

Sortieren

Thomas Röfer

Permutationen
 Naives Sortieren
 Sortieren durch Einfügen, Auswählen, Vertauschen, Mischen
 QuickSort
 Comparator

Rückblick „Suchen“

Identität/Flache/Tiefe Gleichheit

a → [1] → [2] null
 b → [1] → [2] null

Lineare Suche: [1 5 8 9 11]

Binäre Suche: [1 5 8 9 11]

Aufwand

	Zm	Wm	Min	Max
1	2	1	0	0
2	3	3	0	1
3	5	6	1	3
4	8	10	3	6
5	12	15	6	10
6	17	21	10	15
7	23	28	15	21
8	30	36	21	28
9	38	45	28	36
10	47	55	36	45

Asymptotische Komplexität

Typische Aufwandsklassen

- $O(1)$: swap()
- $O(\log n)$: binarySearch()
- $O(n)$: linearSearch()
- $O(n \log n)$: mergeSort()
- $O(n^2)$: selectSort()
- $O(n!)$: naiveSort()

Pl-2: Sortieren

2

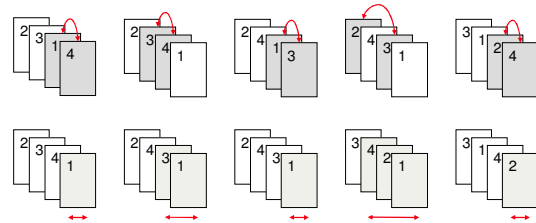
Permutieren nach Dijkstra

- › **Ziel**
 - › Eine Funktion, die aus einer gegebenen Permutation die jeweils nächste erzeugt
 - › Voraussetzung: Es existiert eine Ordnungsrelation zwischen den Elementen der zu permutierenden Folge
- › **Algorithmus**
 - › Folge $a_0 \dots a_{n-1}$
 - › Finde das letzte a_i , das kleiner als a_{i+1} ist
 - › Finde das letzte a_j , das größer als a_i ist
 - › Vertausche die beiden
 - › Drehe die Reihenfolge der Elemente ab a_{i+1} um
- › **Aufwand**
 - › Im Mittel $O(1)$
 - › im Mittel werden nur 1,76 Elemente durchsucht und umgedreht

Pl-2: Sortieren

3

Permutieren nach Dijkstra



Pl-2: Sortieren

4

Permutieren nach Dijkstra

```
class Permutation
{
    static void nextPerm(Comparable[] a)
    {
        int i;
        for(i = a.length - 2; i >= 0 &&
            a[i].compareTo(a[i+1]) >= 0; --i)
            ;
        if(i >= 0)
        {
            int j;
            for(j = a.length - 1;
                a[j].compareTo(a[i]) <= 0; --j)
                ;
            swap(a, i, j);
        }
        reverse(a, i + 1, a.length - 1);
    }
}
```

```
static private void reverse(
    Comparable[] a, int l, int r)
{
    while(l < r)
        swap(a, l++, r--);
}

static void swap(Object[] o,
    int a, int b)
{
    Object temp = o[a];
    o[a] = o[b];
    o[b] = temp;
}
```

Pl-2: Sortieren

5

Sortieren

- › **Definition**
 - › Eine sortierte Folge S ist eine *Permutation* einer Folge T
 - › $S = perm(T)$
 - › S bezeichnet man als *aufsteigend sortiert*, wenn jedes Element größer oder gleich seinem Vorgänger in der Folge ist
 - › $s_j \leq s_{j+1} \leq s_{j+2} \dots \leq s_{n-1}$
 - › S bezeichnet man als *absteigend sortiert*, wenn jedes Element kleiner oder gleich seinem Vorgänger in der Folge ist
 - › $s_j \geq s_{j+1} \geq s_{j+2} \dots \geq s_{n-1}$
- › **Sortierkriterium**
 - › Es wird immer nach einem Sortierkriterium sortiert (Sortierschlüssel)
 - › Der Sortierschlüssel kann sich aus mehreren Teilen zusammensetzen
 - › z.B. Name, Vorname, Telefonnummer
 - › Je komplexer der Sortierschlüssel, desto komplexer die zum Sortieren notwendigen Vergleiche
- › **Indizes**
 - › Zu einer Menge von Daten können mehrere sortierte *Indizes* vorgehalten werden

