17 Anwendungen: Sortieren, Suchen, Mischen

Gliederung

Vorbemerkungen und Variablentausch

Sortieren, Suchen und Mischen kommen besonders in der kommerziellen Datenverarbeitung recht häufig vor. Der Zeitanteil wird in diesem Bereich auf 30% bis 50% geschätzt.

Sortieren, Suchen und Mischen setzen indizierte Variablen (Arrays, Listen) voraus.

Beim Sortieren müssen sehr häufig (indizierte) Variablen getauscht werden.

In Turbo-Pascal steht für das Tauschen von zwei Variablen keine Standard-Prozedur zur Verfügung. Man muß selbst den sog. "Dreieckstausch" programmieren. Dazu ist eine Hilfsvariable vom gleichen Datentyp notwendig.

Beispiel:

```
.............
x := 'Huber';
y := 'Meier';

Temp := x;     { temporäre Hilfsvariable für Dreieckstausch }
x    := y;
y    := Temp;

WriteLn(x);    { |Meier  }
WriteLn(y);    { |Huber  }
.............
```

Hinweis: Die Turbo-Pascal-Standardfunktion "*Swap*" vertauscht das niederwertige mit dem höherwertigen Byte eines Integer- oder Word-Ausdrucks. Dieses "*Swap*" ist für die hier genannte Aufgabe nicht geeignet.

17.1 Das Sortieren von Daten

Für das Sortieren gibt es viele Verfahren (Bubble-Sort, Sortieren durch Einfügen, Heap-Sort, Shell-Sort, Quick-Sort, Misch-Sort, ...). Sie unterscheiden sich im Programmieraufwand, in der Leistung (Zeitbedarf) und im Speicherbedarf.

Im allgemeinen gilt das »Quick-Sort« als das effektivste Verfahren. Dieses Verfahren wird im Praktikum behandelt.

Es gilt die Regel: Je leistungsfähiger, desto aufwendiger!

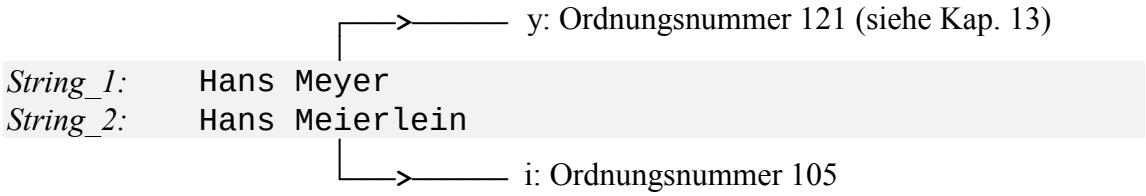
Ausnahmen gibt es viele. Das optimale Verfahren kann man ohne Kenntnis des Umfanges und der Konstellation der Daten nicht sicher angeben.

Es zeigt sich, daß leistungsfähige Verfahren bei kleinen Datenmengen in bezug auf Sortierzeit ungünstiger abschneiden als einfache Verfahren. Bei kommerziellen Anwendungen liegen fast immer große Datenmengen vor. Einfache Verfahren führen in diesen Fällen zu unzumutbaren Zeiten.

Kernpunkt des Sortierens ist der Vergleich von zwei Daten. Bei numerischen Daten ist die Entscheidung unstrittig. Bei Strings muß nach (sinnvollen) Vereinbarungen vorgegangen werden; die Länge eines Strings ist offensichtlich nicht das entscheidende Kriterium. Man denke an die Suche im Telefonbuch!

Beim Vergleich von zwei Strings werden intern die ASCII-Ordnungsnummern der einzelnen Zeichen miteinander verglichen und zwar von links beginnend. Tritt zum erstenmal ein Unterschied auf, dann ist die Ordnungsnummer entscheidend dafür, ob der eine String größer oder kleiner ist als der andere. Die Länge der Strings ist unerheblich. Man beachte, daß auch das Leerzeichen (Space, Blank) ein codiertes Zeichen ist (Ordnungsnummer 32). Zwei Strings sind nur dann gleich, wenn sie an allen Stellen gleiche Zeichen, d.h. gleiche Ordnungsnummern aufweisen. Sie sind dann zwangsweise auch gleich lang.

Das folgende Beispiel zeigt den String-Vergleich:



Somit ist *String1* größer als *String2*.

Allgemeines Sortierproblem bei Sonderzeichen: Die deutschen Sonderzeichen (Umlaute und Scharf-S) besitzen im IBM-Zeichensatz Ordnungsnummern > 127. Sie sind somit nach 'Z' bzw. 'z' angeordnet, siehe Kap. 13. Deshalb ist z.B. 'Ärger' größer als 'Zirkus'. Dieses Problem ist ggf. durch eigene programmtechnische Maßnahmen zu lösen, z.B. wie folgt: Vor dem Vergleich der beiden Strings diese auf temporäre Strings kopieren, darin die Umlaute durch Umschreibungen zu ersetzen sind ('AE' statt 'Ä', 'ue' statt 'ü', 'ss' statt 'ß' usw. Dazu u.a. die Funktion "Pos" nach Kap. 14 einsetzen). Zum Sortier-Vergleich benutzt man die beiden temporären Strings, tauscht dann aber im gegebenen Fall die originalen Strings.

Nachstehend wird ein sehr einfaches Verfahren, das sog. Bubble-Sort-Verfahren, gezeigt. Weitere Verfahren werden im Praktikum behandelt.

17.1.1 Das BubbleSort-Verfahren

Der Grundgedanke: Man durchlaufe wiederholt den Array von Anfang an bis zum vorletzten Element und vergleiche das Element i mit dem Nachfolger-Element $i + 1$. Ist das i-te Element größer als sein Nachfolger, so werden beide getauscht. Man merke sich, wenn ein Tausch stattgefunden hat. Hat bei einem Durchgang kein Tausch mehr stattgefunden, so ist der Array sortiert und der Vorgang kann beendet werden.

