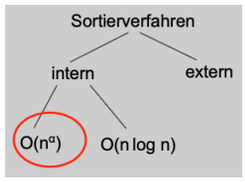


VL03/04 Elementare Sortieralgorithmen

Sortieren

- **Mehr als ein Viertel der kommerziell verbrauchten Rechenzeit entfällt auf Sortiervorgänge.**
- Gegeben: Jeder Satz besitzt einen **Schlüssel**.
- Zwischen den Schlüsseln ist eine **Ordnungsrelation** definiert Beispiel "<" oder "<="
- Außer Schlüsseln können Sätze weitere Komponenten als eigentliche Information beinhalten.

Internes und externes Sortieren - Beispiel „Sortieren nummerierter Karten":

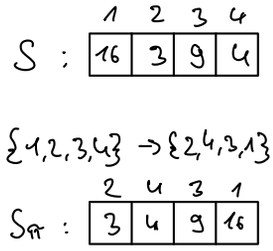
Internes und externes Sortieren wird auch **Sortieren von Arrays**
und **Sortieren von sequentiellen Files (Stapel)** genannt.

- Das Strukturieren der Karten als Array entspricht ihrem Auslegen vor dem Sortierenden, so dass jede Karte sichtbar und effizient zugreifbar ist. - **intern**
- Strukturieren als File bedeutet, dass von jedem Stapel nur die oberste Karte sichtbar ist. Kann aber nötig sein, wenn der Tisch zu klein ist, um alle Karten auszulegen. – **extern**

Das Sortierproblem

- Gegeben ist eine Folge von Sätzen $s_1, \ldots, s_N$ || jeder Satz s_i hat einen Schlüssel k_i (engl. key).

- **Viele Details noch offen:**
  - Was bedeutet „Sätze sind gegeben"? schriftlich?, auf Festplatte?, im Hauptspeicher?
  - Ist die **Anzahl** der Sätze vor dem Sortieren überhaupt bekannt?
  - Ist das **Spektrum** der vorkommenden Schlüssel bekannt?
  - Welche **Operationen** sind erlaubt um die Permutation zu bestimmen?
  - Wie geschieht die Umordnung?
  - Wird bei **Gleichheit** der Schlüssel umgeordnet oder nicht?

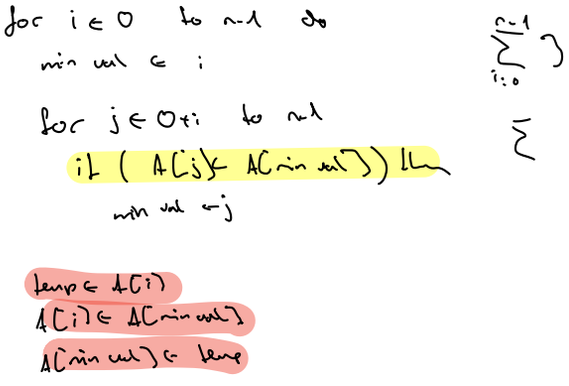


- Annahme zunächst: Die Menge der zu sortierenden Datensätze passt vollständig in den Hauptspeicher.

- **Gegeben: Sätze in unsortiertem Array**
  - im Hauptspeicher, ihre Anzahl ist bekannt, alle Elemente sichtbar, alle Elemente effizient zugreifbar
- **Erlaubter Speicherplatzverbrauch:**
  - Die Umstellung zur Ordnung der Elemente soll **am Ort** ausgeführt werden (also möglichst speichereffizient),

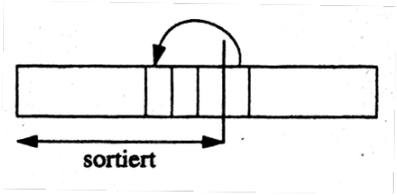
- Wünschenswert: **die Sortiermethode soll stabil sein.**
  - Eine Sortiermethode heisst stabil, wenn die relative Ordnung der Elemente mit gleichem Schlüssel beim Sortieren unverändert bleibt.
- Stabilität beim Sortieren vor allem dann erwünscht, wenn die Elemente bereits nach einem zweitrangigen Schlüssel sortiert sind, d.h. nach Eigenschaften, die nicht durch den (Haupt-)Schlüssel selbst ausgedrückt werden.
  - Beispiel: Alle Personen mit Namen „Meier" sind bereits nach ihrem Vornamen sortiert, Sortierung einer Liste mit allen Meiers, Müller, Schmitz, Adam, etc. soll die interne Meier- Reihenfolge nicht ändern.

