Sortieralgorithmen

Greedy Sortieren:
Sortieren durch Auswahl, Einfügen und Austauschen
Divide-and-Conquer-Sortieren:
Quicksort und merge sort

Einleitung und Problemstellung

- Aufgabe: Sortiere Folge *F* aufsteigend (bzw. absteigend)
- Klassifikation elementarer Sortierverfahren
  - greedy
  - divide-and-conquer
- Implementierungen basieren auf Folgen als
  - Reihungen (*arrays*)
  - Listen (*linked lists*)
- Wichtige Gesichtspunkte:
  - Laufzeitkomplexität (asymptotisch)
  - Speicherplatz:
    - Bei Reihungen Operationen „in place“ bevorzugt
    - Bei jeder Rekursion Platz für Parameter und Variablen

Reihungen vs. Listen

- Aufgabe: Überführe unsortierte Folge *F* in sortierte Folge *S*.
- Implementierung von *F*, *S* als Listen
  - Vorteil: Elemente von *F* nach *S* bewegen durch Umketten (kein zusätzlicher Speicher)
  - Nachteil: kein wahlfreier Zugriff (random access)
- Implementierung von *F*, *S* als Reihung (array)
  - Vorteil: wahlfreier Zugriff
  - Nachteil: Schrumpfen von *F* und Wachsen von *S* zunächst nicht trivial, bei naiver Lösung zeit- und speicheraufwändig

Grundprinzipien elementarer Sortierverfahren: Greedy

- Sortieren durch Auswahl (*selection sort*)
  - Finde in *F* kleinstes Element und füge es an das Ende von *S* an.
- Sortieren durch Einfügen (*insertion sort*)
  - Nehme erstes Element aus *F* und füge es an richtiger Stelle in *S* ein.
- Sortieren durch Austauschen (*bubble sort*)
  - Gehe von links nach rechts durch *F* und vertausche benachbarte Elemente, falls falsch geordnet. Wiederhole den Schritt, bis Folge fertig sortiert.

Grundprinzipien elementarer Sortierverfahren: divide-and-conquer

- Quicksort
  - Teile F in Teilfolgen F_1, F_2, wobei Elemente in F_1 kleiner als Elemente in F_2 sind.
  - Sortiere F_1, F_2 rekursiv
  - Füge sortierte Teilfolgen zusammen.
- Sortieren durch Mischen (*merge sort*)
  - Teile F in Teilfolgen F_1, F_2
  - Sortiere F_1, F_2 rekursiv
  - Bilde S durch iteratives Anhängen des kleineren der jeweils ersten Elemente aus F_1, F_2

Einordnung elementarer Sortierverfahren

	Implementierung array	Implementierung list
greedy	selection sort bubble sort	insertion sort
divide-and-conquer	quicksort	merge sort

Generische Implementierung von Sortierverfahren

- Algorithmen „parametrisiert“ durch Vergleichsoperator
- Interface Comparable im Paket java.lang
  - Aufruf Vergleichsmethode a.compareTo(b) liefert Zahl <0, = 0, >0 für a<b, a=b, a>b
- Muster für Objekte vom Referenztyp Comparable

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


Generische Implementierung von Sortierverfahren

- Muster für Aufruf in Klassenmethode (hier bei Suchverfahren):

```
public static int binarySearch(
                                Comparable[] f,
                                Comparable a,
                                int l,
                                int r) {
  int p = (l+r)/2;
  int c = f[p].compareTo(a);
  ...
```

Selection sort: Sortieren durch Auswahl

- Prinzip: Solange es Elemente in der Folge *F* gibt, finde in *F* das kleinste Element und überführe es ans Ende der Folge *S*.

SelectSort *(F)*
// Die unsortierte Folge *F* wird in die sortierte Folge *S* überführt.
1. **Initialisiere:** S = ∅.
2. **Trivialfall:** if (*F* == ∅) return (*S*)
3. **Reduktion:** Minimum aus F nach S überführen.
   a = select *(F)*
   S = appendElem (*S*,a);
   F = deleteElem *(F*,a*);*
4. **Iteriere:** Weiter bei 2.