Stellt man sich den Array als eine senkrechte Leiter vor, so wandern die kleineren Elemente bei jedem Durchgang um eine Sprosse nach oben, vergleichbar mit aufsteigenden Luftblasen. Daher der Name BubbleSort.

BubbleSort zeigt günstiges Zeitverhalten, wenn die Daten weitgehend vorsortiert sind. Für größere Datenmengen ist BubbleSort im allgemeinen nicht gut geeignet. Das folgende Demo-Programm zeigt BubbleSort.

```
program Pas17011;  { Kap. 17.1: Sortieren, BubbleSort }

const
  iMax = 8;

var
  x:        array[1..iMax] of Integer;
  TempX:    Integer; { Hilfsvariable für Integer-Dreiecks-Tausch }
  StrA:     array[1..iMax] of string[20];
  TempStr:  string[20]; { Hilfsvariable für String-Dreiecks-Tausch  }
  i:        Byte;
  Sortiert: Boolean;

begin
  x[1] := 7; x[2] := 3; x[3] := 6; x[4] := 5;    { Demo-Daten ... }
  x[5] := 3; x[6] := 4; x[7] := 9; x[8] := 0;

  StrA[1] := 'Huber';           StrA[2] := 'huber';
  StrA[3] := 'Huber Anton';     StrA[4] := 'Aumann';
  StrA[5] := 'Ängstlich';       StrA[6] := 'Zeppelin';
  StrA[7] := 'Maus';            StrA[8] := 'Mausilein';

  repeat    { Integer sortieren }
    Sortiert := True;   { vorerst nur kühne Behauptung }
    for i := 1 to iMax - 1 do
      if x[i] > x[i + 1] then
         begin
           TempX    := x[i];           { Drei-  }
           x[i]     := x[i + 1];       { ecks-  }
           x[i + 1] := TempX;          { tausch }
           Sortiert := False;
         end;
  until Sortiert;

  repeat    { Strings sortieren }
    Sortiert := True;
    for i := 1 to iMax - 1 do
      if StrA[i] > StrA[i + 1] then
         begin
           TempStr     := StrA[i];      { Drei-  }
           StrA[i]     := StrA[i + 1];  { ecks-  }
           StrA[i + 1] := TempStr;      { tausch }
           Sortiert    := False;
```

```
        end;
  until Sortiert;

  for i := 1 to iMax do
    WriteLn(i,': ', x[i], i:10, ': ', StrA[i]);

      { Die Bildschirmausgabe:           }
      { |1: 0         1: Aumann          }
      { |2: 3         2: Huber           }
      { |3: 3         3: Huber Anton     }
      { |4: 4         4: Maus            }
      { |5: 5         5: Mausilein       }
      { |6: 6         6: Zeppelin        }
      { |7: 7         7: huber           }
      { |8: 9         8: Ängstlich       }
end.
```

BubbleSort wird auch im späteren Demo-Programm "Pas17012.PAS" als Verfahren 2 gezeigt.

17.1.2 Das MinimumSort-Verfahren

Wird als Verfahren 1 im späteren Demo-Programm "Pas17012.PAS" gezeigt

Das erste Element des Vektors wird zunächst als kleinstes Element angenommen. Dieses wird mit allen folgenden Elementen verglichen. Wenn ein kleineres Element auftaucht, dann werden beide Elemente getauscht.

Nach dem 1. Durchgang steht das kleinste Element an der Spitze des Vektors. Für den weiteren Verlauf braucht man nur den restlichen Vektor betrachten.

Enthält der Vektor *n* Elemente, dann ist das Sortieren nach (*n* - 1) Durchgängen abgeschlossen, wie das folgende numerische Beispiel mit *n* = 6 zeigt (das fett-kursiv gesetzte Element wird zunächst als kleinstes Element für den jeweiligen Durchgang angenommen).

	Original	*i* = 1	*i* = 2	*i* = 3	*i* = 4	*iMax* = *n* - 1 = 5
j = 1	3	~~***3***~~, ~~2~~, ~~1~~, 0	**0**	**0**	**0**	**0**
j = 2	7	7	~~**7**~~, ~~3~~, ~~2~~,**1**	**1**	**1**	**1**
j = 3	2	~~2~~, 3	~~3~~, 7	~~7~~, ~~3~~, **2**	**2**	**2**
j = 4	9	9	9	9	~~**9**~~, ~~7~~, **3**	**3**
j = 5	1	~~1~~, 2	~~2~~, 3	~~3~~, 7	~~7~~, 9	~~**9**~~, **7**
jMax = *n* = 6	0	~~0~~, 1	~~1~~, 2	~~2~~, 3	~~3~~, 7	~~7~~, **9**

Nach der Entwicklung des MinimumSort-Algoritmus mache man sich Gedanken über eine Optimierung (Verfahren 4 im Demo-Programm "Pas17012.PAS"), die mit wenig Zusatzaufwand erreicht wird. Der Grundgedanke der Optimierung besteht darin, zuerst nur den Index des kleinsten Elementes zu suchen; mit dessen Kennntnis braucht bei jedem Durchgang höchstens einmal getauscht werden.

17.1.3 Das QuickSort-Verfahren

Wird als Verfahren 3 im späteren Demo-Programm "Pas17012.PAS" gezeigt.

Quicksort gilt im allgemeinen als das effektivste Verfahren, wenn größere Datenmengen zu sortieren sind. Das Verfahren stammt von C. A. Hoare.

Bei Quick-Sort wird (wie bei allen höheren Sortierverfahren) die Tatsache ausgenutzt, daß das Austauschen von Elementen über größere Distanzen effizienter ist als über kürzere Distanzen. BubbleSort ist bei dieser Betrachtung sehr ungünstig, da immer nur benachbarte Elemente ausgetauscht werden.

