3 Methodisches Programmieren im Kleinen und Suchen/Sortieren

Lernziele:

Vorgehen beim Erstellen kleiner Programme
Effizienzbetrachtung, Test, Dokumentation
Standardkenntnisse Sortieren, wenig über Suchen

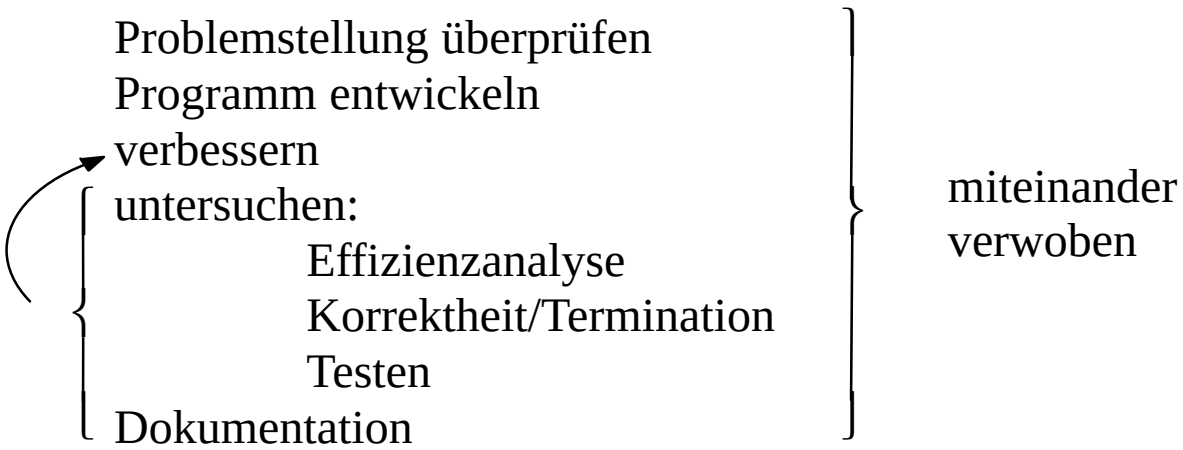
Die Entwicklung kleiner Programme

Vorgehen beim Lösen

– bisher:

Problemformulierung (meist gegeben: Aufgabe)
direkt Programm erstellt:
Hilfsmittel: schrittweise Verfeinerung
Wahl geeigneter Bezeichner
Verwendung von Kommentaren
dabei möglichst gute Gestaltung der Bedieneroberfläche

– systematisches Vorgehen bei kleinen Programmen



Eigenschaften der Lösung

– Problem → zugehöriges Programm
es gibt (unendl.) viele Lösungen

Lösungen: schwach / stark } äquivalent: gleiches Ergebnis / gleiche interne Zustandsübergänge

– welche Eigenschaft soll Lösung haben?

- Zuverlässigkeit
  Korrektheit: löst geg. Aufgabenstellung
  Robustheit gegen falsche Eingabedaten
  Ausfallsicherheit gegen Soft-/Hardwarefehler
- Benutzerfreundlichkeit: UI–Gestaltung
  Verständlichkeit
  Angemessenheit
  vernünftiges Fehlerverhalten
- Flexibilität
  Übertragbarkeit (Portabilität)
  Anpaßbarkeit (Veränderung, Erweiterung)
- Verständlichkeit des Programms
  Lesbarkeit
  Einfachheit
- Effizienz
  Laufzeit
  Speicherplatz
  wichtig sind auch Prozeßaspekte

Sortieren als Voraussetzung für effiziente Suche

Allgemeines

– häufig auftretendes Problem
  bei Übersetzung in Symboltabelle
  in Datenbanken bei großen Informationsbeständen
  in bel. Anwendungsprogrammen

– hier nur einfache Form
  in Feld im Hauptspeicher (internes Suchen,
    Random–Access Zugriffsstruktur)
  später bessere Möglichkeiten (→ Kap. 5)
  Suchen insb. auf kompl. Daten, oft auf ext. Speichermedien

– liegen die zu suchenden Datenelemente sortiert vor?
  nein: sequentielle Suche
  ja: binäre Suche

– zu suchende Elemente Suchinfo (mit Schlüssel)
  der Einfachheit halber Suchinfo nur Schlüssel
  setzen diesen `unsigned int`:
    könnte bel. Typ mit Vergleichsop. sein

– Motivation für Beschäftigung mit Sortieren in diesem Kapitel

Lineare (sequentielle) Suche

– Programm

```
enum zustT{gef, nicht_gef};
zustT status;
unsigned const dimVal=100; int Wert=... ;
typedef unsigned int itemT; // Aenderbarkeit
typedef int indexT;
indexT anzElem;
typedef itemT it_FT[dimVal];
it_FT feld;

// feld erhält Werte, z.B. über EA;
// anzElem ist die Anzahl der Elemente;
// Feldelemente: a[0], ..., a[anzElem–1]

void linSuche(it_FT &f, itemT ges_Wert,
              indexT ia, indexT ie,
              indexT &rsltInd, zustT &zust) {
  // besser it_FT &f,
  // d.h. call by reference
  // zu suchender Wert: ges_Wert;
  // rsltInd ist Index im Feld, bei Erfolg;
   indexT index, indexT og=ie+1;
   index = og; // Vorbes. ausserhalb
   for (indexT i=ia; i<og; i++) {
      if (f[i]==ges_Wert) {
         index=i; break;
      };
   };
   if (index < og) {
      rsltInd = index; zust = gef; }
   else {
      zust = nicht_gef;
   };
};
```

Aufruf:

```
...
// z.B. im Hauptprogramm
linSuche(feld, Wert, 0, anzElem–1,
         i, status);
if (status==gef) {
   cout << ”Wert ” << Wert << ” an Stelle ”
        << i << ” gefunden.” << endl; }
else {
   cout << ”Wert nicht gefunden.” << endl;
};
```

– Analyse

Sei n die Anzahl der Komponenten:

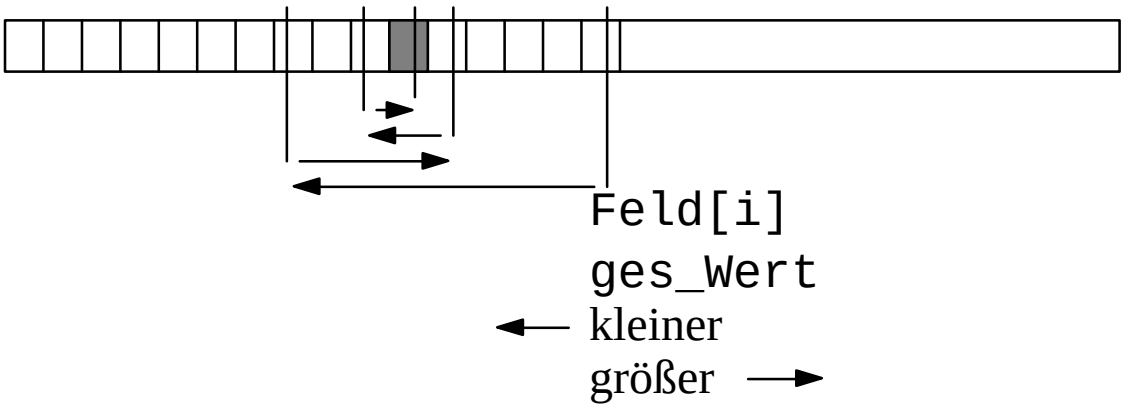
Im schlechtesten Fall muß das ganze Feld durchsucht werden: `linSuche` ist O(n)-Algorithmus (Anzahl der Vergleiche)

