Programmieren / Algorithmen & Datenstrukturen 2

Fortgeschrittenes Sortieren

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Fortgeschrittenes Sortieren

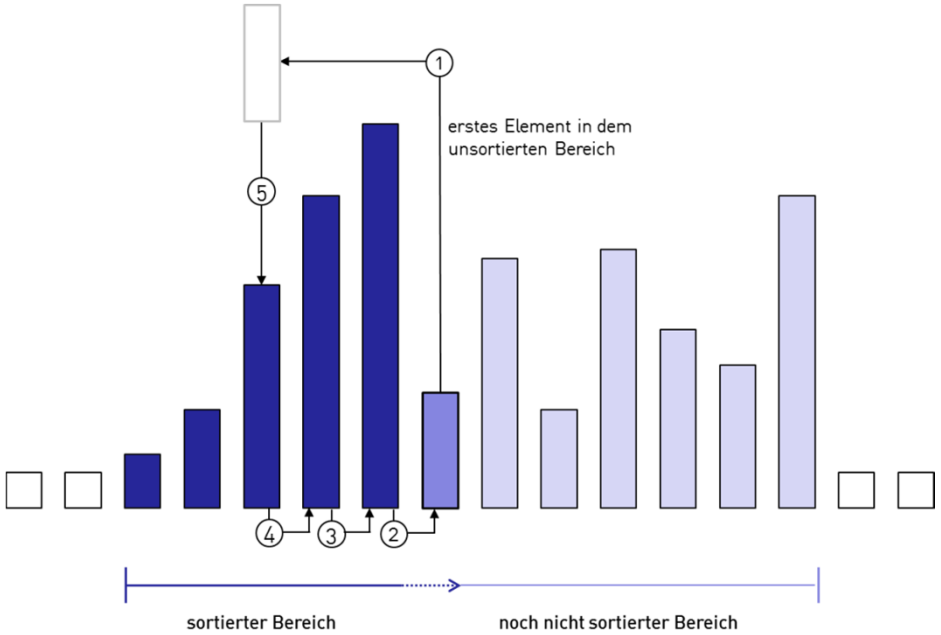
Inhalt.

- Templates
- Abgeleitete Klassen
- Testgetriebene Programmierung
- Container, Iteratoren und Algorithmen der StdLib
- Fortgeschrittenes Suchen
- Fortgeschrittenes Sortieren
- Grafische Benutzeroberflächen

Wiederholung: Sortieren durch direkte Einfügung

Wir hatten schon das Sortieren durch direkte Einfügung ("Insertion Sort") implementiert.

- Sortieren durch direkte Einfügung:




```
01 void insSort( vector<T>& vV, int ui, int oi ) {
02     int i{}, j{}; T tmp {};
03     for( i=oi; ui<i; --i ) swap_if( vV[i-1], vV[i] );
04     for( i=ui+2; i<=oi; ++i ) {
05         tmp = vV[i];
06         for( j=i; tmp<vV[j-1]; --j ) vV[j] = vV[j-1];
07         vV[j] = tmp;
08     }
09 }
```

Shellsort

Shellsort ist eine wirkungsvolle und trotzdem einfache Erweiterung des Sortierens durch direkte Einfügung.

- Eine Schwachstelle des Sortierens durch direkte Einfügung:
  - Für Elemente, die sich in den ursprünglichen, unsortierten Daten weit entfernt von ihrer späteren, endgültigen Position innerhalb der sortierten Daten befinden, müssen viele benachbarte Elemente einzeln vertauscht werden.
  - Sie erinnern sich: genau falsch herum vorsortierte Eingangsdaten sind der ungünstigste Fall für das Sortieren durch direkte Einfügung.
- Shellsort beseitigt diese Schwachstelle:
  - Elemente, die weit voneinander entfernt liegen, werden zuerst direkt vertauscht.
- Die Shellsort-Grundidee: sog. h-sortierte Daten.
  - Daten werden h-sortiert genannt, wenn man durch Entnahme jedes h-ten Elements sortierte Daten erhält.
- Erzeugt man h-sortierte Daten anfangs mit großen Werten für h, werden Elemente unmittelbar über große Abstände bewegt.
- Shellsort erzeugt h-sortierte Daten durch direkte Einfügung, mit einer Abfolge von der Größe nach abnehmenden h-Werten, die mit $h = 1$ (und d.h. komplett sortierten Daten) endet.

