4. Sortierverfahren

Einführung

Elementare Sortierverfahren
- Sortieren durch direktes Auswählen (Straight Selection Sort)
- Sortieren durch Vertauschen (Bubble Sort)
- Sortieren durch Einfügen (Insertion Sort)

Shell-Sort (Sortieren mit abnehmenden Inkrementen)

Quick-Sort

Auswahl-Sortierung (Turnier-Sortierung, Heap-Sort)

Sortieren durch Streuen und Sammeln

Merge-Sort

Externe Sortierverfahren
- Ausgeglichenes k-Wege-Merge-Sort
- Auswahl-Sortierung mit Ersetzung (Replacement Selection)

Anwendungen

Sortierung ist fundamentale Operation in zahllosen System- und Anwendungsprogrammen: dominierender Anteil in Programmlaufzeiten

Beispiele
- Wörter in Lexikon
- Bibliothekskatalog
- Kontoauszug (geordnet nach Datum der Kontobewegung)
- Sortierung von Studenten nach Namen, Notendurchschnitt, Semester, ...
- Sortierung von Adressen / Briefen nach Postleitzahlen, Ort, Straße ...
- indirekte Nutzung u.a. bei Duplikaterkennung (z.B. # unterschiedlicher Wörter in einem Text)

Naive Implementierung zur Duplikaterkennung

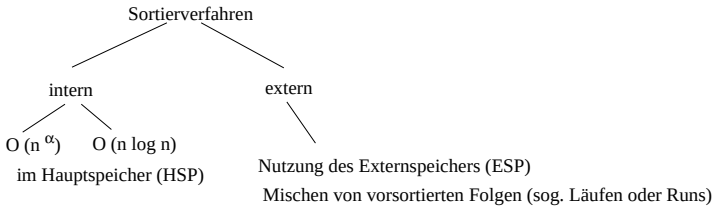
```
public static boolean duplicates( Object [ ] A ) {
  for( int i = 0; i < A.length; i++ )
    for( int j = i + 1; j < A.length; j++ )
      if( A[i].equals( A[j] ) )
        return true;   // Duplicate found
  return false;        // No duplicates found
}
```

Sortierverfahren

allgemeine Problemstellung
- Gegeben: Folge von Sätzen S_1, ... S_n, wobei jeder Satz S_i Schlüssel K_i besitzt
- Gesucht: Permutation π der Zahlen von 1 bis n, welche aufsteigende Schlüsselreihenfolge ergibt:

$$K_{\pi(1)} \leq K_{\pi(2)} \leq \ldots \leq K_{\pi(n)}$$

Interne vs. externe Sortierverfahren



einfache vs. spezielle Sortierverfahren
- Annahmen über Datenstrukturen (z.B. Array) oder Schlüsseleigenschaften

Sortierverfahren (2)

Wünschenswert: *stabile* Sortierverfahren, bei denen die relative Reihenfolge gleicher Schlüsselwerte bei der Sortierung gewahrt bleibt

Speicherplatzbedarf am geringsten für Sortieren am Ort ("in situ")

Weitere Kostenmaße
- #Schlüsselvergleiche C (C_{min}, C_{max}, C_{avg})
- #Satzbewegungen M (M_{min}, M_{max}, M_{avg})

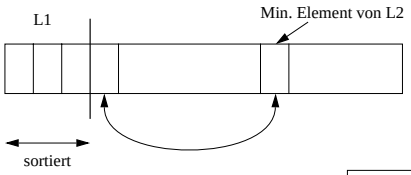
Satz:

Jedes allgemeine Sortierverfahren, welches zur Sortierung nur Vergleichsoperationen zwischen Schlüsseln (sowie Tauschoperationen) verwendet, benötigt sowohl im mittleren als auch im schlechtesten Fall wenigstens Ω (n • log n) Schlüsselvergleiche

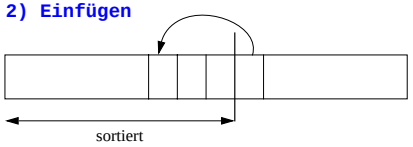
Klassifizierung von Sortiertechniken

Sortieren durch . . .