Im Mittel, wenn gesuchter Wert im Feld enthalten ist, benötigt man $\frac{n}{2}$ Vergleiche

Binäre Suche

– `linSuche` setzt keine Sortierung voraus
  Sortierung bringt im Mittel keinen Effizienzgewinn

– Idee binäre Suche



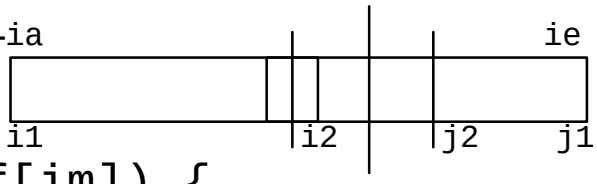
fortgesetzte Intervallschachtelung des Suchbereichs

– rekursives Programm (vereinfacht!)

```
void binSucheR(it_FT &f, item ges_Wert,
               index ia, index ie,
               index &ind, zust &z) {
 // Suche in Feldabschnitt f[ia]..f[ie],
 // 0<=ia, ie<=n–1
   index im; // mittlerer Index
   if (ia > ie) { // fuer Beendigung
      z = nicht_gef; return;
   } else {
      im = (ia + ie)/2;
   }
   if (f[im] == ges_Wert) {
      z = gef; ind = im;
   } else if (ges_Wert > f[im]) {
      binSucheR(f,ges_Wert,im+1,ie,ind,z);
   } else { // ges_Wert < f[im]
      binSucheR(f,ges_Wert,ia,im–1,ind,z);
   }
}
...
binSucheR(a, Wert, 0, n–1, i, status);
```

– iterative Fassung

```
void binSucheI(it_FT &f, item ges_Wert,
               index ia, index ie,
               index &ind, zust &z) {
   // Voraussetzung ia <= ie
   index i=ia, j=ie, im;
   while (true) {
      im = (i+j)/2; // mittlerer Index
      if (ges_Wert==f[im]){
         ind = im; z = gef;
         return; }
      else if (ges_Wert > f[im]) {
         i = im+1;} // Intervallschachtelung
      else { // ges_Wert < f[im]
         j = im–1;   // durch Indexveraend.
      }
      if (i>j) { // fuer Beendigung
         z = nicht_gef; return;
      }
   }
}
... // wie oben
binSucheI(a, Wert, 0, n–1, i, status);
```



– Vergleich: iterative Fassung einiges schneller!

- Messung: 1000 Elemente
  etwa 50% schneller als rekursive

– Analyse: Effizienzbetrachtung

In jedem Schritt wird die Hälfte ausgesondert:
Anzahl der noch zu betrachtenden Elemente
z.B. n=100, 49, 24, 11, 5, 2, 1 max. 7 Schritte
Anzahl zu untersuchender Fälle

gegenüber 100 im schlechtesten Fall
bzw. 50 im Mittel

Allgemein: $\lceil ld(n) \rceil$ Schritte im schlechtesten Fall

Wird durch Messung bestätigt:

1000 Elemente, alle Elemente 10000mal gesucht (10^7 Läufe):

`linSuche`		`binSuche`(iterativ)
1050 sec	Faktor 30	35 sec

Sortieren: Allgemeines

Präzisierung der Aufgabenstellung

– Problem:

In einer Datei befindet sich eine unbekannte Anzahl von Zahlen. Diese sollen gelesen und nach ihrer Größe sortiert ausgedruckt werden.

Unklarheiten:

- Mehrfachvorkommnisse: erlaubt, mehrfach aufgeführt?
- Sortierung: aufsteigend, absteigend?
- Sortierung in einem Feld: max. Anzahl bekannt? (internes Sortieren/externes Sortieren)
- Welcher Typ d. einzelnen Elemente: `int`, `float`, etc. ?
- Format: Wie ausdrucken?
- Feste Randbedingungen: Zahlen < größte in Basismaschine darstellbare Zahl

Vollständigkeit:

- Problemspezifikation unvollständig: Probleme bei Nutzung
- unvollst. Problemspezifikation
  Information aus anderer Quelle besorgen
  Spezifikation ergänzen

Widerspruchsfreiheit

– neue Formulierung:

In einer Datei befinden sich max. 100 nicht notwendigerweise verschiedene `unsigned`–Zahlen. Diese Zahlen sollen gelesen, der Größe nach aufsteigend sortiert und entsprechend ihrer Vorkommenshäufigkeit einzeln ausgegeben werde und zwar je 10 pro Zeile.

```
unsigned const dimVal=100;
typedef unsigned int item; // Aenderbarkeit
typedef int index;
typedef item it_FT[dimVal];
int n; it_FT a; unsigned pro_Z;

void main() {
   zahlenEinlesen   // Pseudo–
   zahlenSortieren  // anwei–
   zahlenDrucken    // sungen
}

{ // Zahlen einlesen:
   item x; n = 0;
   cin >> x;
   while(!cin.eof()) {
   // Zahl fuer Zahl in den Behaelter:
      a[n] = x; n++;
      if(n==dimVal) { break; }
      cin >> x;
   }
}

{ // Zahlen drucken:
   cout << "Die Zahlen in aufsteigender "
        << "Reihenfolge:" << endl;
   for (index k=0; k<n; k++) {
      cout << a[k] << " ";
      if (((k+1)%10)==0) { cout << endl; }
   }
}
```

Sortieren: Übersicht

– Hauptteil Sortieren
    einfache Verfahren: n^2
    gute Verfahren: n log n

    Sortierstrategien (s.u.):
        Einfügen
        Auswählen
        Vertauschen
        Verschmelzen

```
// Sortieren:
...

// Ergänzung der Deklarationen
// je nach Sortierverfahren (s.u.)

// Ausformulierung des Sortierens
// Da aufwendiger als andere Teile: Prozedur
```

– Def.: *Sortieren*:

Anordnen einer Menge von Objekten in bestimmte Ordnung

Gegeben $a_1, a_2, \ldots, a_n$

Gesucht Anordnung $a_{k_1}, a_{k_2}, \ldots, a_{k_n}$ mit

$$f(a_{k_1}) \leq f(a_{k_2}) \leq \ldots \leq f(a_{k_n})$$

Schlüssel wird nicht erst berechnet, ist Komponente:

```
struct Info_Element {
   item key;
   .
   . // weitere Komponenten
   .
};
```

`item` irgendein Typ mit Vergleichsoperationen, z.B. skalarer Typ, nicht notwendigerweise `unsigned int`;

– Sortieren also nicht Einteilung in Klassen (Sorten)

– Anwendungen:

- Voraussetzung für effizientes Suchen (s.o.)
- Datenbanksysteme, Betriebssysteme, Compiler, Anwendungen
- Ersatz für direkte Adressierung auf seq. Dateien

⇒ Verfahren mit geringem Speicher- und Laufzeitaufwand gesucht

– klassisches Kapitel (wie auch Suchen; nur Eingangsbeispiel)

für Leistungsanalyse von Algorithmen:
Tradeoff, Algorithmenstrukturen

– kann sein, daß $a_i = a_{i+1}$ etc.

