

Einführung in die Informatik 2

– Sortieren –

Sven Kosub

AG Algorithmik/Theorie komplexer Systeme
Universität Konstanz

E 202 | Sven.Kosub@uni-konstanz.de | Sprechstunde: nach Vereinbarung

Sommersemester 2010

Einleitung und Problemstellung

Aufgabe: Sortiere Folge F von Elementen aufsteigend (bzw. absteigend)

Klassifikation elementarer Sortierverfahren:

- greedy
- divide-and-conquer

Implementierung der Folgen als:

- Reihungen (*array*)
- Listen (*linked list*)

Gütekriterien von Sortieralgorithmen:

- Laufzeit
 - Anzahl der Vergleiche von Folgeelementen
 - Asymptotik
- (zusätzlicher) Speicherplatz
 - Bevorzugung von Strategien „in place“ (bei Reihungen)
 - Platz für Parameter und Variablen (bei Rekursionen)

Aufgabe: Überführe unsortierte Folge F in sortierte Folge S

Implementierung der Folgen F , S als **Listen**:

- **Vorteil:** Elemente von F nach S überführen durch Umketten (kein zusätzlicher Speicher)
- **Nachteil:** kein direkter Zugriff auf Folgeelemente

Implementierung der Folgen F , S als **Reihungen**:

- **Vorteil:** direkter Zugriff auf Folgeelemente
- **Nachteil:** Schrumpfen von F und Wachsen von S zunächst zeit- und speicherplatzaufwändig

Selection Sort: Sortieren durch Auswahl

- Finde kleinstes Element a von F
- Füge a am Ende von S ein

Insertion Sort: Sortieren durch Einfügen

- Nehme erstes Element a aus F heraus
- Füge a an die richtige Position in S ein

Quicksort

- Teile F in Teilfolgen F_1 und F_2 , wobei Elemente von F_1 kleiner sind als Elemente von F_2
- Sortiere Teilfolgen F_1 und F_2 rekursiv
- Füge sortierte Teilfolgen zusammen

Mergesort: Sortieren durch Mischen

- Teile F in Teilfolgen F_1 und F_2
- Sortiere Teilfolgen F_1 und F_2 rekursiv
- Füge sortierte Teilfolgen zusammen durch iteratives Anhängen des kleineren der jeweils ersten Elemente von F_1 und F_2

Generische Implementierung von Sortierverfahren

- Algorithmen „parametrisiert“ durch Vergleichsoperator
- Interface Comparable im Paket java.lang mit Methode compareTo:

$$a.compareTo(b) < 0 \iff a < b$$

$$a.compareTo(b) = 0 \iff a = b$$

$$a.compareTo(b) > 0 \iff a > b$$

```
public class MyInteger implements Comparable<MyInteger> {
    int data;
    public int compareTo(MyInteger obj){
        if (this.data<obj.data) return(-1);
        if (this.data==obj.data) return(0);
        return(1);
    }
}
```

Generische Implementierung von Sortierverfahren

- Algorithmen „parametrisiert“ durch Vergleichsoperator
- Interface Comparable im Paket java.lang mit Methode compareTo:

`a.compareTo(b) < 0` \iff $a < b$

`a.compareTo(b) = 0` \iff $a = b$

`a.compareTo(b) > 0` \iff $a > b$

```
public class MyInteger implements Comparable<MyInteger> {
    int data;
    public int compareTo(MyInteger obj){
        return(this.data-obj.data);
    }
}
```

Selection Sort

Idee: Solange F nicht leer ist, finde kleinstes Element a von F und füge a am Ende von S ein

Sortiere $F = (21, 5, 3, 1, 17)$ aufsteigend:

$F_0 = (21, 5, 3, 1, 17)$	$S_0 = ()$
$F_1 = (21, 5, 3, 17)$	$S_1 = (1)$
$F_1 = (21, 5, 3, 17)$	$S_1 = (1)$
$F_2 = (21, 5, 17)$	$S_2 = (1, 3)$
$F_2 = (21, 5, 17)$	$S_2 = (1, 3)$
$F_3 = (21, 17)$	$S_3 = (1, 3, 5)$
$F_3 = (21, 17)$	$S_3 = (1, 3, 5)$
$F_4 = (21)$	$S_4 = (1, 3, 5, 17)$
$F_4 = (21)$	$S_4 = (1, 3, 5, 17)$
$F_5 = ()$	$S_5 = (1, 3, 5, 17, 21)$

