Programme und ausgewählte Algorithmen aus dem Buch:

Algorithmen
- von Hammurapi bis Goedel

Autoren:
Jochen Ziegenbalg
Oliver Ziegenbalg
Bernd Ziegenbalg

4., überarbeitete und erweiterte Auflage:
Verlag Springer Spektrum

ISBN 978-3-658-12362-8

Die Programme sind in der Reihenfolge aufgeführt, wie sie im Buch auftreten.

```
Sortieren(Kartenmenge);
  Suche die niedrigste Karte.
  Lege sie als erste Karte des sortierten Teils
    auf den Tisch.
  Solange der unsortierte Rest nicht leer ist,
    tue folgendes:
    [Suche die niedrigste Karte im unsortierten Rest.
     Lege sie rechts neben die höchste Karte
     des sortierten Teils.]
Ende Sortieren.
```

```
Sortieren(Kartenmenge);
  Wenn weniger als zwei Karten vorhanden, dann fertig;
  sonst
    wähle eine Karte aus der unsortierten Menge
      (als "Trennelement")
      und lege sie auf den Tisch;
    lege der Reihe nach
      alle Karten mit einem größeren Wert rechts
      und alle Karten mit einem kleineren Wert links
        neben die ausgewählte Karte
        und lege alle Karten mit demselben Wert
          auf die ausgewählte Karte;
    nenne die Teilhaufen KL (für "kleinere"),
                         GL (für "gleich große") und
                         GR (für "größere") Karten;
    führe das Verfahren
      Sortieren(KL) und
      Sortieren(GR)
      durch und füge die Karten aus GL
        zwischen dem Ergebnis von Sortieren(KL)
        und Sortieren(GR) ein
Ende Sortieren.
```

```
Heron(a)
  Hilfsvariable:  x, y, xneu, yneu;
  (* Vereinbarung von lokalen Hilfsvariablen *)
  x:=a;  y:=1;     (*  :=  Wertzuweisung  *)
  Solange |x^2 - a|  >  0.000001 tue folgendes:
    [ xneu := (x+y)/2;   (* Die eckigen Klammern    *)
      yneu := a/xneu;    (* legen den Gueltigkeits- *)
      x := xneu;         (* bereich der Solange-    *)
      y := yneu ];       (* Kontrollstruktur fest.  *)
  Rueckgabe(x)           (* x wird als Funktions-   *)
Ende.                    (* wert zurueckgegeben.    *)
```

```
1:  Heron[a_] :=
2:    Module[{x=a, y=1, xneu, yneu},
3:      While[Abs[x^2-a] > 0.000001,
4:         xneu = (x+y)/2;
5:         yneu = a/xneu;
6:         x = xneu;
7:         y = yneu ];
8:      Return[x] ]
```

```
heron(a) :=
  block([x : a],
    while abs(a-x*x) > 0.000001  do  x : (x+a/x)/2,
    x);
```

```
1: EuklidSubtraktionsform(a, b)
2:   Solange a und b beide von Null verschieden sind,
3:     fuehre folgendes aus:
4:     Wenn a ? b, so ersetze a durch a-b,
5:                 sonst ersetze b durch b-a.
6:   Die uebrig bleibende, von Null verschiedene ganze
7:   Zahl ist der gesuchte groesste gemeinsame
8:   Teiler GGT(a, b).