Sortierverfahren heißen stabil, falls Reihenfolge von Elementen mit gleichem Schlüssel beim Sortieren unverändert bleibt.

Wichtig, da gleiche Elemente nach anderem Kriterium geordnet sein können.

– Klassifikation

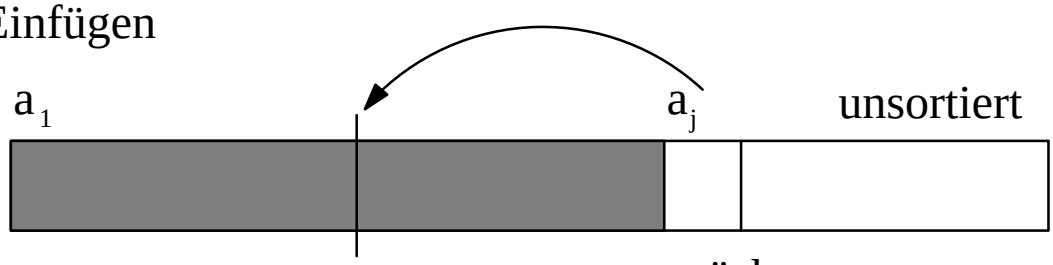
- internes Sortieren (auf Feldern) / externes Sortieren (auf seq. Dateien)
  (1) (2)
- direkte (einfache) Verfahren / verbesserte Verfahren
  (3) (4)
- stabile / instabile Verfahren
  (5) (6)
- einmaliges Sortieren(fester Datenbestand)
  (7)

  dauerndes Sortieren (veränderlicher Datenbestand)
  (8)

im folgenden (1), bedingt (3), (4), (5), (6), (7)
(8) kommt später im Zusammenhang mit Baumsuchen,
ext. Sortieren (2) wird nicht besprochen;

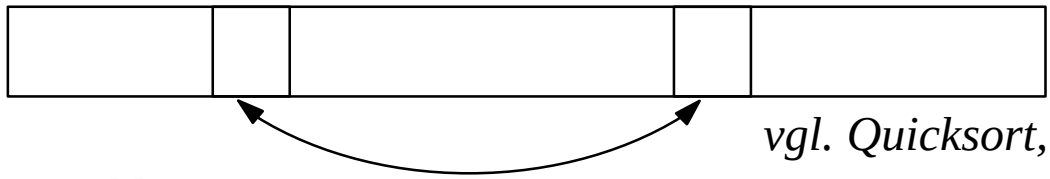
– Sortierprinzipien:

Einfügen



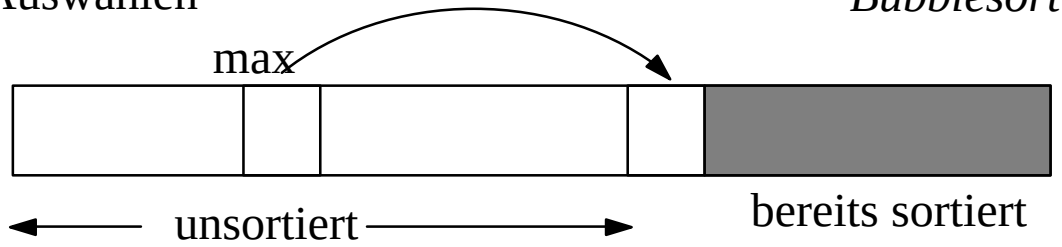
vgl. StraightInsertion

Austauschen



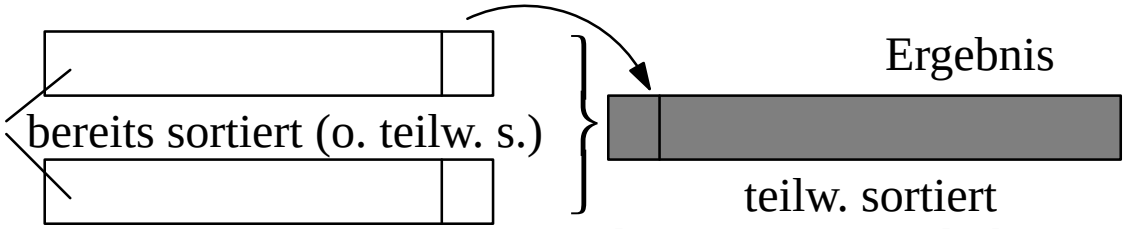
vgl. Quicksort,
Bubblesort

Auswählen



vgl. StraightSelection

Mischen

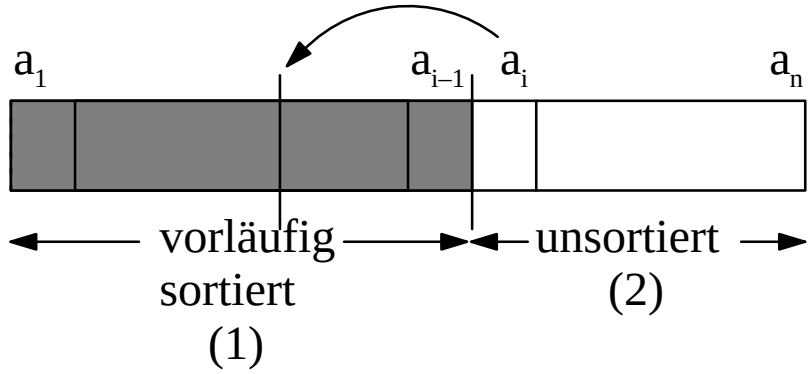


vgl. MergeSort (nicht bespr.)

