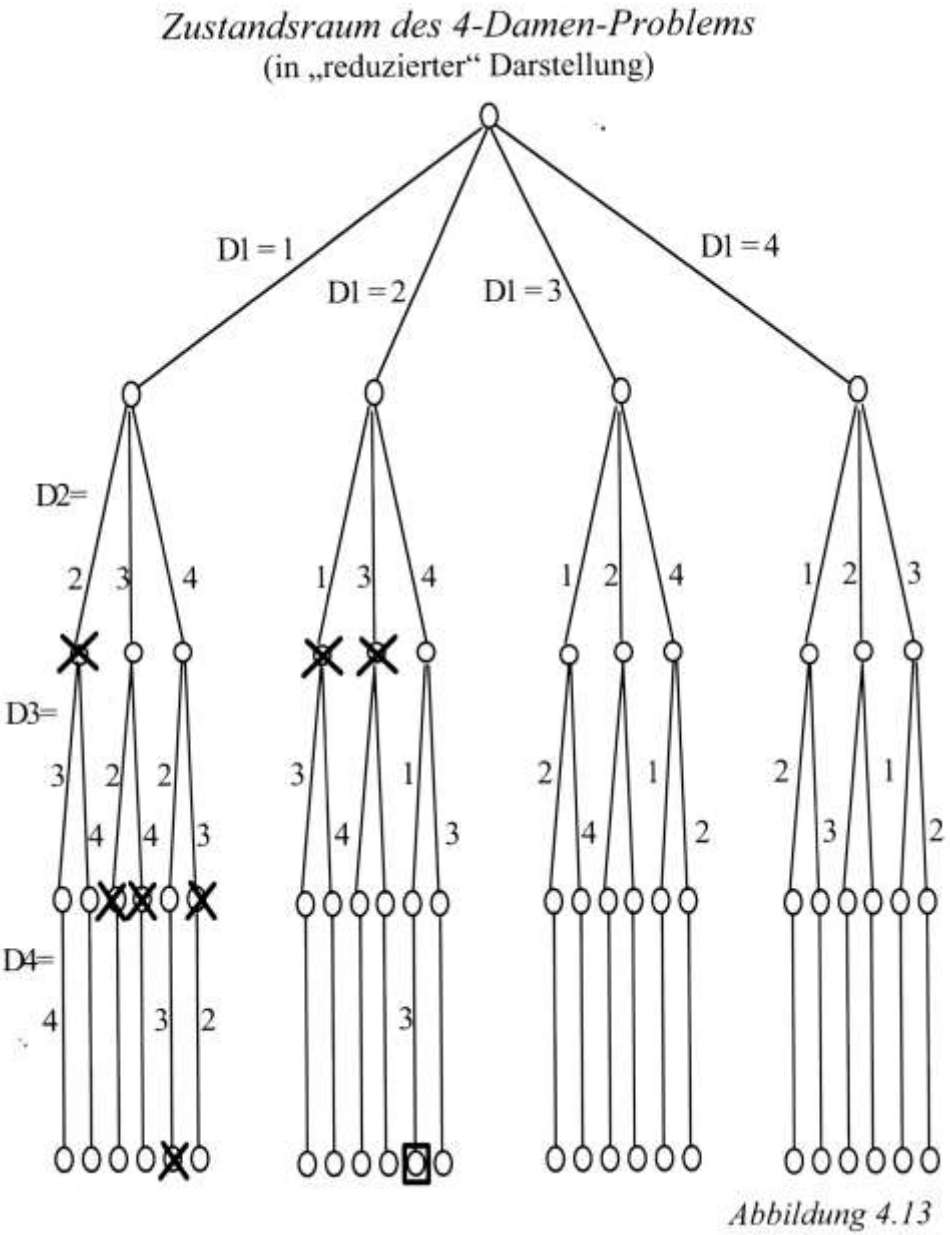


Abbildung 4.13

Verfahren des „Backtracking“:

Man durchläuft die Zweige des Zustandsbaumes (Tiefensuche), bis man auf eine Lösung oder einen „verbotenen“ Knoten stößt. Wenn man einen verbotenen Knoten erreicht hat, geht man eine Stufe zurück (Rückzug – backtracking) und probiert von dort aus die nächste Kante. Wenn es keine weitere Kante mehr gibt, geht man noch eine Stufe höher usw. Will man alle Lösungen haben, darf man natürlich nach der ersten Lösung nicht aufhören, sondern gibt die Lösung aus und geht dann weiter wie beschrieben.

Rekursiver Ansatz für das Damenproblem:

```
procedure SetzeDame (Spalte)

  wenn Spalte > n dann gib Lösung aus
  sonst
    für Zeile von 1 bis n tue
      wenn die Dame dort (Zeile, Spalte) nicht geschlagen werden kann
      dann  setze die Dame in dieses Feld
            SetzeDame (Spalte + 1)          ←    rekursiver Aufruf – nächste Spalte
            nimm die Dame wieder weg
    weiter

end
```

Der „Spielablauf“ für n=4

1. Zug:	D1=1			
2. Zug:		D2=2 (verboten)		
3. Zug:		D2=3		
4. Zug:			D3=2 (verboten)	
5. Zug:			D3=4 (verboten)	
6. Zug:		D2=4		
7. Zug:			D3=2	
8. Zug:				D4=3 (verboten)
9. Zug:			D3=3 (verboten)	
10. Zug:	D1=2			
11. Zug:		D2=1 (verboten)		
12. Zug:		D2=3 (verboten)		
13. Zug:		D2=4		
14. Zug:			D3=1	
15. Zug:				D4=3 (erste Lösung)

Für das klassische Schachbrett liefert das Programm 92 Lösungen:

Lösung 1
```
|x| | | | | | | |
| | | | | | |x| |
| | | | |x| | | |
| | | | | | | |x|
| |x| | | | | | |
| | | |x| | | | |
| | | | | |x| | |
| | |x| | | | | |
```

Lösung 2
```
|x| | | | | | | |
| | | | | | |x| |
| | | |x| | | | |
| | | | | | |x| |
| | | | | | | |x|
| |x| | | | | | |
| | | | |x| | | |
| | |x| | | | | |
```

Lösung 3
```
|x| | | | | | | |
| | | | | |x| | |
| | | | | | | |x|
| | |x| | | | | |
| | | | | | |x| |
| | | |x| | | | |
| |x| | | | | | |
| | | | |x| | | |
```

n	Anzahl der Lösungen
4	2
5	10
6	4
7	40
8	92
9	352
10	724
11	2680
12	14200

Lösung 4
```
|x| | | | | | | |
| | | | |x| | | |
| | | | | | | |x|
| | | | | |x| | |
| | |x| | | | | |
| | | | | | |x| |
| |x| | | | | | |
| | | |x| | | | |
```

.....

Lösung 92
```
| | |x| | | | | |
| | | | | |x| | |
| | | |x| | | | |
| |x| | | | | | |
| | | | | | | |x|
| | | | |x| | | |
| | | | | | |x| |
|x| | | | | | | |
```

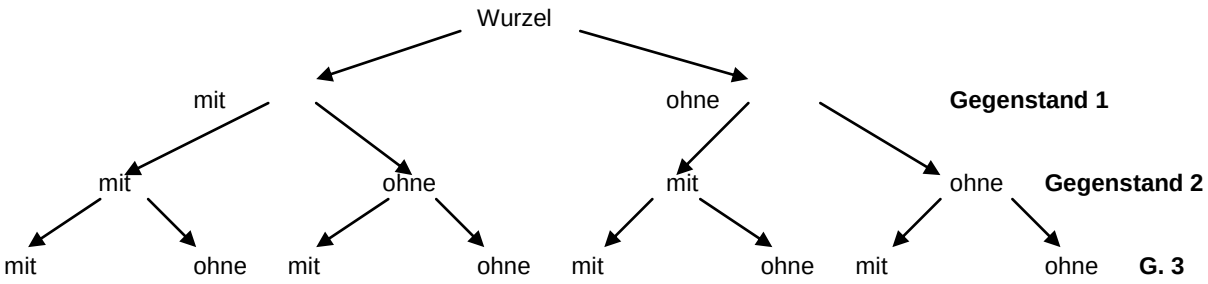
Bei 12 Damen waren es schon 14200 Lösungen und mein Rechner brauchte über 50 Sekunden.