- **Erlaubte Operationen**, um Permutation zu bestimmen?
  - Zugelassen ist nur Ergebnis des Vergleichs zweier Schlüssel.



Grundlegende Ideen zum Sortieren:

Sortieren durch Auswählen/Einfügen



Variante A: Sortieren durch Einfügen

1. Wähle erstes Element des unsortierten Teils
2. füge es an richtiger Stelle in sortierten Teil der Liste ein.

Beispiel am Pseudocode - straight insertion

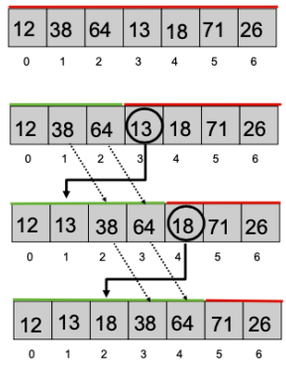
```
Algorithm StraightInsertion(A,n)
for i←1 to n-1 do
   pos_found←0;
   j←i;
   temp←A[i];                          M
   while(j>0) and (pos_found=0)do
     if(A[j-1]<temp)                   C
        pos_found←1;
     else
        A[j]←A[j-1];                   M
        j←j-1;
   A[j]←temp;                          M
```

Option 1: Vergleiche (C)

- Zählen, wie oft Elemente miteinander verglichen werden
- Kosten des Algorithmus ergibt sich aus der Anzahl der Vergleiche

Option 2: Bewegungen (M)

- Gezählt wird, wie oft ein Satz von einer Stelle zur anderen bewegt werden muss
- Kosten des Algorithmus ergibt sich aus der Anzahl der Bewegungen



Laufzeit-Analyse

Anzahl Vergleiche von Datensätzen (C = Comparison):

$$C_{min} = (n-1)\cdot 1 \in O(n)$$

$$C_{max} = \sum_{i=1}^{n}(i-1) = \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1 = \frac{n(n-1)}{2} - n = \frac{1}{2}n^2 - \frac{3}{2}n \in O(n^2)$$

$$C_{max} = \sum_{i=1}^{n-1}\sum_{j=0}^{i} 1 = \sum_{i=1}^{n-1} i$$

$$= \frac{n-1}{2}\cdot n$$

$$= \frac{n^2}{2} - \frac{n}{2} \in O(n^2)$$

Anzahl Bewegungen von Datensätzen (M = Movement):

$$M_{min} = (n-1)\cdot 2 \in O(n)$$

$$M_{max} = \sum_{i=1}^{n}(i+2) \in O(n^2)$$

$$M_{max} = \sum_{i=1}^{n-1}\left(2 + \sum_{j=1}^{i} 1\right) = \sum_{i=1}^{n-1} 2 + \sum_{i=1}^{n-1} i$$

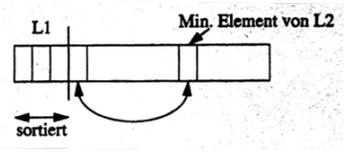
$$= 2\cdot(n-1) + \left(\frac{n-1}{2}\right)\cdot n$$

