SHELL.WEB
Ein Beispiel ein Beispiel für Literate Programming in Deutscher Sprache

Felix Gärtner

Version 0.5

Juli 1993 (zuletzt überarbeitet September 1998)

	Section	Page
Vorbemerkungen zu WEB	1	1
Einführung	7	2
Ein Überblick über das Programm	11	3
Das Beiwerk	16	4
Das Prinzip von Shellsort	26	6
Die Programmierung von Shellsort	32	7
Bezeichnerindex	42	9

1. Vorbemerkungen zu WEB. Dieser Text ist ein Computerprogramm. Auf den ersten Blick mag dies nicht so erscheinen, doch das liegt daran, daß dieses Programm in einer Form geschrieben wurde, wie sie noch unüblich ist. Das Leitbild (oder *Paradigma*), dem diese Form entspricht, ist das des *literate programming* (man könnte es vielleicht mit "literarischem Programmieren" übersetzen). Literate Programming ist ein neuer Versuch, die Lesbarkeit und Verständlichkeit von Computerprogrammen zu verbessern.

Das nun folgende Programm ist also ein Beispiel für ein *literate program*. Es ist die Realisierung eines Sortierverfahrens, welches nach seinem Erfinder D. E. Shell *Shellsort* genannt wurde.

2. Die Idee hinter Literate Programming ist, daß man beim Programmieren in Gedanken einem Menschen den Sinn, Zweck und die Wirkungsweise seines Programms zu erklären versucht anstatt in erste Linie einen Computer zu einem bestimmten Verhalten zu bewegen. Dies wird zuvorderst erreicht durch eine Umkehrung der Prioritäten beim Schreiben von Programmcode: Statt Programmcode mit eingestreutem Kommentar zu schreiben, schreibt man Kommentar mit eingestreutem Programmcode!

Zusätzlich enthält die Idee des Literate Programming auch, daß die Einzelteile eines Programms in einer solchen Reihenfolge vorgestellt werden, die nur von didaktischen Kriterien beeinflußt ist und nicht von den Notwendigkeiten eines Compilers. Ziel ist es, einen Programmierstil zu ermutigen, der sein Hauptaugenmerk auf eine klare Struktur und die Verständlichkeit eines Programmes legt und nicht hauptsächlich darauf, ein Programm nur zum Laufen zu bringen. Dies geht am besten, wenn man Programme als *works of literature*, also als Literatur im weitesten Sinne auffaßt. Darum auch der Name: *Literate* Programming.

3. Moderne Programmiersprachen erlauben die Umsetzung dieser Ideen nur sehr eingeschränkt. Darum werden Hilfsmittel (Tools) benötigt, die echtes Literate Programming erlauben. Ein solches Tool ist unter dem Namen WEB bekannt und stammt von dem norwegischen Stanford-Professor Donald Knuth, der als erster dieses Forschungsgebiet beschrieben und benannt hat. WEB verbindet die Programmiersprache PASCAL und die Formatierungssprache TEX. TEX wird für die Dokumentation benutzt und PASCAL zur Beschreibung des Programmcodes.

Manchmal wird ein Literate Program auch mit dem Terminus WEB bezeichnet. Dies rührt her von der Tatsache, daß man sich ein solches Literate Program als ein Netz (engl. *web*)von einzelnen Modulen vorstellen kann.

4. Ein WEB ist im wesentlichen ein PASCAL-Programm, welches in kleine Teile zerlegt worden ist. Diese Einzelteile sind dann so angeordnet, daß es für einen Menschen einfacher ist, das gesamte Programm zu verstehen. Eigentlich sind PASCAL-Programme und WEB-Programme das gleiche. Nur wird es für einen Leser einfacher sein das PASCAL-Programm zu verstehen, wenn er es in der WEB-Form liest (und wenn der Programmierer eine gute Reihenfolge für die Vorstellung der Einzelteile gewählt hat).