Shellsort

Veranschaulichung: h-sortierte Daten.

- Beispiel für aufsteigend 5-sortierte Daten:

23	**4**	**13**	**324**	**2**	**24**	**17**	**14**	**677**	**55**	**24**	**99**	**15**	**931**	**90**	**25**	**100**	**16**	**971**	**90**
1	*2*	*3*	*4*	*5*	*1*	*2*	*3*	*4*	*5*	*1*	*2*	*3*	*4*	*5*	*1*	*2*	*3*	*4*	*5*
23	4	13	324	2	**24**	17	14	677	55	**24**	99	15	931	90	**25**	100	16	971	90
23	**4**	13	324	2	24	**17**	14	677	55	24	**99**	15	931	90	25	**100**	16	971	90
23	4	**13**	324	2	24	17	**14**	677	55	24	99	**15**	931	90	25	100	**16**	971	90
23	4	13	**324**	2	24	17	14	**677**	55	24	99	15	**931**	90	25	100	16	**971**	90
23	4	13	324	**2**	24	17	14	677	**55**	24	99	15	931	**90**	25	100	16	971	**90**

  - Man kann sich vorstellen, dass h-sortierte Daten aus h unterschiedlichen, unabhängig voneinander sortierten Daten bestehen, die sich - wie oben veranschaulicht – systematisch überlagern.
  - Noch anschaulicher ist es vielleicht, sich h-sortierte Daten als eine Tabelle mit h Spalten vorzustellen, die spaltenweise sortiert ist (vgl. rechts).

23	4	13	324	2
24	17	14	677	55
24	99	15	931	90
25	100	16	971	90

- Aufgabe: geben Sie ein *eigenes* Beispiel für absteigend 3-sortierte Daten, die weder 2- noch 1-sortiert sind (mindestens 20 Elemente).

Shellsort

Eigenschaft des h-Sortierens.

- Wichtige Beobachtung: das Ergebnis des h-Sortierens von k-sortierten Daten ist sowohl h-sortiert als auch k-sortiert (für beliebige h und k).
  - Obwohl diese Eigenschaft intuitiv einzuleuchten scheint, ist der Beweis nicht ganz einfach (vgl. weiterführend etwa bei D. Knuth).
- Beispiel zur Veranschaulichung:

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E			
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3			
A	A	E	N	G	E	P	L	I	R	S	M	X	T	O	erst 3-sortiert		
A	A	E	L	G	E	O	M	I	R	P	N	X	T	S	dann 4-sortiert		
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3			
A	A	E	L	G	E	O	M	I	R	P	N	X	T	S		3,4	

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E			
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3			
A	I	A	G	E	L	E	M	P	S	N	R	T	X	O	erst 4-sortiert		
A	E	A	E	I	L	G	M	O	S	N	P	T	X	R	dann 3-sortiert		
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3			
A	E	A	E	I	L	G	M	O	S	N	P	T	X	R		4,3	

Shellsort

Beispiel: Shellsort mit der Abstandsfolge $\ldots, 13, 4, 1$ für h bearbeitet die Zeichenkette ASORTINGEXAMPLE.

- Eingangsdaten

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E

- 13-Sortierung

1	2	3	4	5	6	7	8	9	10	11	12	13	1	2
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E

1	2	3	4	5	6	7	8	9	10	11	12	13	1	2
A	E	O	R	T	I	N	G	E	X	A	M	P	L	S

  - Ergebnis:

1	2	3	4	5	6	7	8	9	10	11	12	13	1	2
A	E	O	R	T	I	N	G	E	X	A	M	P	L	S

- 4-Sortierung

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3
A	E	O	R	E	I	N	G	P	X	A	M	T	L	S

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3
A	E	O	R	E	I	N	G	P	L	A	M	T	X	S

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3
A	E	A	R	E	I	N	G	P	L	O	M	T	X	S

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3
A	E	A	G	E	I	N	M	P	L	O	R	T	X	S