```

```
1:  EuklidSub[a0_, b0_] :=
2:    Module[{a=a0, b=b0},
3:      While[Not[ a*b == 0],
4:        (* solange a und b beide
5:           von Null verschieden sind *)
6:        Print[a, "   ", b];
7:        If[a >= b, a = a-b, b = b-a ] ];
8:      Return[a+b] (* Jetzt ist einer der
9:                   Summanden gleich Null *) ]
```

```
EuklidDiv[a0_, b0_] :=
  Module[{a=a0, b=b0},
    While[Not[ a*b == 0],
      Print[a, "   ", b];
      If[a >= b, a = Mod[a, b], b = Mod[b,a] ] ];
    Return[a+b] (* Einer der Summanden ist Null *) ]



1:  EuklidSubRek[a_, b_] :=
2:    (Print[a, "   ", b];
3:     Which[a == 0, b,
4:           b == 0, a,
5:           a >= b, EuklidSubRek[a-b, b],
6:           a <  b, EuklidSubRek[a, b-a] ] )



EuklidDivRek[a_, b_] :=
  (Print[a, "   ", b];
   Which[a == 0, b,
         b == 0, a,
         a >= b, EuklidDivRek[Mod[a,b], b],
         a <  b, EuklidDivRek[a, Mod[b,a]] ] )



EuklidReg[a_, 0] = a;
EuklidReg[a_, b_] := EuklidReg[b, Mod[a, b]];



euclid_div(a, b) :=
  block([x : a, y : b],
    while not(x*y = 0) do
      if x > y then x : mod(x, y) else y : mod(y, x),
    return(x+y) );



Eratosthenes(UpperLimit) :=
  /* etwa wie in BYTE - nur richtig */
  block([A, i, k],
    A : make_array(fixnum, UpperLimit+1),
    for i : 0 thru UpperLimit do (A[i] : i),
    A[1] : 0,
    i : 2,
    while i*i <= UpperLimit do
     (k : i+i,
      while k <= UpperLimit do
        (A[k] : 0,
         k : k+i),
      i : i+1),
    delete(0, listarray(A)) );



nexts[s_] := r * Sqrt[2 - 2*Sqrt[1 - (s/(2*r))^2]];
```

```
s[n_] := Nest[nexts, s3, Log[2, n/3]];
S[n_] := s[n] / Sqrt[1 - (s[n]/(2*r))^2];
u[n_] := n * s[n];
U[n_] := n * S[n];



TableForm[
  Table[
    {n, 3*2^n, N[u[3*2^n]/(2*r)], N[U[3*2^n]/(2*r)]},
    {n, 0, 11} ] ]



Pi_Archimedes_Wolff(steps) :=
  block([r:1, se, su, ue, uu, i, n:3],
    se : sqrt(3),  /* initial values             */
    ue : 3 * se,   /* for the "triangle"-polygon */
    su : 2 * sqrt(3),
    uu : 3 * su,
    printf(true, "~2d ~10d ~13, 10h ~13, 10h ~43,
           40h ~%", 0, n, ue/2, uu/2, se*se),
    for i : 1 step 1 thru steps do
      (n  : n * 2,
       se : r*sqrt(2-2*sqrt(1-(se/(2*r))*(se/(2*r)))),
       ue : n * se,
       su : se / sqrt(1 - (se/(2*r)) * (se/(2*r)) ),
       uu : n * su,
       printf(true, "~2d ~10d ~13,10h ~13,10h ~43,
              40h ~%", i, n, ue/2, uu/2, se*se) ),
    bfloat((ue/2+uu/2)/2) );



GoldbachZerlegungen(n) :=
 block([test : 2, G : [] ],
  while 2 * test <= n do
   (if primep(n - test)
     then G : append(G, [[test, n - test]]),
    test : next_prime(test)),
  G );



GoldbachVermutungGegenbeispiel(start) :=
  block([i : start, gefunden : false],
   while not gefunden do
   (if is(GoldbachZerlegungen(i) = [])
     then return(i),
    i : i+2) );
```

```
Stammbruch[a_, b_] :=
  Module[{x, n, t},
    n=1; x=a/b; t={Floor[x]}; x=x-Floor[x];
    While[x>0,
      n=n+1;
      If[x>=(1/n), x=x-1/n; t=Append[t,1/n] ] ];
    Return[t] ]



StammbruchEffizienter[a_, b_] :=
  Module[{x, n, t},
    x=a/b; t={Floor[x]}; x=x-Floor[x];
    While[x>0,
      n=Ceiling[1/x];
      x=x-1/n;
      t=Append[t,1/n] ];
    Return[t] ]



f(n) := if n = 1 then 1 else n*f(n-1)



fib[n_] := Which[n==0, 0,