Ein WEB-Programm besteht also aus einer Folge von aufsteigend numerierten Abschnitten (beginnend mit Abschnitt 1). Jeder Abschnitt sollte so klein sein, daß er für sich selbst verständlich ist. Die Abschnitte bestehen im wesentlichen aus zwei Teilen: Zuerst kommt eine informelle Beschreibung, was in dem Abschnitt beabsichtigt ist; anschließend folgen ein paar Zeilen Programmcode. Jeder der beiden Teile kann auch leer sein.

5. Der Programmcode in einem Abschnitt kann einen Namen tragen. Dieser Name erscheint in Klammern vor dem Code wie beispielsweise bei ⟨Initialisierungen 14⟩. Die Nummer 14 steht für denjenigen Programmabschnitt, in dem der zugehörige Code definiert wurde.

Der Code von so benannten Abschnitten kann in anderen Programmteilen wieder benutzt werden und taucht dann einfach wieder mit dem in Klammern eingeschlossenen Namen auf.

Der PASCAL-Text wird durch WEB in einer einheitlichen und etwas übersichtlicheren Weise formatiert. Reservierte Worte erscheinen in Fettdruck, Bezeichner in kursiver Schrift. Zur Verbesserung der Lesbarkeit werden Worte wie `and` ($\land$) und `or` ($\lor$) oder die Zuweisung `:=` ($\leftarrow$) durch spezielle Zeichen ersetzt.

6. Den Abschluß eines WEB-Programmes bildet ein automatisch generierter Index, in dem alle Bezeichner und Schlagwörter enthalten sind.

Doch genug der Vorrede. Lasset uns beginnen mit Shellsort in WEB-Form.

7. Einführung. Das nun folgende Programm ist also ein Beispiel für ein *literate program.* Es ist die Realisierung eines Sortierverfahrens, welches nach seinem Erfinder D. E. Shell *Shellsort* genannt wurde. Die Quelle hierfür findet sich in den *Communications of the ACM* (2), Nr. 7 (1959) auf den Seiten 30 bis 32. Meine Beschreibung folgt einer Realisierung von Niklaus Wirth in *Algorithmen und Datenstrukturen in Modula-2*, Stuttgart 1986.

Das Sortierverfahren wird andernorts auch bezeichnet mit "Sortieren durch Einfügen mit abnehmender Schrittweite". Es beruht auf der Methode des "Sortierens durch Einfügen". Hierbei wird wie bei jedem Sortierprozeß ausgegangen von einer Folge von n Schlüssels $a_1, a_2, \ldots, a_n$, die gemäß einer Ordnungsrelation $\leq$ in die richtige Reihenfolge gebracht werden sollen.

8. Sortieren durch Einfügen teilt die Folge von Schlüsseln mit Hilfe eines Indexes i in zwei Teile ein. Der erste Teil $a_1, \ldots, a_{i-1}$ ist schon sortiert, der zweite Teil $a_i, \ldots, a_n$ noch unsortiert. Nun werden nacheinander alle Elemente der zweiten Teilfolge an die richtige Stelle in die erste eingefügt.

Sei also die Folge von Schlüsseln gegeben durch ein **array** $[1 \mathinner{..} n]$ **of** *integer* mit Namen A. Der Algorithmus lautet dann also etwa:

for $i \leftarrow 2$ **to** n **do**
  begin $x \leftarrow A[i]$;
  $\langle$ Fuege x am geeigneten Ort in $A[1], \ldots, A[i-1]$ ein 74 $\rangle$
  end;

9. Zur Bestimmung des entsprechenden Ortes ist es angebracht, zwischen Vergleichen und Bewegungen abzuwechseln, d. h. x nach links wandern zu lassen, mit dem nächsten Element a_j zu vergleichen und entweder x einzufügen oder a_j nach rechts zu schieben und nach links fortzufahren. Der Prozess ist beendet, wenn ein Element a_j mit kleinerem Schlüssel gefunden wird, oder wenn das linke Ende der geordneten Teilfolge erreicht wird.