- Je nachdem, ob man mit select ein minimales oder maximales Element auswählt, erhält man eine schwach aufsteigend oder schwach absteigend sortierte Folge *S*.

Selection sort: Beispiel

- Sortiere $F_0 = (21, 5, 3, 1, 17)$
  - schwach aufsteigend
  - wähle jeweils Minimum in *F*

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

Selection sort: Listenimplementierung

- public class MyObject implements Comparable {. . .}
- public class Node {Object data; Node next;}
- public class OrderedList {
  private Node head;
  public OrderedList sort ( ) {. . .}

Interne Hilfsmethoden

- int findMin () {. . .}
  - F.findMin() bestimmt Index des minimalen Elements von OrderedList F
- void insertLast (int a)
  - S.insertLast(a) fügt Element mit Index a an das Ende von S an
- void deleteElem (int a)
  - F.deleteElem(a) löscht Element mit Index a aus Liste F
- Aufwand jeweils = *O(n),* wenn *n* = Anzahl der Objekte in Liste

Selection sort: Beispielimplementierung

```
public class OrderedList {
// ...
  /**
   * Gibt Liste zurück, die die Elemente von this
   * in nicht absteigend sortierter Reihenfolge enthält.
   * Methode ist als selection sort implementiert.
   */
public OrderedList selectSort() {
    int min;
    OrderedList newList = new OrderedList();

    while( head != null ) // Liste ist nicht leer
    {
      min = findMin();
      newList.insertLast(min);
      deleteElem(min);
    }
    return (newList);
  }
}
```

- Komplexität: n Iterationen der while-Schleife mit je $O(n)$ Aufwand ergibt insgesamt $O(n^2)$

Selection sort: Array Implementierung I

- Reihung data[0], ..., data[n-1] partitioniert in
  - sortierter Anfang S: data[0], ..., data[next-1]
  - unsortierter Rest F: data[next], ..., data[n-1]
- Finde Index min des minimalen Elementes in F
- Vertausche data[next] mit data[min]
- Inkrementiere next und iteriere solange next<n
- S wächst nach rechts, F schmilzt von links ab.

Selection sort: array Implementierung II

```
public class OrderedArray {
// ...
  Comparable[] data;
  void selectSort() {
  int min; // Initialize
  // Reduce F and grow S
  for (int next = 0;
       next < data.length-1;
       next++) {
    min = findMin(next,data.length-1);
    swap(next,min);
  }
 }
}
```

- findMin(i,j) berechnet Index des minimalen Elementes in data[i], ..., data[j]
- swap(i,j) vertauscht data[i] mit data[j]

Insertion sort: Sortieren durch Einfügen

- Nehme jeweils das erste Element aus F und füge es in S an der richtigen Stelle ein.

  InsertSort(F)
  // Die unsortierte Folge F wird in die sortierte Folge S überführt.

  1. **Initialisiere:** $S = \varnothing$
  2. **Trivialfall:** if ($F == \varnothing$) return(S).
  3. **Reduziere** F : ein Element aus F sortiert nach S überführen.
     a = takeFirst(F);
     S = insertSorted(S,a);
  4. **Iteriere:** Weiter bei 2.

- Zur Implementierung:
  - F.takeFirst() bestimmt das erste Element aus F und löscht es anschließend aus F,
  - S.insertSorted(a) fügt Element a in S an der richtigen Stelle ein.

