# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

### Speicherstelle der Daten

1. **Intern**: Interne Sortieralgorithmen, wenn zu sortierende Daten im Arbeitsspeicher, also z.B. in einem Array.
2. **Extern**: Externe Sortieralgorithmen, wenn zu sortierende Daten z.B. auf Peripheriegerät (Magnetband, Festplatte, ...)
   → Anders als bei internen Sortieralgorithmen (direkter Zugriff auf Elemente) hier nur sequenzielle Zugriffe auf die Elemente (wie z.B. mit Vor- und Rückspulen) mögl.
3. **Index-sequenziell**: Zu sortierende Elemente in Dateien, wobei zu jedem Element Datensatz existiert, in dem nicht nur Daten, sondern zusätzlich Schlüssel (key) sind.
   → Datensätze so sortieren, dass sich die Schlüssel in einer entspr. Reihenfolge befinden.

# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

### Leistungsfähigkeit der Sortieralgorithmen

- **Laufzeit liegt abhängig vom verwendeten Sortieralgorithmus zwischen $n \cdot \log n$ und $n^2$.**
- **Speicherplatz**
  - **Kein zusätzlicher Speicherplatz: Solche Algorithmen sortieren Elemente direkt am Ort und benötigen keinen wesentlichen zusätzlichen Speicherplatz.**
  - **Zusätzl. Speicherplatz von n Zeigern bzw. Referenzen: Solche Algorithmen unterhalten sich n zusätzliche Zeiger auf die einzelnen Elemente (z.B. verkettete Liste).**
  - **Doppelter Speicherplatz: Solche Algorithmen benutzen beim Sortieren Kopien der zu sortierenden Daten.**

# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

### Stabilität bei Sortieralgorithmen

Ein Sortieralgorithmus gilt als **stabil**, wenn er bei Elementen, die nach Sortierkriterium gleich sind, die zuvor vorliegende Reihenfolge der Elemente relativ zueinander beibehält.

→ Hat man z.B. eine alphabetisch sortierte Liste von Personen, so garantiert stabiler Sortieralgorithmus, der diese Liste nach Alter sortiert, dass gleichaltrige Personen danach immer noch alphabetisch geordnet sind.

→ Bei instabilem Sortieralgorithmus nicht gewährleistet.

# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

### Bubble-Sort

```
------ vor bubble_sort ------------------------------
 19 96 30 11 73  2 99 72 69 73
 1. Durchlauf:   2 96 30 19 73 11 99 72 69 73
 2. Durchlauf:   2 11 96 30 73 19 99 72 69 73
 3. Durchlauf:   2 11 19 96 73 30 99 72 69 73
 4. Durchlauf:   2 11 19 30 96 73 99 72 69 73
 5. Durchlauf:   2 11 19 30 69 96 99 73 72 73
 6. Durchlauf:   2 11 19 30 69 72 99 96 73 73
 7. Durchlauf:   2 11 19 30 69 72 73 99 96 73
 8. Durchlauf:   2 11 19 30 69 72 73 73 99 96
 9. Durchlauf:   2 11 19 30 69 72 73 73 96 99
------ nach bubble_sort ---------------------------
  2 11 19 30 69 72 73 73 96 99
```

```
void bubble_sort(int n, int z[]) {
   for (int i=0; i<n-1; i++)
      for (int j=i+1; j<n; j++)
         if (z[i] > z[j]) {
            int t = z[i];    z[i] = z[j];    z[j] = t;
         }
```

# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

### Verbesserter Bubble-Sort

**Verbesserter Bubble-Sort erkennt, dass ein Array bereits sortiert ist und vermeidet so unnötige Durchläufe:**

```
void bubble_sort(int n, int z[]) {
    boolean sortiert = false;
    for (int i=0; i < n-1 && !sortiert; i++) {
        sortiert = true;
        for (int j = n-1; j > i; j--)
            if (z[j] < z[j-1]) {
                sortiert = false;
                int t = z[j]; z[j] = z[j-1]; z[j-1] = t;
            }
    }
}
```

# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

### Insert-Sort (Sortieren durch direktes Einfügen)

```
void insert_sort(int n, int z[]) {
    for (int i=1; i < n; i++) /* fuegt i. te Eleme
        for (int j=i; j > 0 && z[j] < z[j-1]; j--) {
            int t = z[i];    z[i] = z[j];    z[j] = t; /
        }
}
```

```
---- vor insert_sort ----------------------
 61 96 24 45 87 29  4 14 31 31

 1. Durchlauf:  61 96 24 45 87 29  4 14 31 31
 2. Durchlauf:  24 61 96 45 87 29  4 14 31 31
 3. Durchlauf:  24 45 61 96 87 29  4 14 31 31
 4. Durchlauf:  24 45 61 87 96 29  4 14 31 31
 5. Durchlauf:  24 29 45 61 87 96  4 14 31 31
 6. Durchlauf:   4 24 29 45 61 87 96 14 31 31
 7. Durchlauf:   4 14 24 29 45 61 87 96 31 31
 8. Durchlauf:   4 14 24 29 31 45 61 87 96 31
 9. Durchlauf:   4 14 24 29 31 31 45 61 87 96
---- nach insert_sort ---------------------
 4 14 24 29 31 31 45 61 87 96
```

# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

### Insert-Sort (Sortieren durch direktes Einfügen)

Sehr leistungsfähig ist Insert-Sort bei bereits sortierten Daten, in die man wenige andere Daten einsortieren muss.

→ Man hängt diese Daten an Ende des sortierten Arrays an, bevor man es dann mit Insert-Sort sortieren lässt.

→ Bei großen vorsortierten Datenmengen, wie Telefonbuch, in das neue Telefondaten einzumischen sind, ist der Insert-Sort sogar den später vorgestellten komplizierten Sortieralgorithmen überlegen.

# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

**Select-Sort (Sortieren durch direktes Auswählen)**

```
void select_sort(int n, int z[]) {
    int i, j, h, t, k;
    for (int i=0; i < n-1; i++) { /* sucht i. tes k
        h = i;
        for (int j=i+1; j < n; j++)
            if (z[h] > z[j])
                h = j;   /* Merke neue Position */
        if (h != i) {
            int t = z[h];    z[h] = z[i];    z[i] = t;
        }
    }
}
```

**Zuerst kleinstes Element an 1. Stelle setzen → wieder kleinstes Element an 2. Stelle setzen, ... Allgemein sucht man für i = 1, 2, 3,…, n Pos., an der kleinstes Elem. im noch unsortierten Bereich (i..n) steht und tauscht es gegen z[n].**

# Elementare Sortieralgorithmen

## Grundsätzliches zu Sortieralgorithmen

### Select-Sort (Sortieren durch direktes Auswählen)

```
------ vor select_sort ------------------------
 24 74 24 86 59 96 13 76  7 39
 1. Durchlauf:   7 74 24 86 59 96 13 76 24 39
 2. Durchlauf:   7 13 24 86 59 96 74 76 24 39
 3. Durchlauf:   7 13 24 86 59 96 74 76 24 39
 4. Durchlauf:   7 13 24 24 59 96 74 76 86 39
 5. Durchlauf:   7 13 24 24 39 96 74 76 86 59
 6. Durchlauf:   7 13 24 24 39 59 74 76 86 96
 7. Durchlauf:   7 13 24 24 39 59 74 76 86 96
 8. Durchlauf:   7 13 24 24 39 59 74 76 86 96
 9. Durchlauf:   7 13 24 24 39 59 74 76 86 96
------ nach select_sort ---------------------
  7 13 24 24 39 59 74 76 86 96
```

# Elementare Sortieralgorithmen

## Zeitmessungen

### Zeitverhalten bei kleinen Datensätzen

```
.... Guenstigster Fall:
Bubble–Sort:    0.00 Sek.
Insert–Sort:    0.00 Sek.
Select–Sort:   19.18 Sek.
.... Durchschnittlicher Fall:
Bubble–Sort:   59.47 Sek.
Insert–Sort:   28.73 Sek.
Select–Sort:   20.46 Sek.
.... Unguenstigster Fall:
Bubble–Sort:   58.39 Sek.
Insert–Sort:   57.92 Sek.
Select–Sort:   20.63 Sek.
```

- Select-Sort braucht in allen 3 Fällen etwa die gleiche Zeit.
- Im durchschnittl. Fall braucht verbesserter Bubble doppelt so viel Zeit wie andere.
- Im ungünstigsten Fall braucht Select-Sort nur halb so viel Zeit wie Bubble und Insert.