Der **Aufwand** (in Abhängigkeit von n):
Es ist n-mal das Problem zu lösen, n-1 Damen in einem (n-1)*(n-1) – Feld zu platzieren (+ mehr Zeit für die Prüfung der Diagonalen) - also **Aufwand (n) >= n !**

41.4 Rucksack – Problem (knapsack problem)

Gegeben ist eine Menge von Gegenständen, von denen jeder ein bestimmtes Gewicht G_i und einen bestimmten Wert W_i hat. Man stelle nun eine „Ladung“ (für besagten Rucksack oder praktisch relevanter einen LKW) zusammen, die bei vorgegebener Beschränkung des Gesamtgewichts einen möglichst großen Wert hat.

Wenn man vorerst die Gewichtsbeschränkung ignoriert, kann man die alle Möglichkeiten für unterschiedliche Ladungen in Form eines Binärbaums darstellen:



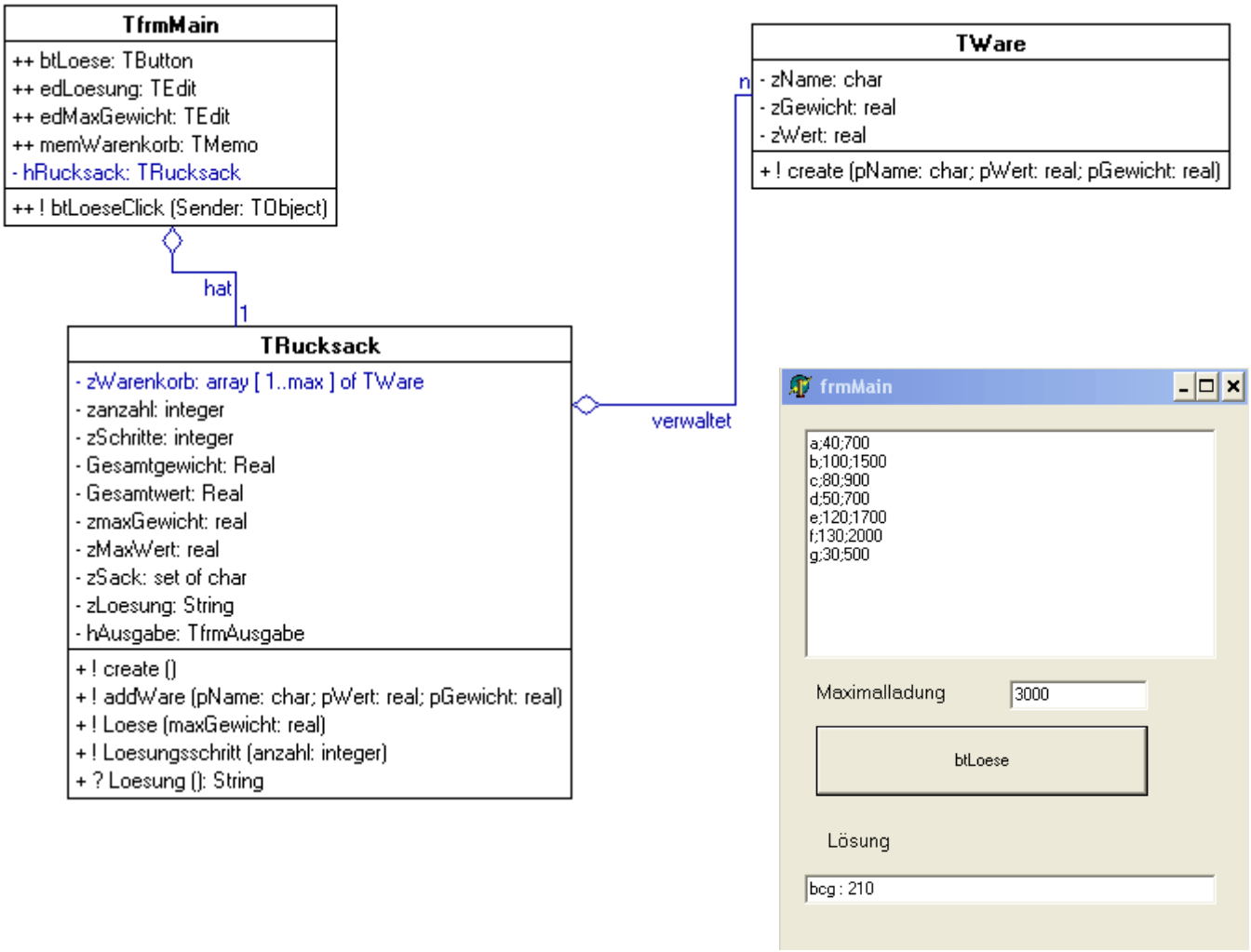
usw.

Bei drei Gegenständen gibt es $8 = 2^3$ Wege durch den Baum, bei n Gegenständen $\mathbf{2^n}$.

Wenn man Teilbäume wegen Überschreitung der Kapazität nicht berechnet, spart man etwas an Aufwand; es ändert aber nichts daran, dass der Aufwand mit n exponentiell wächst.

Für kleinere Werte von n kann man die exakte Lösung mit Durchmusterung aller Möglichkeiten (Wege durch den obigen Baum) finden.

Das UML – Diagramm für ein Programm:



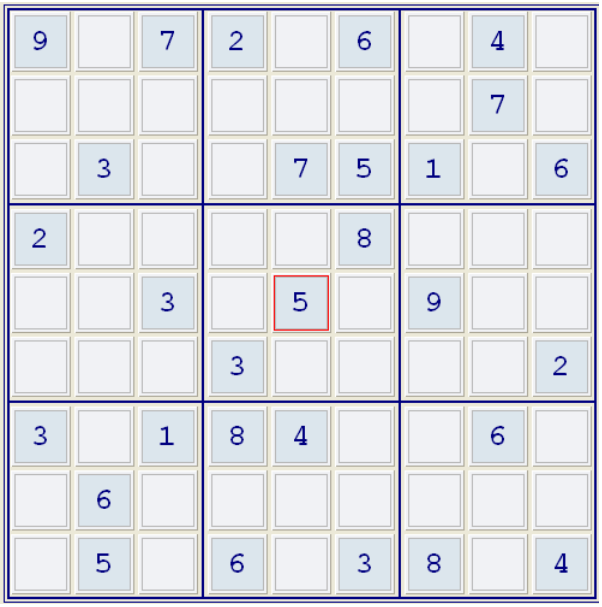
Der Algorithmus prüft, ob der jeweilige Gegenstand vom Gewicht her noch passt.
Wenn ja, fügt er den Gegenstand dem Rucksack hinzu und geht weiter.
Ohne den Gegenstrand wird in jedem Fall noch einmal weiter gesucht