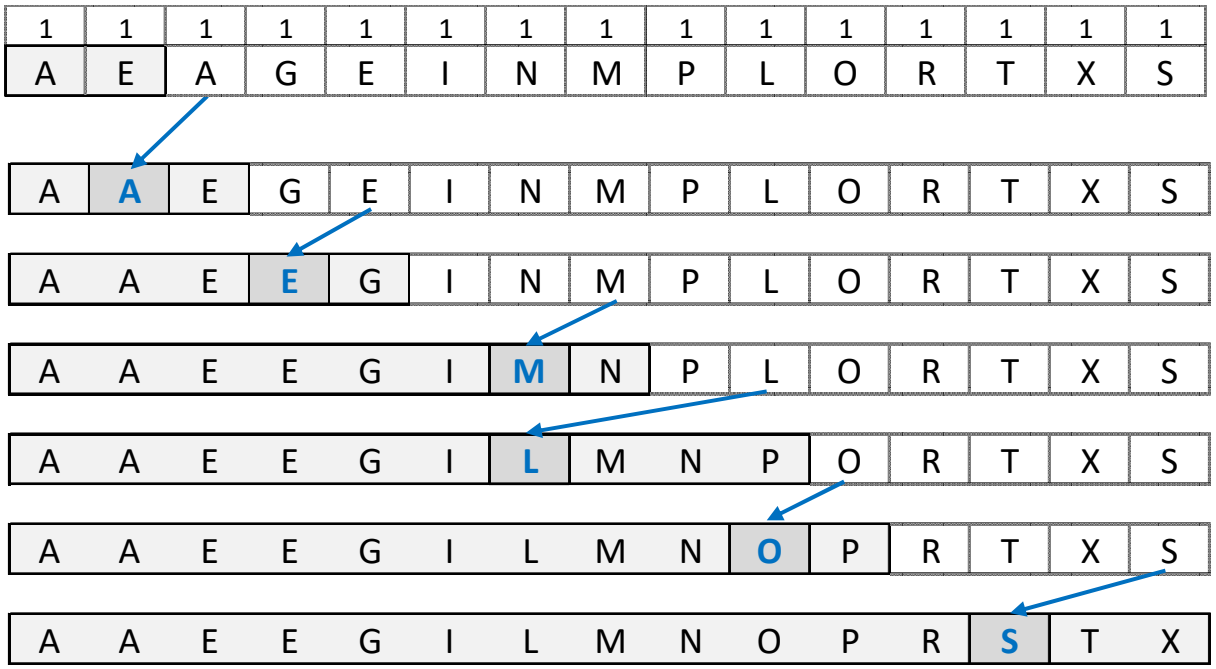
  - Ergebnis:

1	2	3	4	1	2	3	4	1	2	3	4	1	2	3
A	E	A	G	E	I	N	M	P	L	O	R	T	X	S

Shellsort

Beispiel: Shellsort mit der Abstandsfolge …, 13, 4, 1 für h bearbeitet die Zeichenkette ASORTINGEXAMPLE.

- 1-Sortierung



  - Ergebnis:

- 1-Sortieren ist das normale Sortieren durch direkte Einfügung.
- Da Shellsort dafür sorgt, dass die Eingangsdaten für den letzten Schritt (für das 1-Sortieren) schon h-sortiert sind (für alle bisher verwendeten Abstände h), muss kein Element mehr weit bewegt/oft getauscht werden (im Beispiel höchstens um drei Positionen).

Shellsort

Shellsort lässt sich durch Anpassung des Quellcodes aus dem schon besprochenen Insertion Sort implementieren.

- Mögliche Implementierung:

```
01 template<class T> void shellSort( vector<T>& vV, int ui, int oi ) {
02    int i{}, j{}, h{}, d{};
03    T tmp{};
04    while( h <= (oi-ui+1)/9 ) h = (3*h)+1; // Startwert fuer h
05    for( ; 0 < h; h /= 3 ) {
06       for( d = 0; d < h; ++d ) {
07           // kleinstes Element jeweils an den Anfang:
08           for( i = oi-d; ui+h <= i; i -= h ) swap_if( vV[i-h], vV[i] );
09           // sortieren:
10           for( i += h; i <= oi; i += h ) {
11              j = i; tmp = vV[i];
12              while( tmp < vV[j-h] ) { vV[j] = vV[j-h]; j -= h; }
13              vV[j] = tmp;
14           }
15       }
16    }
17 }
```

in der innersten Schleife

- Dieses Programm verwendet die Abstandsfolge $n_{i+1} = 3n_i+1$
  $\ldots, 1093, 364, 121, 40, 13, 4, 1$

Shellsort

Abstandsfolgen im Shellsort Verfahren.