Insertion sort: Beispiel

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

Insertion sort: listenbasiert

```
public class OrderedList {
  // ...
  /**
   * Gibt Liste zurück, die die Elemente von this
   * in nicht absteigend sortierter Reihenfolge enthält.
   * Methode ist als insertion sort implementiert.
   */
  public OrderedList insertSort() {
    Comparable first;
    OrderedList resList;

    while( head!=null ) // Liste ist nicht leer
      {
       first = takeFirst();
       resList.insertSorted(first);
      }
    return resList;
  }
}
```

- Komplexität: $O(n^2)$

Insertion sort: arraybasiert

- Reihung data[0], ..., [n-1] partitioniert in
  - sortierter Anfang *S:* data[0], ..., data[next-1]
  - unsortierter Rest *F:* data[next], ..., data[n-1]
- Wähle Element data[next] zum Einfügen
- Suche Einfügestelle absteigend von next-1 bis 0
- Schiebe dabei jedes zu große Element eine Position nach hinten bis Einfügestelle gefunden

bubble sort: Sortieren durch Austauschen

- Geht man von links nach rechts durch eine Folge *F* und vertauscht alle benachbarten Elemente, welche falsch geordnet sind, so steht am Schluss das größte Element von *F* am Ende der Folge.
- Das größte Element steigt dabei in der Folge wie eine Blase nach oben (*bubble sort*).
- Diesen Durchgang wiederholt man mit dem Rest der Folge *F*, bis die ganze Folge geordnet ist.
- *S* bildet sich also rechts von *F* und dehnt sich nach links aus.

bubble sort: Beispiel

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

Bubble sort: Arrayimplementierung

```
public class OrderedArray {
  Comparable[] data; // array of data
  // ...
  /**
   * Swaps the contents of data[i] and data[j].
   */
  void swap(int i, int j) {
    Comparable tmp = data[i];
    data[i]=data[j];
    data[j]=tmp;
  }
  /**
   * Sorts the array data in ascending order.
   */
  public void bubbleSort() {
    for(int j = data.length-1; j > 0; j--)
      for( int i = 1; i <= j; i++)
        // compare element at position j in sorted list
        // move greatest up
        if( data[i-1].compareTo(data[i]) > 0 )
          swap(i-1,i);
  }
}
```

- Komplexität: $O(n^2)$

divide-and-conquer: Prinzip

Das abstrakte Divide-and-Conquer-Prinzip sieht folgendermaßen aus:

1. Teile das Problem (*divide*).
2. Löse die Teilprobleme (*conquer*).
3. Kombiniere die Teillösungen (*join*).

Beim Sortieren gibt es zwei Ausprägungen:

Hard split / easy join: Dabei wird die gesamte Arbeit beim Teilen des Problems verrichtet und die Kombination ist trivial, das heißt, F wird so in F_1 und F_2 partitioniert, daß $S = S_1 S_2$. Dieses Prinzip führt zum *Quicksort*-Algorithmus.
Easy split / hard join: Dabei ist die Aufteilung $F = F_1 F_2$ trivial und die ganze Arbeit liegt beim Zusammensetzen von S_1 und S_2 zu S. Dieses Prinzip führt zum *Mergesort*-Algorithmus.

quicksort: Allgemeines

- Einer der besten und meist genutzten Sortieralgorithmen
- Hauptvorteile:
  - sortiert Array in place
  - *O(n log n)* Komplexität im Mittel
  - *O(log n)* zusätzlicher Speicherplatz auf Stack
  - in Praxis schneller als andere Verfahren mit derselben asymptotischen Komplexität *O(n log n)*

quick sort: Prinzip

- Wähle und entferne Pivotelement p aus F
- Zerlege F in Teilfolgen F_1, F_2 so dass
  - $e_1 < p \leq e_2$ für alle e_1 in F_1, e_2 in F_2
  - Sortiere (rekursiv) F_1 zu S_1 und F_2 zu S_2
- Gesamtlösung ist $S_1\, p\, S_2$

quicksort: Zerlegung in Teilfolgen

1. Für Zerlegung in F_1, F_2 ist p mit jedem Element von $F \setminus \{p\}$ zu vergleichen
2. Zerlegung soll durch Vertauschen von Elementen geschehen (kein Zusatzspeicher, in place!)
3. Wähle p als rechtes Randelement von F
4. Suche $i = \arg\min_i F[i] \geq p$, $j = \arg\max_j F[j] < p$
5. Paar $F[i]$, $F[j]$ „steht falsch“, also vertausche $F[i]$ mit $F[j]$
6. Terminierung (ohne Vertauschen!), wenn erstmals $i > j$, sonst gehe zum Suchschritt 4
7. Vertausche p mit linkem Rand von F_2 (also mit $F[i]$)