1) Auswählen

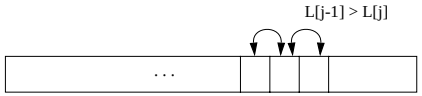


2) Einfügen

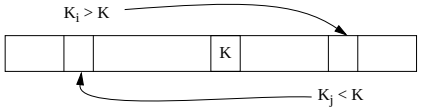


3) Austauschen

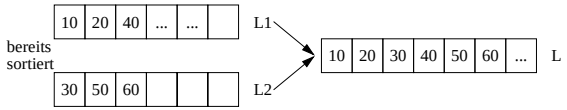
a) lokal



b) entfernt



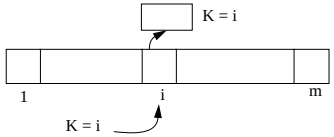
4) Mischen



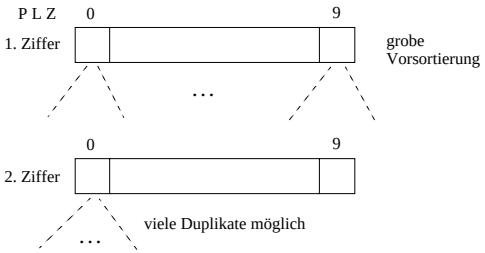
Klassifizierung von Sortiertechniken (2)

5. Streuen und Sammeln

- begrenzter Schlüsselbereich m, z. B. 1 - 1000
- relativ dichte Schlüsselbelegung n ≤ m
- Duplikate möglich (n > m)
- lineare Sortierkosten !



6. Fachverteilen (z. B. Poststelle)



Basisklasse für Sortieralgorithmen

```
public abstract class SortAlgorithm {

  static void swap (Object A[], int i, int j) {
    Object o = A[i];
    A[i] = A[j];
    A[j] = o;
  }

  /** Sortiert das übergebene Array;
   *  Element an Position 0 bleibt unberücksichtigt (Stopper...)!
   */
  public static void sort (Orderable A[]) {}

  public static void print(Orderable A[]) {
    for (int i=1; i<A.length; i++)
      System.out.println(A[i].getKey());
  }
}
```

auf zu sortierenden Objekten muß Ordnung definiert sein

Sortieren durch Auswählen (Selection Sort)

Idee

- Auswahl des kleinsten Elementes im unsortierten Teil der Liste
- Austausch mit dem ersten Element der unsortierten Teilliste

27	75	99	3	45	12	87

Sortieren durch Auswählen (2)

Sortierprozedur

```
public class SelectionSort extends SortAlgorithm {

  public static void sort (Orderable A[]) {
    for (int i = 1; i < A.length-1; i++) {
      int min = i;        // Suche kleinstes Element
      for (int j = i+1; j < A.length; j++) {
        if (A[j].less(A[min])) {
          min = j;
        }
      }
      swap (A, i, min);  // tausche aktuelles mit kleinstem Element
    }
  }
}
```

Anzahl Schlüsselvergleiche

$C_{min}(n) = C_{max}(n) =$

Anzahl Satzbewegungen (durch Swap):

$M_{min}(n) = M_{max}(n) =$

in-situ-Verfahren, nicht stabil

Sortieren durch Vertauschen (Bubble Sort)

Idee

- Vertauschen benachbarter Elemente, die nicht in Sortierordnung
- pro Durchgang wandert größtes Element der noch unsortierten Teilliste nach “oben”
- Sortierung endet, wenn in einem Durchgang keine Vertauschung mehr erfolgte

1						N
27	75	99	3	45	12	87

Variation: Shaker-Sort (Durchlaufrichtung wechselt bei jedem Durchgang)

Bubble Sort (2)

```
public class BubbleSort extends SortAlgorithm {

  public static void sort (Orderable A[]) {
    for (int i=A.length-1; i>1; i--) {
      for (int j=1; j<i; j++) {
        if (A[j].greater(A[j+1]))
          swap (A, j, j+1);  // Vertauschen benachbarter Elemente
      }
    }
  }
}
```

Kosten

$C_{min}(n) = C_{max}(n) = C_{avg}(n) =$

$M_{min}(n) =$

$M_{max}(n) =$

$M_{avg}(n) =$

in-situ-Verfahren, stabil

Vorsortierung kann genutzt werden

Sortieren durch Einfügen (Insertion Sort)

Idee

- i-tes Element der Liste x (1. Element der unsortierten Teilliste) wird an der richtigen Stelle der bereits sortierten Teilliste (1 bis i-1) eingefügt
- Elemente in sortierter Teilliste mit höherem Schlüsselwert als x werden verschoben

1						N
27	75	99	3	45	12	87

Insertion Sort (2)

```
public class InsertionSort extends SortAlgorithm {

  public static void sort (Orderable A[]) {
    for (int i = 2; i < A.length; i++) {
      A[0] = A[i];  // Speichere aktuelles Element an reservierter Pos.
      int j = i - 1;
      while ((j > 0) && A[j].greater(A[0])) {
        A[j+1] = A[j];  // Verschiebe größeres Elem. eine Pos. nach rechts
        j--;
      }
      A[j+1] = A[0];   // Füge zwischengespeichertes Elem. ein
    }
  }
}
```