Ja, es ist wieder Backtracking angesagt:

```
procedure TRucksack.Loesungsschritt(anzahl:integer);
  var c : char;
  begin

    if anzahl <= zanzahl then begin

      if Gesamtgewicht + zWarenkorb[anzahl].zGewicht <= zmaxGewicht then begin
            //Gegenstand hinzuzfügen
         zSack := zSack + [zWarenkorb[anzahl].zName];
         Gesamtwert := Gesamtwert + zWarenkorb[anzahl].zWert;
         Gesamtgewicht := Gesamtgewicht+ zWarenkorb[anzahl].zGewicht;

         if Gesamtwert > zMaxWert then begin
           //notiere neuen Lösungskandidaten
         end;
            //rekursiver Aufruf
         Loesungsschritt(anzahl+1);
            //Gegenstand wieder herausnehmen
         Gesamtgewicht := Gesamtgewicht- zWarenkorb[anzahl].zGewicht;
         Gesamtwert := Gesamtwert - zWarenkorb[anzahl].zWert;
         zSack := zSack - [zWarenkorb[anzahl].zName];
      end;
            //zweiter Weg ohne den Gegenstand
      Loesungsschritt(anzahl+1);
    end;
  end;
```

Bei demselben Warenkorb hängt die Anzahl der zu durchlaufenden Rekursionsschritte von der Rucksackkapazität ab.
Wenn alles hineinpasst, durchläuft der Algorithmus bei 7 Gegenständen die erwarteten 128 Wege.
Ist der Rucksack sehr klein, so bleibt im Extremfall nur noch ein Weg übrig.
Bei 10 Gegenständen sind es dann bis zu 1024 Schritte (eben 2^n).

41.5 Sudoku, das Spiel



Die einfache **Spielregel**:

Die vorgegebenen Zahlen sind so zu ergänzen, dass
1. in jeder Zeile,
2. in jeder Spalte und
3. in jedem der neun Quadrate

die Zahlen 1,2,3,…9 vorkommen. Da sie alle erscheinen sollen, kann jede Zahl dort jeweils nur einmal vorkommen.

Wenn man ein gestelltes Rätsel lösen will, kann man davon ausgehen, dass die Lösung eindeutig ist.
Wenn man hingegen neue Rätsel entwickeln will, ist es wichtig festzustellen, ob die Lösung eindeutig ist; man wird versuchen, eine zweite Lösung zu bestimmen. Alle Lösungen zu finden macht bei Sudoku wenig Sinn.

Wie viele Möglichkeiten gibt es überhaupt, die Zahlen zu verteilen. In einem naiven Ansatz gehen wir davon aus, dass in einer Zeile eine Permutation der Zahlen von 1 bis 9 steht:
9 ! Möglichkeiten (9*8*7*6*5*4*3*2*1 = 362 880).
Bei neun unabhängigen Zeilen wären das dann $(9!)^9 = 362880^9$ Möglichkeiten. Diese Zahl möchte ich lieber nicht ausrechnen($\approx 10^{50}$). Sie ist auch viel zu groß, weil die Zeilen eben nicht unabhängig sind. Sie zeigt aber auch, dass es hier nicht darum geht, ein paar Möglichkeiten durchzuprobieren.

Bei einer einigermaßen geschickten Strategie setzt man eine Ziffer erst, wenn es keine Widersprüche zu den bisher gesetzten Ziffern gibt.

41.5.1 Lösung durch Backtracking (rekursiv)

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

Ansatz wie beim Damenproblem:

Dort hatten wir n Spalten und jeweils n Möglichkeiten, die Dame zu setzen.
Beim Sudoku sind es 81 Felder und jeweils 9 mögliche Werte.

Wir setzen in das erste Feld (links oben) der Reihe nach (Schleife) die Zahlen 1 bis 9 ein, prüfen, ob die Spielregeln erfüllt sind, und gehen jedes Mal zum nächsten Feld, setzen der Reihe nach die Zahlen 1 bis 9 ein, usw. Wenn die Spielregeln nicht erfüllt sind, wird der jeweilige Zweig nicht weiter verfolgt.

Das Sudoku – Spielfeld ist zweidimensional, also ein `array [1..9, 1..9] of integer`.
Bei einem zweiten Ansatz fand ich es für die Formulierung des rekursiven Ansatzes leichter, die Zellen hintereinander anzuordnen, also mit einer Nummerierung zu versehen wie nebenstehend.
Vorteil dieses Ansatzes: Man kann von Feld n zu Feld n+1 gehen, ohne an den Übergang zur nächsten Zeile denken zu müssen.
Nachteil: Die Überprüfung der Zeilen und Spalten ist nicht so einfach.
Durch die Nummerierung von 0 anfangend kann man den Nachteil wettmachen: `Feldnummer mod 9` liefert nämlich die Spaltennummer, `Feldnummer div 9` die Zeilennummer.
Beim Damenproblem wurden jeweils die Zeile und die Diagonalen durchlaufen und überprüft. Hier fand ich es einfacher und effizienter, eine Buchführung über die schon gesetzten Zahlen zu machen und sich für jede Zeile, Spalte und jedes Quadrat zu merken, welche Zahlen gesetzt sind.
In Delphi geht das am elegantesten mit Mengen (set of 1..9). Mit dem „in“ – Operator prüft man, ob eine Zahl in der Menge vorhanden ist. (Alternative wäre ein array [1..9] of boolean, in dem true angibt, dass die jeweilige Zahl vorkommt.)

```
type TSudoku = class
  private
    merkZeile,                                          „Buchführung“
    merkSpalte,
    merkKasten : array[0..8] of set of 1..9;

    SudokuFeld : array[0..80] of integer;               Das Spielfeld

    procedure testen(n, test : integer; var fertig:boolean);
```

```
    procedure loesen(n : integer; var fertig:boolean);

    procedure AllesMerken;                                  Die „Aufgabenstellung" =
                                                            vorgegebene Zahlen in die
                                                            Buchführung eintragen
  public
    constructor create;
    procedure Loese;
end;
```

Die beiden entscheidenden Methoden (aufgeteilt aus Gründen der Übersichtlichkeit):

```
procedure TSudoku.loesen(n : integer; var fertig: boolean);
var geloest : boolean; testzahl : integer;
begin
  if n < 81 then begin

    if SudokuFeld[n] > 0 then loesen(n+1, fertig)          vorgegebene Zahl --> weiter
    else begin
      testzahl := 1;
      while (not fertig) and (testzahl <= 9) do begin      Schleife von 1 bis 9

        testen (n, testzahl, fertig);                      untersuchen, ob die Zahl passt
                                                           dort ggf. rekursiv weiter
        inc(testzahl,1);
      end;
    end;

  end else fertig := true;
end;


procedure TSudoku.testen(n, testzahl : integer; var fertig: boolean);
var geloest : boolean; t : integer;
begin
  geloest := false;

  if (testzahl >0) and (testzahl <10) and
     (not (testzahl in merkZeile[n div 9])) and           wenn Zahl noch nicht vorhanden
     (not (testzahl in merkSpalte[n mod 9])) and
     not (testzahl in merkKasten[kastenNr(n)])
  then begin                                               Zahl setzen
     merkZeile[n div 9] :=
           merkZeile[n div 9] + [testzahl];                zu den Mengen hinzufügen
     merkSpalte[n mod 9] :=
           merkSpalte[n mod 9] + [testzahl];
     merkKasten[kastenNr(n)] :=
           merkKasten[kastenNr(n)] + [testzahl];
     F[n] := testzahl;                                     in das n-te Feld setzen

     loesen(n+1, geloest);                                 rekursiver Aufruf

     if not geloest then begin                             Backtracking zurueck
        merkZeile[n div 9] :=
              merkZeile[n div 9] - [testzahl];
        merkSpalte[n mod 9] :=
              merkSpalte[n mod 9] - [testzahl];
        merkKasten[kastenNr(n)] :=
              merkKasten[kastenNr(n)] - [testzahl];
        F[n] := 0;                                         0 in das n-te Feld
     end;
  end;

  fertig := geloest;
end;
```

41.5.2 Iterative Lösung

Bei dem rekursiven Aufruf im Rahmen des Backtracking wird jedes Mal eine Art Spielstand gespeichert. Bei der Lösung oben ist es der Parameter n, der angibt, an welcher Stelle des Sudokufeldes wir gerade sind, und der Wert von Testzahl – die Zahl, die an der Stelle n gerade probiert wird.