$\langle$ Fuege x am geeigneten Ort in $A[1], \ldots, A[i-1]$ ein 74 $\rangle \equiv$
  $j \leftarrow i$; { Index für das nach links Wandern }
  while $(x < A[j-1]) \wedge (j > 1)$ **do**
    begin $A[j] \leftarrow A[j-1]$; $j \leftarrow j-1$;
    end;
  $A[j] \leftarrow x$; { Wert x einsetzen }

10. Um bei dieser Form des Algorithmus keine Bereichsüberschreitungen innerhalb des Feldes A zu bekommen, muß natürlich A mit den Grenzen $[0 \mathinner{..} n]$ definiert werden.

Mit der Idee dieses Algorithmus können wir jetzt daran gehen, Shellsort zu Untersuchen.

11. Ein Überblick über das Programm. Hauptteil dieses Programmes ist das Sortierverfahren Shellsort. Das Sortierverfahren soll als eine Prozedur *shellsort* realisiert werden. Natürlich ist es wenig nützlich, eine solche Prozedur alleine zu schreiben: Ein gewisses Beiwerk sollte schon vorhanden sein, um die Sortier-Prozedur auch austesten zu können.

Dieses Beiwerk will ich hier bereitstellen. Es besteht hauptsächlich aus der Bereitstellung von Testdaten, auf die der Sortierer angewendet werden kann, und aus einer Ausgabefunktion für diese Daten auf dem Bildschirm.

12. Wie jedes PASCAL-Programm hat auch unser Programm hier die übliche Form: Zuerst kommen Definitionen, Variablen, usw. Anschließend folgen Prozeduren und Funktionen, und abschließend das Hauptprogramm.

program *shell*(*output*); { Es wird nur ausgegeben }
  const ⟨ Konstanten des Programmes 13 ⟩
  var ⟨ Variablen des Programmes 14 ⟩
    ⟨ Hilfsprozeduren 17 ⟩
    ⟨ Die eigentliche Shellsort-Prozedur 33 ⟩
    begin ⟨ Das Hauptprogramm 15 ⟩
    end.

13. Die Prozedur *shellsort* soll auf einer globalen Reihung A von Schlüsseln arbeiten, diese also sortieren. Die Schlüssel dieser Reihung sollen der Einfachheit halber nur aus *integer*-Zahlen bestehen. Die Größe dieser Reihung wird durch eine globale Konstante n festgelegt.

⟨ Konstanten des Programmes 13 ⟩ ≡
  $n = 100$; { maximale Anzahl von Schlüsseln }

See also section 20.

This code is used in section 12.

14. ⟨ Variablen des Programmes 14 ⟩ ≡
A: **array** $[0 \mathrel{..} n]$ **of** *integer*; { die zu sortierende Reihung }

This code is used in section 12.

15. Das Hauptprogramm schreibt sich mit den obigen Überlegungen fast von selbst.

⟨ Das Hauptprogramm 15 ⟩ ≡
  ⟨ Fuelle die Reihung A mit Testdaten und zeige sie an 0 ⟩
  ⟨ Sortiere die Reihung mittels Shellsort 0 ⟩
  ⟨ Gebe die sortierten Daten auf dem Bildschirm aus 0 ⟩

This code is used in section 12.

16. Das Beiwerk. Da das Beiwerk, sprich die Beschaffung der einzelnen Zahlen und deren Ausgabe auf dem Bildschirm, nicht so besonders interessant sind, wollen wir sie lieber früher als später gleich abhandeln.

17. Das Füllen der Reihung A mit Testdaten ist eine einfache Angelegenheit. Sie wird durch die Prozedur *initialize* geleistet. Testdaten sollen un unserem Falle positive, ganze Zufallszahlen sein.