Direkte Sortierverfahren

Sortieren durch Einfügen

– Idee noch einmal:



– Bsp.:

Anfangswerte	44	55	12	42	94	18	06	67
i=2	44	55	12	42	94	18	06	67
i=3	12	44	55	42	94	18	06	67
i=4	12	42	44	55	94	18	06	67
i=5	12	42	44	55	94	18	06	67
i=6	12	18	42	44	55	94	06	67
i=7	06	12	18	42	44	55	94	67
i=8	06	12	18	42	44	55	67	94

– Rohform:

```
for(int i=2; i<=n; i++) {
   // vom 2. Element an
   x=a[i];
   füge x geeignet in a[1]..a[i] ein
};
```

– nicht Stelle auffinden und dann verschieben,
  sondern Aufsuchen und verschieben in einem:
    von rechts nach links im Feldteil (1)
      Rechtsschieben, falls linkes Element noch größer;
      in die Lücke füllen, falls linkes Element kleiner / gleich
– Terminationsbedingungen
  • kleineres / gleiches Element gefunden
  • hinteres Ende erreicht

```
void StraightInsertion(it_FT &f, index ia,
                       index ie) {
 // direktes Verfahren, Sortieren durch
 // Einfuegen
   index j;
   item x;
   for (index i=ia+1; i<=ie; i++) {
      x = f[i]; // Zwischenspeicher
      j = i;
      // x einsortieren
      while ((j>ia) && (x<f[j-1])) {
         f[j] = f[j-1];
         j--;
      };
      f[j] = x;
   };
};
...
StraightInsertion(a, 0, n-1);
```

– Analyse: (C Anzahl der Vergleiche, M Anzahl der Bewegungen)

- Durchschnittliche Anzahl der Schlüsselvergleiche
  beim Einfügen von a_i ist $\frac{i}{2}$ (Mitte zwischen 1 und i–1)

$$\Rightarrow C_{\emptyset} = \sum_{i=2}^{n} C_i = \frac{1}{2}\left[\sum_{i=2}^{n} i\right] = \frac{1}{2}\left[\left[\sum_{i=1}^{n} i\right] - 1\right] = \frac{1}{2}\left[\frac{n(n+1)}{2} - 1\right] = \frac{n^2+n-2}{4}$$

- Durchschnittliche Anzahl von Bewegungen
  $M_i = C_i + 2$ (inkl. Marke setzen)

$$\Rightarrow M_{\emptyset} = \frac{n^2+n-2}{4} + (n-2+1)\cdot 2 = \frac{n^2+n+8n-2-8}{4} = \frac{n^2+9n-10}{4}$$

- bester Fall: alles geordnet

$$C_{best} = \sum_{i=2}^{n} 1 = n-1$$

$$M_{best} = \sum_{i=2}^{n} 3 = 3\cdot(n-1)$$

- schlechtester Fall: umgekehrt geordnet

$$C_{worst} = \sum_{i=2}^{n} i = \frac{n^2+n-2}{2}$$

$$M_{worst} = \frac{n^2+n-2}{2} + (n-2+1)\cdot 2 = \frac{n^2+5n-6}{2}$$

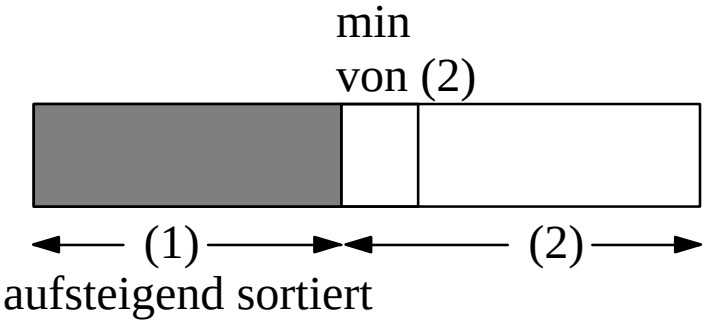
– Resumee:

typisches $O(n^2)$ - Verfahren, mit natürlichem Verhalten
(bester, durchschnittlicher, schlechtester Fall)

zur O-Notation, s.u.

Sortieren durch Auswählen

– direktes Auswählen




```
void StraightSelection(it_FT &f, index ia,
                       index ie) {
 // direktes Verfahren,
 // Sortieren durch Auswaehlen
   index k;
   item x;
   for (index i=ia; i<=ie-1; i++) {
      k = i; x = f[i];
      for (index j=i+1; j<=ie; j++) {
         if (f[j]<x) {
            k = j; x = f[j];
         }
      }
      f[k] = f[i]; f[i] = x;
   }
}
...
StraightSelection(a, 0, n-1);
```

– Analyse:

$C_{\substack{best \\ \emptyset \\ worst}} \in O(n^2)$ $\qquad$ $M_{best} = 3 \cdot (n-1)$

$M_{worst} \in O(n^2)$

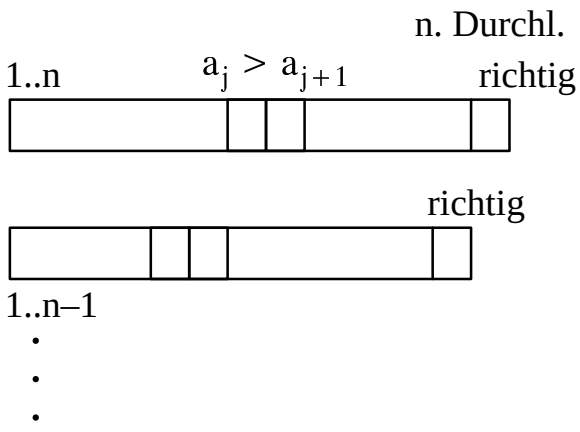
Somit: Wie kann die Anzahl der Vergleiche reduziert werden? Kann von jedem Durchlauf mehr Information zurückbehalten werden, als das kleinste Element? (siehe Heapsort in Literatur)

Verbesserung, Dokumentation, Test

Sortieren durch Vertauschen:

– Bubblesort

größere Blase steigt auf, bis sie durch eine noch größere Blase aufgehalten wird




```
void BubbleSort (it_FT &f, index ia,
                 index ie) {
// direktes Verfahren; Sortieren durch
// Vertauschen
bool getauscht;
item hilf;
   do {
      getauscht = false;
      for (index i=ia; i<ie; i++) {
         if (a[i]>a[i+1]) {
            // Vertausche
            hilf = a[i];
            a[i] = a[i+1];
            a[i+1] = hilf;
            getauscht = true;
         }
      }
   } while (getauscht);
}
```

vollständiges Programm

```
const dimVal=100;
typedef item it_FT[dimVal];
it_FT a;
index n=0;
item x;
unsigned pro_Z=10;

void Bubblesort(it_FT &f, index ia, index ie);

void main() { // Sortieren mit Bubblesort
   // Zahlen einlesen
   while (!cin.eof()) {
      cin >> x;
      a[n] = x; n++;
   };

   BubbleSort(a, 0, n–1);

   // Zahlen drucken:
   {  cout << ”Die Zahlen in aufsteigender ”
           << ”Reihenfolge:” << endl;
      for (index j=0; j<=n–1; j++) {
         // Ausgabe 10 pro Zeile;
         cout << a[j] << ” ”;
         if (((j+1)%pro_Z)==0){cout << endl;};
      };
   };
}

// Hier Prozedur für BubbleSort
```