- Shellsort läuft mit bestimmten Abstandsfolgen ziemlich effizient,
- etwa mit der in der Beispiel-Implementierung verwendeten Folge, die von D. Knuth stammt und mit weniger als $O(N^{3/2})$ Vergleichen auskommt.
- Eine im ungünstigsten Fall nachweislich bessere Folge stammt von R. Sedgewick: $\ldots, 4193, 1073, 281, 77, 23, 8, 1$
  - Erzeugungsregel: $n_{j-1} = 4^{j-1} + 3(2^{j-2}) + 1$ (mit $j > 1$ und Startwert $n_0 = 1$).
  - Diese Folge kommt mit weniger als $O(N^{4/3})$ Vergleichen aus.
- Neben der Maßgabe, möglichst wenige Abstände zu verwenden, sind auch arithmetische Eigenschaften der Abstände zu beachten (wie z.B. der ggT).
  - Elementfolgen, die h-sortiert und k-sortiert sind, sind auch $(i \cdot h + j \cdot k)$-sortiert für alle $i, j \in \{ 0, 1, 2, 3, \ldots \}$.
  - Eine ungeschickt gewählte Folge wäre auch $\ldots, 32, 16, 8, 4, 2, 1$, denn Elemente an ungeraden Positionen werden dann erst ganz zum Schluss mit Elementen an geraden Positionen verglichen.

Shellsort

Abstandsfolgen im Shellsort Verfahren.

- Die Abstandsfolgen für Shellsort lassen sich auch fest einprogrammieren:

```
template<class T> void shlsrt_RSe( vector<T> vV ) {
   // Mit einer fest einprogrammierten Abstandsfolge
   // Implementierung nach Sedgewick:
   const int n{ 16 };
   const std::array<int,n> incs { 1391376, 463792, 198768, 86961,
                                    33936,  13776,   4592,  1968,
                                      861,    336,    112,    48,
                                       21,      7,      3,     1 };
   T tmp{};
   int h{}, i{}, j{};
   for( int k{}; k < n; ++k ) {
      h = incs[k];
      for( i=h; i<vV.size(); ++i ) {
         for( j=i; j>=h; j-=h ) {
            if( vV[j] < vV[j-h] ) {
               tmp = vV[j]; vV[j] = vV[j-h]; vV[j-h] = tmp; // tauschen
            }
            else break;
         }
      } // h
   } // k
}
```

Shellsort

Eigenschaften von Shellsort.

- Die Leistungseigenschaften von Shellsort kann man nur ungenau beschreiben.
  - Seit das Verfahren 1959 von D. Shell vorgestellt wurde war bis heute (Stand 2015) noch niemand in der Lage, es zu analysieren.
  - Einige Messdaten (Werte in 1000, außer N):

N	comparisons	$N^{1.289}$	2.5 N lg N
5,000	93	58	106
10,000	209	143	230
20,000	467	349	495
40,000	1022	855	1059
80,000	2266	2089	2257

Bildquelle: Sedgewick.

Shellsort

Eigenschaften von Shellsort.

- Shellsort ist kaum von der ursprünglichen Ordnung der Eingangsdaten abhängig,
  - im Unterschied zum Sortieren durch direkte Einfügung, aus dem Shellsort hervorgegangen ist.
- Shellsort ist kein stabiles Sortierverfahren,
  - d.h. die Ordnung nach früheren Schlüssel in den Daten wird beim Sortieren nach weiteren Schlüsseln nicht bewahrt.
- Die wesentlichen Vorteile von Shellsort:
  - Shellsort benötigt nur ein kurzes Programm, das mit etwas Geduld gut zum Laufen gebracht werden kann.
  - Sortierverfahren, die noch effizienter als Shellsort arbeiten, sind weniger leicht verständlich und es ist deutlich schwieriger, sie gut zu implementieren.

LEERE SEITE

Quicksort

Quicksort arbeitet mit der rekursiven Zerlegung der Daten in zwei Teile.