Das Sortiervorgang beginnt beim QuickSort nicht mit dem ersten Element, sondern sondern mit einem mittleren. Der Vektor wird dann vom Anfang und vom Ende her durchsucht, bis vor dem mittleren Element ein größeres und nach ihm ein kleineres Element auftaucht. Diese beiden Elemente werden dann getauscht. Der Vorgang wird wiederholt, bis sich die beiden Indizies treffen. Der Vektor ist dann in zwei Hälften aufgeteilt, die linke Hälfte hat kleinere Elemente als das Element in der Mitte, die rechte Häfte besitzt die größeren Elemente.

Mit dem gleichen Verfahren werden dann beide Hälften getrennt bearbeitet. Der Vorgang wird solange wiederholt, bis schließlich die "Hälften" nur noch aus einem Element bestehen. Dann ist der Vektor sortiert.

In Pascal kann Quicksort mit einer rekursiven Unter-Prozedur relativ einfach dargestellt werden.

Es existiert ein nicht-rekursives QuickSort-Verfahren, das mit einem Integer-Hilfsarray arbeitet. Dieses Verfahren ist anzuwenden, wenn die Programmiersprache keine Rekursion erlaubt oder wenn man die Nachteile der Rekursion (Belastung des Stack-Speichers) vermeiden möchte.