Verbesserungen

– Verbesserung 0. Schritt:
  eine noch unsinnigere Lösung wäre gewesen:
    n-maliger Durchlauf, unabhängig davon, ob im letzten Durchlauf vertauscht wurde
– Verbesserung 1. Schritt:

i	a					
undef.	16	33	94	82	6	58
1	16	33	94	82	6	58
2	16	33	94	82	6	58
3	16	33	82	94	6	58
4	16	33	82	6	94	58
5	16	33	82	6	58	94

1. Durchlauf größte Zahl am richtigen Platz
2. Durchlauf zweitgrößte Zahl am richtigen Platz usw.

- brauchen nur max. n–1 Durchläufe,
  letzte Zahl dann automatisch am richtigen Platz
  $\Rightarrow$ letzter Durchlauf überflüssig (hatten wir schon)
- brauchen beim k-ten Durchlauf nur bis n–k zu laufen
- Abbruchkriterium *nicht* überflüssig: wollen rechtzeitig aufhören, wenn nicht mehr vertauscht wurde

– Verbesserung 2. Schritt:

k	i	a					
1	1	16	33	94	82	6	58
	...	...					
1	5	16	32	82	6	58	94
2	1	16	33	82	6	58	94
2	2	16	33	82	6	58	94
2	3	16	33	6	82	58	94
2	4	16	33	6	58	82	94
3	1	16	33	6	58	82	94
3	2	16	6	33	58	82	94
3	3	16	6	33	58	82	94

überflüssig bis n–k zu laufen, wenn im letzten Durchlauf nicht bis dorthin vertauscht wurde

– Neuer Sortierteil (Rumpf von Bubblesort)
  rechtes Ende pro Durchlauf nicht nur um 1 verkleinern:

```
...
int rechtesEnde, letztesI;
...
letztesI = ie;
do {
   rechtesEnde = letztesI; letztesI=ia–1;
   for (int i=0; i<rechtesEnde; i++) {
      if (f[i] > f[i+1]) {
         hilf = f[i]; f[i] = f[i+1];
         f[i+1] = hilf; letztesI = i;
      }
   }
} while (letztesI!=ia–1);
```

– Analyse: Effizienzbetrachtung Bubblesort

best–case–Analyse: geordnet
1 Schleifendurchlauf
n–1 Vergleiche, 0 Vertauschungen
hier Bubblesort gut!

worst–case–Analyse: umgekehrt geordnet
n–1 Durchläufe
k–ter Durchlauf:
innere Schleife (n–k)–mal vertauscht

$$\sum_{k=1}^{n-1}(n-k) = \sum_{k=1}^{n-1} k = \frac{n^2-n}{2}$$

Anzahl der Vergleiche/
Vertauschungen

$\Rightarrow$ Bubblesort $O(n^2)$

d.h. $\exists$ Konstante a, b, c mit

$$f^{WC}_{Zeit}(\text{Bubblesort, Feld\_der\_Laenge n}) < an^2 + bn + c$$

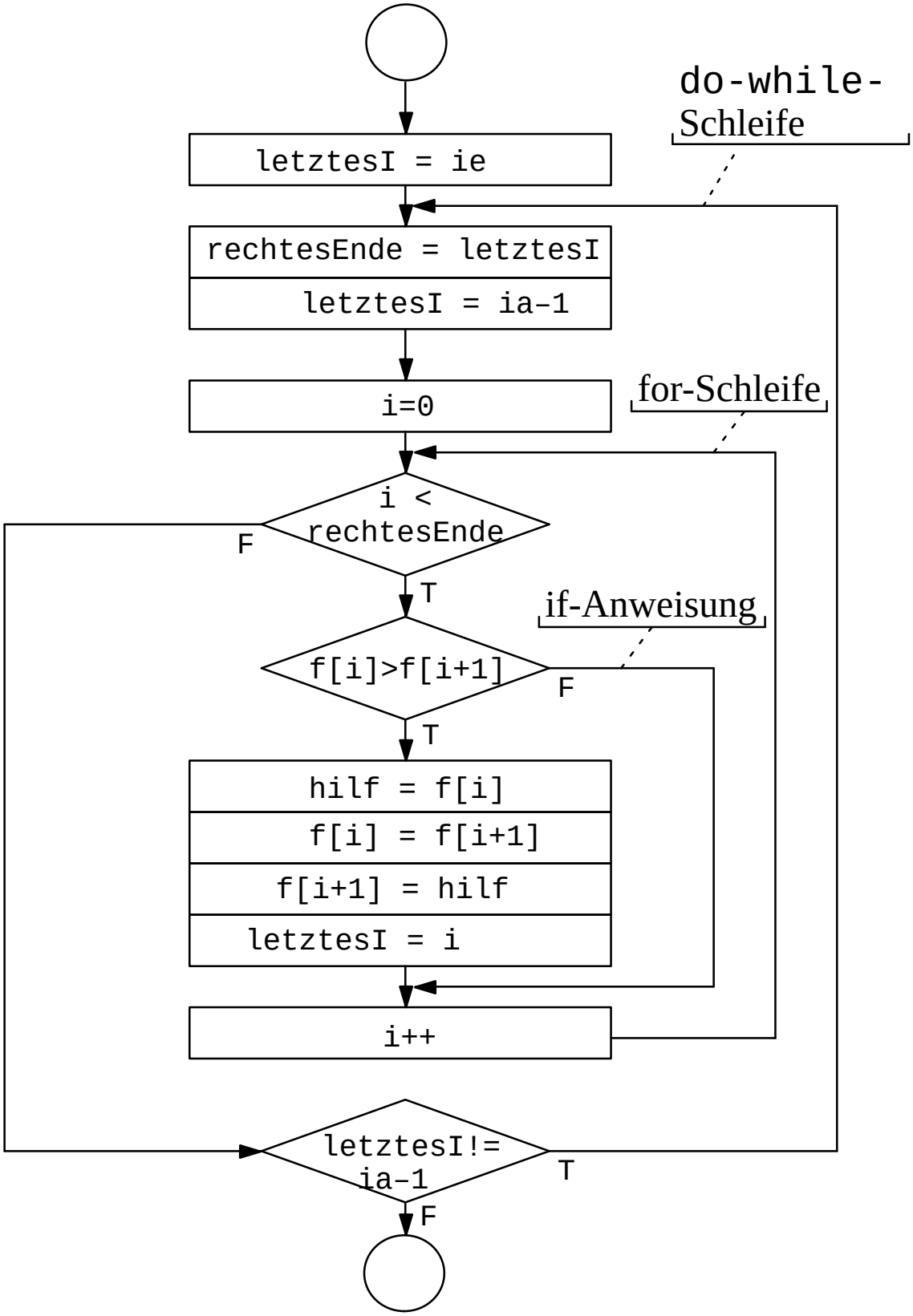
Konstanten nicht ausgerechnet: Vorsicht!