- Das grundlegende Verfahren ist rekursiv:
  - Zerlegung der Daten in zwei Teile und
  - Sortierung der Teildaten unabhängig voneinander durch
    - Zerlegung der Teildaten in zwei Teile und
    - Sortierung der Teil-Teildaten unabhängig voneinander durch
      - Zerlegung der Teil-Teildaten in zwei Teile und
      - Sortierung der Teil-Teil-Teildaten unabhängig voneinander durch
        - Zerlegung ... usw.
  - Die Rekursion endet, sobald Teil-Teil-...-Teildaten nur noch aus einem Element bestehen.
- Damit das Verfahren funktioniert ist es entscheidend, wie genau die Zerlegung in zwei Teildaten geschieht.
- Bei einer geeigneten Zerlegung müssen drei Invarianten erfüllt sein:
  - Ein Element `i` kommt an seine endgültige Position (das "Pivot-Element"),
  - kein Element im einen Teil (in `ui` bis `i-1`, "unterer" Teil) ist größer als `i`,
  - kein Element im anderen Teil (in `i+1` bis `oi` ,"oberer" Teil) ist kleiner als `i`,

Quicksort

Allgemeiner Ansatz zur Implementierung der geeigneten Zerlegung.

(1) Ein Pivot-Element an (irgend) einem Index `i` aus den Daten wird gewählt.

(2) Die Daten werden durchsucht,
- vom unteren Index `ui` ausgehend, bis ein Element gefunden wird, das größer `v[i]` ist (oder gleich groß, obwohl das nicht intuitiv ist),
- vom oberen Index `oi` ausgehend, bis ein Element gefunden wird, das kleiner `v[i]` ist (oder gleich groß, obwohl das nicht intuitiv ist).

(3) Die beiden gefundenen Elemente sind offensichtlich fehl am Platz und werden daher vertauscht.

(4) Schritte (2) und (3) werden so lang wiederholt, bis sich die beiden Zeiger treffen, die von "unten" und von "oben" gestartet waren.
- Alle Elemente "unterhalb" von dem Zeiger, der am unteren Index gestartet war, sind dann offensichtlich nicht größer als `v[i]`.
- Alle Elemente "oberhalb" von dem Zeiger, der am oberen Index gestartet war, sind dann offensichtlich nicht kleiner als `v[i]`.

(5) Die Prozedur ruft sich jetzt selbst für den unteren Teil und den oberen Teil auf, solange noch mehr als ein Element in dem jeweiligen Datenteil ist.

Quicksort

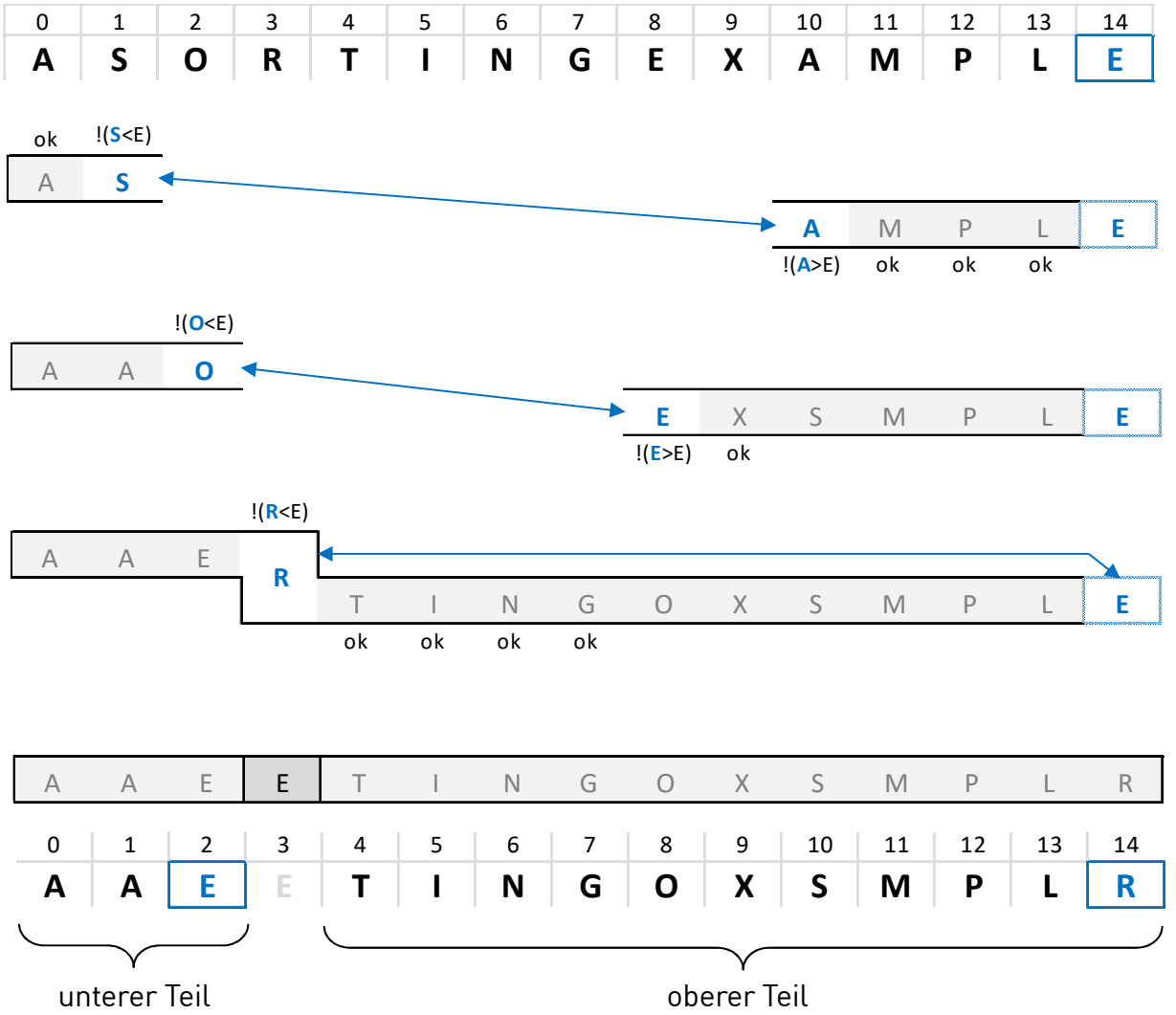
Möglichkeit einer einfachen Implementierung der Zerlegung.