Das Demo-Programm "Pas17012.PAS"

```
program Pas17012; { Verschiedene Sortier-Algorithmen. Sortierzeit    }
                  { Hier nur für Zufall-Strings                       }
                  { Turbo-Pascal, 31090693. Dr. K. Haller, FHM, DR    }
{$M 65520, 0, 655350 }   { Compilerbefehl $M (Memory), Stackspeicher }
    { auf Maximalwert 65520 für Rekursion bei "rekursivem QuickSort" }
uses
  CRT, DOS;
const
  ZeitMin      =  4.0;  { Mindest-Sortierzeit in Sekunden, s. später}
  nMax         = 2500;  { Maximale Vektorlänge begrenzen.           }
  StringLaenge =   10;  { Der String-Vektor schluckt viel Speicher! }
                        { Außerdem werden in diesem Demo-Programm    }
                        { immer 2 Vektoren angelegt, damit später    }
                        { die Daten zur Demonstration auch in un-    }
                        { sortierter Form ausgegeben werden können. }
type
  Verfahren    = (Ende, MinimumSort1, BubbleSort,
                  QuickSort, MinimumSort2);
  StringVektor = array[1..nMax] of string[StringLaenge];
var
```

```
  Sortierverfahren: Verfahren;
  s, s_unsortiert:  StringVektor; { "s_unsortiert" steht für   }
                                  { unsortierten Vektor. Für   }
                                  { spätere Ausgabe notwendig. }
  n:                LongInt;      { n = aktuelle Vektorlänge   }
  MitZeitmessung:   Boolean;
  Zeit, ZeitGesamt: Real;
  AnzahlTausch:     LongInt;      { Anzahl der Vertauschungen  }
  Wiederholungen:   Word;
{ ------------------------------------------------------------------ }

procedure WriteXY(Spalte, Zeile: Byte; Meldung: string);
begin
  GotoXY(Spalte, Zeile);
  Write(Meldung);
end;

function Uhrzeit: Real;
var
  hh, mm, ss, ss100: Word;
begin
  GetTime(hh, mm, ss, ss100);
  Uhrzeit := 3600.0*hh + 60*mm + ss + 0.01*ss100;
  { "3600.0" damit Real-Mulitiplikation erzwungen wird,    }
  { sonst Overflow-Error bei strenger Compiler-Einstellung }
end;  { von Funktion "Uhrzeit" }

procedure Menue(var Sortierverfahren: Verfahren; var n: LongInt);
var
  Ch:         Char;
  nStr:       string;
  Fehlercode: Integer;
begin
  ClrScr;
  WriteXY(02,  2, 'Sortierverfahren. Hier Sortierzeit für ' +
                  'Zufalls-Strings mit ');
  WriteLn(StringLaenge, ' Zeichen.    kha ');
  WriteXY(20,  4, 'Sortierverfahren');
  WriteXY(20,  5, '----------------');
  WriteXY(20,  6, '1   MinimumSort1');
  WriteXY(20,  7, '2   BubbleSort  ');
  WriteXY(20,  8, '3   QuickSort   ');
  WriteXY(20,  9, '4   MinimumSort2');
  WriteXY(20, 10, 'Esc Ende, auch 0');
  WriteXY(20, 11, '----------------       Zeitmessung (j/n):   j');
  WriteXY(20, 12, '                       Eingabe n (1..');
  Write(nMax, '): ');
  GotoXY( 20, 12);

  repeat
    Ch := ReadKey;
    if Ch = #27 then Ch := '0';
  until Ch in ['0'..'4'];
  Write(Ch);
```

```
  case Ch of
  '0': begin Sortierverfahren := Ende; Exit; end; { >>>>>>>>>>>> }
  '1': Sortierverfahren := MinimumSort1;
  '2': Sortierverfahren := BubbleSort;
  '3': Sortierverfahren := QuickSort;
  '4': Sortierverfahren := MinimumSort2;
  end;

  repeat
    GotoXY(64, 11);
    Ch := ReadKey;
    if Ch = #13 then Ch := 'j';
  until UpCase(Ch) in ['J', 'N'];
  Write(Ch);
  if UpCase(Ch) = 'J'
     then MitZeitmessung := True
     else MitZeitmessung := False;

  repeat
    GotoXY(64, 12); ClrEoL;
    GotoXY(64, 12);
    ReadLn(nStr);  { Numerik-Eingabe abgesichert !! }
    Val(nStr, n, Fehlercode)
  until (n >= 1) and (n <= nMax) and (Fehlercode = 0);
end; { von Prozedur "Menue" }

procedure MinimumSort1String(n: Word;
                             var s: StringVektor;
                             var AnzahlTausch: LongInt);
var
  i, j:  Word;
  sTemp: string;  { Hilfsvariable für Tauschen }
begin
  AnzahlTausch := 0;
  for i := 1 to n - 1  do
    for  j := i + 1  to  n  do
      if s[j] < s[i]  then
         begin
           sTemp := s[i];    { Drei-  }
           s[i]  := s[j];    { ecks-  }
           s[j]  := sTemp;   { tausch }
           Inc(AnzahlTausch);
         end;
end; { von Prozedur "MinimumSort1String" }

procedure MinimumSort2StringOpt(n: Word;
                                var s: StringVektor;
                                var AnzahlTausch: LongInt);
var               { Freiwillige Zusatzaufgabe für die Studenten }
  i, j,
  iMin:  Word;
  sTemp: string;  { Hilfsvariable für Tauschen }
```

```
begin
  AnzahlTausch := 0;
  for i := 1 to  n - 1  do
    begin
      iMin := i;  { Vorerst }
      for j := i + 1  to  n  do
        if s[j] < s[iMin]
           then iMin := j;
      if iMin > i then
         begin
           sTemp   := s[iMin];   { Drei-  }
           s[iMin] := s[i];      { ecks-  }
           s[i]    := sTemp;     { tausch }
           Inc(AnzahlTausch);
         end;
    end;
end; { von Prozedur "MinimumSort2StringOpt" }

procedure BubbleSortString(n: Word;
                           var s: StringVektor;
                           var AnzahlTausch: LongInt);
var
  i:        Word;
  sTemp:    string;  { Hilfsvariable für Tauschen }
  Sortiert: Boolean;
begin
  AnzahlTausch := 0;
  repeat
    Sortiert := True;  { vorerst nur kühne Behauptung }
    for i := 1  to  n - 1  do
      if s[i] > s[i + 1] then
         begin
           sTemp    := s[i];        { Drei-  }
           s[i]     := s[i + 1];    { ecks-  }
           s[i + 1] := sTemp;       { tausch }
           Sortiert := False;
           Inc(AnzahlTausch);
         end;
  until Sortiert
end; { von Prozedur "BubbleSortString" }

procedure QuickSortString(n: Word;
                          var s: StringVektor;
                          var AnzahlTausch: LongInt);
procedure  QSortString(Links, Rechts: Word; { Unterprozedur }
                       var AnzahlTausch: LongInt);  { lokal }
var
  i, j:    Word;
  sMitte:  string;    { Mittenelement String }
  sTemp:   String;    { Hilfsvariable für Tauschen }
begin
  i := Links;
  j := Rechts;
  sMitte := s[ (Links + Rechts) div 2 ]; { Element etwa in der Mitte }
  repeat
    while s[i] < sMitte  do  Inc(i);
```

```
    while s[j] > sMitte  do  Dec(j);
    if i <= j then
       begin
         sTemp := s[i];  { Drei-  }
         s[i]  := s[j];  { ecks-  }
         s[j]  := sTemp; { tausch }
         Inc(i);
         Dec(j);
         Inc(AnzahlTausch);
       end;
  until i > j;
  if Links  < j
     then QSortString(Links, j     , AnzahlTausch); { rekurs. Aufruf }
  if Rechts > i
     then QSortString(i,     Rechts, AnzahlTausch); { rekurs. Aufruf }
end;  { von Unter-Prozedur "QSortString" }

begin
  AnzahlTausch := 0;
  QSortString(1, n, AnzahlTausch);
end; { von Prozedur "QuickSortString" }

procedure ZufallsStrings(n:            Word;
                         var s:        StringVektor;
                         StringLaenge: Byte);
var
  i:          Word;    { Index für das Zufallswort i }
  j:          Byte;
  nStr:       string;
  Fehlercode: Integer;

function Zufallszahl(Anfangswert, Endwert: Integer): Integer;
begin
  Zufallszahl := Anfangswert + Random(Endwert + 1 - Anfangswert);
end;  { von lokaler Funktion "Zufallszahl" }

begin
  for i := 1 to n do  { 1. Zeichen groß }
    begin             { Rest klein      }
      s[i] :=          Chr(Zufallszahl(Ord('A'), Ord('Z')));
      for j := 2 to StringLaenge do
        s[i] := s[i] + Chr(Zufallszahl(Ord('a'), Ord('z')));
    end;
end; { von Prozedure "ZufallsStrings" }

procedure Ausgabe;  { Benützt außer "i" und "iStr" }
                    { nur globale Variablen!       }
var
  i:    Word;
  iStr: string[4];
begin
  GotoXY(1, WhereY); ClrEoL;
  WriteXY(20, WhereY,     'Nr    unsortiert   sortiert');
  WriteXY(20, WhereY + 1, '----------------------------' + #13#10);

  for i := 1 to n do
    begin
      Str(i, iStr);
```

```
      while Length(iStr) < 4 do
        iStr := '0' + iStr;
      GotoXY(20, WhereY);
      WriteLn(iStr, '  ', s_unsortiert[i], '   ', s[i]);
    end;
  WriteXY(20, WhereY, '----------------------------' + #13#10);
  GotoXY(10, WhereY);
  Write('Sortierverfahren:      ');
  case Sortierverfahren of
    MinimumSort1:  WriteLn('MinimumSort1, einfach  ');
    BubbleSort:    WriteLn('BubbleSort             ');
    QuickSort:     WriteLn('QuickSort              ');
    MinimumSort2:  WriteLn('MinimumSort2, optimiert');
  end;

  GotoXY(10, WhereY);
  WriteLn('Anzahl Vertauschungen: ', AnzahlTausch);
  if MitZeitmessung then
     begin
       GotoXY(10, WhereY);
       WriteLn('Die Sortierzeit: ', Zeit:12:4, ' s');
       GotoXY(10, WhereY);
       WriteLn('Wegen genauerer Zeitmessung wurde der Sortier' +
               'vorgang ', Wiederholungen, '-mal wieder-');
       GotoXY(10, WhereY);
       WriteLn('holt. Die angegebene Zeit gilt aber für  e i n e n ' +
               ' Sortiervorgang.');
     end;
  WriteLn;
  WriteXY(10, WhereY, 'Weiter mit Tastendruck ... ');
  repeat
  until ReadKey <> '';
end; { von Prozedure "Ausgabe" }

begin   { ====== Hauptprogramm ================================== }
  TextBackGround(Blue); TextColor(Yellow); ClrScr;

  repeat
    Menue(Sortierverfahren, n);

    if Sortierverfahren = Ende then Halt;  { >>>>>>>>>>>>>>>> }

    ZufallsStrings(n, s, StringLaenge);

    s_unsortiert := s;  { unsortierter Vektor ... }
                        { ... für spätere Ausgabe }
    WriteXY(10, WhereY + 2, 'Sortierung läuft. Bitte warten ... ');

    Wiederholungen := 0;
    ZeitGesamt     := 0.0;

    repeat
      Inc(Wiederholungen);
      Zeit := Uhrzeit;
      s := s_unsortiert; { Array für Wiederholung regenerieren, }
         { sonst würde BubbleSort bevorzugt. Die Zeit fürs     }
         { "Regenerieren" müßte man abziehen; relatativ klein. }
      case Sortierverfahren of
        MinimumSort1:  MinimumSort1String(   n, s, AnzahlTausch);
        BubbleSort:    BubbleSortString(     n, s, AnzahlTausch);
```

```
        QuickSort:     QuickSortString(      n, s, AnzahlTausch);
        MinimumSort2:  MinimumSort2StringOpt(n, s, AnzahlTausch);
      end;
      Zeit       := Uhrzeit    - Zeit;
      ZeitGesamt := ZeitGesamt + Zeit;
      if not MitZeitmessung then Break;
    until ZeitGesamt >= ZeitMin;  { Damit genauere Zeiten ... }
    Zeit := ZeitGesamt/Wiederholungen;
    Ausgabe;
  until Sortierverfahren = Ende;  { Ausstieg aber weiter oben }
end. { ============================================================ }
```