Es gibt bessere Verfahren: $O(n \log n)$

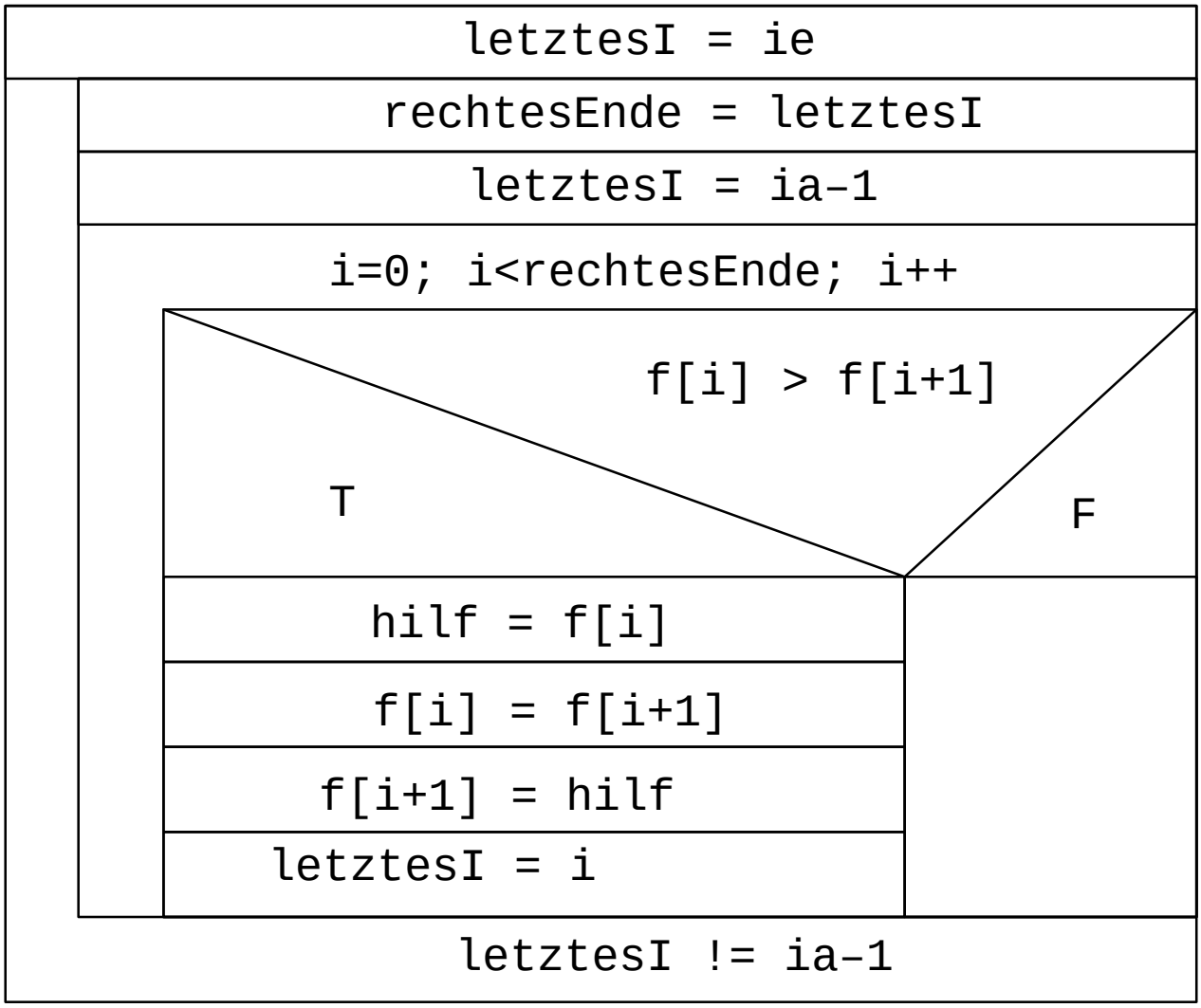
Allgemeines

- bisher: im Programm durch Kommentare
  Programm selbst durch Lesbarkeit
- “parallele” Notation: Flußdiagramme
  Struktogramme } (s.u.)
- große Projekte: Anforderungsdefinition
  Entwurfsspezifikation
  Rationales
  etc.