                 n==1, 1,
                 n>1,  fib[n-1]+fib[n-2] ]



fibit[n_] :=
  Module[{f0=0, f1=1, f2=1, i=0},
    While[i<n, (i=i+1; f0=f1; f1=f2; f2=f0+f1)];
    Return[f0] ]



hanoi[n_, start_, hilf_, ziel_] :=
  If[ n==1, {{start, ziel}},
            Join[hanoi[n-1, start, ziel, hilf],
                 {{start, ziel}},
                 hanoi[n-1, hilf, start, ziel]] ]




kleiner[TE_, L_] :=
  Which[
    L=={}, {},
    First[L] < TE,
      Prepend[kleiner[TE, Drop[L, 1]], First[L] ],
    First[L] >= TE, kleiner[TE, Drop[L, 1]] ]

gleich[TE_, L_] :=
  Which[
    L=={}, {},
    First[L] == TE,
      Prepend[gleich[TE, Drop[L, 1]], First[L] ],
    First[L] != TE, gleich[TE, Drop[L, 1]] ]
```

```
groesser[TE_, L_] :=
  Which[
    L=={}, {},
    First[L] > TE,
      Prepend[groesser[TE, Drop[L, 1]], First[L] ],
    First[L] <= TE, groesser[TE, Drop[L, 1]] ]

quicksort[L_] :=
  If[L=={}, {},
            Join[quicksort[kleiner[First[L], L]],
                 gleich[First[L], L],
                 quicksort[groesser[First[L], L]] ]]



sep[L_, TE_] :=    (* fuer separiere *)
  Module[{L1=L, KL={}, GL={}, GR={} },
    While[L1 != {},   (* != ... ungleich *)
      Which[
        First[L1] < TE, AppendTo[KL, First[L1]],
        First[L1]== TE, AppendTo[GL, First[L1]],
        First[L1] > TE, AppendTo[GR, First[L1]] ];
      L1 = Rest[L1] ];
    Return[List[KL, GL, GR]] ]

qs[L_] :=    (* quicksort unter Verwendung von sep *)
  If[L=={}, {},
    Module[{S=sep[L, First[L]], KL, GL, GR},
      KL=First[S];  GL=First[Rest[S]];  GR=Last[S];
      Return[Join[qs[KL], GL, qs[GR] ] ] ] ]




Wurzel[Baum_] := First[Baum]
Folgebaeume[Baum_] := Rest[Baum]



ts[B_] := ts1[Wurzel[B], Folgebaeume[B]]
          (* Tiefensuche *)
ts1[W_, BB_] :=
  (Print[W];
   If[Not[BB=={}],
      ts1[Wurzel[First[BB]],
          Join[Folgebaeume[First[BB]], Rest[BB]] ] ] )



tsf[B_] :=
  If[B=={}, {},
     Prepend[
       Apply[Join, Map[tsf, Folgebaeume[B]]],
                   Wurzel[B] ] ]
```

```
bs[B_] := bs1[Wurzel[B], Folgebaeume[B]]
          (* Breitensuche *)
bs1[W_, BB_] :=
 (Print[W];
  If[Not[BB=={}],
     bs1[Wurzel[First[BB]],
         Join[Rest[BB],
              Folgebaeume[First[BB]] ] ] ] )



bsf[B_] := bsf1[Wurzel[B], Folgebaeume[B]]
bsf1[W_, BB_] :=
  If[BB=={}, {W},
     Prepend[
       bsf1[Wurzel[First[BB]],
           Join[Rest[BB], Rest[First[BB]]] ],
       W ] ]



Warenkorb = {{a, 40, 700}, {b, 100, 1500},
             {c, 80, 900}, {d, 50, 700},
             {e, 120, 1700}, {f, 130, 2000},
             {g, 30, 500} }
Name[G_] := First[G]           (* G:  Einzel-Gut *)
Wert[G_] := First[Rest[G]]
Gewicht[G_] := Last[G]
Namen[Bag_] := Map[Name, Bag]  (* Bag: Teilmenge der
                                       Gueter *)
Werte[Bag_] := Map[Wert, Bag]
Gesamtwert[Bag_] := Apply[Plus, Werte[Bag]]
Gewichte[Bag_] := Map[Gewicht, Bag]
Gesamtgewicht[Bag_] := Apply[Plus, Gewichte[Bag]]

Optimum[B1_, B2_] :=
  If[Gesamtwert[B1] >= Gesamtwert[B2], B1, B2]

Rucksack[W_, L_] :=
  (* W:  Waren;  L:  Gewichts-Limit *)
  Module[{G1, GR, W1, WR},
    (*  Print[Namen[W], "   ", L];  *)
    If[W == {}, {},
    ( G1 = Gewicht[First[W]];
      GR = Gesamtgewicht[Rest[W]];
      W1 = Wert[First[W]];