Die folgenden Excel-Tabellen wurden mit den Ausgaben von "Pas17012.PAS" erstellt.

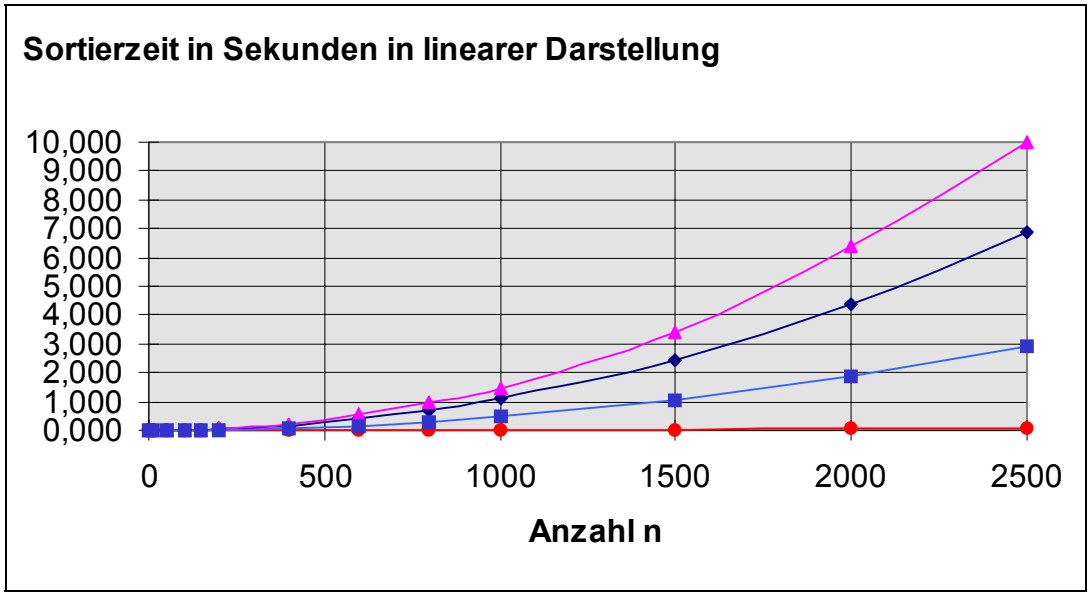
Das erste Diagramm zeigt die Rechenzeiten als $f(n)$ in linearer Darstellung, das zweite in doppelt-logarithmischer Darstellung, das noch deutlicher die Leistungsfähigkeit von QuickSort bei großen Datenmengen zeigt.

Sortierzeiten für Zufallsstrings mit 10 Zeichen. Excel-Tabelle				
Nach Programm "Pas17021.PAS", Rechner Pentium P-166, 16060693, Dr. K. Haller				
Anzahl n	**MinimumSort1** ohne Optimierung	**BubbleSort**	**QuickSort** rekursiv	**MinimumSort2** mit Optimierung
0	0,000	0,000	0,000	0,000
10	0,002	0,002	0,002	0,002
50	0,005	0,006	0,002	0,004
100	0,013	0,017	0,003	0,007
150	0,027	0,034	0,004	0,012
200	0,046	0,065	0,005	0,021
400	0,172	0,239	0,009	0,076
600	0,385	0,549	0,012	0,167
800	0,697	0,998	0,017	0,294
1000	1,085	1,483	0,020	0,464
1500	2,445	3,430	0,031	1,043
2000	4,400	6,370	0,041	1,867
2500	6,860	10,000	0,053	2,940
Vertausch. bei n = 2500	**1.568.494**	**1.548.390**	**7.237**	**2.491**

