IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(08 - Einfache Sortierverfahren)

Prof. Dr. Susanne Albers

Sortieren

Motivation, Einführung

- Datenbestände müssen sehr oft sortiert werden, etwa um Zuordnungen herzustellen oder um darin suchen zu können.
- Im täglichen Leben werden ebenfalls oft Dinge sortiert.
  Mögliche Strategien etwa beim Sortieren von Spielkarten sind:

  1. Alle Karten kommen in die Hand, anschließend werden benachbarte Karten (in korrekte Reihenfolge) getauscht, bis alle korrekt sind → Bubble Sort
  2. Die jeweils niedrigste Karte wird vom Tisch in die Hand genommen und an die schon sortierten dort angefügt → Selection Sort
  3. Karten werden in beliebiger Reihenfolge aufgenommen und jeweils an der richtigen Stelle (die zu suchen ist) in die sortierten in der Hand eingefügt → Insertion Sort

Rahmenbedingungen beim Sortieren

- Die zu sortierenden Datensätze können beliebig strukturiert sein.
- Der Sortierschlüssel ist Teil jedes Datensatzes. Davon kann es mehrere geben.
- Das Sortieren ändert die Reihenfolge von Datensätzen oder beschreibt (etwa durch Angabe einer Folge von Operationen) wie die korrekte Reihenfolge erreicht werden kann.

Internes Sortieren: Anzahl der Datensätze ist so begrenzt, dass alle gleichzeitig in den Speicher passen.

Externes Sortieren: Anzahl der Datensätze ist zu groß, um sie alle gleichzeitig im Speicher zu halten. Kann auf Internes Sortieren zurückgeführt werden:

1. Die zu sortierende Datei wird in Teile zerlegt, die jeweils intern sortiert werden können.
2. Die Teile werden sortiert.
3. Die Ergebnisdatei wird aus den sortierten Teilen zusammengemischt: Dazu ist ein Datensatz pro Teil im Speicher ausreichend, der kleinste wird zum Ergebnis hinzugefügt und durch seinen Nachfolger aus der Ursprungsdatei ersetzt.

Klassifikationsmöglichkeiten für Sortierverfahren

Methode des Verfahrens

- Vertauschen, Einfügen, Auswählen
- Vorsortierung ausnutzen
- Rekursion, Iteration
- Verwendung spezieller Datenstrukturen

Effizienz

- $O(n^2)$, $O(n \log n)$, $O(n)$
- Verwendung von zusätzlichem Speicher

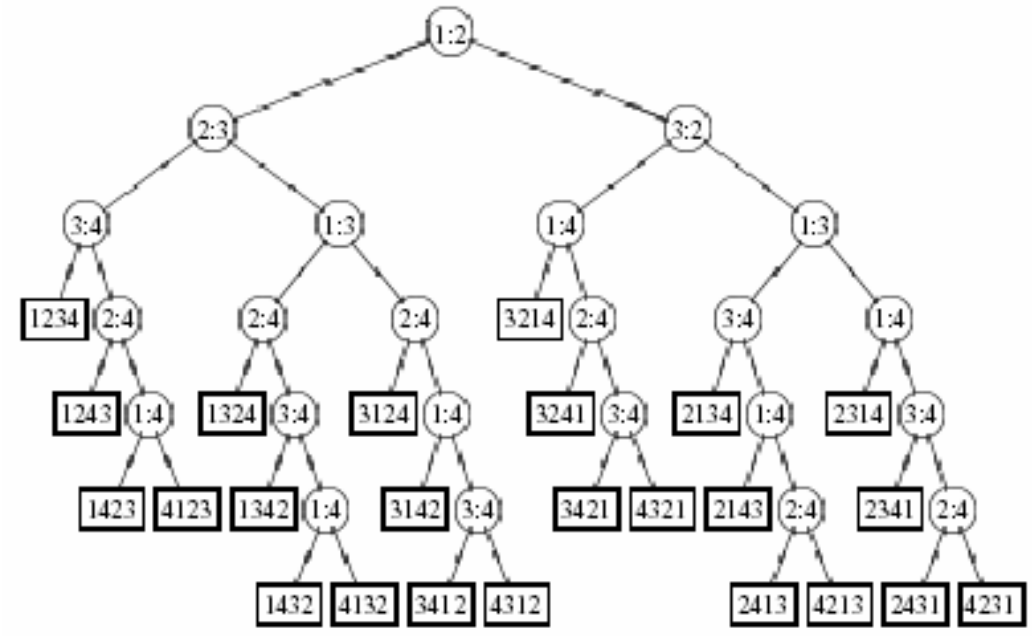
Allgemeine Verfahren oder Verfahren mit speziellen Voraussetzungen