$$= 2n - 2 + \frac{n^2}{2} - \frac{n}{2}$$

$$= \frac{n^2}{2} + \frac{3}{2}n - 2 \in O(n^2)$$

Variante B: Sortieren durch Auswählen

1. Wähle nächstgrößtes Element aus unsortieren Teil der Liste
2. füge es an sortierten Teil an



Beispiel am Pseudocode - straight selection

```
Algorithm StraightSelection(A,n)
for i←0 to n-1 do
    pos_min←i;
    for j←i+1 to n-1 do
       if(A[j]<A[pos_min])             C
          pos_min=j;

    help←A[pos_min];                   M
    A[pos_min]←A[i];                   M
    A[i]←help;                         M
```

$$C_{max} = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n-1-(i+1)+1$$

$$= \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1$$

$$= n^2 - \left(\frac{n-1}{2}\right)\cdot n - (n-1+1)$$

$$= \frac{n^2}{2} - \frac{n}{2} \in O(n^2)$$

Anzahl von durchlaufen einer Schleife

Ende - Anf + 1

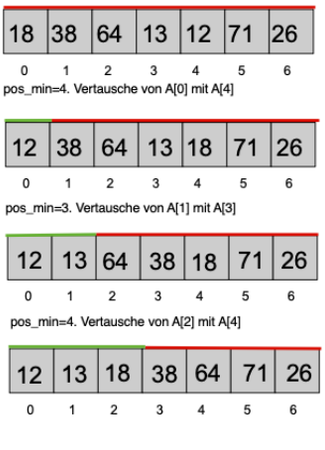
Gaußsche Summenformel:

$$\sum_{i=1}^{n} i = \left(\frac{n}{2}\right)\cdot(n+1)$$

Laufzeit-Analyse

$$M_{min}(n) = M_{max}(n) = M_{avg}(n) = 3(n-1)$$

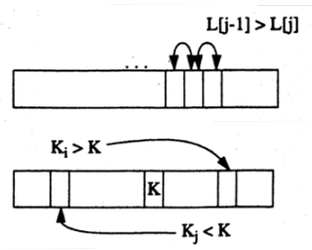
$$C_{min}(n) = C_{max}(n) = C_{avg}(n) = \sum_{i=1}^{n}(n-i) = n^2 - \sum_{i=1}^{n} i = n^2 - \frac{n(n+1)}{2} \in O(n^2)$$



Bemerkungen:

- Jeder Algorithmus zur Bestimmung des Minimums von n Schlüsseln, der allein auf Schlüsselvergleichen beruht, muss mindestens n-1 Schlüsselvergleiche ausführen.
- **Sortieren durch Auswahl ist günstig, wenn Satzbewegungen aufwändig und Schlüsselvergleiche günstig (schnell) sind.**

Sortieren durch Austauschen



Variante C: Sortieren durch Austauschen

1. Vergleiche Elemente miteinander
2. Bei falscher Reihenfolge werden sie vertauscht.

- Kann mit direkt benachbarten Elementen geschehen
- Vertauschungen können über größere Strecken passieren

- **Bubblesort**
- **Shellsort**

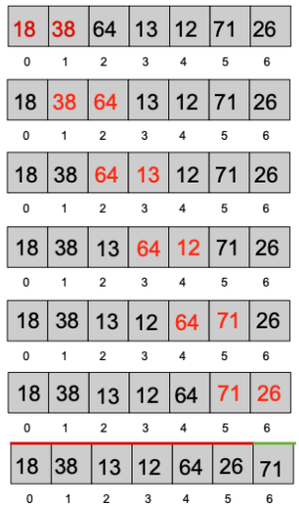
Bubble-Sort

```
Algorithm SimplestBubbleSort(A,n)
for j←0 to n-2 do
   for i←0 to n-2-j do
      if(A[i]>A[i+1])
        temp←A[i];
        A[i]←A[i+1];
        A[i+1]←temp;
```

C
M
M
M

Verbesserungsmöglichkeit:

- Merken ob überhaupt eine Vertauschung stattgefunden hat.
- Wenn nicht, Abbruch → fertig.