Bei der iterativen Lösung muss dieser Spielstand auch gespeichert werden. Im Prinzip kann man das mit einem Stack machen, der bei jedem Weitergehen (~ rekursiver Aufruf) die Daten aufnimmt. Beim Zurückgehen (~ Rückkehr aus dem rekursiven Aufruf) kann man diese Daten dann weiter verwenden und wird sie vom Stack löschen.

Bei unserem Problem hier können wir das Tiefersteigen im Zustandsbaum und das Zurückkommen, wenn man in eine Sackgasse geraten ist, dadurch realisieren, dass wir n (Stelle im Sudokufeld) herauf- und herunterzählen.

Dann müssen wir noch speichern, bei welcher Zahl wir an der Stelle n gerade waren – das aber ist SudokuFeld[n] !

Bleibt noch eine Komplikation: Wenn die Zahl schon bei der Aufgabenstellung gesetzt wurde, dürfen wir sie natürlich nicht verändern. Dafür können wir ein

```
gesetzt: array[0..80] of boolean;                    //gesetzte Zahl
```

verwenden, in dem wir nachschauen können, ob die Zahl verändert werden darf.

Der zentrale Algorithmus in Wortform:

loesen (iterativ)

Wir gehen vorwärts, n ist 1

Solange n < 81 und n > = -1 ist

vorwärts: n+1 sonst n-1

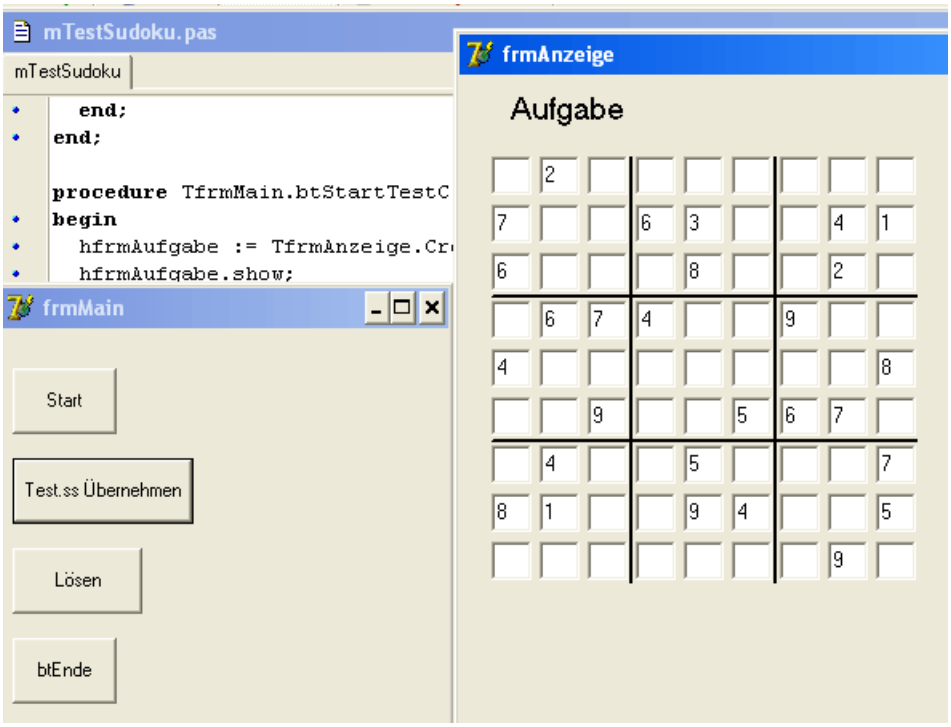
Wenn die Zahl nicht in der Aufgabenstellung steht

Beim Vorwärtsgehen fangen wir mit Testzahl = 1 an,
sonst müssen wir die vorher getestete Zahl (Sudokufeld[n])
aus den Mengen heraus nehmen und mit der nächsten Zahl weiter machen.

Wir suchen die nächste Zahl, die passt.

Wenn wir eine passende Zahl gefunden haben,
fügen wir sie den Testmengen hinzu
notieren sie als Sudokufeld[n] und
gehen vorwärts
sonst (keine passende Zahl gefunden) sind wir in einer Sackgasse
. und gehen zurück.

41.5.3 Programmoberfläche



Anforderungen: Man muss Aufgaben stellen können und Ergebnisse sehen.

Ich wollte die Aufgabenstellungen von einem bekannten Sudokuspiel (simplesudoku) einlesen können, aber auch von Hand Eingaben oder Änderungen machen können, bevor das Spiel gelöst wird.
Dafür ist am besten ein Formular mit einer entsprechenden Zahl von Edit-Komponenten geeignet.
Natürlich wird man diese Edit-Komponenten nicht von Hand auf dem Formular aufziehen. Das Auslesen der Zahlen im Formular und das Anzeigen der Lösung soll ja bitte auch automatisch in einer Schleife erledigt werden.

Also ein

```
edAnzeige : array[0..80] of TEdit;
```

Das Erzeugen und Positionieren der Edit – Komponenten, aber auch Schreib- und Lesevorgänge können jetzt mit Zählschleifen erledigt werden.
Die Linien zum Abteilen der neun Quadrate sind inzwischen Panels (Farbe.clblack, Breite 4); diese sind „wischfest“ im Gegensatz zu einfachen Linien, die man auf den Canvas zeichnet.
Für die Lösung kann man ein weiteres derartiges Fenster erzeugen.

14 Algorithmen

---

42 Grenzen von Algorithmen

Was ist Komplexität, was ist Effizienz?

Bei welchen Gelegenheiten (Unterrichtsvorhaben) wird die Effizienz von Algorithmen untersucht?

Welche Art von Effizienz ?

Definitionen aus dem Duden Informatik:

> *Ein Algorithmus heißt **effizient,** wenn er ein vorgegebenes Problem mit möglichst geringem Ressourcenverbrauch (Zeit, Speicher) löst.*
>
> *Die **Zeiteffizienz** beschreibt den Zeitaufwand zur Lösung eines Problems in Abhängigkeit von der Problemgröße n. Dadurch ist es möglich, von den speziellen Eigenschaften der konkreten Rechenanlage, auf der das betrachtete Programm läuft, zu abstrahieren. Diese Abstraktion wird vertieft, indem nicht etwa die Zeitdifferenzen oder die Anzahl der Elementaroperationen in Programmen, sondern die Takte einer abstrakten Turingmaschine zur Analyse herangezogen werden.*
>
> *Verfahren, deren Zeitverhalten als Polynom in n ausgedrückt werden kann, haben **polynomialen Aufwand** und r ist der Grad des Polynoms*
> $T(n) = a_r n^r + a_{r-1} n^{r-1} + \ldots + a_1 n + a_0$ *mit* $r \varepsilon N, a_r, \ldots, a_0 \varepsilon R, a_r \neq 0.$
> *Wenn r= 1, spricht man von **linearem Aufwand**. r = 2 bzw. r= 3 bedeuten **quadratischer bzw. kubischer Aufwand**.*
>
> *Ein Verfahren besitzt **exponentiellen Aufwand**, wenn die folgende funktionale Beziehung gilt:*
> $T(n) = c \bullet z^n$, *wobei z, c ε R, c≠ 0, z> 1.*

O – Notation (Asymptotische Ordnung)

> *Die **asymptotische Ordnung** O(f), sprich: „groß-O von f", einer Funktion*
> *f:* $N \rightarrow R_0^+$
> *ist die Menge aller Funktionen t :* $N \rightarrow R_0^+$, *die nach oben durch ein positives reelles Vielfaches von f beschränkt werden, wenn n hinreichend groß ist.*
> $O(f) = \{ t \mid t: N \rightarrow R_0^+$ *und es gibt ein* $c \varepsilon R^+$ und ein $n \varepsilon N$, so dass für alle $n \geq n_0$ $(t(n) \leq c \bullet f(n))\}$

Einfache Sortierverfahren (Bubblesort) haben ein Zeitverhalten, das mit $O(n^2)$ skaliert., bei Quicksort ist es $O(n \bullet ld(n))$. Für die folgenden Betrachtungen habe ich das einfachste Sortierverfahren gewählt.