Kosten

C_{min} (n) =

C_{max} (n) =

M_{min} (n) =

M_{max} (n) =

in-situ-Verfahren, stabil

Vergleich der einfachen Sortierverfahren

#Schlüsselvergleiche

	C_{min}	C_{avg}	C_{max}
Direktes Auswählen	n (n - 1) / 2	n (n - 1) / 2	n (n - 1) / 2
Bubble Sort	n (n - 1) / 2	n (n - 1) / 2	n (n - 1) / 2
Einfügen	n - 1	$(n^2 + 3n + 4) / 4$	$(n^2 + n - 2) / 2$

#Satzbewegungen

	M_{min}	M_{avg}	M_{max}
Direktes Auswählen	3 (n - 1)	3 (n - 1)	3 (n - 1)
Bubble Sort	0	3n (n - 1) / 4	3n (n - 1) / 2
Einfügen	3 (n - 1)	$(n^2 + 11n + 12) / 4$	$(n^2 + 5n - 6) / 2$

Sortieren von großen Datensätzen (Indirektes Sortieren)

Indirektes Sortieren erlaubt, Kopieraufwand für *jedes* Sortierverfahrens auf lineare Kosten O (n) zu beschränken

Führen eines Hilfsfeldes von Indizes auf das eigentliche Listenfeld

Liste A [1..n]

Pointerfeld P [1..n] (Initialisierung P[i] = i)

Schlüsselzugriff: A [i] -> A [P[i]]

Austausch:
```
swap (A, i, j) ->
swap (P , i, j)
```

	1	2	3	4	5
A	D	A	C	E	B
P					

Sortierung erfolgt lediglich auf Indexfeld

abschließend erfolgt linearer Durchlauf zum Umkopieren der Sätze

Shell-Sort (Shell, 1957)
(Sortieren mit abnehmenden Inkrementen)

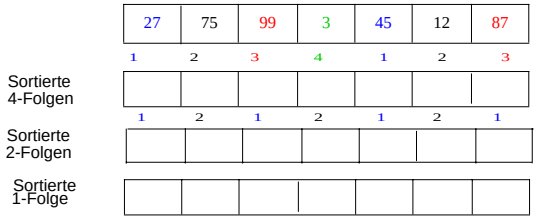
Idee

- Sortierung in mehreren Stufen “von grob bis fein”
- Vorsortierung reduziert Anzahl von Tauschvorgängen

Vorgehensweise:

- Festlegung von t Inkrementen (Elementabständen) h_i mit $h_1 > h_2 > \ldots . h_t = 1$ zur Bildung von disjunkten Teillisten
- Im i-ten Schritt erfolgt unabhängiges Sortieren aller h_i-Folgen (mit Insertion Sort)
- Eine Elementbewegung bewirkt Sprung um h_i Positionen
- Im letzten Schritt erfolgt “normales” Sortieren durch Einfügen

Beispiel:
$h_1 = 4$; $h_2 = 2$; $h_3 = 1$



Shell-Sort (2)

Aufwand wesentlich von Wahl der Anzahl und Art der Schrittweiten abhängig

Insertion Sort ist Spezialfall (t=1)

Knuth [Kn73] empfiehlt

- Schrittweitenfolge von 1, 3, 7, 15, 31 . . . mit $h_{i-1} = 2 \cdot h_i + 1$, $h_t = 1$ und $t = \lfloor \log_2 n \rfloor - 1$
- Aufwand O ($n^{1.2}$)

Andere Vorschläge erreichen O ($n \log^2 n$)

- Inkremente der Form $2^p 3^q$

Quick-Sort

Hoare, 1962

andere Bezeichnung: Partition-Exchange-Sort

Anwendung der Divide-and-Conquer-Strategie auf Sortieren durch Austauschen:

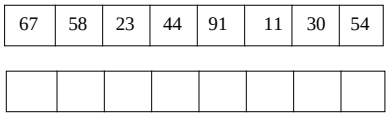
- Bestimmung eines Pivot-Elementes x in L
- Zerlegung der Liste in zwei Teillisten L1 und L2 durch Austauschen, so daß linke (rechte) Teilliste L1 (L2) nur Schlüssel enthält, die kleiner (größer oder gleich) als x sind

- Rekursive Sortierung der Teillisten, bis nur noch Listen der Länge 1 verbleiben