Laufzeit-Analyse

$$C_{min}(n) = C_{max}(n) = \sum_{j=1}^{n}(\sum_{i=1}^{n-j}(1)) = \sum_{j=1}^{n}(n-j)$$
$$= \sum_{j=1}^{n} n - \sum_{j=1}^{n} j$$
$$= n^2 - \frac{1}{2}n^2 - \frac{1}{2}n \in O(n^2)$$

$$M_{max}(n) = \sum_{j=1}^{n}(\sum_{i=1}^{n-j}(3)) = \sum_{j=1}^{n} 3(n-j) = 3\sum_{j=1}^{n} n - 3\sum_{j=1}^{n} j = 3n^2 - \frac{3}{2}n^2 - \frac{3}{2}n \in O(n^2)$$

$M_{min} = 0$

Bubble Sort ist im Allgemeinen der schlechteste elementare Sortieralgorithmus.

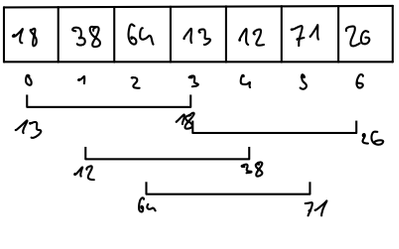
Shell Sort

- Anstatt durch mehrfache Verschiebungen, bringe die Elemente durch **größere Sprünge** an die richtige Position.
- **Definition**: $k_1,k_2,...,k_n$ sei eine Folge von Schlüsseln.
  → Man nennt diese Folge **h-sortiert**, wenn für alle i, $1 \le i \le n-h$ gilt: $k_i \le k_{i+h}$.
- Durch die Vorsortierung sorgen wir dafür das große und kleine Schlüssel schon ungefähr an die richtige Stelle kommen, das heißt wir haben im letzten Durchgang nur noch sehr wenig zu verschieben. Kleine Feinheiten!

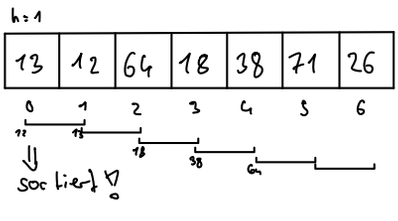
Analyse Shell Sort

- Stellt einige sehr schwierige mathematische Probleme.
- Es ist bekannt, dass die Schrittweiten **nicht** Vielfache voneinander sein sollen.
- **Es gilt:** Wenn eine k-sortierte Sequenz i-sortiert wird, bleibt sie k-sortiert. stabil
- Es ist beweisbar, dass für die Folge 1,3,7,15,31,... das Verfahren $\in O(n^{1,5})$ ist.
  - Bewiesen: Es kann keine Folge geben mit O(n log n)

Folge 1,3,7,15,31,...
h=3



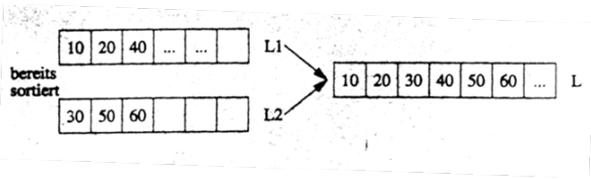
⇒



Sortieren durch Mischen

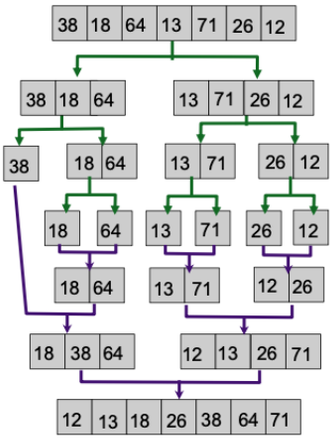
Variante D: Sortieren durch Mischen

- wenn Teile der Liste bereits sortiert sind, lassen sich diese zu einem größeren ebenfalls sortierten Teil zusammen-mischen.
- relativ effizient, rekursiv anwendbar.