Beim Rucksackproblem ist der Zeitaufwand in $O(2^n)$, wenn wir einen großen Rucksack haben, in den viele Gegenstände passen.

	Bubblesort	Rucksackproblem
as. Ordnung der Bearbeitungszeit	$O(n^2)$	$O(2^{n)}$
für n=100	10000	$1{,}26 \bullet 10^{30}$
Zuwachs bei n=101 (einer mehr)	$101^2 = 100^2 + 200 + 1 = 10000 + 201$ (ca. 5% mehr)	$2^{n+1} = 2^n \bullet 2$ (100% mehr; Verdoppelung des Aufwandes)
Verdoppelung von n	$(2n)^2 = \mathbf{4}\ n^2$ Vervierfachung des Aufwandes	$2^{2n} = 2^{n+n} = \mathbf{2^n} \bullet 2^n$

Relevanz:
Nehmen wir an, ein Lösungsschritt lässt sich in 1000 Takten erledigen, dann leistet ein Gigaherz – Rechner 1 000 000 Schritte in der Sekunde. Es sei **n = 100**:
Die Lösungszeit für Bubblesort ist dann vernachlässigbar.
Beim Rucksackproblem rechnet unser Gigaherz – Rechner dann $1{,}26 \bullet 10^{30} / 1\,000\,000 = 1{,}26 \bullet 10^{24}$ Sekunden.

Das sind bei 31 536 000 Sekunden im Jahr etwa $4 \cdot 10^{16}$ Jahre, das sind 3 Millionen mal das Alter des Universums.
Bei 101 Gegenständen dauert es doppelt so lange ...
Über die Verdoppelung reden wir nicht mehr.

Viele Gegenstände sind das ja nicht – wenn man einen LKW mit Hilfsgütern bepacken oder ein Raumschiff für eine längere Reise ausrüsten will.

Komplexitätsklassen P und EXP

> ***P*** *ist die Klasse aller Probleme, die mithilfe deterministischer Algorithmen in polynomialer Zeit gelöst werden können.*

Man nennt P auch die „Klasse der praktisch lösbaren Probleme“

> ***EXP*** *ist die Klasse aller Probleme, die mithilfe deterministischer Algorithmen in exponentieller Zeit gelöst werden können.*

Probleme, die in EXP, aber nicht in P liegen, sind „praktisch unlösbar“. Wie oben beim Rucksackproblem gezeigt, kann man für sehr kleines n die Lösung noch berechnen, aber der Aufwand wächst so schnell, dass für interessante Größenordnungen eine vollständige (exakte) Lösung in vernünftiger Zeit nicht mehr berechnet werden kann. Eine **Verdoppelung** der Rechenleistung von Computern – wie sie bisher in regelmäßigen Zeitabständen stattfand - erlaubt es auch nur, für **einen** Gegenstand mehr die Lösung (in der selben Zeit) zu berechnen.

Weil Exponentialfunktionen schließlich (für genügend großes x) größere Werte liefern als jede Polynomfunktion, ist P in EXP enthalten.

Komplexitätsklasse NP

Zwischen P und EXP liegt eine Komplexitätsklasse, zu der sehr viele relevante Aufgabenstellungen gehören, auch wenn die Definition der Klasse NP sehr theoretisch klingt. (Wikipedia: P-NP-Problem, ähnlich auch im Duden Informatik)

> *Die Komplexitätstheorie definiert jedoch noch weitere Maschinenmodelle neben der deterministischen Turingmaschine. Ein wichtiges Modell ist die* ***nichtdeterministische Turingmaschine****, welche eine Erweiterung der deterministischen Variante darstellt. Eine nichtdeterministische Turingmaschine hat zu jedem Zeitpunkt eventuell mehrere Möglichkeiten, ihre Berechnung fortzusetzen, der Rechenweg ist deshalb nicht eindeutig bestimmbar. Es handelt sich dabei um ein theoretisches Modell, das nicht gleichwertig zu realen Computern ist. Sein Nutzen ist in diesem Zusammenhang, dass durch nichtdeterministische Turingmaschinen eine weitere Komplexitätsklasse unterschieden werden kann, welche innerhalb von EXP liegt und ihrerseits P einschließt.*
>
> *Die* ***Komplexitätsklasse NP*** *ist die Menge aller von nichtdeterministischen Turingmaschinen in Polynomialzeit lösbaren Probleme.*

Ziegenbalg hingegen stellt fest:

> *Die Komplexitätsklasse NP (...) umfasst alle Entscheidungsprobleme, für die ein vorliegender Lösungsvorschlag in polynomialer Zeit* ***verifiziert*** *werden kann.*

Beispiele für Probleme aus NP: Neben dem Rucksackproblem das Teilsummen-Problem, das Stundenplan – Problem, das Erfüllbarkeitsproblem für boolesche Ausdrücke in konjunktiver Normalform (SAT) und viele Graphenprobleme (Hamiltonkreis, Traveling Salesman Problem, Cliquen, Knotenüberdeckung)

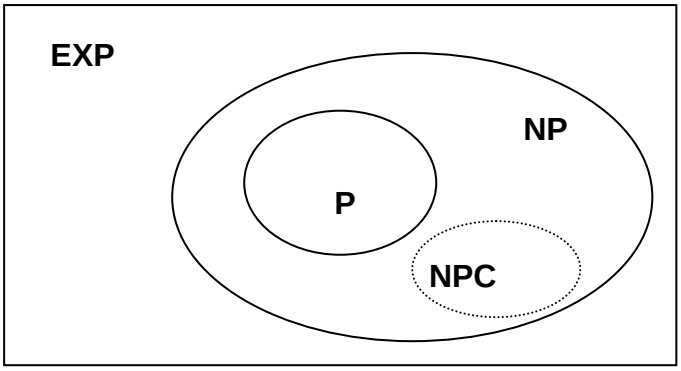
NP = P ?

Wenn ein Problem in polynomialer Zeit gelöst werden kann, ist das ja gleichzeitig auch die Verifikation der Lösung.
Also gilt sicher $P \subseteq NP$

Der Nachweis, dass ein Problem existiert, das zu NP gehört, für das es aber **keinen Lösungsalgorithmus** mit polynomialem Zeitaufwand gibt, ist bisher nicht geführt worden. Ebensowenig konnte man für eines der „NP – Probleme“ eine Lösung mit polynomialem Aufwand finden.
Es handelt sich um eine der wichtigsten offenen (mathematisch – informatischen) Fragestellungen, ein „Millennium-Problem“, für dessen Lösung ein hoher Preis ausgesetzt worden ist (Clay Mathematics Institute):

NP-vollständige Probleme

1971 gelang dem theoretischen Informatiker Stephen Cook der Nachweis, dass das Erfüllbarkeitsproblem für boolesche Ausdrücke in konjunktiver Normalform (SAT) in NP liegt und darüber hinaus: Wenn *dieses* Problem in polynomialer Zeit lösbar sein sollte, so folgt daraus, dass *jedes* Problem in der Komplexitätsklasse NP in polynomialer Zeit lösbar wäre. Das Erfüllbarkeits-Problem ist also „mindestens so schwierig" wie jedes andere in NP liegende Problem. (Ziegenbalg)
Man hat bis heute für etliche weitere Probleme den entsprechenden Nachweis erbracht. Diese Probleme bezeichnet man als **NP-vollständig** (**NP C**omplete) oder **NP-schwer** (**NP hard**)



Der amerikanische Informatiker Richard M. Karp fand 21 Probleme, von denen er die NP-Vollständigkeit nachwies.
(R. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, Complexity of Computer Computations, pages 85–103. Plenum Press, 1972.)