Pl-2: Sortieren

6

Naives Sortieren

- Algorithmus**
 - Erzeuge so lange Permutationen, bis eine entsprechend dem Kriterium sortiert ist
- Aufwand**
 - Bester Fall: $O(n)$
 - vorsortiert
 - Schlechtester Fall: $O(\frac{1}{2}n \cdot n!) = O(n \cdot n!)$
 - erst letzte Permutation ist sortiert
 - Nicht-Sortiertheit wird im Mittel nach der Hälfte der Vergleiche festgestellt
 - Durchschnittlicher Fall: $O(\frac{1}{2}n \cdot n! / 2) = O(n \cdot n!)$
 - Sortierte Folge wird nach der Hälfte der Permutationen gefunden
 - Nicht-Sortiertheit wird im Mittel nach der Hälfte der Vergleiche festgestellt

```

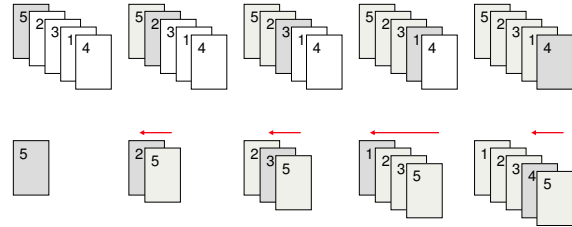
static void naiveSort(
    Comparable[] a)
{
    while (!sorted(a))
        Permutation.nextPerm(a);
}

static private boolean sorted(
    Comparable[] a)
{
    for (int i = 1; i < a.length;
        ++i)
        if (a[i].compareTo(a[i-1]) < 0)
            return false;
    return true;
}
    
```

PI-2: Sortieren

7

Sortieren durch Einfügen



PI-2: Sortieren

8

Sortieren durch Einfügen

- Algorithmus**
 - Füge alle Elemente der Reihe nach in eine ursprünglich leere Folge sortiert ein
 - Problem bei Reihungen: Einfügen heißt, dass alle Elemente hinter der Einfügeposition verschoben werden müssen
- Aufwand**
 - Doppelter Speicherplatzverbrauch
 - Bester Fall: $O(n)$
 - vorsortiert
 - Schlechtester Fall: $O(n^2)$
 - rückwärts sortiert
 - Durchschnittlicher Fall: $O(\frac{1}{2}n^2) = O(n^2)$
 - Einsortieren im Mittel in der Mitte

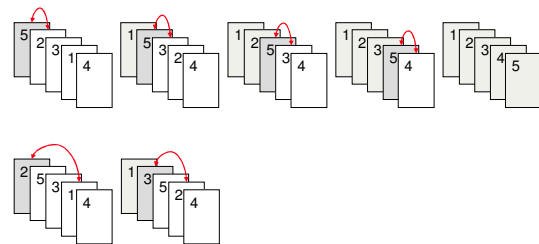
```

static void insertSort(Comparable[] a)
{
    Comparable[] b =
        (Comparable[]) a.clone();
    for (int i = 0; i < b.length; ++i)
    {
        int j = i;
        while (j > 0 &&
            a[j-1].compareTo(b[i]) > 0)
        {
            a[j] = a[j-1];
            --j;
        }
        a[j] = b[i];
    }
}
    
```

PI-2: Sortieren

9

Sortieren durch Auswählen



PI-2: Sortieren

10

Sortieren durch Auswählen

- Algorithmus**
 - Falls nur noch ein Element übrig ist, ist die Folge sortiert.
 - Ansonsten
 - Suche nach dem kleinsten Element und stelle es an den Anfang,
 - sortiere den Rest.
- Aufwand**
 - In jedem Durchgang wird ein Element ausgewählt
 - Um das Element auszuwählen, müssen im Mittel $n/2$ Elemente durchsucht werden
 - Durchschnittlicher Fall: $O(\frac{1}{2}n \cdot n) = O(\frac{1}{2}n^2) = O(n^2)$

```

static void selectSort(Comparable[] a)
{
    for (int i = 0; i < a.length - 1; ++i)
        for (int j = i + 1; j < a.length; ++j)
            if (a[i].compareTo(a[j]) > 0)
                swap(a, i, j);
}

static void swap(Object[] o,
    int a, int b)
{
    Object temp = o[a];
    o[a] = o[b];
    o[b] = temp;
}
    
```