⟨ Hilfsprozeduren 17 ⟩ ≡
procedure *initialize*; { Initialisiert A mit Zufallszahlen }
  var i: *integer*; { Hilfszähler }
  begin ⟨ Vorarbeiten zur Generierung von Zufallszahlen 19 ⟩
  $A[0] \leftarrow 0$; { Uninteressante Eintragung in A }
  for $i \leftarrow 1$ **to** n **do**
    begin ⟨ Setze $A[i]$ auf einen Zufallswert 18 ⟩
    end;
  end; { von *initialize* }

See also section 21.

This code is used in section 12.

18. Oben habe ich gesagt, daß dieses Programm vollständig in Standard-PASCAL geschrieben worden ist. Nun, das war nur fast richtig. Das eine Prozent Turbo-PASCAL in diesem Programm steckt hier in diesem Modul. In Standard-PASCAL gibt es leider keine einfache Möglichkeit, Zufallszahlen zu erzeugen. Drum greife ich hier auf die Turbo-PASCAL-Funktionen *randomize* und *random* zurück.

Die Prozedur *randomize* initialisiert den Zufallszahlen-Generator des Compilers neu. $zahl = random(a)$ liefert eine ganze Zahl aus dem Bereich $0 \leq zahl < a$. Der Zahlenbereich, aus dem die Zufallszahlen stammen, soll zwischen 0 und n liegen.

⟨ Setze $A[i]$ auf einen Zufallswert 18 ⟩ ≡
  $A[i] \leftarrow random(n)$;

This code is used in section 17.

19. ⟨ Vorarbeiten zur Generierung von Zufallszahlen 19 ⟩ ≡
  randomize;

This code is used in section 17.

20. Damit bliebe als Rest des Beiwerkes nur noch die Ausgabefunktion zu schreiben. Sie soll realisiert werden als eine Prozedur *print_table* und soll den Inhalt der Reihung seitenweise in rr Zeilen und cc Spalten pro Seite ausgeben. Jede Spalte ist ww Zeichen breit. In unserem Falle soll $rr = 24$, $cc = 8$ und $ww = 10$ sein.

⟨ Konstanten des Programmes 13 ⟩ +≡
  $rr = 24$; { Anzahl der Zeilen pro Seite }
  $cc = 8$; { Anzahl der Spalten pro Seite }
  $ww = 10$; { Anzahl der Zeichen pro Spalte }

21. ⟨ Hilfsprozeduren 17 ⟩ +≡
procedure *print_table*; { Zeigt die Testdaten auf dem Bildschirm an }
  var ⟨ Hilfsvariablen für die Ausgabe 22 ⟩
    begin ⟨ Gebe die Testdaten auf den Bildschrim aus 23 ⟩
    end; { von *print_table* }

22. Wir brauchen noch ein paar Hilfsvariablen zur Steuerung der Ausgabe. Die Variable *page_number* wird immer die Nummer der aktuell angezeigten Seite enthalten. *page_offset* ist ein Zeiger in die Reihung A derart, daß $A[page\_offset]$ gerade immer die erste Zahl ist, die auf dieser Seite angezeigt wird. Ähnlich dazu enthält $A[row\_offset]$ gerade diejenige Zahl, die am Anfang einer gegebenen Zeile steht.

⟨ Hilfsvariablen für die Ausgabe 22 ⟩ ≡
page_number: *integer*; { aktuelle Seitennummer }
page_offset: *integer*; { Index für den ersten Eintrag auf der aktuellen Seite }
row_offset: *integer*; { Index für den ersten Eintrag der aktuellen Spalte }
c: *integer*; { Hilfsvariable, durchläuft die Anzahl der Spalten }

This code is used in section 21.

23. Mit den eben vorgestellten Hilfsvariablen kann man nun die Ausgabeprozedur vervollständigen.

⟨ Gebe die Testdaten auf den Bildschrim aus 23 ⟩ ≡
```
page_number ← 1; page_offset ← 1;
while page_offset ≤ n do
  begin ⟨ Gebe eine Seite Zahlen aus 24 ⟩
  page_number ← page_number + 1; page_offset ← page_offset + rr * cc;
  end;
```
This code is used in section 21.