Der folgende Baum (Quelle: Wikipedia) zeigt Karps 21 Probleme, inklusive der zugehörigen Reduktion, die er nutzte, um ihre NP-Vollständigkeit nachzuweisen. Zum Beispiel wurde die NP-Vollständigkeit des Rucksackproblems gezeigt, indem das Problem der exakten Überdeckung darauf reduziert wurde.

* SATISFIABILITY: das Erfüllbarkeitsproblem der Aussagenlogik für Formeln in Konjunktiver Normalform
  Das Erfüllbarkeitsproblem der Aussagenlogik (SAT, von englisch satisfiability) ist ein Entscheidungsproblem. Es fragt, ob eine aussagenlogische Formel erfüllbar ist. Anwendungen finden sich unter anderem in der Komplexitätstheorie, Verifikation und im Entwurf von logischen Schaltungen. Insbesondere in der Komplexitätstheorie wird mit SAT auch nur der Spezialfall von Formeln bezeichnet, die in konjunktiver Normalform vorliegen. Das Erfüllbarkeitsproblem der Aussagenlogik ist in exponentieller Zeit in der Anzahl der Variablen entscheidbar, zum Beispiel durch das Aufstellen einer Wahrheitstabelle. Ob es auch einen Algorithmus gibt, der SAT in Polynomialzeit löst, ist nicht bekannt. Die Forschung beschäftigt sich mit der Entwicklung möglichst effizienter Verfahren (sogenannter SAT-Solver).

  o CLIQUE: Cliquenproblem
    Es ist gefragt, ob es zu einem einfachen Graphen G und einer Zahl n eine Clique der Mindestgröße n in G gibt; das heißt, ob G zumindest n Knoten hat, die alle untereinander paarweise verbunden sind.

    + SET PACKING: Mengenpackungsproblem
      Das Mengenpackungsproblem (oft mit set-packing-Problem notiert) ist ein Entscheidungsproblem der Kombinatorik.
      Es fragt, ob zu einer endlichen Menge U und n Teilmengen Sj von U eine Anzahl von mindestens $k \leq n$ *paarweise disjunkter Teilmengen Sj existieren.*

    + VERTEX COVER: Knotenüberdeckungsproblem
      Das Knotenüberdeckungsproblem (oft mit VERTEX COVER notiert) ist ein Problem der Graphentheorie.
      Es fragt, ob zu einem gegebenen einfachen Graphen und einer natürlichen Zahl k eine Knotenüberdeckung der Größe von höchstens k existiert. Das heißt, ob eine aus maximal k Knoten bestehende Teilmenge U der Knotenmenge gibt, sodass jede Kante des Graphen mit mindestens einem Knoten aus U verbunden ist.

SET COVERING: Mengenüberdeckungsproblem

Das Mengenüberdeckungsproblem (oft mit set-covering-Problem notiert) ist ein Entscheidungsproblem der Kombinatorik.
Es fragt, ob zu einer Menge U und n Teilmengen Sj von U und einer natürlichen Zahl $k \leq n$ *eine Vereinigung von k oder weniger Teilmengen Sj existiert, die der Menge U entspricht (Überdeckung).*
Als Optimierungsproblem formuliert, wird eine Überdeckung mit möglichst kleiner Anzahl der Teilmengen Sj gesucht oder, falls den Teilmengen Sj Kosten cj zugeordnet sind, eine Überdeckung mit geringsten Kosten.

FEEDBACK ARC SET: Feedback Arc Set

Der Begriff Feedback Arc Set stammt aus der Graphentheorie und bezeichnet eine Menge von Kanten, durch deren Entfernung aus einem Graphen dieser azyklisch, d.h. kreisfrei wird.(bei gerichteten Graphen).

FEEDBACK NODE SET: Feedback Vertex Set

In the mathematical discipline of graph theory, a feedback vertex set of a graph is a set of vertices whose removal leaves a graph without cycles. In other words, each feedback vertex set contains at least one vertex of any cycle in the graph. The feedback vertex set problem is an NP-complete problem in computational complexity theory. It was among the first problems shown to be NP-complete. It has wide applications in operating system, database system, genome assembly, and VLSI chip design.

DIRECTED HAMILTONIAN CIRCUIT:

*Ein **Hamiltonkreis** ist ein geschlossener Pfad in einem Graphen, der jeden Knoten genau einmal enthält. Ob ein solcher Kreis in einem gegebenen Graph besteht, ist ein fundamentales Problem der Graphentheorie. Im Gegensatz zum leicht lösbaren Eulerkreisproblem, bei dem alle Kanten genau einmal durchlaufen werden, ist das **Hamiltonkreisproblem** NP-vollständig.*

* UNDIRECTED HAMILTONIAN CIRCUIT: siehe Hamiltonkreisproblem

o 0-1 INTEGER PROGRAMMING: siehe Integer Linear Programming

Nichtlineare und ganzzahlige Optimierung
Viele Anwendungsprobleme lassen sich mit kontinuierlichen Variablen nicht sinnvoll modellieren, sondern erfordern die Ganzzahligkeit einiger Variablen. Beispielsweise können keine 3,7 Flugzeuge gekauft werden, sondern nur eine ganze Anzahl, und ein Bus kann nur ganz oder gar nicht fahren, aber nicht zu zwei Dritteln. Bei der Verwendung von Branch-and-Cut zur Lösung eines solchen ganzzahligen linearen Optimierungsproblems müssen sehr viele ähnliche lineare Programme hintereinander als Unterproblem gelöst werden. Eine optimale ganzzahlige Lösung eines linearen Programms zu finden ist NP-vollständig, aber parametrisierbar in der Anzahl der Variablen. Es ist sogar NP-vollständig, irgendeine ganzzahlige Lösung eines linearen Programms zu finden. Auch zur Lösung nichtlinearer Optimierungsprobleme gibt es Algorithmen, in denen lineare Programme als Unterproblem gelöst werden müssen

o 3-SAT: siehe 3-SAT

3-SAT ist eine Variante des Erfüllbarkeitsproblems der Aussagenlogik (Erfüllbarkeit engl.: satisfiability, kurz SAT).
Es beschäftigt sich mit der Frage, ob eine in konjunktiver Normalform vorliegende aussagenlogische Formel F, die höchstens 3 Literale pro Klausel enthält, erfüllbar ist. Ein Beispiel für eine solche Formel:

$$F = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge (x_1 \vee \overline{x_2})$$

Gesucht ist nun eine Belegung der Variablen x1 bis x4 mit 0 oder 1, für die F den Wert 1 (wahr) annimmt. Falls es eine solche Belegung gibt, ist F erfüllbar, sonst nicht. Wie bei allen NP-vollständigen Problemen ist es "einfach", einen Lösungskandidaten auf seine Gültigkeit zu überprüfen, hier also festzustellen, ob eine vorgegebene Belegung der Variablen die Formel erfüllt. Das Auffinden eines gültigen Lösungskandidaten ist jedoch im Allgemeinen "schwierig", da heute keine Methode bekannt ist, eine erfüllende Belegung in polynomieller Zeit zu finden.