Realisierung der Zerlegung

- Duchlauf des Listenbereiches von links über Indexvariable i, solange bis L [i] >= x vorliegt
- Duchlauf des Listenbereiches von rechts über Indexvariable j, solange bis L [j] <= x vorliegt
- Austausch von L[i] und L [j]
- Fortsetzung der Durchläufe bis i > j gilt

Quick-Sort-Beispiel



Quick-Sort (2)

```
public class QuickSort extends SortAlgorithm {

  public static void sort (Orderable A[], int l, int r) {
    int i=l;
    int j=r;
    if (r <= l) return;  // Listenlänge < 2 -> Ende
    A[0] = A[(l+r)/2];   // speichere Pivot-Element in Position 0
    do {
      while (A[i].less(A[0])) i++;
      while (A[j].greater(A[0])) j--;
      if (i <= j) {
        swap (A, i, j);
        i++; j--;
      }
    } while (i <= j);
    sort (A, l, j);
    sort (A, i, r);
  }

  public static void sort (Orderable A[]) {
    sort(A, 1, A.length-1);
  } }
```

In-situ-Verfahren; nicht “stabil”

Quick-Sort (3)

Kosten am geringsten, wenn Teillisten stets gleichlang sind, d.h. wenn das Pivot-Element dem mittleren Schlüsselwert (Median) entspricht

- Halbierung der Teillisten bei jeder Zerlegung
- Kosten O (n log n)

Worst-Case

- Liste der Länge k wird in Teillisten der Längen 1 und k-1 zerlegt (z.B. bei bereits sortierter Eingabe und Wahl des ersten oder letzten Elementes als Pivot-Element)
- Kosten O (n^2)

Wahl des Pivot-Elementes von entscheidender Bedeutung

Sinnvolle Methoden

- mittleres Element
- Mittlerer Wert von k Elementen (z.B. k=3)

=> bei fast allen Eingabefolgen können Kosten in der Größenordnung O (n log n) erzielt werden

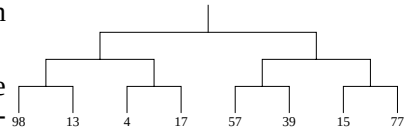
Zahlreiche Variationen von Quick-Sort
(z.B. Behandlung von Duplikaten, Umschalten auf elementares Sortierverfahren für kleine Teillisten)

Turnier-Sortierung

Maximum- bzw. Minimum-Bestimmung einer Sortierung analog zur Siegerermittlung bei Sportturnieren mit KO-Prinzip

- paarweise Wettkämpfe zwischen Spielern/Mannschaften
- nur Sieger kommt weiter
- Sieger des Finales ist Gesamtsieger

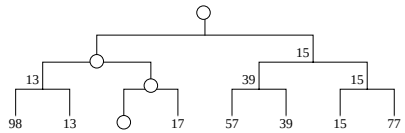
Zugehörige Auswahlstruktur ist ein binärer Baum



Aber: der zweite Sieger (das zweite Element der Sortierung) ist nicht automatisch der Verlierer im Finale

Stattdessen: Neuaustragung des Wettkampfes auf dem Pfad des Siegers (ohne seine Beteiligung)

- Pfad für Wurzelelement hinabsteigen und Element jeweils entfernen
- Neubestimmung der Sieger



Turnier-Sortierung (2)

Algorithmus TOURNAMENT SORT:

“Spiele ein KO-Turnier und erstelle dabei einen binären Auswahlbaum”

FOR I := 1 **TO** n **DO**

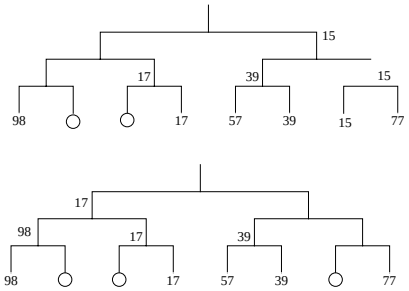
“Gib Element an der Wurzel aus”

“Steige Pfad des Elementes an der Wurzel hinab und lösche es”

“Steige Pfad zurück an die Wurzel und spiele ihn dabei neu aus”

END

Weitere Schritte im Beispiel



Turnier-Sortierung (3)

Anzahl Vergleiche (Annahme $n = 2^k$)

- #Vergleiche für Aufbau des initialen Auswahlbaumes

$$2^{k-1} + 2^{k-2} + \ldots + 1 = \sum_{i=0}^{k-1} 2^i = 2^k - 1 = n - 1$$

- pro Schleifendurchlauf Absteigen und Aufsteigen im Baum über k Stufen:
  $2k = 2 \log_2 n$ Vergleiche