24. Am Anfang jeder Seite wird eine kleine Kopfzeile ausgegeben.

⟨ Gebe eine Seite Zahlen aus 24 ⟩ ≡
```
write('Die␣', n : 1, '␣Testzahlen␣zu␣Shellsort␣'); write('(Seite␣', page_number : 1, ')'); writeln;
writeln; { eine Leerzeile hinter dem Titel }
for row_offset ← page_offset to page_offset + rr − 1 do
  begin ⟨ Gebe eine Zeile Zahlen aus 25 ⟩
  end;
```
This code is used in section 23.

25. Die erste Zeile auf der ersten Seite enthält die Zahlen

$$A[1], A[1 + rr], A[1 + 2 * rr], \ldots$$

Ein ähnliches Muster trifft dann auch für die anderen Zeilen der Ausgabe zu.

⟨ Gebe eine Zeile Zahlen aus 25 ⟩ ≡
```
for c ← 0 to cc − 1 do
  begin if row_offset + c * rr ≤ m then write(A[row_offset + c * rr] : ww);
  end;
writeln;
```
This code is used in section 24.

26. Das Prinzip von Shellsort. Nachdem nun das Beiwerk abgehandelt ist können wir uns endlich der eigentlichen Shellsort-Prozedur zuwenden, um die es ja hauptsächlich hier gehen soll. Zunächst will ich also das Prinzip von Shellsort erklären. Anschließend können wir uns auf die Implementierung stürzen.

27. Vorausschicken will ich eine Definition:

Definition: Eine Folge von Zahlen $a_1, a_2, \ldots, a_n$ heißt k-sortiert, falls gilt:

$$\forall i \in [1 \ldots n] : a_i \leq a_i + k$$

Mit anderen Worten bedeutet dies, daß alle Zahlen sich in der richtigen Reihenfolge befinden, die genau k Schritte auseinanderliegen. Zum Beispiel ist die Zahlenfolge

$$44, 2, 13, 63, 4, 17, 77$$

schon 3-sortiert, weil alle Zahlenfolgen, die den Abstand 3 haben, bereits sortiert sind.

28. Die Idee von Shellsort ist, daß man die gegebene Zahlenfolge zuerst grob vorsortiert, zum Beispiel 8-sortiert. Anschließend wird zum Beispiel 4-sortiert, dann 2-sortiert und zum Ende nochmal 1-sortiert.

Man wird sich zunächst wundern, daß die Notwendigkeit mehrerer Durchläufe, die jeweils alle Elemente betreffen, nicht mehr Aufwand verursacht als erspart. Aber jede k-Sortierung betrifft entweder relativ wenige Elemente, oder die Elemente sind bereits zum größten Teil vorsortiert.

29. Es ist klar, daß die Methode eine korrekt sortierte Zahlenfolge liefert, da eventuelle Fehler bei einer k-Sortierung durch die abschließende 1-Sortierung ausgebügelt werden. Es ist auch klar, daß prinzipiell jede Folge von Schrittweiten verwendet werden kann, solange abschließend eine 1-Sortierung erfolgt. Nicht klar ist allerdings, wie die Wahl der Schrittweiten am besten zu treffen wäre, um möglichst wenig Vergleiche und Vertauschungen vornehmen zu müssen.

30. Allgemein benötigt Shellsort also eine Folge von Schrittweiten. Die t Schrittweiten will ich bezeichnen mit

$$h_1, h_2, \ldots, h_t$$

wobei verlangt ist, daß

$$h_t = 1, \text{und} \forall i \in [1 \ldots t-1] : h_{i+1} < h_i$$

Jede h-Sortierung wird hierbei als ein Sortieren durch Einfügen programmiert und zwar so, wie es in der Einleitung beschrieben wurde. Darum nennt man Shellsort auch "Sortieren durch Einfügen mit abnehmender Schrittweite".