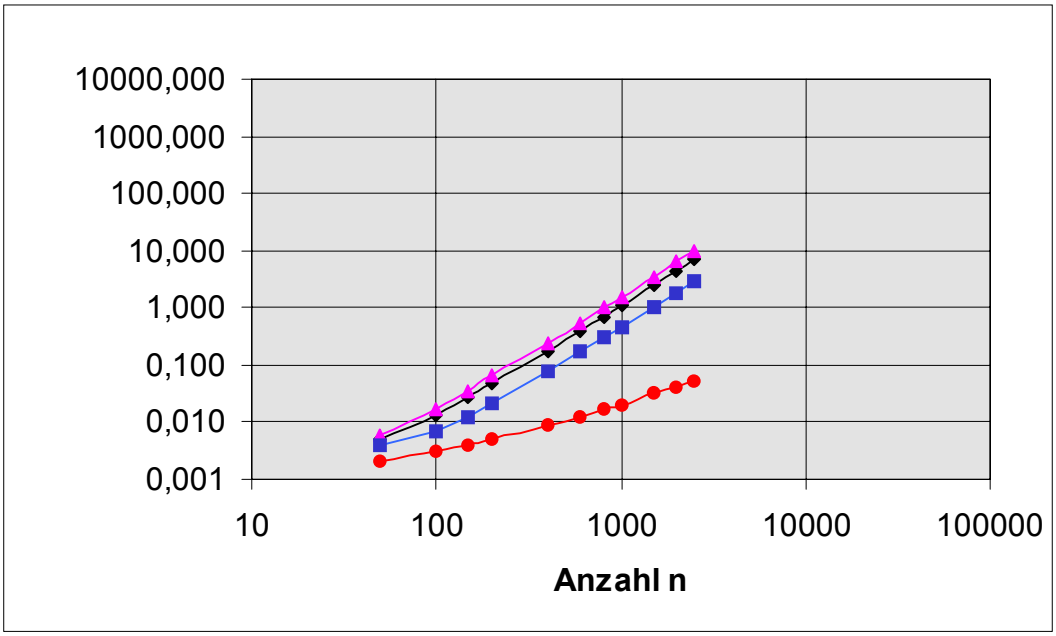
Die Sortierzeiten, von oben nach unten für

- BubbleSort
- MinimumSort1
- MinimumSort2
- QuickSort

a) in linearer Darstellung:



b) in doppelt-logarithmischer Darstellung:



17.2 Das Suchen in Listen

Beim Suchen geht es darum, die Stelle (= Index) eines Elementes in einer Liste (eindimensionaler Array) zu ermitteln oder nachzuweisen, daß das Element in der Liste enthalten bzw. nicht enthalten ist.

Wenn die Liste nicht sortiert ist, muß sequentiell jedes Element mit dem gesuchten Element verglichen werden. Hat die Liste den Umfang ***n***, so sind im ungünstigsten Fall ***n***-Vergleiche (Zugriffe) notwendig, im statistischem Mittel ***n***/2. Das sequentielle Suchen ist somit sehr zeitaufwendig.

Ein ungleich schnelleres Suchen wird erzielt, wenn die Liste sortiert vorliegt und die Methode des **binären Suchens** eingesetzt wird. Die Anzahl z_{Max} der maximalen Zugriffe reduziert sich dann auf $z_{Max} = ld(n + 1)$. »ld« bedeutet logarithmus dualis, Logarithmus zur Basis 2. Dezimalzahlen sind auf den nächsten ganzzahligen Wert aufzurunden.

Die folgende Tabelle zeigt den Zusammenhang $z_{Max} = f(n)$:

Umfang der Liste ***n***	Anzahl der maximalen Zugriffe z_{Max} beim binären Suchen
2 .. 3	2
4 .. 5	3
8 .. 15	4
16 .. 31	5
32 .. 63	6
64 .. 127	7
128 .. 255	8
256 .. 511	9
512 .. 1023	10
1024 .. 2047	11
2048 .. 4095	12
4096 .. 8191	13
8192 .. 16383	14
16384 .. 32767	15
32768 .. 65535	16

Mit $z_{Max} = 27$ Zugriffen könnte ein Element in einer sortierten Liste mit dem Umfang $n = 2^{27} - 1$ = ca. 134 Millionen gesucht werden. Diese Zahl ist weit größer als die Einwohnerzahl der Bundesrepublik (ca. 85 Mio). Zu bedenken ist aber, daß statische Arrays in Pascal maximal 65536 Elemente enthalten können und auch nicht mehr Speicherplatz in Bytes belegen dürfen.

Das binäre Suchen kann vereinfacht wie folgt beschrieben werden:

Man betrachte ein Element, das etwa in der Mitte der Liste steht. Wenn keine Übereinstimmung mit dem Such-Element vorliegt sind zwei Fälle zu unterscheiden: Ist das Such-Element kleiner als das Mitten-Element, so ist in der linken Teilhälfte weiterzusuchen, anderenfalls in der rechten. Von dem zutreffendem Teilfeld betrachte man wieder das Mitten-Element, wenn keine Übereinstimmung mit dem Such-Element vorliegt, ist in dem zutreffenden Teil des Teilfeldes weiterzusuchen usw., bis entweder

das Such-Element gefunden wird oder die Ober- und Untergrenze des Teilfeldes zusammenfallen. In diesem Fall ist das gesuchte Element nicht in der Liste enthalten.