      WR = Gesamtwert[Rest[W]];
      Which[
        G1 <= L, Optimum[
                   Prepend[Rucksack[Rest[W], L-G1],
                           First[W]],
                   Rucksack[Rest[W], L] ],
        True, Rucksack[Rest[W], L] ]) ] ]
```

```
damen(n) :=
 (Position : make_array(fixnum, n+1),
  fillarray(Position, makelist(1, j, 0, n+1)),
  ZL_frei : make_array(fixnum, n+1),
  fillarray(ZL_frei, makelist(1, j, 0, n+1)),
  HD_frei : make_array(fixnum, 2*(n+1)),
  fillarray(HD_frei, makelist(1, j, 0, 2*(n+1))),
  ND_frei : make_array(fixnum, 2*(n+1)),
  fillarray(ND_frei, makelist(1, j, 0, 2*(n+1))),
  anzahl_loesungen : 0,
  plaziere_damen_ab_spalte(1, n),
  anzahl_loesungen
 );

plaziere_damen_ab_spalte(s, n) :=
 for z : 1 thru n do
  (if (ZL_frei[z]=1 and
       HD_frei[z-s+n]=1 and
       ND_frei[s+z]=1)
    then
     (Position[s]:z,
      ZL_frei[z]:0,
      HD_frei[z-s+n]:0,
      ND_frei[s+z]:0,
      if s<n then plaziere_damen_ab_spalte(s+1, n)
       else
        (anzahl_loesungen : anzahl_loesungen+1,
         print("Loesung: ",
                rest(listarray(Position), 1) ) ),
      ZL_frei[z]:1,     /* Freigabe für die Suche */
      HD_frei[z-s+n]:1, /* nach weiteren Lösungen */
      ND_frei[s+z]:1 ),
    anzahl_loesungen
  );




vollstaendiger_Satz() :=
  block([sammler_array, r, i :0 ],
    make_random_state(true),
    sammler_array : make_array(fixnum, 7),
    fillarray(sammler_array, makelist(0, j, 0, 6)),
        /* den sammler_array mit Nullen auffuellen */
    while
      is(apply("*",
          rest(listarray(sammler_array))) = 0)
    do (i : i+1,
        r : random(6)+1,
        sammler_array[r] : sammler_array[r]+1,
        if verbose then print(r) ),
    return([i, rest(listarray(sammler_array))]) ) ;
```

```
Ziegenproblem(AnzahlDerVersuche) :=
 block(wert_tuer : 0, offen_tuer : 0,
       wahl1 : 0, wahl2 : 0,
       summe1 : 0, summe2 : 0,
  make_random_state(true),
  for i : 1 thru AnzahlDerVersuche do
   (wert_tuer : random(3)+1,
    wahl1 : random(3)+1,
    offen_tuer : random(3)+1,
    while ( is(offen_tuer = wert_tuer) or
            is(offen_tuer = wahl1) )
      do offen_tuer : random(3)+1,
    wahl2 :
      first(
       listify(
        setdifference({1,2,3},{wahl1, offen_tuer}))),
    if is(wahl1 = wert_tuer) then summe1 : summe1+1,
    if is(wahl2 = wert_tuer) then summe2 : summe2+1,
    if verbose then
      print(i,"  ", wert_tuer, "  ", wahl1, "  ",
            offen_tuer, "  ", wahl2, "  ",
            summe1, "  ", summe2) ),
  return([summe1, summe2])  )



fib(n) :=
  if n<=1 then n else fib(n-1)+fib(n-2)

fib_it(n) :=
  block([i : 0, f0 : 0, f1 : 1, f2 : 1],
    while i < n do
      (i : i+1, f0 : f1, f1 : f2, f2 : f1 + f0),
    f0 )



spot[a_, n_] :=   (* schnelles Potenzieren *)
  Which[
    n==0, 1,
    Mod[n, 2]==0, spot[a^2, n/2],
    True, a*spot[a, n-1] ]



T[n_] :=
  Which[
    n==0, 0,
    Mod[n, 2]==0, T[n/2] + 1,
    True, T[n-1] + 1]



Basis[n_, b_]  :=
  If[n == 0,
       {},
       Append[Basis[Quotient[n, b], b], Mod[n, b] ]  ]
```

```
FOR X = 0.95 TO 1 STEP 0.01
PRINT X
NEXT X
```

```
Program integer_arithmetik_beispiel;
var a, b: integer;
begin
 a := 32767;
 b := a + 1;
 writeln(a);
 writeln(b)
end.