31. Die Korrektheit des Verfahrens ist oben schon gezeigt worden. Zur Effizienz des Verfahrens soll an dieser Stelle nur gesagt werden, daß eine mathematische Analyse für die Sequenz

$$1, 3, 7, 15, 31 \ldots$$

von Schrittweiten ein Aufwand für Vertauschungen berechnet wurde, der zu $n^{1.2}$ proportional ist, wenn n die Länge der zu sortierenden Reihung ist. Dies ist eine wesentliche Verbesserung gegenüber herkömmlichen Sortiermethoden, die einen zu n^2 proportionalen Aufwand haben. Wenn man den Aufwand aber vergleicht mit schnellen Sortiermethoden wie *Quicksort*, zeigt sich doch noch die relative Langsamkeit von Shellsort (Aufwand Quicksort im mittleren Falle $n \cdot \log n$).

32. Die Programmierung von Shellsort. An dieser Stelle nun dürfte das Verfahren von Shellsort klar sein. Nun gilt es, den eben informell beschriebenen Algorithmus programmiersprachlich zu beschreiben und dabei möglichst wenige Fehler zu machen.

33. Wir wollten Shellsort als eine Prozedur programmieren. Sie hat folgende Form:

⟨ Die eigentliche Shellsort-Prozedur 33 ⟩ ≡
procedure *shellsort*; { Realisierung des Sortierers Shellsort }
  const ⟨ lokale Konstanten von Shellsort 34 ⟩
  var ⟨ lokale Variablen von Shellsort 35 ⟩
    begin ⟨ Initialisierung von lokalen Variablen in Shellsort 36 ⟩
    ⟨ Sortiere die Reihung A nach dem Shellsort-Verfahren 37 ⟩
    end; { von *shellsort* }

This code is used in section 12.

34. Es soll insgesamt t Schrittweiten geben, die in dem Feld h gespeichert werden. In unserem Falle ist $t = 4$ und wir wählen

$$h[1] = 9, h[2] = 5, h[3] = 3, h[4] = 1$$

⟨ lokale Konstanten von Shellsort 34 ⟩ ≡
  $t = 4$; { Anzahl der Schrittweiten }

This code is used in section 33.

35. ⟨ lokale Variablen von Shellsort 35 ⟩ ≡
h: **array** $[1 \mathrel{..} t]$ **of** *integer*; { Die Schrittweiten persönlich }

See also sections 38 and 41.

This code is used in section 33.

36. ⟨ Initialisierung von lokalen Variablen in Shellsort 36 ⟩ ≡
  $h[1] \leftarrow 9$; $h[2] \leftarrow 5$; $h[3] \leftarrow 3$; $h[4] \leftarrow 1$;

This code is used in section 33.

37. Shellsort beginnt also mit einer $h[1]$-Sortierung, fährt dann fort mit einer $h[2]$-Sortierung, usw.

⟨ Sortiere die Reihung A nach dem Shellsort-Verfahren 37 ⟩ ≡
  for $m \leftarrow 1$ **to** t **do**
    begin $k \leftarrow h[m]$; ⟨ k-sortiere die Reihung A 39 ⟩
    end;

This code is used in section 33.

38. Natürlich müssen die eben benutzten Indexvariablen m und k noch deklariert werden.

⟨ lokale Variablen von Shellsort 35 ⟩ +≡
m, k: *integer*; { Hilfsvariablen }

39. Die k-Sortierung wird erreicht, indem alle k Teilfolgen der Reihung A gleichzeitig k-sortiert werden. Wer hier näher hinsieht, erkennt das Schema des Sortierens durch Einfügen aus der Einleitung.