- Gesamtaufwand:

Platzbedarf für Auswahlbaum

$$2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1 = 2n - 1$$

Heap-Sort (Williams, 1964)

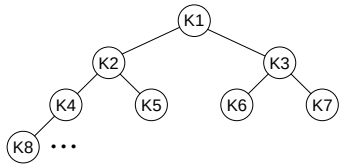
Reduzierung des Speicherplatzbedarfs gegenüber Turnier-Sort, indem "Löcher" im Auswahlbaum umgangen werden

Heap (Halde):

- hier: binärer Auswahlbaum mit der Eigenschaft, daß sich das größte Element jedes Teilbaumes in dessen Wurzel befindet
- => Baumwurzel enthält größtes Element

Vorgehensweise

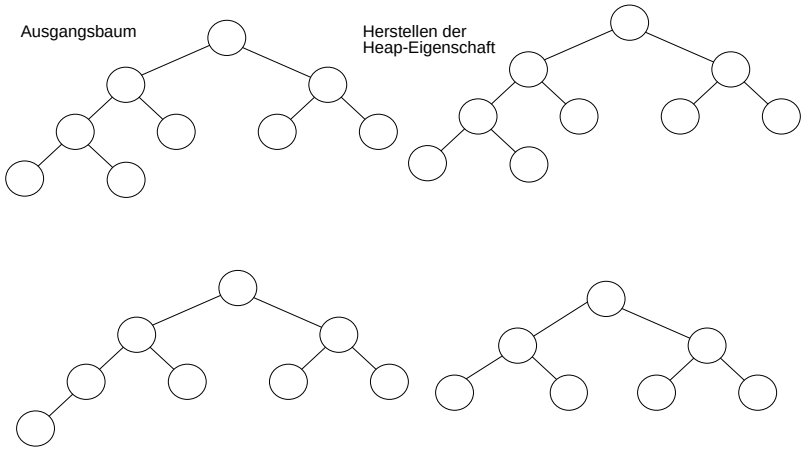
- Repräsentation der Schlüsselfolge $K_1, K_2, K_3, \ldots$ in einem Binärbaum minimaler Höhe nach nebenstehendem Schema



- *Herstellen der Heap-Eigenschaft:* ausgehend von Blättern durch "Absinken" von Knoteninhalten, welche kleiner sind als für einen der direkten Nachfolgerknoten
- Ausgabe des Wurzelelementes
- Ersetzen des Wurzelelementes mit dem am weitesten rechts stehenden Element der untersten Baumebene
- Wiederherstellen der Heap-Eigenschaft durch "Absenken" des Wurzelelementes; Ausgabe der neuen Wurzel usw. (solange bis Baum nur noch 1 Element enthält)

Heap-Sort (2)

Beispiel für Schlüsselfolge 57, 16, 62, 30, 80, 7, 21, 78, 41



Heap-Sort (3)

Effiziente Realisierbarkeit mit Arrays

- Wurzelelement = erstes Feldelement A [1]
- direkter Vaterknoten von A [i] ist A [i / 2]
- *Heap-Bedingung:* A [i] <= A [i/2]

Beispiel (Ausgangsliste 57, 16, 62,30, 80, 7, 21, 78, 41)

Heap-Darstellung

1	2	3	4	5	6	7	8	9
80	78	62	57	16	7	21	30	41

Algorithmus

1. Aufbau des Heaps durch Absenken aller Vaterknoten A[n / 2] bis A [1]
2. Vertauschen von A [1] mit A [n] (statt Entfernen der Wurzel)
3. Reduzierung der Feldlänge n um 1
4. Falls n > 1:
   - Absenken der neuen Wurzel A [1]
   - Gehe zu 2

1	2	3	4	5	6	7	8	9

Heap-Sort (4)

```
public class HeapSort extends SortAlgorithm {
  /** Absenken des i-ten Elements im Bereich 2*i bis n */
  static void sink (Orderable A[], int i, int n) {
    while (2*i <= n) {       // i hat Sohn
      int j = 2*i;           // j zeigt auf linken Sohn
      if ((j < n) && A[j].less(A[j+1]))
        j = j + 1;           // rechter Sohn ist größer als linker
      if (A[i].less(A[j])) { // aktuelles Element kleiner als Söhne
        swap(A,i,j);         // tausche mit dem größten Sohn
        i = j;               // i zeigt wieder auf aktuelles Element
      } else i = n; }}       // Söhne kleiner als Element -> Ende

  /** Stelle Heap-Eigenschaften im Array her. */
  static void makeHeap (Orderable A[]) {
    int n = A.length-1;
    for (int i = n/2; i > 0; i--)
      sink(A,i,n); }

  public static void sort (Orderable A[]) {
    int n = A.length-1;
    makeHeap(A);                      // stelle Heap-Eigenschaft her
    for (int i = n; i > 1; i--) {     // sortiere
      swap(A,1,i);
      sink(A,1,i-1); }}}
```