Selection Sort: Implementierung als Liste (in Java)

```
public class OrderedList {
    // Verwende: public class Node {Object data; Node next; }
    private Node head;
    // Implementiere findMin(): Rückgabe ist Element mit kleinstem Wert
    public Node findMin() { ... };
    // Implementiere deleteElem: Entferne Element a aus Liste
    public void deleteElem(Node a) { ... };
    // Implementiere insertLast: Hänge Element a an Liste an
    public void insertLast(Node a) { ... };
    public OrderedList selectionSort() {
        Node min;
        OrderedList newList = new OrderedList();
        while (head != null) {
            min = findMin();
            deleteElem(min);
            newList.insertLast(min);
        }
        return(newList);
    }
}
```

Selection Sort: Implementierung als Liste (Beurteilung)

Anzahl der Vergleiche:

- findMin benötigt $i - 1$ Vergleiche bei i Elementen in der Liste
- deleteElem und insertLast benötigen keine Vergleiche
- while-Schleife wird n -mal durchlaufen bei anfänglich n Elementen in der Liste
- insgesamt:

$$\sum_{i=1}^n (i - 1) = \frac{n(n + 1)}{2} - n = \frac{n(n - 1)}{2} = O(n^2)$$

Selection Sort: Implementierung als Reihung (in Java)

```
public class OrderedArray {
    Comparable[] data;
    // Implementiere findMin: Rückgabe ist Index des kleinsten Elementes
    public int findMin(int left, int right) { ... };
    // Implementiere swap: Vertausche zwei Elemente des Arrays
    public void swap(int a, int b) { ... };

    public void selectionSort() {
        int min;
        for (int next=0; next < data.length-1; next++) {
            min = findMin(next,data.length-1);
            swap(next,min);
        }
    }
}
```

Selection Sort: Implementierung als Reihung (Beurteilung)

Anzahl der Vergleiche:

- findMin benötigt $i - 1$ Vergleiche bei i Elementen in der Reihung
- swap benötigt keine Vergleiche
- for-Schleife wird n -mal durchlaufen bei n Elementen in der Reihung
- insgesamt:

$$\sum_{i=1}^n (i - 1) = O(n^2)$$

Beachte: Sortierung erfolgt *in place*

Insertion Sort

Idee: Solange F nicht leer ist, nehme erstes Element aus F und füge es an die richtige Position in S ein

Sortiere $F = (21, 5, 3, 1, 17)$ aufsteigend:

$F_0 = (21, 5, 3, 1, 17)$	$S_0 = ()$
$F_1 = (5, 3, 1, 17)$	$S_1 = (21)$
$F_1 = (5, 3, 1, 17)$	$S_1 = (21)$
$F_2 = (3, 1, 17)$	$S_2 = (5, 21)$
$F_2 = (3, 1, 17)$	$S_2 = (5, 21)$
$F_3 = (1, 17)$	$S_3 = (3, 5, 21)$
$F_3 = (1, 17)$	$S_3 = (3, 5, 21)$
$F_4 = (17)$	$S_4 = (1, 3, 5, 21)$
$F_4 = (17)$	$S_4 = (1, 3, 5, 21)$
$F_5 = ()$	$S_5 = (1, 3, 5, 17, 21)$

Insertion Sort: Implementierung als Liste (in Java)

```
public class OrderedList {
    // Verwende: public class Node {Object data; Node next; }
    private Node head;
    // Implementiere takeFirst: Nehme erstes Element aus F heraus
    public Node takeFirst() { ... };
    // Implementiere insertSorted: Füge Element a an richtiger Position ein
    public void insertSorted(Node a) { ... };
    public OrderedList insertionSort() {
        Node first;
        OrderedList resList = new OrderedList();
        while (head != null) {
            first = takeFirst();
            resList.insertSorted(first);
        }
        return(resList);
    }
}
```

Insertion Sort: Implementierung als Liste (Beurteilung)

Anzahl der Vergleiche:

- takeFirst benötigt keine Vergleiche
- insertSorted benötigt maximal i Vergleiche bei i Elementen in der Liste vor dem Einfügen
- while-Schleife wird n -mal durchlaufen bei anfänglich n Elementen in der Liste
- insgesamt:

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Insertion Sort: Implementierung als Reihung (in Java)

```
public class OrderedArray {
    Comparable[] data;
    // Implementiere insertSorted: Füge Element richtig in sortiertem
    // Array-Bereich ein und verschiebe Bereich rechts von der
    // Einfügeposition um eine Position nach rechts
    public void insertSorted(Comparable data, int left, int right)
        { ... };

    public void insertionSort() {
        for (int first=1; first < data.length; first++) {
            insertSorted(data[first],0,first-1);
        }
    }
}
```


Insertion Sort: Implementierung als Reihung (Beurteilung)

Anzahl der Vergleiche:

- insertSorted benötigt maximal $1 + \log i$ Vergleiche für binäre Suche bei i Elementen im Array-Bereich vor dem Einfügen
- for-Schleife wird $(n - 1)$ -mal durchlaufen bei n Elementen in der Reihung
- insgesamt:

$$\sum_{i=1}^{n-1} (1 + \log i) = O(n \log n)$$

Vorsicht: Anzahl der Vergleiche nur halbe Wahrheit

- insertSorted benötigt zusätzlich bis zu n Rechtsverschiebungen
- **Damit:** $O(n^2)$ Zugriffe auf Feldelemente, d.h. quadratische Laufzeit

Beachte: Sortierung erfolgt *in place*

Divide-and-Conquer-Ansätze beim Sortieren

Divide-and-Conquer-Prinzip:

- Teile das Problem (*divide*)
- Löse die Teilprobleme (*conquer*)
- Kombiniere die Lösungen der Teilprobleme (*join*)

Ausprägungen beim Sortieren:

- **Hard split/easy join:**
 - gesamter Aufwand beim Teilen des Problems F in F_1 und F_2
 - Kombination trivial, d.h., S ist Konkatenation von S_1 und S_2
 - Prinzip von Quicksort
- **Easy split/hard join:**
 - Teilen des Problems F in F_1 und F_2 trivial
 - gesamter Aufwand beim Kombinieren von S_1 und S_2 zu S
 - Prinzip von Mergesort

1961 von Sir Charles Anthony Richard Hoare veröffentlicht:

- C. A. R. Hoare: Algorithm 63: partition. *Communications of the ACM*, **4**(7):321, 1961.
- C. A. R. Hoare: Algorithm 63: Quicksort. *Communications of the ACM*, **4**(7):321, 1961.

Prinzip:

- Wähle und entferne **Pivotelement** p aus F
- Zerlege F in F_1 und F_2 mit $a < p \leq b$ für alle $a \in F_1$ und $b \in F_2$
- Sortiere rekursiv F_1 zu S_1 und F_2 zu S_2
- Kombiniere Gesamtlösung als $S_1 p S_2$

Realisierung des Aufteilungsschrittes (**partition**):

- Für Zerlegung in F_1 und F_2 ist p mit jedem Element von $F \setminus \{p\}$ zu vergleichen
- Zerlegung durch Vertauschung von Elementen *in place*
- Wähle rechtestes Element von F als Pivotelement p
- Bestimme kleinstes i mit $F[i] \geq p$
- Bestimme größtes j mit $F[j] < p$ (sonst setze $j = -1$)
- Paar $F[i]$ und $F[j]$ steht „falsch“; vertausche $F[i]$ mit $F[j]$
- Terminierung (ohne Vertauschung!), wenn erstmals $i > j$ gilt, sonst Wiederholung der letzten drei Schritte
- Vertausche p mit linkem Rand von F_2 , d.h. mit $F[i]$

Teile Folge $F = (9, 3, 6, 7, 2, 8, 1, 5)$ auf:

$(9, 3, 6, 7, 2, 8, 1, 5)$

$(9, 3, 6, 7, 2, 8, 1, 5)$

$(1, 3, 6, 7, 2, 8, 9, 5)$

$(1, 3, 6, 7, 2, 8, 9, 5)$

$(1, 3, 2, 7, 6, 8, 9, 5)$

$(1, 3, 2, 7, 6, 8, 9, 5)$

$(1, 3, 2, 5, 6, 8, 9, 7)$

Damit sind die Teilfolgen $F_1 = (1, 3, 2)$ und $F_2 = (6, 8, 9, 7)$.

Quicksort: Implementierung in Java

```
public class OrderedArray {
    Comparable[] data;
    public void quickSort(int left, int right) {
        int i=left; j=right-1;
        // Rekursionsabbruch
        if (left>=right) return;
        // Aufteilungsschritt
        Comparable pivot = data[right];
        while (data[i].compareTo(pivot) < 0) i++;
        while (j>left && data[j].compareTo(pivot) >= 0) j--;
        while (i<j) {
            swap(i,j);
            while (data[i].compareTo(pivot) < 0) i++;
            while (j>left && data[j].compareTo(pivot) >= 0) j--;
        }
        swap(i,right);
        // Rekursionsaufrufe
        quickSort(left,i-1);
        quickSort(i+1,right);
    }
}
```

Quicksort: Beurteilung

Anzahl der Vergleiche:

- Aufteilungsschritt für k Elemente benötigt $k - 1$ Vergleiche
- im **schlechtesten** Fall ist Pivotelement immer Maximum oder Minimum, damit n Rekursionsebenen:

$$\sum_{k=1}^n (n - k) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = O(n^2)$$