PI-2: Sortieren

11

BubbleSort



PI-2: Sortieren

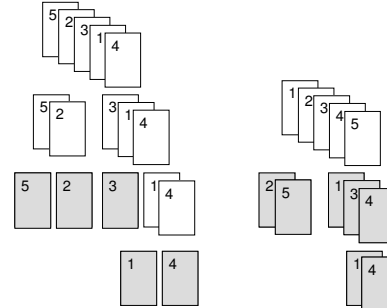
12

BubbleSort

- Algorithmus**
 - Gehe der Reihe nach durch alle Elemente und vertausche zwei jeweils aufeinander folgende, wenn ihre Reihenfolge falsch ist
 - Wiederhole dies so lange, bis keine Vertauschungen mehr notwendig sind
- Aufwand**
 - Günstigster Fall: $O(n)$
 - vorsortiert
 - Schlechtester Fall: $O(n^2)$
 - rückwärts sortiert (maximale Anzahl von Vertauschungen)
 - Durchschnittlicher Fall: $O(n^2)$
 - im Mittel $\frac{1}{2}n^2$ Vertauschungen

```
static void bubbleSort(Comparable[] a)
{
    boolean swapped;
    do
    {
        swapped = false;
        for(int j = 1; j < a.length; ++j)
            if(a[j].compareTo(a[j - 1]) < 0)
            {
                swap(a, j, j - 1);
                swapped = true;
            }
    } while(swapped);
}
```

Sortieren durch Mischen



Sortieren durch Mischen

- Algorithmus**
 - Enthält die Folge kein oder nur ein Element, ist sie sortiert
 - Ansonsten
 - Teile die Folge in zwei gleich große Hälften
 - Sortiere die Hälften
 - Mische die sortierten Hälften wieder zusammen
- Mischen**
 - Vergleiche die beiden ersten Elemente der Hälften
 - Entnimme das kleinere von beiden und füge es dem Ergebnis hinzu
 - Wiederhole so lange, bis beide Hälften leer sind
- Aufwand**
 - Folge kann $\log_2(n)$ mal in Hälften zerlegt werden
 - Auf jeder Ebene der Zerlegung werden alle Elemente einmal gemischt: $O(n)$
 - Insgesamt (in jedem Fall): $O(n \log n)$
 - Platzkomplexität: $2n$ Elemente beim Mischen

Sortieren durch Mischen

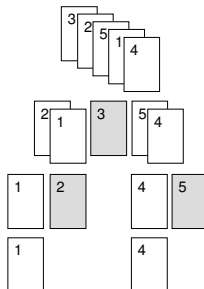
```
class Queue
{
    void push(Object) ...
    Object pop() ...
    Object top() ...
    int length() ...
}

static Queue mergeSort(Queue q)
{
    if(q.length() > 1)
    {
        Queue q2 = new Queue();
        balance(q, q2);
        return merge(mergeSort(q),
                    mergeSort(q2));
    }
    else
        return q;
}
```

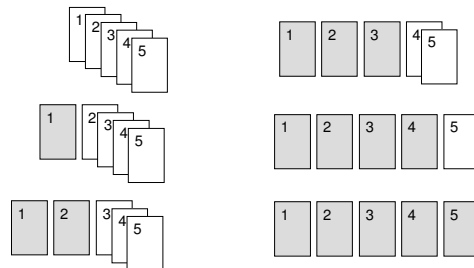
```
static private void balance(Queue q1,
                          Queue q2)
{
    while(q1.length() > q2.length())
        q2.push(q1.pop());
}

static private Queue merge(Queue q1,
                          Queue q2)
{
    Queue q = new Queue();
    while(q1.length() > 0 || q2.length() > 0)
    {
        if(q2.length() == 0 || q1.length() > 0
           && ((Comparable) q1.top())
              .compareTo(q2.top()) < 0)
            q.push(q1.pop());
        else
            q.push(q2.pop());
    }
    return q;
}
```