- Für eine einfache Implementierung, die einigermaßen gut ans Laufen gebracht werden kann:
  - Als Pivot-Element `v[oi]` wählen.
  - Zuerst mit dem "unteren" Zeiger suchen, bei `ui` anfangen.
  - Danach mit dem "oberen" Zeiger suchen, bei `oi-1` anfangen.
  - Elemente vertauschen wie beschrieben, bis sich die Zeiger treffen.
  - Dann das Pivot-Element `v[oi]` mit dem untersten Element des oberen Teils vertauschen,
    - d.h. mit dem Element, auf das der untere Zeiger zeigt.
- Eine solche vereinfachte Implementierung funktioniert und dient der Übung und dem Verständnis dieses wichtigen Grundalgorithmus.
- Eine vollwertige Implementierung von Quicksort nach den anerkannten Regeln der Technik ist recht zeitaufwändig,
  - u.a. weil Quicksort sehr genau erforscht ist und daher die anerkannten Regeln der Technik bei diesem Verfahren weit fortgeschritten sind.

Quicksort

Veranschaulichung der vereinfachten Implementierung am Beispiel.

- Eingangsdaten
- Tausch
- Tausch
- Letzter Tausch
  Oberer Zeiger
  trifft unteren Zeiger
- Zwischenergebnis
- Rekursive Aufrufe



Quicksort

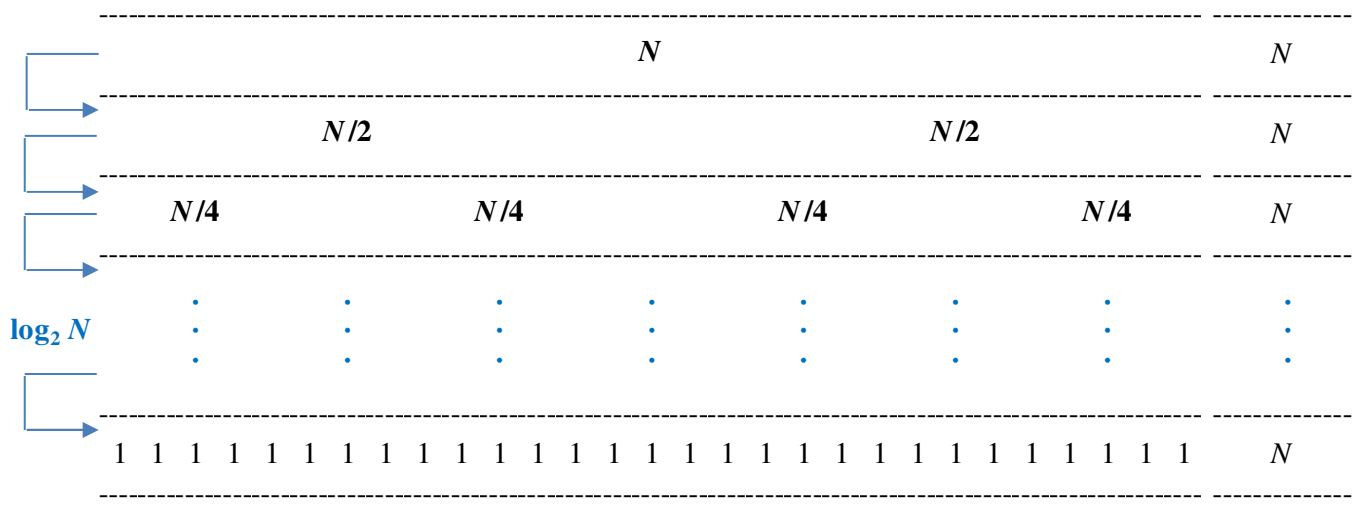
Zum Laufzeitverhalten.