```

```
1:  spot[a_, n_] :=   (* schnelles Potenzieren *)
2:    Which[
3:      n==0, 1,
4:      Mod[n, 2]==0, spot[a^2, n/2],
5:      True, a*spot[a, n-1] ]
```

```
aegyptische_Multiplikation_iterativ(a,b) :=
  block([a1:a, b1:b, c:0],

    while a1 > 1 do
     (if verbose then print(a1, "  ", b1, "  ", c),
      if mod(a1, 2)=0
         then (a1 : a1/2, b1 : b1*2)
         else (a1 : a1-1, c : c+b1) ),
    if verbose then print(a1, "  ", b1, "  ", c),
    return(b1+c) )
```

```
24:    CLC    (CLear Carry flag)
173:   LDA    (LoaD Accumulator)
109:   ADC    (ADd to aCcumulator)
141:   STA    (STore Accumulator)
96:    RTS    (ReTurn from Subroutine)
```

```
CLC       lösche den Überlauf-Speicher
LDA 850   lade den Inhalt von Zelle 850 in den
          Akkumulator
ADC 851   addiere den Inhalt von Zelle 851 zum
          Akkumulator
STA 852   speichere den Akkumulatorinhalt in Zelle 852
RTS       springe an die Stelle zurück, an der sich
          der „Computer“ vor Ausführung dieser
          Additions-Routine befunden hat
```

```
1:  kmw(0, _, []).
2:  kmw(N, [A1 | AT], [A1 | XT]) if
3:    N > 0,
4:    N1 = N-1 and
5:    kmw(N1, [A1 | AT], XT).
6:  kmw(N, [_ | AT], X) if
7:    N > 0,
8:    kmw(N, AT, X).
```

```
Selektion(P, K)  :=
  (* P: Ausgangs-Population,
     K: Fitness-Kriterium    *)
  Erzeuge auf der Basis des Kriteriums K aus P eine
  neue Population P'.
  (Selektionsmethode:  z.B. der direkte Vergleich -
  „Duell").
  Ergebnis:  Die so entstandene neue Population P'.

Crossover(P, CW)  :=
  (* P:  Ausgangs-Population,
     CW: Rekombinations- bzw.
         Crossoverwahrscheinlichkeit *)
  Führe eine Rekombination der Erbanlagen auf der
  Basis der jeweiligen „Rekombinationswahrschein-
  lichkeit" CW durch.  Die Population P der „Eltern"
  wird durch die jeweilige Population P' der Nach-
  kommen ersetzt.
  Ergebnis:  Die so entstandene Generation P' der
             Nachkommen.

Mutation(P, MW)  :=
  (* P:  Ausgangs-Population,
     MW: Mutationswahrscheinlichkeit *)
  Mutiere die Erbanlagen der Individuen aus P auf der
  Basis der gegebenen Mutationswahrscheinlichkeit MW.
  Ergebnis:  Die auf diese Weise mit einem neuen Satz
             von Erbanlagen ausgestattete Population.
```

```
EvolutinaererAlgorithmus(P, K, CW, MW, S);
  (* P:  Ausgangspopulation
     K:  Fitness-Kriterium
     CW: Rekombinationswahrscheinlichkeit(en)
     MW: Mutationswahrscheinlichkeit(en)
     S:  Stop-Kriterium *)
  Wiederhole
    P := Selektion(P, K);
    P := Crossover(P, CW);
    P := Mutation(P, MW);
  bis das Stop-Kriterium S erfüllt ist.
  Ergebnis:  Die neue Population P.
```

```
StaedteListe  =  { {1, 7}, {2, 3}, {2, 12}, {3, 9},
   {5, 1}, {5, 12}, {7, 5}, {8, 2}, {8, 10}, {9, 6},
   {10, 1}, {10, 12}, {11, 9}, {12, 4}, {12, 11} }

InitialPopulation =
 {{8, 6, 12, 7, 13, 4, 1, 10, 5, 14, 15, 2, 11, 9, 3},
  {11, 2, 9, 10, 3, 8, 14, 12, 13, 1, 4, 6, 5, 15, 7},
  {13, 11, 12, 7, 4, 1, 6, 15, 5, 8, 2, 3, 10, 9, 14},
  ...