- Flußdiagramm zu Bubblesort-Rumpf



- Struktogramm zu Bubblesort-Rumpf



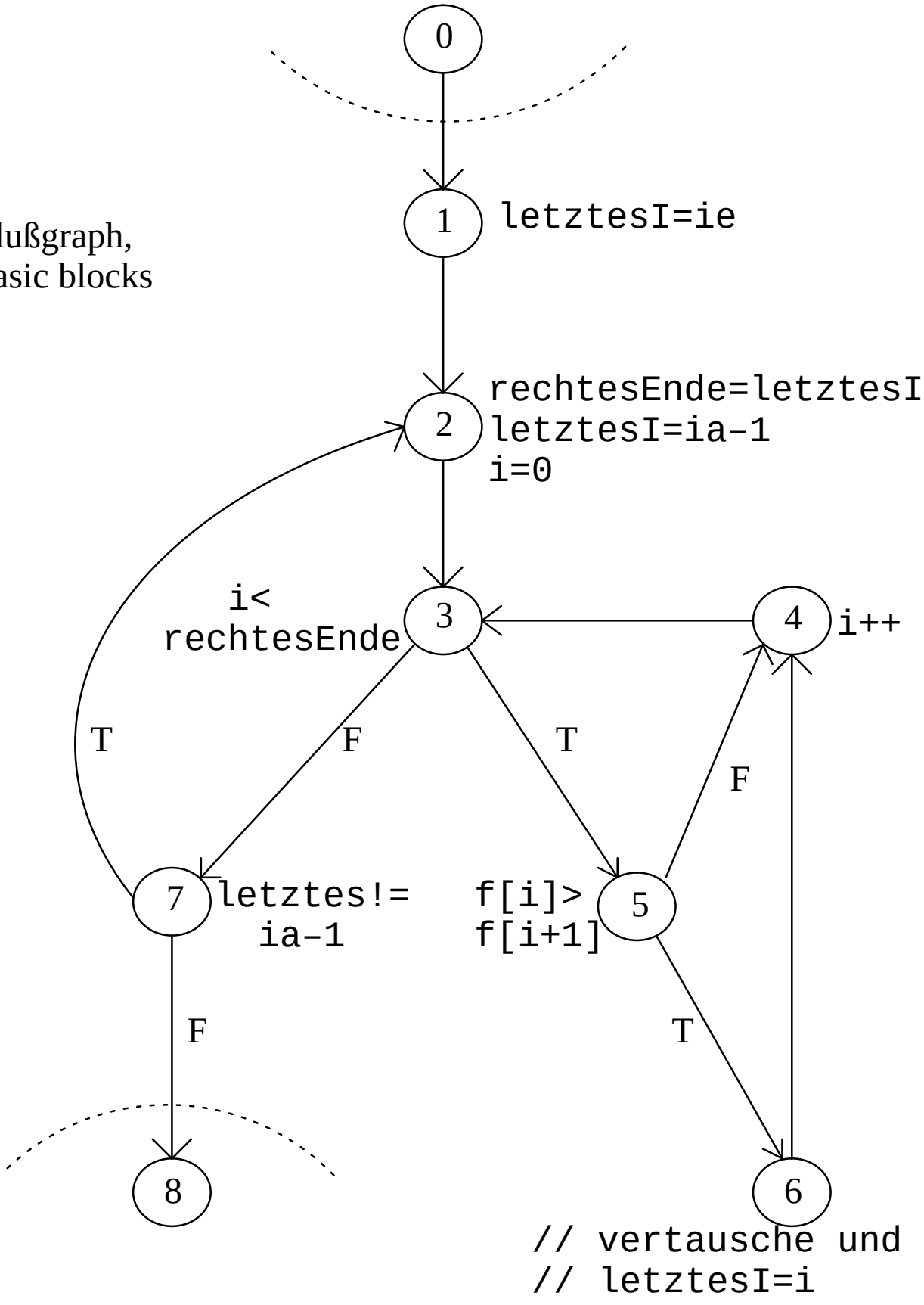
– Testen und Testdatenermittlung

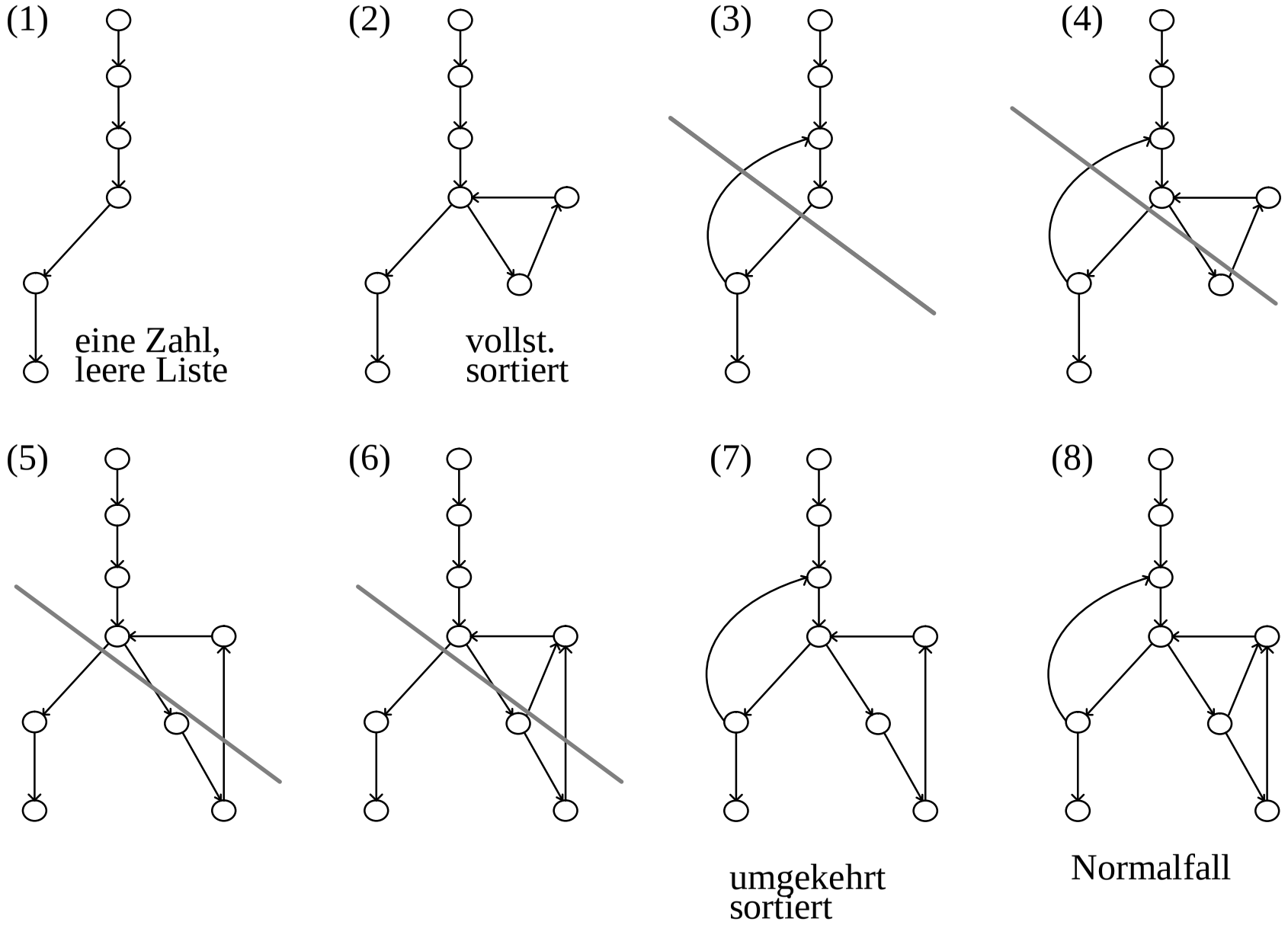
- black–box–Test (intuitiver Ansatz)
  z.B. Normalfall-Extremfall-Betrachtung

Normalfall	16	31	94	82	6	58
	...					
Extremfall	1	2	3	4	5	6
	6	5	4	3	2	1
	5	5	5	5	5	5
	6					
	leere Eingabe					

- white–box–Test

Flußgraph,
basic blocks




(1)
eine Zahl,
leere Liste
(2)
vollst.
sortiert
(3)
(4)
(5)
(6)
(7)
umgekehrt
sortiert
(8)
Normalfall

- Testen und Durchführbarkeit (white–box):

  Ermittlung aller dyn. Programmpfade nicht möglich!

  Ermittlung aller Programmpfade (bis auf Vielfache von Schleifendurchläufen, wie oben), nur bei kleinen Programmen möglich.

  Kantenüberdeckung und Bedingungsüberdeckung üblich (Überdeckungsverfahren nicht vorgeführt)!

- Testen und Korrektheit:

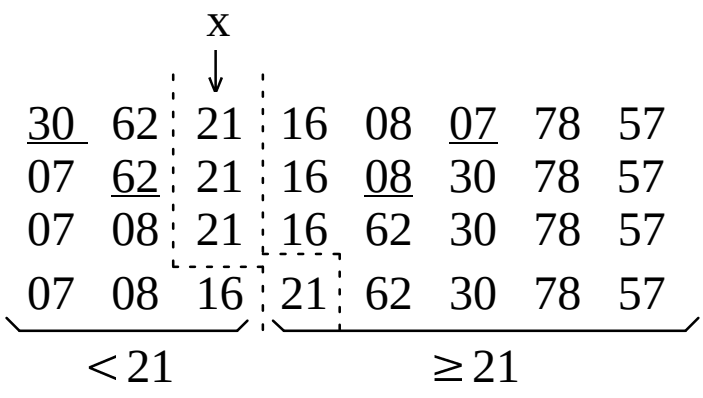
  “Durch Testen kann nur die Anwesenheit aber nicht die Abwesenheit von Fehlern gezeigt werden.” (Dijkstra)

Effizientes Verfahren: Quicksort

Entwicklungsidee

– Können wir ein verbessertes Verfahren finden, das die Anzahl der Vertauschungen verkleinert?
  Ansatz: Bewegungen eines falsch plazierten Elements um größere Strecke, nicht nur eine Stelle nach links oder rechts

– Idee:
  Quicksort (Hoare 1962): gebräuchlichste interne Sortiermethode
  Zeiger i,j (anfangs i=ia (=0), j=ie (=n–1)); wähle Index *eines* Elementes x aus.
  Durchlaufe von links bis f_i mit $f_i > x$, von rechts bis $f_j < x$;
  Vertausche beide Elemente f_i, f_j; setze i=i+1, j=j–1;
  Wiederhole bis i>j.