⟨ k-sortiere die Reihung A 39 ⟩ ≡
  for $i \leftarrow k + 1$ **to** n **do**
    begin $x \leftarrow A[i]$; ⟨ Fuege x am geeigneten Ort in $A[1], \ldots, A[i - k]$ ein 40 ⟩
    end;

This code is used in section 37.

40. Zur Bestimmung des Ortes, an dem x eingefügt werden soll, wechseln wir wieder zwischen Vergleichen und Bewegungen ab. x wandert nun aber immer k Schritte nach links durch.

Der Prozess ist beendet, wenn ein Element $A[j]$ gefunden wurde, das kleiner ist als x, oder wenn der Index j kleiner als k geworden ist, d. h. das linke Ende der Reihung erreicht ist.

$\langle$ Fuege x am geeigneten Ort in $A[1], \ldots, A[i-k]$ ein 40 $\rangle \equiv$
  $j \leftarrow i$; { Index für das Wandern nach links }
  while $(x < A[j-k]) \wedge (j > k)$ **do**
    begin $A[j] \leftarrow A[j-k]$; $j \leftarrow j-k$;
    end;
  $A[j] \leftarrow x$; { Den Wert x hier einsetzen }

This code is used in section 39.

41. Natürlich dürfen wir nicht vergessen, wieder die eben benutzten Hilfsvariablen zu deklarieren.

$\langle$ lokale Variablen von Shellsort 35 $\rangle +\equiv$
j: *integer*; { Index für das Wandern nach links }
x: *integer*; { Speichert das zu verschiebende Element }

42. Bezeichnerindex.

cc: 20, 23, 25.
i: 17.
initialize: 17.
integer: 8, 13, 14, 17, 22, 35, 38, 41.
output: 12.
page_number: 22, 23, 24.
page_offset: 22, 23, 24.
print_table: 20, 21.
random: 18.
randomize: 18, 19.
row_offset: 22, 24, 25.
rr: 20, 23, 24, 25.
shell: 12.
shellsort: 11, 13, 33.
write: 24, 25.
writeln: 24, 25.
ww: 20, 25.
zahl: 18.

⟨ k-sortiere die Reihung A 39 ⟩ Used in section 37.
⟨ Das Hauptprogramm 15 ⟩ Used in section 12.
⟨ Die eigentliche Shellsort-Prozedur 33 ⟩ Used in section 12.
⟨ Fuege x am geeigneten Ort in $A[1], \ldots, A[i-k]$ ein 40 ⟩ Used in section 39.
⟨ Fuelle die Reihung A mit Testdaten und zeige sie an 0 ⟩ Used in section 15.
⟨ Gebe die Testdaten auf den Bildschrim aus 23 ⟩ Used in section 21.
⟨ Gebe die sortierten Daten auf dem Bildschirm aus 0 ⟩ Used in section 15.
⟨ Gebe eine Seite Zahlen aus 24 ⟩ Used in section 23.
⟨ Gebe eine Zeile Zahlen aus 25 ⟩ Used in section 24.
⟨ Hilfsprozeduren 17, 21 ⟩ Used in section 12.
⟨ Hilfsvariablen für die Ausgabe 22 ⟩ Used in section 21.
⟨ Initialisierung von lokalen Variablen in Shellsort 36 ⟩ Used in section 33.
⟨ Konstanten des Programmes 13, 20 ⟩ Used in section 12.
⟨ Setze $A[i]$ auf einen Zufallswert 18 ⟩ Used in section 17.
⟨ Sortiere die Reihung mittels Shellsort 0 ⟩ Used in section 15.
⟨ Sortiere die Reihung A nach dem Shellsort-Verfahren 37 ⟩ Used in section 33.
⟨ Variablen des Programmes 14 ⟩ Used in section 12.
⟨ Vorarbeiten zur Generierung von Zufallszahlen 19 ⟩ Used in section 17.
⟨ lokale Konstanten von Shellsort 34 ⟩ Used in section 33.
⟨ lokale Variablen von Shellsort 35, 38, 41 ⟩ Used in section 33.