- Wählt man als Pivot-Element immer das Element am höchsten Index, so läuft die Zerlegung in zwei Teile so schnell wie der Indexzugriff auf dieses Element und das Speichern des Indexwerts.
  - $O(1)$ (konstant).
- Jedes Nicht-Pivot-Element wird dann mit dem Pivot-Element verglichen.
  - $O(N)$, wenn Indexzugriff und Vergleich $O(1)$ laufen.
- Weiter hängt die Laufzeit wesentlich davon ab, wie die übrigen Elemente relativ zum Pivot-Element liegen,
  - d.h. wie groß die beiden Datenteile sind.
- Man kann sich intuitiv klar machen:
  - der günstigste Fall liegt vor, wenn zwei gleichgroße Teile entstehen,
  - der ungünstigste Fall liegt vor, wenn ein Teil "leer" ist und der andere Teil die $N–1$ übrigen Elemente enthält.

Quicksort

Zum Laufzeitverhalten: best case.

- Im günstigsten Fall liegt das Pivot-Element genau in der Mitte.
- Dann gilt die Rekursion $B_N = 2 \cdot B_{N/2} + N$
- Anschaulich:

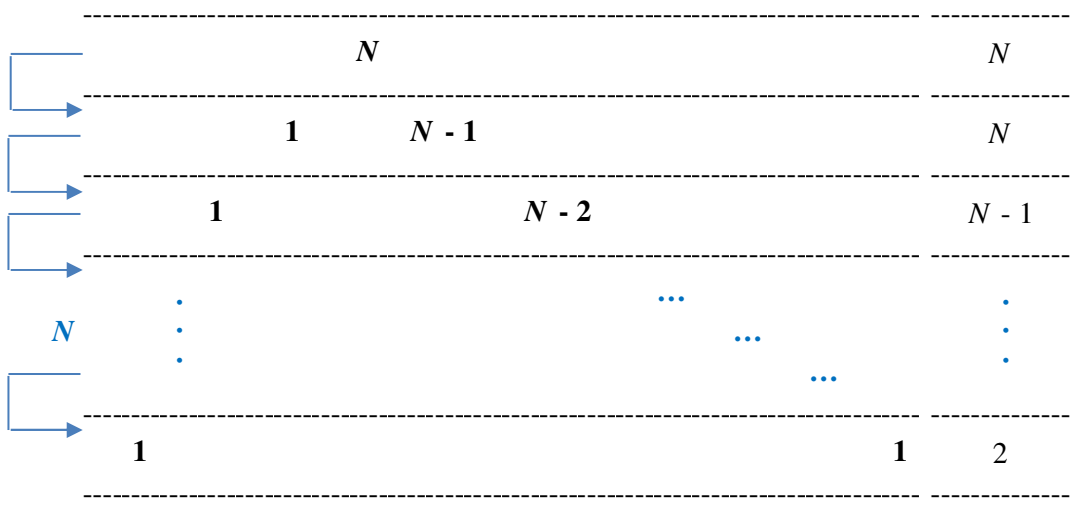


- $O(\, N \log_2 N \,)$

Quicksort

Zum Laufzeitverhalten: worst case.