Heap-Sort (5)

Für $2^{k-1} \le n < 2^k$ gilt

- Baum hat die Höhe k
- Anzahl der Knoten auf Stufe i $(0 \le i \le k-1)$ ist 2^i

Kosten der Prozedur Sink

- Schleife wird höchstens h - 1 mal ausgeführt (h ist Höhe des Baumes mit Wurzelelement i)
- Zeitbedarf O(h) mit h <= k

Kosten zur Erstellung des anfänglichen Heaps

- Absenken nur für Knoten mit nicht-leeren Unterbäumen (Stufe k-2 bis Stufe 0)
- Kosten für einen Knoten auf der Stufe k-2 betragen höchstens 1· c Einheiten, die Kosten für die Wurzel (k-1) · c Einheiten.
- max. Kosten insgesamt

$$2^{k-2} \cdot 1 \cdot c + 2^{k-3} \cdot 2 \cdot c + \ldots + 2^0 \cdot (k-1) \cdot c = \sum_{i=1}^{k-1} c \cdot i \cdot 2^{k-i-1} = c \cdot 2^{k-1} \sum_{i=1}^{k-1} i \cdot 2^{-i} < c \cdot \frac{n}{2} \cdot \sum_{i=1}^{k-1} i \cdot 2^{-i}$$

- Gesamtkosten O (n)

Kosten der Sortierung (zweite FOR-Schleife): n-1 Aufrufe von Sink mit Kosten von höchstens O (k)=O ($\log_2$ n)

maximale Gesamtkosten von HEAPSORT: O(n) + O(n $\log_2$n) = O(n $\log_2$n).

Sortieren durch Streuen und Sammeln (Distribution-Sort, Bucket-Sort)

lineare Sortierkosten O (n) !

kein allgemeines Sortierverfahren, sondern beschränkt auf kleine und zusammenhängende Schlüsselbereiche 1..m

Die Verteilung der Schlüsselwerte wird für alle Werte bestimmt und daraus die relative Position jedes Eingabewertes in der Sortierfolge bestimmt

Vorgehensweise

- Hilfsfeld COUNT [1..m]
- Bestimmen der Häufigkeit des Vorkommens für jeden der m möglichen Schlüsselwerte („Streuen“)
- Bestimmung der akkumulierten Häufigkeiten in COUNT
- „Sammeln“ von i = 1 bis m: falls COUNT [i] > 0 wird i-ter Schlüsselwert COUNT [i]-mal ausgegeben

Beispiel: m=10; Eingabe 4 0 1 1 3 5 6 9 7 3 8 5 (n=12)

Kosten

- Keine Duplikate
  a) Streuen C_1 . n
  b) Sammeln C_2 . m
- mit Duplikaten: d
  a) Streuen C_1 . n
  b) Sammeln C_2 (m+d)

Distribution Sort (2)

Algorithmus

```
public class SimpleDistributionSort {
  /** Sortiert ein Feld von Werten im Bereich 1 bis m */
  public static void sort (int[] A) {
    int i,j,k;                    // Laufvariablen
    int m = 1000;                 // m mögliche Schlüsselwerte (0 bis 999)
    int n = A.length-1;           // Position 0 des Arr. wird nicht betrachtet
    int[] count = new int[m];     // Zähler für die Anzahl der einzelnen Zeichen
    for (i=0; i<m; i++) count[i] = 0;  // Zähler initialisieren
    for (i=1; i<=n; i++)               // Zählen der Zeichen (Streuphase)
      count[A[i]]++;
    k=0;
    for (i=0; i<m; i++)           // Sammelphase
      for (j=0; j<count[i]; j++)
        A[k++] = i;               // schreibe i count[i] mal in A ein
  }
}
```


Distribution-Sort / Fachverteilen

Verallgemeinerung: Sortierung von k-Tupeln gemäß lexikographischer Ordnung („Fachverteilen“)

Lexikographische Ordnung:

$A = \{a_1, a_2, \ldots, a_n\}$ sei Alphabet mit gegebener Ordnung $a_1 < a_2 < \ldots < a_n$
die sich wie folgt auf A* fortsetzt:

$v \leq w$ $(v, w \in A^*)$: $\Leftrightarrow$ $(w = v\,u$ mit $u \in A^*)$ oder
$(v = u\,a_i\,u'$ und $w = u\,a_j\,u''$ mit $u, u', u'' \in A^*$ und $a_i, a_j \in A$ mit $i < j)$

Die antisymmetrische Relation ”≤” heißt lexikographische Ordnung.