- im **besten Fall** wird Reihung genau in der Mitte geteilt (Pivotelement ist der Median), dann gilt für $n = 2^k$:

$$\begin{aligned} T_{\text{best}}(n) &= T_{\text{best}}(2^k) \leq n + 2 \cdot T_{\text{best}}(2^{k-1}) \\ &\leq n + 2 \cdot \left(\frac{n}{2} + 2 \cdot T_{\text{best}}(2^{k-2}) \right) = 2n + 4 \cdot T_{\text{best}}(2^{k-2}) \\ &\leq \dots \leq k \cdot n + 2^k \cdot T_{\text{best}}(2^0) = k \cdot n + n \cdot c \\ &= n \cdot \log n + c \cdot n = O(n \cdot \log n) \end{aligned}$$

Quicksort: Beurteilung

- im **Mittel** benötigt Quicksort $O(n \cdot \log n)$ Vergleiche
- praktische Güte des Verfahren hängt ab von Wahl des Pivotelementes: **Median-von-Drei-Quicksort**
- Sortierung erfolgt *in place*
- für Rekursionen $O(n)$ zusätzlicher Speicherplatz im schlechtesten Fall und $O(\log n)$ Speicherplatz im besten und im mittleren Fall
- in Praxis schneller als viele Sortierverfahren mit Worst-Case-Komplexität $O(n \cdot \log n)$

Quicksort ist einer der besten und meist genutzten Sortieralgorithmen

Mergesort

Prinzip:

- Teile F in Teilfolgen F_1 und F_2
- Sortiere Teilfolgen F_1 und F_2 rekursiv
- Füge sortierte Teilfolgen zusammen durch iteratives Anhängen des kleineren der jeweils ersten Elemente von F_1 und F_2

Sortiere Folge $F = (9, 3, 6, 7, 2, 8, 1, 5)$ aufsteigend:

(9,3,6,7,2,8,1,5)

(9,6,3,7) (2,8,1,5)

(9,6) (3,7) (2,8) (1,5)

(9) (6) (3) (7) (2) (8) (1) (5)

(6,9) (3,7) (2,8) (1,5)

(3,6,7,9) (1,2,5,8)

(1,2,3,5,6,7,8,9)

Mergesort: Implementierung in Java

```
public class OrderedList {
    // Verwende: public class Node {Comparable data; Node next; }
    Node head;
    int length;
    public void mergeSort() {
        OrderedList aList=new OrderedList();
        OrderedList bList=new OrderedList();
        Node aChain, bChain, tmp;
        if ( (head==null) || (head.next==null) || length==1 ) return;
        aChain=head;
        tmp=head;
        for (int i=0; i<length/2; i++) tmp=tmp.next;
        bChain=tmp.next;
        aList.head=aChain; aList.length=length/2;
        bList.head=bChain; bList.length=length-aList.length;
        aList.mergeSort();
        bList.mergeSort();
        merge(aList,bList);
    }
    :
}
```

Mergesort: Implementierung in Java

```
:
private void merge(OrderedList a, OrderedList b) {
    Node tmp;
    if (a.head==null) { head=b.head; length=b.length; return; }
    if (b.head==null) { head=a.head; length=a.length; return; }
    if (a.head.data.compareTo(b.head.data)>0)
        { head = b.head; b.head = b.head.next; }
    else { head=a.head; a.head=a.head.next }
    tmp=head;
    while (a.head != null && b.head != null) {
        if (a.head.data.compareTo(b.head.data)>0)
            { tmp.next = b.head; b.head = b.head.next; }
        else { tmp.next=a.head; a.head=a.head.next }
        tmp=tmp.next;
    }
    if (a.head==null) tmp.next = b.head;
    else tmp.next=a.head;
    length=a.length+b.length;
}
}
```

Mergesort: Beurteilung

Anzahl der Listenzugriffe (zur Vereinfachung $n = 2^k$ Elemente in Liste):

- Halbieren der Liste benötigt $n/2$ Listenzugriffe
- Mischen zweier Listen der Größen n und m benötigt maximal $n + m - 1$ Vergleiche und damit $O(n)$ Listenzugriffe auf jeder Rekursionsebene (unabhängig von Listenzahl)
- auf jeder Rekursionsebene verdoppelt sich die Anzahl der Listen, damit k Rekursionsebenen
- **insgesamt:** $O(n \cdot \log n)$ Listenzugriffe im schlechtesten Fall

Beachte:

- $O(\log n)$ zusätzlicher Speicher für Rekursionen notwendig (besser: **iterativer Mergesort**)
- $O(n)$ zusätzlicher Speicher für gekapselte Zwischenlisten (umgehbar)