  {7, 15, 5, 14, 4, 1, 9, 11, 2, 10, 8, 3, 6, 12, 13},
  {10, 13, 6, 9, 4, 2, 7, 3, 8, 5, 1, 12, 14, 11, 15},
  {8, 11, 9, 4, 5, 2, 6, 3, 7, 12, 10, 1, 13, 15, 14},
  {13, 11, 1, 12, 5, 2, 9, 14, 6, 7, 15, 3, 10, 4, 8},
  {11, 10, 3, 6, 12, 7, 5, 9, 15, 1, 13, 14, 2, 4, 8},
  {9, 13, 15, 7, 3, 12, 11, 8, 6, 14, 5, 1, 2, 4, 10},
  {3, 12, 5, 1, 10, 2, 4, 7, 11, 14, 8, 9, 13, 15, 6},
  {11, 15, 2, 5, 12, 14, 7, 8, 10, 3, 13, 6, 1, 9, 4},
  {15, 8, 5, 9, 13, 3, 6, 12, 14, 4, 11, 2, 1, 10, 7},
  {2, 9, 5, 8, 14, 1, 7, 13, 11, 6, 4, 12, 10, 15, 3},
  {10, 3, 8, 12, 2, 6, 11, 14, 7, 9, 1, 15, 5, 4, 13},
  {14, 4, 9, 13, 3, 11, 2, 12, 8, 1, 6, 7, 15, 10, 5},
  {6, 2, 8, 5, 15, 13, 3, 9, 10, 14, 7, 4, 1, 12, 11},
  {15, 11, 14, 13, 7, 1, 6, 8, 3, 4, 10, 2, 5, 9, 12},
  {6, 8, 9, 15, 7, 1, 5, 11, 4, 3, 10, 12, 13, 14, 2},
  {10, 12, 2, 14, 9, 6, 5, 1, 8, 7, 15, 11, 4, 13, 3},
  {13, 4, 8, 12, 10, 5, 3, 15, 7, 14, 2, 1, 6, 11, 9},
  {3, 14, 6, 7, 2, 1, 4, 10, 5, 13, 11, 12, 9, 15, 8},
  {1, 7, 6, 5, 3, 2, 4, 14, 13, 12, 8, 15, 9, 10, 11},
  {13, 6, 12, 11, 8, 9, 4, 7, 1, 2, 14, 3, 5, 10, 15},
  {3, 5, 4, 10, 9, 2, 14, 7, 6, 8, 12, 15, 13, 1, 11},
  {12, 8, 13, 5, 9, 1, 6, 14, 10, 11, 2, 7, 15, 3, 4}}

Distanz[S1_, S2_] :=
  (* Abstand zwischen den Staedten S1 und S2 *)
  Sqrt[(S1[[1]]-S2[[1]])^2 + (S1[[2]]-S2[[2]])^2] //N

Kosten[Rundreise_] :=
  (* Kosten der Rundreise bezueglich der
     (globalen) StaedteListe  *)
  (Sum[Distanz[StaedteListe[[Rundreise[[i]] ]],
               StaedteListe[[Rundreise[[i+1]]] ]],
        {i, 1, Length[Rundreise]-1} ]
    + Distanz[StaedteListe[[Last[Rundreise]]],
              StaedteListe[[First[Rundreise]]]]) //N



OptimaleLoesung[Population_] :=
  Module[{KL, OptimalesElement, PosOpt},
         (* KL:  Kosten-Liste
            PosOpt: Position optimales Elements *)
    KL = Map[Kosten, Population];
    OptimalesElement = Min[KL];
    PosOpt = First[
              First[Position[KL, OptimalesElement]]];
    Return[Population[[PosOpt]]] ]
```

```
Selektion[Pop_] :=
  Module[{NeuPop={OptimaleLoesung[Pop]}},
    (* Diese Initialisierung dient nur dazu, den
       Selektionsdruck zu erhoehen *)
    For[j=1, j<Length[Pop], j=j+1,
      r1 = Random[Integer, {1, Length[Pop]} ];
      r2 = Random[Integer, {1, Length[Pop]} ];
           (* Zwei Zufallszahlen werden ausgewählt *)
      If[Kosten[Pop[[r1]]] <= Kosten[Pop[[r2]]],
           NeuPop = Append[NeuPop, Pop[[r1]]],
           NeuPop = Append[NeuPop, Pop[[r2]]] ] ];
             (* Die beiden zufällig ausgewählten
                Lösungsvorschläge werden verglichen
                und der mit der kürzeren Tour
                wird ausgewählt, d.h. zu NeuPop
                hinzugefügt *)
    Return[NeuPop] ]



Crossover[Pop_, CW_] :=
  Module[{NeuPop={},
          Schnitt1, Schnitt2, Segment1, Segment2,
          kind1, kind2 },
    For[j=1, j<Length[Pop], j=j+2,
      elt1 = Pop[[j]];
      elt2 = Pop[[j+1]];
        (* Das Elternpaar wird aus der Population