Sortierung erfolgt in k Schritten:

- Sortierung nach der letzten Stelle
- Sortierung nach der vorletzten Stelle
  ...
- Sortierung nach der ersten Stelle

Beispiel: Sortierung von Postleitzahlen
67663, 04425, 80638, 35037, 55128, 04179, 79699, 71672

Distribution-Sort (4)

```
public class  DistributionSort{
  public static void sort (Orderable[] A, int m, int k) {
    for (int i=k; i > 0; i--) {    // läuft über alle Schlüsselpositionen, beginnend mit der letzten
      int[] count = new int[m];
      for (int j=0; j < m; j++) count[j] = 0;             // count initialisieren
      for (int j=1; j < A.length; j++)
        count[key(i, k, A[j])]++; // Bedarf für Fachgröße bestimmen
      for (int j=1; j < m; j++) count[j] += count[j-1];   // Aufsummieren
      for (int j=m-1; j > 0; j--)
        count[j] = count[j-1]+1;  // Beginn der Fächer im Array bestimmen
      count[0] = 1;               // Fach für Objekt j beginnt an Pos. count[j]

     Orderable B = new Orderable[A.length];
      for (int j=1; j < A.length; j++) {  // Einordnen und dabei Fachgrenze
        int z=key(i, k, A[j]);            // anpassen
        B[count[z]++] = A[j];
      }
      System.arraycopy(B, 0, A, 0, A.length); // kopiere B -> A
    }
  }
}
```

Aufwand:

Merge-Sort

anwendbar für internes und externes Sortieren

Ansatz (Divide-and-Conquer-Strategie)

- rekursives Zerlegen in jeweils gleich große Teilfolgen
- Verschmelzen (Mischen) der sortierten Teilfolgen

Algorithmus (2-Wege-Merge-Sort)

```
public class MergeSort extends SortAlgorithm {
  static void sort (Orderable A[], int l, int r) {
    if (l < r) {
      int m = (l+r+1)/2;     // auf Mitte der Liste zeigen
      sort (A, l, m-1);      // linke Teil-Liste sortieren
      sort (A, m, r);        // rechte Teil-Liste sortieren
      merge(A, l, m, r);     // linken und rechten Teil mischen
    }
  }

  public static void sort (Orderable A[]) {
    sort(A, 1, A.length-1);
  }
}
```

Merge-Sort (2)

Mischen (Prozedur Merge)

- pro Teilliste wird Zeiger (Index) auf nächstes Element geführt
- jeweils kleinstes Element wird in Ergebnisliste übernommen und Indexzeiger fortgeschaltet
- sobald eine Teilliste „erschöpft“ ist, werden alle verbleibenden Elemente der anderen Teilliste in die Ergebnisliste übernommen
- lineare Kosten

```
static void merge(Orderable A[], int l, int m, int r) {
  Orderable[] B = new Orderable[r-l+1];
  for (int i=0, j=l, k=m; i < B.length; i++)
    if ((k > r) || (j < m) && A[j].less(A[k]))
      B[i] = A[j++];
    else
      B[i] = A[k++];
  for (int i=0; i < B.length; i++) A[l+i] = B[i];
}
```

Beispiel: Eingabe 7, 4, 15, 3, 11, 17, 12, 8, 18, 14

Merge-Sort (3)

Direktes (reines, nicht-rekursives) 2-Wege-Merge-Sort

- zunächst werden benachbarte Elemente zu sortierten Teillisten der Länge 2 gemischt
- fortgesetztes Mischen benachbarter (sortierter) Teillisten
- Länge sortierter Teillisten verdoppelt sich pro Durchgang
- Sortierung ist beendet, sobald in einem Durchgang nur noch zwei Teillisten verschmolzen werden

Beispiel: 7, 4, 15, 3, 11, 17, 12, 8, 18, 14

Merge-Sort (4)

Natürliches 2-Wege-Merge-Sort:

- statt mit 1-elementigen Teillisten zu beginnen, werden bereits anfangs möglichst lange sortierte Teilfolgen („runs“) verwendet
- Nutzung einer bereits vorliegenden (natürlichen) Sortierung in der Eingabefolge