+ CHROMATIC NUMBER: graph coloring problem

Eine Färbung eines ungerichteten Graphen ordnet jedem Knoten bzw. jeder Kante im Graphen eine Farbe zu.
In der Graphentheorie beschäftigt man sich meist nur mit sogenannten „zulässigen“ oder „gültigen“ Färbungen (siehe unten), und versucht, Algorithmen zu entwickeln, die für einen vorgegebenen Graphen eine gültige Färbung mit möglichst wenig Farben finden. Probleme aus der diskreten Mathematik, aber auch außermathematische Fragestellungen lassen sich manchmal in ein Färbungsproblem übersetzen, daher ist die Existenz oder Nichtexistenz solcher Algorithmen auch außerhalb der Graphentheorie von Interesse.

CLIQUE COVER: Covering by cliques

The clique cover problem (also sometimes called partition into cliques) is the problem of determining whether the vertices of a graph can be partitioned into k cliques. Given a partition of the vertices into k sets, it can be verified in polynomial time that each set forms a clique, so the problem is in NP. The NP-completeness of clique cover follows by reduction from GRAPH k-COLOURABILITY.

EXACT COVER: Problem der exakten Überdeckung

Das Problem der exakten Überdeckung (englisch Exact Cover) ist ein Entscheidungsproblem der Kombinatorik. Es gehört zu den 21 klassischen NP-vollständigen Problemen, von denen Richard M. Karp 1972 gezeigt hat, dass sie NP-vollständig sind.
Gegeben ist eine Menge X und eine Menge S, die Teilmengen von X enthält, also $S \subseteq \mathcal{P}(X)$*, wobei* $\mathcal{P}(X)$ *die Potenzmenge von X bezeichnet.*
Gesucht ist eine Teilmenge U von S, deren Disjunkte Vereinigung X ist:

$$X = \bigcup_{X_i \in U} X_i \quad .$$

D. h. jedes Element in X soll in genau einer der Mengen in U vorkommen. Die Mengen in U bilden also eine exakte Überdeckung von X.

* 3-dimensional MATCHING: 3-dimensional matching

In the mathematical discipline of graph theory, a 3-dimensional matching is a generalization of bipartite matching (a.k.a. 2-dimensional matching) to 3-uniform hypergraphs.

* STEINER TREE: Steinerbaumproblem

Das Steinerbaumproblem (oft mit STEINER TREE notiert), ein nach dem Schweizer Mathematiker Jakob Steiner benanntes Problem der Graphentheorie, ist eine Verallgemeinerung des Problems des minimalen Spannbaums. Beim Steinerbaumproblem sucht man in einem umgebenden Graphen einen kleinsten Teilgraphen (den Steinerbaum), welcher eine Menge vorgegebener Endpunkte (die Terminale) miteinander verbindet.

* HITTING SET: Hitting set

Gegeben ist eine Menge von Teilmengen S eines „Universums“ T, gesucht ist eine Teilmenge H von T so, dass jede Menge in S mindestens ein Element aus H enthält. Zusätzlich ist gefordert, dass die Anzahl der Elemente von H einen gegebenen Wert k nicht überschreitet.

* KNAPSACK: Rucksackproblem

Das Rucksackproblem (oft mit RUCKSACK, KNAPSACK bezeichnet) ist ein Optimierungsproblem der Kombinatorik. Aus einer Menge von Objekten, die jeweils ein Gewicht und einen Nutzwert haben, soll eine Teilmenge ausgewählt werden, deren Gesamtgewicht eine vorgegebene Gewichtsschranke nicht überschreitet. Unter dieser Bedingung soll der Nutzwert der ausgewählten Objekte maximiert werden.
Die Entscheidungsvariante des Rucksackproblems fragt, ob ein zusätzlich vorgegebener Nutzwert erreicht werden kann. Sie gehört zur Liste der 21 klassischen NP-vollständigen Probleme, von denen Richard Karp 1972 die Zugehörigkeit zu dieser Klasse zeigen konnte.

o JOB SEQUENCING: Job sequencing

o PARTITION: Partitionsproblem

Die Aufgabenstellung beim Partitionsproblem lautet: Gegeben sei eine (Multi-)Menge von natürlichen Zahlen. Gesucht wird eine Aufteilung dieser Zahlen auf zwei Haufen, so dass die Differenz der Summen der Zahlen in den beiden Haufen möglichst klein ist.

+ MAX-CUT: Max cutNäherungsverfahren

Der maximale Schnitt eines Graphen ist eine Zuordnung seiner Knotenmenge V in zwei Partitionen (S,T), so dass das Gesamtgewicht der zwischen den beiden Partitionen

verlaufenden Kanten maximal wird. Im Gegensatz zum minimalen Schnitt ist das Problem NP-vollständig.

Weitere NP-vollständige Probleme

- **Problem des Hamiltonschen Weges**: Gegeben ist ein (zusammenhängender) Graph. Gesucht ist ein Hamiltonscher Rundweg für diesen Graphen.
- **Problem des Handlungsreisenden** (traveling salesman problem): Gegeben ist ein (zusammenhängender) bewerteter Graph. Gesucht ist eine kosten-optimale Rundreise, die jeden Knoten des Graphen durchläuft.
- **Kasten-Problem** (bin packing problem): Gegeben sind n Gegenstände sowie eine unbestimmte Anzahl von Kästen („bins"); jeder Kasten habe die Kapazität L. Gegenstand Nr. i benötige die Kapazität K(i) (mit „Kapazität" sei z.B. das Gewicht, Volumen oder ähnliches gemeint). Gesucht ist die minimale Anzahl von Kästen, in denen man die gegebenen n Gegenstände verstauen kann. (Kein Gegenstand darf auf mehrere Kästen aufgeteilt werden.)
- **Stundenplan-Problem** (time table problem) Gegeben: (Schul-) Klassen, Fächer, Lehrer und Lehrerinnen Gesucht: Organisation eines Stundenplanes mit möglichst wenig „Zeitlücken".

Näherungsverfahren

- branch and bound mit Schätzfunktion für die Schranke

  Im Optimierungsproblem $f(x) \rightarrow \min!$ *bei* $x \in D$ *sei der zulässige Bereich D eine diskrete Menge, eventuell sogar endlich. Alle zugelassenen Belegungen* $x \in D$ *durchzuprobieren, scheitert meist an inakzeptabel langen Rechenzeiten. Deshalb wird D nach und nach in mehrere Teilmengen aufgespalten (Branch). Mittels geeigneter Schranken (Bound) sollen viele suboptimale Belegungen frühzeitig erkannt und ausgesondert werden, so dass der zu durchmusternde Lösungsraum klein gehalten wird. Im ungünstigsten Fall werden alle Belegungen aufgezählt (vollständige Enumeration).*
- greedy

  Greedy-Algorithmen oder gierige Algorithmen bilden eine spezielle Klasse von Algorithmen, die in der Informatik auftreten. Sie zeichnen sich dadurch aus, dass sie schrittweise den Folgezustand auswählen, der zum Zeitpunkt der Wahl den größten Gewinn bzw. das beste Ergebnis verspricht (z. B. Gradientenverfahren). Um unter den Folgezuständen eine Auswahl zu treffen, wird oft eine Bewertungsfunktion verwendet.
  Greedy-Algorithmen sind meist schnell, lösen viele Probleme aber nicht optimal.
- divide and conquer