A: Nur Schlüsselvergleiche
A: Reihenfolge von Daten ändern
A: Kann Reihenfolge von Daten mit gleichem Sortierschlüssel geändert werden? (Stabilität)
S: Wird eine bestimmte Schlüsselstruktur vorausgesetzt?

Untere Komplexitätsschranke für Sortierverfahren (1)

Satz. *Jeder Sortieralgorithmus für n Elemente, der ausschließlich auf Schlüsselvergleichen und Elementvertauschungen beruht, hat eine worst-case Laufzeit* $\Omega(n \log n)$.

- Am Ende des Sortiervorgangs muss eine von *n!* Permutationen der *n* Elemente erzeugt worden sein.
- Jeder Ausgang eines Vergleiches liefert Informationen um die in Frage kommende Menge von Permutationen in zwei Teile zu zerlegen.



Untere Komplexitätsschranke für Sortierverfahren (2)

- Der entstehende Entscheidungsbaum mit *n!* Blättern ist im besten Fall balanciert.
- Die Höhe des Baumes ist also ≥ ⌈log n!⌉

Mit der Stirlingschen Formel

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

folgt dann der Satz.

Rahmen für Sortierverfahren

- Sortierverfahren sollten immer so geschrieben sein, dass sie mit beliebigen Datentypen operieren können.
- Das bedingt einen gewissen Zusatzaufwand.
- Für die Darstellung der Verfahren wird im Folgenden der Einfachheit halber ein Array von Zeichen (char) sortiert.
- Wir verwenden ein simples Rahmenprogramm.

```
class CharSort {
    public static void main (String args[]) {
        char [] a = " thequickbrownfoxjumpsoverthelazydog".toCharArray();
        System.out.println ("Zeichen-Sortierprogramme\n");
        System.out.println (a);
        System.out.print   ("bubbleSort: ");
        Sorter.bubbleSort  (a);
        System.out.println (a);
    } // main
} // CharSort
```

Die Sorter-Klasse

Die Sorter-Klasse vereint verschiedene Sortier-Algorithmen sowie Hilfs-Methoden:

```
class Sorter {
    private final static void swap (char[] a, int i, int k){
        char h = a[i];
        a[i] = a[k];
        a[k] = h;
    } // swap
    // verschiedene Sortier-Verfahren ...
} // Sorter
```

Die Ausgabe eines Sortierlaufs sieht dann etwa so aus:

```
C:\algo>java CharSort
Zeichen-Sortierprogramme

thequickbrownfoxjumpsoverthelazydog
bubbleSort: abcdeeefghhijklmnoooopqrrsttuuvwxyz
```

Bubble Sort

Wiederholung: Das Blasen-Aufstiegs-Verfahren

```
static void bubbleSort (char[] a){
    int hi = a.length-1;
    for (int k = hi; k > 0; k--){
        boolean test = true;
        for (int i = 0; i < k; i++)
            if (a[i] > a[i+1]) {
                swap (a, i, i+1);
                test = false;
            }
        if (test) break;
    }
} // bubbleSort
```

- Beginnend mit dem höchsten Index wird das Array sortiert.
- Vorsortierung wird ausgenutzt.
- Laufzeiten in $O(n^2)$ im worst und average case sowie $O(n)$ im best case.