Beispiel: 7, 4, 15, 3, 11, 17, 12, 8, 18, 14

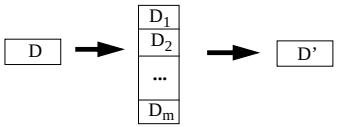
Externes Sortieren

Anwendung, falls zu sortierende Daten nicht vollständig im Hauptspeicher gehalten werden können

Hauptziel: Minimierung von Externspeicherzugriffen

Ansatz

- Zerlegen der Eingabedatei in m Teildateien, die jeweils im Hauptspeicher sortiert werden können
- Interne Sortierung und Zwischenspeicherung der Runs
- Mischen der m Runs (m-Wege-Mischen)



Bewertung

- optimale E/A-Kosten (1 Durchgang)
- setzt Dateispeicherung auf Direktzugriffsmedium (z.B. Magnetplatten) voraus, da ggf. sehr viele temporäre Dateien D_i
- verfügbarer Hauptspeicher muß wenigstens m+1 Seiten (Blöcke) umfassen

Restriktiver: Sortieren mit Bändern

- Ausgeglichenes 2-Wege-Merge-Sort (4 Bänder)
- Ausgeglichenes k-Wege-Merge-Sort (2k Bänder)

Externes Sortieren (2)

Ausgeglichenes 2-Wege-Merge-Sort

- vier Bänder B1, B2, B3, B4; Eingabe sei auf B1; Hauptspeicher fasse r Datensätze
- Wiederholtes Lesen von r Sätzen von B1, interne Sortierung und abwechselndes Ausschreiben der Runs auf B3 und B4 bis B1 erschöpft ist
- Mischen der Runs von B3 und B4 (ergibt Runs der Länge 2r) und abwechselndes Schreiben auf B1 und B2
- Fortgesetztes Mischen und Verteilen bis nur noch 1 Run übrig bleibt

Beispiel 14, 4, 3, 17, 22, 5, 25, 13, 9, 10, 1, 11, 12, 6, 2, 15 (r=4)

- #Durchgänge:

Ausgeglichenes k-Wege-Merge-Sort

- k-Wege-Aufteilen und k-Wege-Mischen mit 2k Bändern
- #Durchgänge:

Externes Sortieren (3)

Mischen bei Mehrwege-Merge-Sort kann durch Auswahlbaum (Turnier-Sortierung oder Heap-Sort) beschleunigt werden:
Replacement Selection Sort

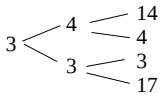
Bei k Schlüsseln kann Entfernen des Minimums und Hinzufügen eines neuen Schlüssels in O (log k) Schritten erfolgen

Auswahlbaum kann auch bei unsortierter Eingabe (initiale Zerlegung) genutzt werden, um Länge der erzeugten Runs zu erhöhen => Sortierung erfordert weniger Durchgänge

- ausgegebenes (Wurzel-) Element wird im Auswahlbaum durch nächstes Element x aus der Eingabe ersetzt
- x kann noch im gleichen Run untergebracht werden, sofern x nicht kleiner als das größte schon ausgegebene Element ist
- Ersetzung erfolgt solange bis alle Schlüssel im Auswahlbaum kleiner sind als der zuletzt ausgegebene (=> neuer Run)
- im Mittel verdoppelt sich die Run-Länge (auf 2r)

Externes Sortieren (4)

Beispiel 14, 4, 3, 17, 22, 5, 25, 13, 9, 10, 1, 11, 12, 6, 2, 15 (r=4)



22, 5, 25, 13, 9, 10, 1, 11, 12, 6, 2, 15

Zusammenfassung

Kosten allgemeiner Sortierverfahren wenigstens O (n log n)

Elementare Sortierverfahren: Kosten O (n^2)

- einfache Implementierung; ausreichend bei kleinem n
- gute Lösung: Insertion Sort

O(n log n)-Sortierverfahren: Heap-Sort, Quick-Sort, Merge-Sort

- Heap-Sort und Merge-Sort sind Worst-Case-optimal (O (n log n))
- in Messungen erzielte Quick-Sort in den meisten Fällen die besten Ergebnisse

Begrenzung der Kosten für Umkopieren durch indirekte Sortierverfahren

Vorteilhafte Eigenschaften

- Ausnutzen einer weitgehenden Vorsortierung
- Stabilität

lineare Sortierkosten in Spezialfällen erreichbar: Streuen und Sammeln

Externes Sortieren

- fortgesetztes Zerlegen und Mischen
- Mehrwege-Merge-Sort reduziert Externspeicherzugriffe
- empfehlenswert: Replacement Selection