Divide and Conquer Prinzip

- **Divide**: Teile das Problem der Größe n>1 in (mindestens) 2 annähernd gleich große Teilprobleme.
- **Conquer**: Löse Teilprobleme auf dieselbe Art (rekursiv)
- **Merge**: Füge die Teillösungen zur Gesamtlösung zusammen



MergeSort (Naive Variante) - rekursiv

```
Algorithm MergeSort(L,n)

  if(n=1) return L;

  for j←1 to n/2 do
     L1←L[j];
  for j←n/2+1 to n do
     L2←L[j];

  L1←MergeSort(L1,n/2);
  L2←MergeSort(L2,n-n/2);

  return MergeTwoLists(L1,L2);
```

O(n log n)

Teile das Problem in 2 Teilprobleme

- **Teile** jedes Teilproblem wieder auf
- Fast alle Teilprobleme sind jetzt *trivial*

Alle Teil-Listen haben jetzt nur noch 1 Element, d.h. sie sind sortiert.

Mische je 2 sortierte Teillisten zu einer ebenfalls sortieren größeren Liste

→ *usw. bis Gesamtliste sortiert*

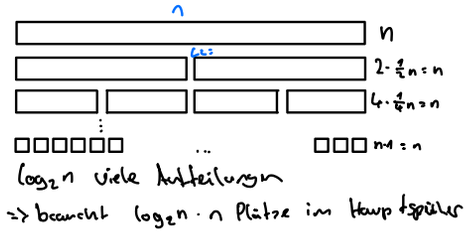
Überlegungen zur Merge Methode

```
Algorithm MergeTwoLists(L1,L2,m,n)
  i←0; j←0
  L1[m]←maxint; L2[n]←maxint;
  for k←0 to m+n-1 do
       if(L1[i]<L2[j])
         L[k]←L1[i]; i←i+1;
       else
         L[k]←L2[j]; j←j+1;
  return L;
```

O(n log n)

- Wenn 3 Listen verfügbar, dann
  - Zähler i für Liste L1, Zähler j für Liste L2, Zähler k für Liste L
  - Länge L1: m
  - Länge L2: n
  - Länge L: m+n

Speicherkomplexität



Beobachtung: Zwei zu mergende Teil-Listen liegen immer nebeneinander

MergeSort (Verbesserte Variante)

- Verbesserte Variante soll weniger oft Kopien anlegen
- Bei jedem Aufruf werden die aktuellen Begrenzungen der Teil-Listen mitgegeben.
  - low, middle, high
- Merge legt nur noch **eine temporäre Liste an**.

```
Algorithm MergeSort(L,lo,hi)

  if(lo>=hi) return L;

  mid←(lo+hi)/2;
L←MergeSort(L,lo,mid);
  L←MergeSort(L,mid+1,hi);

  return Merge(L,lo,mid,hi);
```

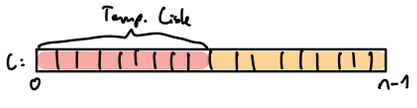
Verbesserte Merge Methode

Idee: Es genügt nur vordere Hälfte in Hilfsarray auszulagern

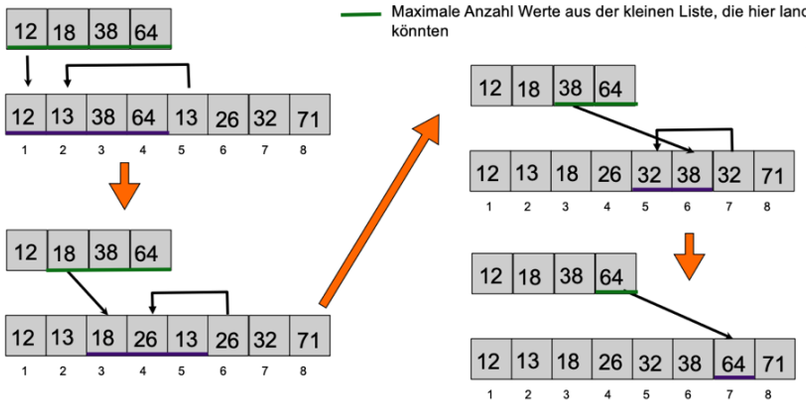
Die hintere Hälfte bleibt im Array A.