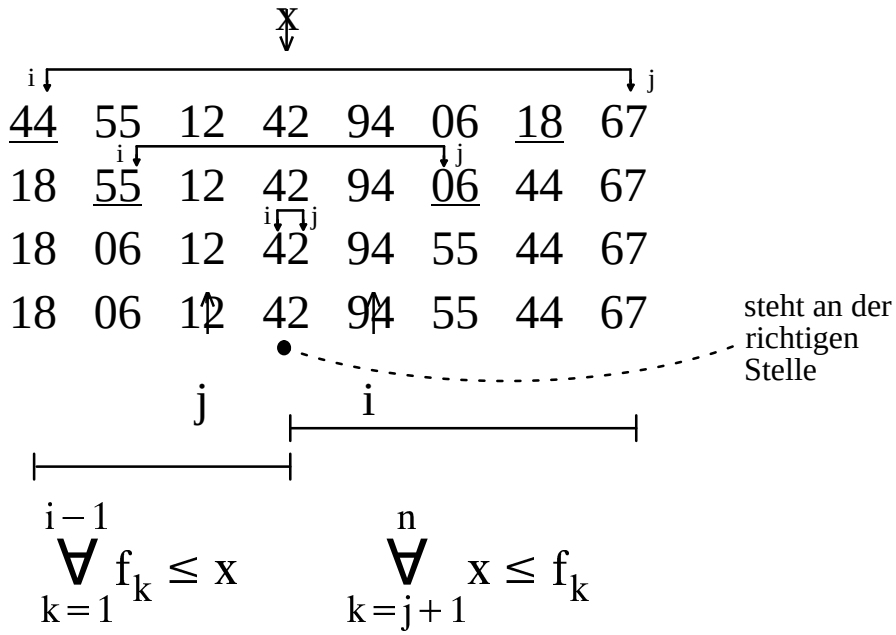


Anwendung auf beide entstandenen Teil-intervalle etc.

Index nach Durchlauf muß nicht mit x übereinstimmen!

– erste Fassung, Rumpf der Prozedur:

```
item x, hilf;
index ia=0, ie=n–1;

// Partition: zerteilt das
// zu sortierende Feld in zwei Abschnitte
{  index i=ia, j=ie;
   waehle ein zufaelliges Element x aus
   f[0]..f[n–1] aus;
   while(i<=j) {
      while (f[i]<x) { i++; }
      while (x<f[j]) { j––; }
      if (i<=j) {
         hilf=f[i]; f[i]=f[j]; f[j]=hilf;
         i++; j––;
      }
   }
}
```



$$\forall_{k=1}^{i-1} f_k \le x \qquad \forall_{k=j+1}^{n} x \le f_k$$

Invariante der while-Anweisung

anfangs klar

bei jedem Vertauschungsschritt/Indexweiterschieben erhalten

=> Ende: Zerlegung in zwei Intervalle mit Schlüsseln kleiner-gleich bzw. Schlüsseln größer-gleich

Lösung: rekursive Anwendung auf Teilintervalle

– Programm

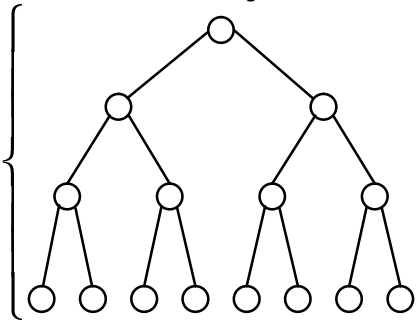
```
...
void QuickSort(it_FT &f, index ia, index ie);
  // Prozedurspezifikation
void Sort(it_FT &f, index L, index R);
  // Prozedurspezifikation:
  // Anschluß irgendein Suchverfahren

void main() {
   ...
   // Einlesen: ...
   Sort(a, 0, n–1);
   /// Ausgabe: ...
}
void Sort (it_FT &f, index L, index R) {
   QuickSort(f, L, R);
}
void QuickSort(it_FT &f,index ia,index ie) {
   index i=ia, j=ie; item x, hilf;
   {  x=f[(ia+ie)/2];//Element mit mittl. Index
      while (i<=j) {
         while (f[i]<x) { i++; }
         while (x<f[j]) { j--; }
         if (i<=j) {
            hilf=f[i]; f[i]=f[j]; f[j]=hilf;
            i++; j--;
         }
      }
      if (ia<j) { QuickSort(f, ia, j); }
      if (i<ie) { QuickSort(f, i, ie); }
   }
}
```

Analyse

– Auswahl von x entscheidend

– best case:
x jeweils "Mittelobjekt"

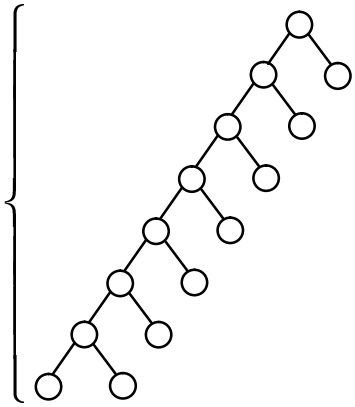
Höhe= $\lceil \log_2 n \rceil$



auf jedem Niveau n Vergleiche
$\Rightarrow \; C_{best} \in O(n \log_2 n)$

– worst case:
x jeweils größtes/kleinstes Element

Höhe = n



$\Rightarrow \; C_{worst} \in O(n^{2)}$

– average case Laufzeit: $O(n \log_2 n)$ Sehr gut!
allerdings sollte n nicht zu klein sein
($\rightarrow$ dir. Methode)

– Auswahl des mittleren Elements
  - Laufzeit gut bis auf pathologische Fälle
  - Mittelobjekt (Median)
    $\leq$ Hälfte der Schlüssel
    $\geq$ Hälfte der Schlüssel
  - exakte Bestimmung würde Sortieren voraussetzen. Wie können wir ungefähr bestimmen und damit schlechtesten Fall verbessern?
  - Näherung in obiger Lösung

    Wähle Mittelobjekt aus $\left\{a_l, a_{\lfloor \frac{l+r}{2} \rfloor}, a_r\right\}$,

    d.h. entsprechenden Index