  Bei einem „teile und herrsche"-Ansatz wird das eigentliche Problem so lange in kleinere und einfachere Teilprobleme zerlegt, bis man diese lösen („beherrschen") kann. Anschließend wird aus diesen Teillösungen eine Lösung für das Gesamtproblem (re-)konstruiert.
  Die Suche in sortierten Listen kann nach diesem Prinzip erfolgen. Hierzu wird das Element in der Mitte der Liste mit dem gesuchten Eintrag verglichen. Anschließend muss nur noch in dem vorhergehenden oder nachfolgenden Teil weiter gesucht werden. Dazu kann dieser erneut auf diese Art und Weise halbiert werden. Diese Suchmethode, heutzutage bekannt als Binäre Suche, geht bereits auf die Babylonier zurück.
  Der Euklidische Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen kann als eine einfache Art des „teile und herrsche" Prinzips betrachtet werden. Hierbei wird das Problem iterativ vereinfacht, indem man „gemeinsame" Teile entfernt.
  „Teile und herrsche" ist eines der wichtigsten Prinzipien für effiziente Algorithmen. Dabei wird ausgenutzt, dass bei vielen Problemen der Lösungsaufwand sinkt, wenn man das Problem in kleinere Teilprobleme zerlegt. Dies lässt sich meist durch Rekursive Programmierung umsetzen, bei der die Teilprobleme wie eigenständige Probleme gleichzeitig parallel oder sequenziell (einzeln nacheinander) behandelt werden, bis sie auf triviale Lösungen zurückgeführt sind oder der Restfehler hinreichend klein ist. Bei manchen Algorithmen steckt dabei die Kernidee im Schritt des „Teilens", während die „Rekombination" einfach ist (beispielsweise Quicksort). In anderen Verfahren (beispielsweise Mergesort) ist das Teilen

einfach, während die Rekombination die Kernidee des Algorithmus enthält. In manchen Algorithmen sind beide Schritte komplex.

- stochastische Verfahren

  Ein randomisierter Algorithmus (auch stochastischer oder probabilistischer Algorithmus) verwendet – im Gegensatz zu einem deterministischen Algorithmus – Zufallsbits, um seinen Ablauf zu steuern. Es wird nicht verlangt, dass ein randomisierter Algorithmus immer effizient eine richtige Lösung findet. Randomisierte Algorithmen sind in vielen Fällen einfacher zu verstehen, einfacher zu implementieren und effizienter als deterministische Algorithmen für dasselbe Problem. Ein Beispiel, das dies zeigt, ist der AKS-Primzahltest, der zwar deterministisch ist, aber viel ineffizienter und viel schwieriger zu implementieren als beispielsweise der Primzahltest von Solovay und Strassen.
  Las-Vegas
  Randomisierte Algorithmen, die nie ein falsches Ergebnis liefern, bezeichnet man auch als Las-Vegas-Algorithmen. Es gibt zwei Varianten von Las-Vegas-Algorithmen:
  Algorithmen, die immer das richtige Ergebnis liefern, deren Rechenzeit aber (bei ungünstiger Wahl der Zufallsbits) sehr groß werden kann. In vielen Fällen ist der Algorithmus nur im Erwartungsfall schnell, nicht aber im schlimmsten Fall. Ein Beispiel ist die Variante von Quicksort, bei der das Pivotelement zufällig gewählt wird. Die erwartete Rechenzeit beträgt O(nlog n), bei ungünstiger Wahl der Zufallsbits werden dagegen O(n2) Schritte benötigt.
  Algorithmen, die versagen dürfen (= aufgeben dürfen), aber niemals ein falsches Ergebnis liefern dürfen. Die Qualität dieser Art von Algorithmen kann man durch eine obere Schranke für die Versagenswahrscheinlichkeit beschreiben.
  Monte-Carlo
  Randomisierte Algorithmen, die auch ein falsches Ergebnis liefern dürfen, bezeichnet man auch als Monte-Carlo-Algorithmen. Die Qualität von Monte-Carlo-Algorithmen kann man durch eine obere Schranke für die Fehlerwahrscheinlichkeit beschreiben.
  Von randomisierten Algorithmen spricht man nur, wenn man den Erwartungswert der Rechenzeit und/oder die Fehler- bzw. Versagenswahrscheinlichkeit abschätzen kann. Algorithmen, bei denen nur intuitiv plausibel ist, dass sie gute Ergebnisse liefern, oder Algorithmen, bei denen man dies durch Experimente auf typischen Eingaben bewiesen hat, bezeichnet man dagegen als heuristische Algorithmen.
  Bei der Analyse von erwarteter Rechenzeit und/oder Fehlerwahrscheinlichkeit geht man in der Regel davon aus, dass die Zufallsbits unabhängig erzeugt werden. In Implementierungen verwendet man anstelle von richtigen Zufallsbits üblicherweise Pseudozufallszahlen, die natürlich nicht mehr unabhängig sind. Dadurch ist es möglich, dass sich Implementierungen schlechter verhalten, als die Analyse erwarten lässt
  Shor-Algorithmus
  Der Shor-Algorithmus ist ein Algorithmus aus dem mathematischen Teilgebiet der Zahlentheorie, der Mittel der Quanteninformatik benutzt. Er berechnet auf einem Quantencomputer einen nichttrivialen Teiler einer zusammengesetzten Zahl und zählt somit zur Klasse der Faktorisierungsverfahren.
- Sintflut – Algorithmus

  Der Sintflutalgorithmus (englisch great deluge algorithm) ist ein heuristisches Optimierungsverfahren der Informatik. Es ist verwandt mit Simulierter Abkühlung und wird meist für Optimierungsprobleme eingesetzt, die durch ihre hohe Komplexität das vollständige Ausprobieren aller Möglichkeiten und einfache mathematische Verfahren ausschließen.
  Die Idee ist, eine zufällige Suche im Suchraum durch einen steigenden Wasserspiegel mit der Zeit einzuschränken. Dazu werden ein Schwellwert W (Wasserstand) und eine Konstante R (Regen) definiert. Von einem zufälligen Startwert x ausgehend wird nun iterativ ein neuer Wert x' im Suchraum erzeugt und genau dann akzeptiert, wenn er oberhalb von W liegt, d.h. er muss besser sein als W. Er darf aber schlechter sein als x. W wird dabei regelmäßig um R erhöht. Bildlich verkleinern sich dadurch die begehbaren Regionen des Suchraums, so dass der Algorithmus zwar anfänglich lokale Optima überwinden kann, indem er niedere Regionen durchquert, mit der Zeit aber in einen Bergsteigeralgorithmus übergeht.
  Wie auch Simulierte Abkühlung ist der Sintflutalgorithmus in der Regel hinsichtlich des gefundenen (lokalen) Optimums weniger effizient als etwa Evolutionsstrategien, dafür aber nicht so aufwändig.

- Evolutionäre Algorithmen

Ein Evolutionärer Algorithmus (EA) ist ein Optimierungsverfahren, das als Vorbild die biologische Evolution hat. Dabei werden Individuen durch ihre Eigenschaften (i.A. in Zahlenwerten) beschrieben; sie müssen sich bzgl. der Selektionsbedingungen als möglichst geeignet behaupten, und dürfen dementsprechend ihre Eigenschaften vererben - oder eben nicht. Im Laufe mehrerer Durchläufe entwickelt sich so die „Bevölkerung" immer näher an das Optimum.

- Neuronale Netze

Links zur Fortbildung

Bücher:
Algorithmen – von Hammurapi bis Gödel; Ziegenbalg, … ; Verlag Harri Deutsch; Frankfurt 2007

Duden Informatik SII; Berlin 2006

P=NP, NP-Vollständigkeit, Cook … in: Wikipedia

NP-schwere Probleme in LOG IN Nr. 146/147 und 148/149
Dazu: Programm NP-Schule
http://e4.mirz.uni-jena.de/np ??

Vorlesungen zur Theoretischen Informatik an versch. Universitäten (u.a. Jena, Koblenz, Paderborn)

Demoprogramme und viel Material
http://www.gymnasium-odenthal.de