Das folgende Demo-Programm zeigt das binäre Suchen am Beispiel eines String-Arrays:

```
program Pas17021; { Kap. 17.2: Binäres Suchen }

uses
  CRT;

const
  iMax = 10;   { Bei einer Liste mit 8..15 Elementen sind beim }
               { binären Suchen maximal 4 Zugriffe notwendig   }
type
  String25    = string[25];
  StringArray = array[1..iMax] of String25;

var
  Unten,
  Oben,
  Mitte,
  GesuchterIndex,
  Zugriffe:       Word;
  Gefunden:       Boolean;
  StrA:           StringArray;
  Suchelement:    String25;

procedure Daten_einlesen_und_sortieren(var StrA: StringArray);
var
  i:        Word;
  TempStr:  String25;  { Hilfsvariable für Dreiecks-Tausch }
  Sortiert: Boolean;

begin
  StrA[1]  := 'Huber';         StrA[2]  := 'Unentdeckter';
  StrA[3]  := 'Meier';         StrA[4]  := 'Geheimnisträger';
  StrA[5]  := 'Yuppie';        StrA[6]  := 'Stilles Wasser';
  StrA[7]  := 'Bonze';         StrA[8]  := 'Verdächtiger';
  StrA[9]  := 'Unauffindbar';  StrA[10] := 'Strafpunktesammler';
  repeat  { Bubble-Sort }
    Sortiert := True;
    for i := 1 to iMax - 1 do
      if StrA[i] > StrA[i + 1] then
         begin
           TempStr     := StrA[i];     { Drei-  }
           StrA[i]     := StrA[i + 1]; { ecks-  }
           StrA[i + 1] := TempStr;     { tausch }
           Sortiert    := False;
         end;
  until Sortiert;

  for i := 1 to iMax do
    WriteLn(i, ': ', StrA[i]);  { nur für Demo }
end;  { von »Daten_einlesen_und_sortieren« }

begin { Hauptprogramm binäres Suchen }
```

```
  ClrScr;
  Daten_einlesen_und_sortieren(StrA); { Binäres Suchen setzt
                                        sortierte Liste voraus }
  repeat
    WriteLn;
    Write('Eingabe Suchelement, Ende mit RETURN:  ');

    ReadLn(Suchelement);

    if Suchelement = ''  { »''«, wenn nur »Return« }
       then  EXIT;       { >>>>>>>>>>>>>>>>>>>>>>> }

    Unten    := 1;
    Oben     := iMax;
    Zugriffe := 0;
    Gefunden := False;

    repeat
      Inc(Zugriffe);          { nur aus Neugierde Zugriffe zählen }
      Mitte := (Unten + Oben) div 2;  { Element etwa in der Mitte }
      if Suchelement = StrA[Mitte]
         then begin
                Gefunden       := True;
                GesuchterIndex := Mitte;
              end
         else if Suchelement > StrA[Mitte]
                 then Unten := Mitte + 1     { »Oben«  bleibt }
                 else Oben  := Mitte - 1;    { »Unten« bleibt }
    until Gefunden or (Unten > Oben);   { Vergleich unbedingt }
                      { mit »>« und nicht »>=«, sonst Fehler! }

    if Gefunden
       then Write('Das Suchelement »', Suchelement,
                  '« hat den Index: ', GesuchterIndex)
       else Write('Das Suchelement ist in der Liste ',
                  'nicht enthalten');
    WriteLn('. Anzahl der Zugriffe: ', Zugriffe);
  until Suchelement = '';
end.
```

17.3 Das Mischen von sortierten Daten

Die Aufgabenstellung: Es liegen zwei sortierte Listen *I* und *J* vor. Aus diesen Listen soll eine gemeinsame Liste *K* erstellt werden. Die Elemente von *I* und *J* sollen aber so in die Liste *K* eingetragen werden, daß sie ohne Sortiervorgang sortiert ist.

Beispiel mit numerischen Daten:

Aus	Liste *I*:	2, 4, 5, 6, 8, 13, 15
und	Liste *J*:	1, 2, 3, 7
soll werden:	Liste *K*:	1, 2, 2, 3, 4, 5, 6, 7, 8, 13, 15

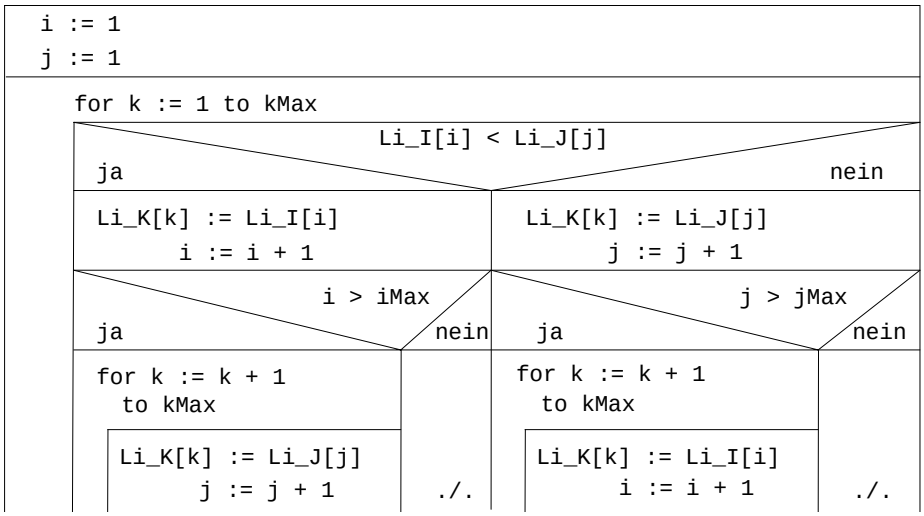
Das Mischen wird dann eingesetzt, wenn man große Dateien zu sortieren hat, die im Arbeitsspeicher des Rechners nicht Platz haben. Man teilt die große Datei in mehrere kleinere externe Dateien auf, sortiert diese nacheinander im Arbeitsspeicher und schreibt die sortierten Teillisten wieder in eine externe Datei (Magnetplatte, Magnetband, Diskette usw.). Dann mischt man zwei Teillisten und schreibt das Ergebnis wieder in eine externe Datei. Durch Wiederholung kann man somit Dateien sortieren, deren Größe nur durch die Kapazität des externen Speichers begrenzt ist.

Bei der Formulierung des Misch-Algorithmus ist zu bedenken, daß die zu mischenden Listen unterschiedlich lang sein können. Unter dieser Berücksichtigung kann man das Mischen wie folgt formulieren:

1. Wiederhole die Vorgänge 2 und 3 so lange, bis die *K*-Liste mit allen Elementen der beiden Listen *I* und *J* gefüllt ist:
2. Vergleiche die beiden ersten Elemente (bzw. die beiden nächsten Elementen bei weiteren Durchläufen) der beiden Listen *I* und *J*.
3. Ist das *I*-Element kleiner als das *J*-Element?
   Wenn ja, dann:
   3a1: Schreibe das *I*-Element in die Liste *K*.
   3a2: Frage ab, ob die *I*-Liste bereits beendet ist.
   Wenn ja, dann: Schreibe alle noch nicht übertragenen *J*-Elemente in die Liste *K*.
   anderenfalls:
   3b1: Schreibe das *J*-Element in die Liste *K*.
   3b2: Frage ab, ob die *J*-Liste bereits beendet ist.
   Wenn ja, dann: Schreibe alle noch nicht übertragenen *I*-Elemente in die Liste *K*.