           ausgewählt *)
      If[Random[Real] < CW,
        Schnitt1 = Random[Integer, {1, Length[elt1]}];
        Schnitt2 = Random[Integer, {Schnitt1,
                                    Length[elt1]}];
          (* Ein Segment wird zufällig ausgewählt *)
        Segment1 = Take[elt1, {Schnitt1, Schnitt2}];
        Segment2 = Take[elt2, {Schnitt1, Schnitt2}];
          (* Diese Segmente werden bei den Eltern
             ausgeschnitten *)
        kind1={}; kind2={};
          (* Die Kinder werden initialisiert *)
        For[i=1, i<=Length[elt2], i=i+1,
          If[FreeQ[Segment1, elt2[[i]]],
               kind1 = Append[kind1, elt2[[i]]] ];
          If[FreeQ[Segment2, elt1[[i]]],
               kind2 = Append[kind2, elt1[[i]]] ] ];
               (* Alle Werte des elt2-Segments werden
                  bei elt1 geloescht.
                  Alle Werte des elt1-Segments werden
                  bei elt2 geloescht.
                  Die uebrigbleibenden Listen sind
                  kind1 und kind2 .
                  Dies kann leider nicht mit der
                  Funktion „Complement“ gemacht
                  werden, da Complement die
                  Ausgabeliste sortiert  *)
        kind1 = Flatten[
                  Insert[kind1, Segment1, Schnitt1]];
        kind2 = Flatten[
                  Insert[kind2, Segment2, Schnitt1]];
```

```
          (* Die Segmente der Eltern werden bei kind1
             und kind2 an ihren urspruenglichen
             Positionen wieder eingesetzt. *)
        NeuPop = Join[NeuPop, {kind1}, {kind2}],
          (* Die Kinder werden in die neue Population
             eingefuegt *)
        NeuPop = Join[NeuPop, {elt1}, {elt2}] ] ];
          (* Falls nicht gepaart wurde, werden die
             Eltern unveraendert uebernommen *)
    Return[NeuPop] ];



Mutation[Population_, MW_] :=
  Module[{Pop=Population, r1, r2, Individuum,
          Hilfsvariable},
    For[i=1, i<=Length[Pop], i=i+1,
      If[Random[Real] < MW,
           (* Der Zufall hat entschieden, dass das
              Element Nr. i mutiert wird *)
         Individuum = Pop[[i]];
         r1 = Random[
                Integer, {1, Length[Individuum]}];
         r2 = Random[
                Integer, {1, Length[Individuum]}];
            (* Zwei Zufallszahlen werden ausgelost *)
         Hilfsvariable = Individuum[[r1]];
         Individuum[[r1]] = Individuum[[r2]];
         Individuum[[r2]] = Hilfsvariable;
           (* Die Städte an den Positionen r1 und r2
              werden ausgetauscht *)
         Pop[[i]] = Individuum ] ];
    Return[Pop]]



TravelingSalesman[CW_, MW_, MaxGen_] :=
  (* Der Modul verwendet die globalen Variablen
     StaedteListe und InitialPopulation;
     die formalen Parameter haben folgende Bedeutung:
       CW: Crossoverwahrscheinlichkeit
       MW: Mutationswahrscheinlichkeit
       MaxGen: maximale Anzahl an Generationen  *)
  Module[{P = InitialPopulation, BL},
    For[n=1, n<=MaxGen, n=n+1,
        P = Selektion[P];