quicksort: Wahl des Pivotelements

- Algorithmus ist schnell, wenn die Teilfolgen F_1, F_2 etwa gleich groß sind
- Im schlimmsten Fall ist eine Teilfolge immer leer (kann vorkommen, wenn Folge vorsortiert ist)
- Daher wählt man häufig p als Median von drei Stichproben, z. B. erstes, mittleres und letztes Element von F
- (b ist Median von a,b,c, wenn $a \leq b \leq c$)

quicksort: Beispiel

(9, 7, 1, 6, 2, 3, 8, 4)
(9, 7, 1, 6, 2, 3, 8), (4)
(3, 7, 1, 6, 2, 9, 8), (4)
(3, 2, 1, 6, 7, 9, 8), (4)
(3, 2, 1, 4, 7, 9, 8, 6)

Nun ist die 4 am richtigen Platz angelangt. Danach geht es weiter mit dem rekursiven Aufruf auf den beiden Teilfolgen:

$F_1 = (3, 2, 1)$ und $F_2 = (7, 9, 8, 6)$

quick sort: Java Implementierung I/II

```
public class OrderedArray {
  Comparable[] data; // array of data
  // ...

  /**
   * Sorts data[l], data[l+1], ... , data[r]
   * in ascending order using the quicksort algorithm
   * Precondition: 0 <= l <= r < data.length.
   */
  public void quickSort(int l, int r) {
    // 0. Initialisiere
    int i = l, j = r-1;
    // 1. Check auf Trivialfall
    if (l >= r) return;
    // Initialisiere pivot
    Comparable pivot = data[r];
    // 2. Teile: Füge pivot an einem Platz p in F (=data),
    // ein, so dass F[i]<F[p] für l<=i<p und
    // F[j] >= F[p] für p<=j<=r.
```


quicksort: Java Implementierung II

```
    // 2.1 Finde äusserstes ungeordnetes Paar F[i], F[j],
    //     i<j, mit F[i] >= pivot und F[j] < pivot und
    //     F[s] < pivot für l<=s<i und F[s] >= pivot
    //     für j<s<=r.
    while( data[i].compareTo(pivot) < 0 ) i++;
    while( j>=l && data[j].compareTo(pivot) >= 0) j--;

    // 2.2 Ordne Paar; finde nächstes ungeordnetes Paar.
    while( i < j ) {
      // i<j impliziert j>=l,
      // daher F[j]<pivot und F[i]>=pivot.
      swap(i,j);
      while( data[i].compareTo(pivot) < 0 ) i++;
      while( data[j].compareTo(pivot) >= 0 ) j--;
    }
    // 2.3 Endgültiger Platz für pivot ist i: es gilt i>j
    //     (und nicht nur i>=j, denn i=j impliziert
    //     pivot<=F[i]<pivot) und F[k]<pivot f¨ur 0<=k<i;
    //     F[k]>=pivot für j<k<=r; wegen i>j folgt
    //     F[k]>=pivot für i<=k<=r.
    swap(i,r);

    // 3. Herrsche: Sortiere links und rechts
    //    vom Ausgangspunkt.
    quickSort(l, i-1);
    quickSort(i+1, r);
  }
}
```


quicksort: Komplexität I

- Anzahl der Vergleiche *T(n), n* = Länge der Eingabe
- Auf jeder Rekursionsebene sind *O(n)* Vergleiche von Datenelementen zu machen (genaue Zahl hängt ab von der Anzahl der swap-Operationen)
- Maximale Anzahl von Rekursionsebenen ist *n* (alle Folgenelemente sind kleiner als Pivotelement)
- Im schlimmsten Fall also $O(n^2)$ Vergleiche

quicksort: Komplexität II

- Im besten Fall sind Teilfolgen gleich groß und ungefähre Rechnung für $n = 2^k$ ergibt

$$T_{best}(n) = T_{best}(2^k) = n + 2\ T_{best}(2^{k-1})$$
$$= n + 2 \cdot n/2 + 4 \cdot T_{best}(2^{k-2}) = n + n + 4\ T_{best}(2^{k-2})$$
$$= \cdots = n + n + \cdots + n + 2^k \cdot T_{best}(1)$$
$$= k \cdot n + 2^k \cdot T_{best}(1) = k \cdot n + n \cdot c$$
$$= n \log_2 n + cn \in O(n \log n)$$