- Verbraucht nur halb soviel zusätzlichen Speicherplatz
- nur halb soviel Zeit zum Kopieren benötigt

Die gemischte Liste steht wieder direkt im Array A



Beispiel im Detail:




```
Algorithm Merge(A,lo,mid,hi)
  i←1; j←lo;

  while j<=mid do
     B[i]=A[j];
     i←i+1; j←j+1;

  i←1; k←lo;
  while k<j and j<=hi do
     if(B[i]<A[j])
       A[k]←B[i]; i←i+1;
     else
       A[k]←A[j]; j←j+1;
     k←k+1;

  while k<j do
     A[k]←B[i]; i←i+1; k←k+1;
  return A;
```

vordere Hälfte von A in Hilfsarray B kopieren

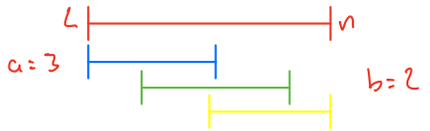
jeweils das nächstgrößte Element zurückkopieren

Rest von B zurück-kopieren falls vorhanden

Master Theorem für Rekursionsgleichungen

- Design von Divide+Conquer Strategien allgemein: $T(n) = \begin{cases} 1 & \text{für } n = 1 \\ a \cdot T(\frac{n}{b}) + d(n) & n > 1 \end{cases}$
- Zerteilung des Problems in a Probleme der Größe n/b, a≥1, b>1.
- und einer positiven Funktion d(n)
- statt $T(\frac{n}{b})$ kann auch $T(\lfloor \frac{n}{b} \rfloor)$ oder $T\lceil \frac{n}{b} \rceil$ stehen.
- **Master Theorem**: falls $d(n) \in O(n^\gamma)$ *mit* $\gamma > 0$:

$$T(n) \in \begin{cases} O(n^\gamma) & a < b^\gamma \\ O(n^\gamma \log_b n) & \text{für } a = b^\gamma \\ O(n^{\log_b a}) & a > b^\gamma \end{cases}$$



a = Anzahl der rekursiven Aufrufe
b = Anzahl der Aufteilungen
d(n) = Laufzeit für's Mischen
γ bestimmen mit der $O(n^\gamma)$

Master Theorem Aufg 3)

a.) $T(n) = 4 \cdot T(\frac{n}{2}) + n$ $\quad a=4, b=2, \gamma=1$ $\quad \in n^{\log_2 4}$

$4 > 2^1$

b.) $T(n) = 4 \cdot T(\frac{n}{2}) + n^2$ $\quad a=4, b=2, \gamma=2$ $\quad \in O(n^2 \log_2 n)$

$4 = 2^2$

c.) $T(n) = 4 \cdot T(\frac{n}{2}) + n^3$ $\quad a=4, b=2, \gamma=3$ $\quad \in O(n^3)$

$4 < 2^3$

d.) $T(n) = 4 \cdot T(\frac{n}{2}) + n \cdot \log_2 n$

$\in O(n \cdot \log n) \in O(n^2)$ $\quad a=4, b=2, \gamma=2$ $\quad \in O(n^2 \log_2 n)$

$4 = 2^2$

2.) a.) $T(n) = 4 \cdot T(\frac{n}{4}) + \text{sqrt}(n)$ $\quad a=4, b=4, \alpha=\frac{1}{2}$

$4 > 4^{\frac{1}{2}} \in O(n^{\log_4 4})$

b.) $T(n) = 25 \cdot T(\frac{n}{5}) + 2n$ $\quad a=25, b=5, \gamma=1$

$25 > 5^1$

c.) $T(n) = 3 \cdot T(\frac{n}{9}) + n^3 - 5$ $\quad a=3, b=9, \gamma=3$

$3 < 9^3$ $\quad O(n^3)$

$T(n) = 3 \cdot T(\frac{n}{3}) + 1$ $\quad a=3, b=3, \gamma=0$

1