- Bei kleinen Datensätzen ist bei Wahl des Algorithm. auf vorliegende Reihenfolge zu sortierender Daten zu achten.

# Elementare Sortieralgorithmen

## Zeitmessungen

### Zeitverhalten bei großen Datensätzen

```
.... Guenstigster Fall:
Bubble-Sort:    0.00 Sek.
Insert-Sort:    0.00 Sek.
Select-Sort:    0.47 Sek.
.... Durchschnittlicher Fall:
Bubble-Sort:   25.12 Sek.
Insert-Sort:   23.13 Sek.
Select-Sort:    0.52 Sek.
.... Unguenstigster Fall:
Bubble-Sort:   46.97 Sek.
Insert-Sort:   46.87 Sek.
Select-Sort:    0.64 Sek
```

- Select-Sort wieder in allen 3 Fällen in etwa gleiche Zeit.
- Im durchschnittl. und im ungünstigsten Fall hat der Select-Sort eindeutig das beste Zeitverhalten.

- Auch bei großen Datensätzen ist bei Wahl des Algorithmus auf vorliegende Reihenfolge der zu sortierenden Daten zu achten.

# Shell-Sort

- Einer der ersten Sortieralgorithmen überhaupt.

- Basiert auf Insert-Sort, vertauscht nicht nur benachbarte Elemente, sondern auch weit voneinander liegende.
  → Folge sich überlappender Insert-Sorts, die Elemente in einer Distanz h voneinander vergleichen und sortieren.
  → h-sortierte Datenmenge bestehend aus h-unabhängigen sortierten Datenmengen, die übereinander liegen.

- Bedeutet, dass nach jedem h-Durchgang alle Daten, die mit Distanz h zueinander liegen, zueinander sortiert sind.
  → Z. B. könnte man nach einem h-Durchgang mit h=7 jedes 7. Element (unabh. vom Startwert) aus Datenmenge entnehmen und man hätte ein sortiertes Teilarray.

# Shell-Sort

## Typischer Code für einen Shell-Sort

```
void shell_sort(int z[], int l, int r) {
    int h, sw[] = { 1391376, 463792, 198768, 86961, 33936, // Folge von h–Distanzen
                    13776, 4592, 1968, 861, 336, 112, 48, 21, 7, 3, 1 };
    for (int k=0; k < sw.length; k++) {
        h = sw[k];
        for (int i = l+h; i <= r; i++) {
            int v = z[i], j = i;
            while (j >= h && z[j–h] > v) { z[j] = z[j–h]; j –= h; }
            z[j] = v;
        }
    }
}
```

**Parameter l und v legen dabei den zu sortierenden Bereich des Arrays fest**

# Shell-Sort

## Weitere Eigenschaften des Shell-Sorts

- Datenmenge, die zugleich 2-sorted und 3-sorted ist, kann mit einem Durchgang und n Vergleichen vollständig sortiert werden, also 1-sorted werden.

- Eine 4-sorted und 6-sorted Datenmenge kann in einem Durchgang mit n Vergleichen 2-sorted gemacht werden.

- Eine 6-sorted und 9-sorted Datenmenge kann in einem Durchgang mit n Vergleichen eine 3-sorted Menge werden.

# Quicksort

- Anfang der 1960er Jahre von C.A.R Hoare gefunden.
- Einer der am häufigsten verwendeten Sortieralgorithmen.
- Im Durchschnitt benötigt er nur n·log n Operationen.
- Vorteil ist nicht nur seine Schnelligkeit, sondern auch sein geringer Speicherbedarf, da er Daten im zu sortier. Array direkt nur mit Hilfe eines kleinen Hilfs-Stacks sortiert.
- Nachteile des Quicksort sind, dass er rekursiv arbeitet und im ungünstigsten Fall $n^2$ Operationen benötigt.