- Komplexität im Mittel ebenfalls $O(n \log n)$.
  (Beweis siehe z.B. in Ottmann/Widmayer)

merge sort: Sortieren durch Mischen

- Sehr gut geeignet für Folgen in Listenform
- Für Arrays problematisch ($O(n)$ zusätzlicher Speicherplatz)
- Komplexität auch im schlimmsten Fall $O(n \log n)$
- Bei merge sort wird die Folge F einfach in zwei gleich große Hälften F_1 und F_2 geteilt, $F = F_1 F_2$.
- Teilfolgen F_1 und F_2 werden rekursiv sortiert mit Ergebnis S_1 und S_2.
- Die sortierten Folgen S_1 und S_2 werden dann in einem linearen Durchgang zur sortierten Folge S *gemischt*, indem man das jeweils kleinste (bzw. größte) Element von S_1 und S_2 an S anfügt.

merge sort: Beispiel

(3, 2, 1, 7, 3, 4, 9, 2)
(3, 2, 1, 7),(3, 4, 9, 2)
(3, 2), (1, 7),(3, 4), (9, 2)
(3),(2), (1),(7),(3),(4),(9),(2)
(2, 3), (1, 7),(3, 4) (2, 9)
(1, 2, 3, 7) (2, 3, 4, 9)
(1, 2, 2, 3, 3, 4, 7, 9)

merge sort: Java Implementierung I

```
public class OrderedList {
  OrderedNode head;
  int length;
  // ...

  /**
   * Sorts this list in non-descending order.
   * Works destructively on the encapsulated
   * node chain starting with head.
   * Implemented as merge sort.
   */
  public void mergeSort() {
    OrderedList aList, bList; // the divided lists
    OrderedNode aChain; // start of first node chain
    OrderedNode bChain; // start of second node chain
    OrderedNode tmp; // working node for split

    // trivial cases
    if( (head==null) || // empty list
        (head.next == null) ) // one element list
      return;
```


merge sort: Java Implementierung II

```
    // divide: split the list in two parts
    aChain = head;
    tmp = head; // init working node for split
    // advance half of the list
    for (int i=0; i < (length-1) / 2; i++)
      tmp=tmp.next;

    // cut chain into aChain and bChain
    bChain=tmp.next;
    tmp.next=null;

    // encapsulate the two node chains in two lists
    aList = new OrderedList();
    aList.head=aChain;
    aList.length=length/2;
    bList = new OrderedList();
    bList.head=bChain;
    bList.length=length - alist.length;

    // conquer: recursion
    aList.mergeSort(); bList.mergeSort();
    // join: merge
    merge(aList, bList);
  }
}
```

merge sort: Anmerkungen

Aus Gründen der Übersichtlichkeit erzeugt dieses Programm im Divide-Schritt jeweils gekapselte Zwischenlisten vom Typ OrderedList. In der Praxis würde man hierauf verzichten und rein auf Knoten-Ketten arbeiten, da insgesamt O(n) Objekte vom Typ OrderedList erzeugt und wieder vernichtet werden (maximal $O(\log n)$ davon sind gleichzeitig aktiv).

merge sort: Komplexität

- Zähle nur Vergleiche bei Länge $n = 2^k$
- Split: $n/2$ Schritte (bewege Node tmp in die Mitte)
- Merge: n Vergleiche
- Rekursionsebenen: $\log_2 n$
- Gesamtkomplexität: $O(n \log_2 n)$