        P = Crossover[P, CW];
        P = Mutation[P, MW];
        BL = OptimaleLoesung[P];
        Print[n, ":   ", BL, "   ", Kosten[BL]] ];
    Return[P] ]
```

```
GewMat = Sum[Table[ms[[i,j]] * ms[[i,k]],
              {j, Length[ms[[1]]]},
              {k, Length[ms[[1]]]}],
          {i, 1, Length[ms]} ];
          (* ms ist der Mustersatz, ms[[5,3]] greift
             z.B. auf das dritte Element des fünften
             Musters zu *)
For[i=1, i<=Length[ms[[1]]], i=i+1, GewMat[[i,i]]=0];
  (* Die Gewichte w[[i,i]] werden auf Null gesetzt *)



Mustererkennung[Eingabemuster_, Block_] :=
 Module[{M = Eingabemuster, B = Block, Zustandalt},
   Zustandalt = M;
   ListDensityPlot[Reverse[Partition[Zustandalt, B]]];
    (* Das Muster wird in seinem derzeitigen Zustand
       auf dem Bildschirm, geteilt in Blöcke der Länge
       B, ausgegeben.*)
   Zustandneu =
    Table[
     Signum[
      Sum[GewMat[[i,j]] * Zustandalt[[j]],
       {j, 1, Length[M]}]],
     {i, 1, Length[M]} ];
  (* Entsprechend der Aktivierungsfunktion Signum wird
     der neue Netzzustand berechnet.  *)
   While[Not[Zustandalt = Zustandneu],
    Zustandalt = Zustandneu;
    ListDensityPlot[
     Reverse[Partition[Zustandneu, B]] ];
    Zustandneu =
     Table[
      Signum[
       Sum[GewMat[[i,j]] * Zustandalt[[j]],
        {j, 1, Length[M]}]],
      {i, 1, Length[M]} ] ]
     (* Während alter und neuer Netzzustand nicht
        übereinstimmen, wird der neue Netzzustand zum
        alten und der Prozess beginnt wieder von
        vorn *)
   Return[Zustandneu] ];



Signum[x_] := If[x >= 0, 1, -1];



TSPNN[SL_, Durchlaeufe_, Lernrate_] :=
 Module[{Sz, Nz, Nb, Signal, AbstandsL, MinAbst,
         ErrZent},
  Sz = Length[SL];     (* Anzahl der Staedte *)
  Nz = 3*Sz;           (* Anzahl der Neuronen *)
  Nb = Table[{N[2*Sin[i*360/Nz Degree]]+5,
              N[2*Cos[i*360/Nz Degree]]+5}, {i, Nz}];
   (* Startanordnung des Neuronenbandes (Kreis) *)
  For[j=0, j <= Durchlaeufe, j=j+1,
   If[Mod[j, 200] == 0, GraphikModul[SL, Nb] ];
   Signal = Random[Integer, {1, Sz}];
```

```
   AbstandsL =
    Table[N[Sqrt[(SL[[Signal, 1]] - Nb[[i, 1]])^2+
                 (SL[[Signal, 2]] - Nb[[i, 2]])^2]],
     {i, Nz}];
     (* Abstandsliste der Neuronen zum Signal *)
   MinAbst = Min[AbstandsL];
   ErrZent =
    First[First[Position[AbstandsL, MinAbst]]];
    (* Neuron mit dem kleinsten Abstand zum Signal *)
   Nb =
    Nb +
     Table[
      Lernrate
       * N[Exp[-((Min[Abs[ErrZent-i],
                     Abs[ErrZent + Abs[Nz-i]] ])^2) /
            (2 * (50 * (0.02^(j/Durchlaeufe)))^2) ] ]
       * (SL[[Signal]] - Nb[[i]]),
      {i, Nz}]
      (* Berechnung der Positionsveränderung der
         Neuronen *)
  ] ];



GraphikModul[Stl_, Neub_] :=
 Module[{Staedte, Abschluss, Ad1, Ad2},
  Staedte = Table[Point[Stl[[i]]],{i, Length[Stl]}];
  Abschluss = Append[Neub,Neub[[1]]];
  Ad1 = ListPlot[Abschluss, PlotJoined->True];
  Ad2 = Show[Ad1,Graphics[{PointSize[0.01],Staedte}]]]
```