QuickSort



QuickSort – Ungünstige Vorsortierung



QuickSort

- ▶ **Algorithmus**
 - ▶ Enthält die Folge kein oder nur ein Element, ist sie sortiert
 - ▶ Ansonsten
 - ▶ entnimmt ein Element (das sog. Pivot-Element),
 - ▶ bilde eine neue Liste aus allen kleineren Elementen und lasse diese sortieren,
 - ▶ bilde eine neue Liste aus allen größeren Elementen und lasse diese sortieren,
 - ▶ Hänge die sortierte Liste der kleineren Elemente, das gewählte Element und die sortierte Liste der größeren Elemente aneinander und liefere diese zurück.
- ▶ **Aufwand**
 - ▶ Günstigster Fall: $O(n \log n)$
 - ▶ beide Teillisten sind jeweils gleich lang
 - ▶ Schlechtester Fall: $O(n^2)$
 - ▶ Eine der Teillisten ist immer leer
 - ▶ Wenn als Pivot-Element immer das erste verwendet wird, ist dies bei Vorsortierung der Fall!
 - ▶ Durchschnittlicher Fall: $O(n \log n)$
 - ▶ Beweis siehe Ottmann / Widmayer
 - ▶ Kein zusätzlicher Speicherbedarf bei Sortierung eines Arrays!

QuickSort einer Warteschlange

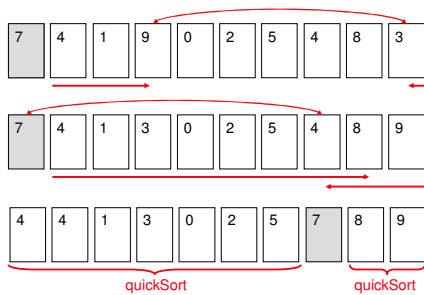
```
static void quickSort(Queue q)
{
    if (q.length() > 0)
    {
        Comparable pivot =
            (Comparable) q.pop();
        Queue q1 = new Queue(),
            q2 = new Queue();
        split(q, pivot, q1, q2);
        quickSort(q1);
        quickSort(q2);
        join(q, pivot, q1, q2);
    }
}

static void split(Queue q, Comparable pivot,
                Queue q1, Queue q2)
{
    while (q.length() > 0)
    {
        if (pivot.compareTo(q.top()) > 0)
            q1.push(q.pop());
        else
            q2.push(q.pop());
    }
}

static void join(Queue q, Object pivot,
                Queue q1, Queue q2)
{
    while (q1.length() > 0)
        q.push(q1.pop());
    q.push(pivot);
    while (q2.length() > 0)
        q.push(q2.pop());
}

```

QuickSort mit Arrays



QuickSort

```
static void quickSort(Comparable[] a)
{
    quickSort2(a, 0, a.length);
}

static private void quickSort2(Comparable[] a, int bottom, int top)
{
    if (bottom + 1 < top)
    {
        swap(a, bottom, (top + bottom) / 2);
        Comparable pivot = a[bottom];
        int i = bottom + 1,
            j = top - 1;
        do
        {
            while (i < top && a[i].compareTo(pivot) < 0) ++i;
            while (j > bottom && a[j].compareTo(pivot) >= 0) --j;
            if (i < j)
                swap(a, i, j);
        } while (i < j);
        swap(a, i - 1, bottom);
        quickSort2(a, bottom, i - 1);
        quickSort2(a, i, top);
    }
}

```

Schnittstelle Comparator

- ▶ **Motivation**
 - ▶ Die Schnittstelle Comparable legt nur genau eine Ordnungsrelation für eine Klasse fest
 - ▶ Die Schnittstelle Comparator erlaubt die Definition beliebiger weiterer Ordnungsrelationen
- ▶ **Generische Schnittstellen**
 - ▶ interface Comparable<T>
 - ▶ interface Comparator<T>
 - ▶ new Comparator<String>()
 - ▶ public int compare(String a, String b)
 - ▶ { return a.compareTo(b); }

```
import java.util.Comparator;

static void selectSort(Object[] a,
                      Comparator c)
{
    for (int i = 0; i < a.length - 1; ++i)
        for (int j = i + 1; j < a.length; ++j)
            if (c.compare(a[i], a[j]) > 0)
                swap(a, i, j);
}

static void selectSort(String[] s)
{
    selectSort(s, new Comparator()
    {
        public int compare(Object a, Object b)
        {
            return ((String) a).compareTo(b);
        }
    });
}

```