# Quicksort

**Prinzip „Teile und Herrsche“ → zerlegt Datenmenge in zwei Teile und sortiert dann beide Teile unabh. voneinander**

- **Array z[p..r] wird in 2 nicht-leere Teilarrays z[p..q] und z[q+1..r] zerlegt, so dass alle Elemente in z[p..q] < als alle in z[q+1..r].**
- **Funktion partition() (für Zerlegung des Arrays in Teilarrays) liefert dabei den Index des so genannten Pivot-Elements, das die Trennstelle zwischen den beiden Teilarrays ist.**
- **Teilarrays werden nun ihrerseits wieder nach dem gleichen Verfahren durch rekursive Aufrufe des Quicksort sortiert.**

# Quicksort

## Typischer Standard-Algorithmus des Quicksort

```
void quick_sort(int z[], int l, int r) {
    if (l < r) {
        int pivot = partition(z, l, r);
        quick_sort(z, l, pivot-1);
        quick_sort(z, pivot+1, r);
    }
}
```

```
int partition(int z[], int l, int r) {
    int x = z[r], i = l-1, j = r;
    while (1) {
        while (z[++i] < x)
            ;
        while (z[--j] > x)
            ;
        if (i < j)
            swap(&z[i], &z[j]);
        else {
            swap(&z[i], &z[r]);
            return i;
        }
    }
}
```

# Quicksort

## Laufzeitverhalten des Quicksort

- **Günstigster Fall (best case) → O(n · lg n)**
  Bei jeder rekursiven Zerlegung der Datenmenge in zwei Teilarrays wird das vorhergehende Array genau halbiert.

- **Durchschnittlicher Fall (average case) → O(1,38n · lg n)**
  Wenn die zu sortierenden Daten zufällig angeordnet sind
  → um etwa 40% mehr Vergleiche als im günstigsten Fall.

- **Ungünstigster Fall (worst case) → $O(n^2)$**
  Datenmenge ist bereits auf- oder absteigend sortiert
  → bei jeder Zerlegung in zwei Teilarrays enthält immer nur das eine Teilarray ein Element.

# Mergesort

- Quicksort zerlegt rekursiv in 2 Teilarrays, die er sortiert.
  → Mergesort geht umgekehrt vor, indem er rekursiv zwei bereits sortierte Teilarrays mischt (merge).
- Vorteil → Laufzeit immer (auch im ungünstig. Fall) $O(n \cdot \lg n)$.
- Größter Nachteil: benötigt zu n prop. zusätzl. Speicherplatz.
- Wenn es um Schnelligkeit geht und genug Speicherplatz vorhanden, ist also Mergesort dem Quicksort vorzuziehen.
- Weiterer Vorteil des Mergesort → lässt sich umändern, dass Zugriff auf Daten nahezu sequenziell (ohne Indizes) mögl.
  → vorteilhaft, wenn nur sequenziell zugreifbar, wie z.B. beim Sortieren verkett. Liste oder auf Geräten mit seq. Zugriff.
- Mergesort anders als Quicksort → stabiler Sortieralgorithm.

# Mergesort

## Rekursiver Mergesort für Arrays

Auch Mergesort hier nach Prinzip „Teile und Herrsche“
→ zerlegt Datenmenge in 2 Teile, sortiert diese beiden Teile rekursiv unabh. voneinander und mischt sie dann:

```
void merge_sort(int z[], int l, int r) {
    if (l < r) {
        int mitte = (l+r)/2;
        merge_sort(z, l, mitte);
        merge_sort(z, mitte+1, r);
        merge(z, l, mitte, r);
    }
}
```

```
void merge(int z[], int l, int m, int r) {
    int i, j, k;
    for (i=m+1; i>l; i--)
        hilf[i-1] = z[i-1];
    for (j=m; j<r; j++)
        hilf[r+m-j] = z[j+1];
    for (k=l; k<=r; k++)
        z[k] = (hilf[i] < hilf[j]) ? hilf[i++] : hilf[j--];
}
```