Selection Sort (1)

- Suche das kleinste der verbleibenden Elemente und hänge es an den bisher sortierten Bereich (durch Tausch) an.
- Am Ende ist alles sortiert.

```
static void selectionSort (char[] a){
    int hi = a.length-1;
    for (int k = 0; k < hi; k++){
        int min = minPos (a, k, hi); // finde minPos im Rest
        if (min != k) swap (a, min, k); // tausche es nach k
    }
} // selectionSort
private static int minPos (char[] a, int lo, int hi){
    int min = lo;
    for (int i = lo+1; i <= hi; i++)
        if (a[i] < a[min]) min = i;
    return min;
} // minPos
```

Selection Sort (2)

- Äußere Schleifendurchläufe: ($n - 1$)
- Bestimmung von minPos (a,k,hi): ($n - k$) Vergleiche.
- Gesamtaufwand: O(n^2)
- Vorteil: Nur ($n - 1$) Vertauschungen, besser als bei Bubble Sort.
- Nachteil: Bei Vorsortierung keine Verbesserung.

Insertion Sort (1)

- Das erste Element des noch unsortierten Bereiches wird genommen und in den schon sortierten an der richtigen Stelle (durch Tausch) eingefügt.
- Die bereits sortierten Elemente oberhalb der Einfügestelle müssen dazu je um eine Position verschoben werden.

```
static void insertionSort (char[] a){
    int hi = a.length-1;
    for (int k = 1; k <= hi; k++)
        if (a[k] < a[k-1]){
            char x = a[k];
            int i;
            for ( i = k; ( (i > 0) && (a[i-1] > x) ) ; i-- )
                a[i] = a[i-1];    // rechts schieben
            a[i] = x;             // einfuegen
        }
} // insertionSort
```

Insertion Sort (2)

- Aufwand wieder in $O(n^2)$ im Mittel und im worst case.
- Es werden viele Vertauschungen ausgeführt.
- Vorsortierung ist von Vorteil.

Shell Sort (1)

- Shell Sort kann als Variante von Insertion Sort angesehen werden (Donald Shell).
- Es werden mehrere Durchgänge gemacht, deren letzter mit Insertion Sort übereinstimmt.
- Frühere Durchgänge erhöhen die Vorsortierung, so dass spätere Durchgänge weniger zu tun haben.
- Insgesamt kann der Aufwand gegenüber Insertion Sort verbessert werden, auch in Größenordnungen. Er liegt unter $O(n^{1.5})$, experimentell belegt in $O(n^{1.25})$.
- Es wird eine (mit h statt 1) parametrisierte Version von Insertion Sort benutzt.

Shell Sort: Beispiel

$h_3 = 5,\ h_2 = 3,\ h_1 = 1$

Eingabefolge:	d	a	f	e	b	c	g
$h_3 = 5$	d	a	f	e	b	c	g
	c	a	f	e	b	d	g
$h_2 = 3$	c	a	f	e	b	d	g
	c	a	d	e	b	f	g
$h_1 = 1$	c	a	d	e	b	f	g
	a	b	c	d	e	f	g

Shell Sort (2)

```
private static void insertionSorth (char[] a, int h){
    int i, hi = a.length-1;
    for (int k = h; k <= hi; k++)
        if (a[k] < a[k-h]){
            char x = a[k];
            for ( i = k; ( (i > (h-1)) && (a[i-h] > x) ) ; i-=h )
                a[i] = a[i-h];
            a[i] = x;
        }
} // insertionSorth
```

Shell Sort (3)

Die parametrisierte Version von Insertion Sort wird nun für immer kleiner werdende Werte von h aufgerufen:

h = . . . , 1093, 364, 121, 40, 13, 4, 1

```
static void shellSort (char[] a){
    int hi = a.length-1;
    int hmax, h;
    for (hmax = 1; hmax < hi ; hmax = 3*hmax+1 );
        // 1, 4, 13, 40, 121, 364, 1093 ....., 3*h+1
    for ( h = hmax/3; h > 0; h /= 3)
        insertionSorth (a, h);
} // shellSort
```

- Der erste Wert von *h* muss kleiner sein als *n*
- Experimentell hat man noch bessere Folgen für *h* ermittelt, etwa:
  16001, 3905, 2161, 929, 505, 209, 109, 41, 19, 5, 1