Das folgende Struktogramm zeigt den Misch-Algorithmus in schematischer Pascal-Schreibweise. Dabei werden folgende Variablennamen benutzt:

`- Li_I[1..iMax]`	sortierte Liste *I*
`- Li_J[1..jMax]`	sortierte Liste *J*
`- Li_K[1..kmax]`	gemischte Liste *K*, sortiert. *kMax = iMax + jMax*
`- i, j, k`	Laufvariablen für Listen *I*, *J* und *K*



Das folgende Demo-Programm zeigt das Mischen von zwei sortierten String-Listen:

```
program Pas17031;  { Kap. 17.3: Mischen von sortierten Daten }

uses
  CRT;

const
  iMax = 5;
  jMax = 8;
  kMax = iMax + jMax;

type
  String15     = string[15];
  StringArrayI = array[1..iMax] of String15;
  StringArrayJ = array[1..jMax] of String15;
  StringArrayK = array[1..kMax] of String15;

var
  ListeI:    StringArrayI;
  ListeJ:    StringArrayJ;
  ListeK:    StringArrayK;
  i, j , k:  Word;

procedure Daten_einlesen_und_sortieren(var ListeI: StringArrayI;
                                       var ListeJ: StringArrayJ);
var
  i:        Word;
  TempStr:  String15;  { Hilfsvariable für Dreiecks-Tausch }
  Sortiert: Boolean;

begin
  ListeI[1] := 'Huber';                  ListeI[2] := 'Zeppelin';
  ListeI[3] := 'Haspert';                ListeI[4] := 'Meyer';
  ListeI[5] := 'Yuppie';
  repeat  { Bubble-Sort, Liste I }
    Sortiert := True;
    for i := 1 to iMax - 1 do
      if ListeI[i] > ListeI[i + 1] then
         begin
```

```
          TempStr       := ListeI[i];     { Drei-  }
          ListeI[i]     := ListeI[i + 1]; { ecks-  }
          ListeI[i + 1] := TempStr;       { tausch }
          Sortiert      := False;
        end;
  until Sortiert;
  GotoXY(5, 1); WriteLn('Liste I ');
  GotoXY(5, 2); WriteLn('---------------');
  for i := 1 to iMax do
    begin
      GotoXY(5, 2 + i);
      Write(i, ': ', ListeI[i]);  { nur für Demo }
    end;

  ListeJ[1] := 'Lippert';                ListeJ[2] := 'Aumann';
  ListeJ[3] := 'Kugler';                 ListeJ[4] := 'Vulpert';
  ListeJ[5] := 'Huber';   { !! s.o. !! } ListeJ[6] := 'Greiner';
  ListeJ[7] := 'Meier';                  ListeJ[8] := 'Berthold';
  repeat  { Bubble-Sort, Liste J }
    Sortiert := True;
    for i := 1 to jMax - 1 do
      if ListeJ[i] > ListeJ[i + 1] then
         begin
          TempStr       := ListeJ[i];     { Drei-  }
          ListeJ[i]     := ListeJ[i + 1]; { ecks-  }
          ListeJ[i + 1] := TempStr;       { tausch }
          Sortiert      := False;
        end;
  until Sortiert;
  GotoXY(25, 1); WriteLn('Liste J ');
  GotoXY(25, 2); WriteLn('---------------');
  for i := 1 to jMax do
    begin
      GotoXY(25, 2 + i);
      Write(i, ': ', ListeJ[i]);  { nur für Demo }
    end;
end;   { von »Daten_einlesen_und_sortieren« }

begin  { Hauptprogramm }
  ClrScr;
  Daten_einlesen_und_sortieren(ListeI, ListeJ);

  { Beginn Mischen }
  i := 1;  { Start-Index für Liste I }
  j := 1;  { Start-Index für Liste J }
  for k := 1 to kMax do
    if ListeI[i] < ListeJ[j]
       then begin
              ListeK[k] := ListeI[i];
              Inc(i);
              if i > iMax then
                 for k := k + 1 to kMax do
                   begin
                     ListeK[k] := ListeJ[j];
                     Inc(j);
                   end;
            end
```

```
      else begin
             ListeK[k] := ListeJ[j];
             Inc(j);
             if j > jMax then
                for k := k + 1 to kMax do
                  begin
                    ListeK[k] := ListeI[i];
                    Inc(i);
                  end;
           end;
  { Ende Mischen }

  GotoXY(45, 1); WriteLn('Gemischte Liste K');
  GotoXY(45, 2); WriteLn('-----------------');
  for k := 1 to kMax do
    begin
      GotoXY(45, 2 + k);
      Write(k:2, ': ', ListeK[k]);  { nur für Demo }
    end;

  repeat
  until KeyPressed;

  { Die Bildschirmausgabe: }
  { ¦     Liste I             Liste J             Gemischte Liste K }
  { ¦     ---------------     ---------------     ----------------- }
  { ¦     1: Haspert          1: Aumann           1: Aumann         }
  { ¦     2: Huber            2: Berthold         2: Berthold       }
  { ¦     3: Meyer            3: Greiner          3: Greiner        }
  { ¦     4: Yuppie           4: Huber            4: Haspert        }
  { ¦     5: Zeppelin         5: Kugler           5: Huber          }
  { ¦                         6: Lippert          6: Huber          }
  { ¦                         7: Meier            7: Kugler         }
  { ¦                         8: Vulpert          8: Lippert        }
  { ¦                                             9: Meier          }
  { ¦                                            10: Meyer          }
  { ¦                                            11: Vulpert        }
  { ¦                                            12: Yuppie         }
  { ¦                                            13: Zeppelin       }
end.
```