- Im ungünstigsten Fall wird das Pivot-Element (ein-elementig) separiert.
- Dann gilt die Rekursion $W_N = W_{N-1} + N$
- Anschaulich:



- $O(N^2)$

Bem.: $2 + 3 + 4 + \cdots + N + N = \frac{N(N+1)}{2} - 1 + N \in O(N^2)$

Quicksort

Zum Laufzeitverhalten: average case.

- Für den durchschnittlichen Fall kann man annehmen, dass der günstigste Fall B und der ungünstigste Fall W immer genau abwechselnd auftreten.
- Dann gilt mit $W_N = B_{N-1} + N$:

$$B_N = 2 \cdot W_{N/2} + N =$$

$$= 2 \cdot (B_{(N/2)-1} + {}^{N}/_{2}) + N =$$

$$= 2 \cdot B_{(N/2)-1} + 2N \quad \in \boldsymbol{O}(\boldsymbol{N \log_2 N});$$

Analog: $W_N = 2 \cdot W_{(N-1)/2} + N - 1 + N =$

$$= 2 \cdot W_{(N-1)/2} + 2N - 1 \quad \in \boldsymbol{O}(\boldsymbol{N \log_2 N})$$

Quicksort

Quicksort gilt als *das* Mehrzweck-Sortierverfahren.

- Seit das grundlegende Verfahren ca. 1960 von C. Hoare vorgeschlagen wurde, ist Quicksort gründlich analysiert worden und die Analysen sind durch empirische Erfahrungen gut bestätigt, so dass die Eigenschaften bekannt sind, z.B.:
  - Quicksort benötigt im Mittel (d.h. zufällige Eingangsdaten) ungefähr $2\,N \ln N$ Vergleiche (ca. $1{,}39\,N \log_2 N$).
  - Die Laufzeit zum Sortieren von N Elementen ist damit, wie gesehen, im Mittel auf zufälligen Inputdaten $O(\,N \log_2 N\,)$.
    - Und damit "theoretisch optimal" (siehe minimale Anzahl von Vergleichen).
  - Im ungünstigsten Fall aber, wie gesehen, $O(\,N^{\,2}\,)$, denn Quicksort benötigt im ungünstigsten Fall ungefähr $N^{\,2}/2$ Vergleiche, bei vorsortierten Eingangsdaten, genauer $N + (N{-}1) + (N{-}2) + \ldots + 2 + 1 = (N{+}1)\,N\,/\,2.$
- Der Aufrufstack bei Quicksort:
  - Bei zufälligen Eingangsdaten ist die max. Stackgröße proportional zu $\log_2 N.$
  - Bei entarteten Daten kann der Stack proportional zu N wachsen.
- Quicksort arbeitet nicht stabil.

Quicksort

Quicksort nach den anerkannten Regeln der Technik beinhaltet weitere wichtige Verbesserungen.

- Einige wichtige, theoretisch und empirisch bestätigten Verbesserungen, die zu den anerkannten Regeln der Technik bei Quicksort gehören:
  - *Vermischen der Eingangsdaten*, bevor das eigentliche Sortierverfahren startet (und somit ungünstigste Fälle praktisch ausschließen).
  - *Kleine Teildaten* (ab ca. 10 Elementen) nicht mit einem weiteren Aufruf von Quicksort behandeln sondern unsortiert lassen, und danach die fast sortierten Elemente in einem weiteren Gesamtdurchlauf durch direktes Einfügen sortieren (direktes Einfügen ist auf fast sortierten Daten sehr effizient).
  - Den *Median* der Eingabedaten *als Pivotelement* wählen (indem der Median aus drei zufällig gewählten Elementen geschätzt wird).
  - *Nicht-rekursiv* implementieren, den Aufrufstack selbst verwalten und immer zuerst mit dem Datenteil weitermachen, der weniger Elemente hat (damit ist garantiert, dass der Stack maximal $\log_2 N$ Einträge hat).
  - *Dreiwegzerlegen* für Daten mit vielen gleichen Schlüsseln: kleiner, gleich oder größer als das Pivotelement.

Mergesort

Das Mergesort Verfahren wurde schon 1945 von J. von Neumann vorgestellt.

- Das Verfahren ist rekursiv:
  - Rekursive Halbierung der Daten,
  - die Rekursion endet bei zwei Elementen,
    - diese müssen nun entweder vertauscht werden oder nicht.
  - Rekursive Vereinigung der Teildaten: ..., 4 x Achtel, 2 x Viertel, 1 x Hälften.

- Es sind drei Grundoperationen erforderlich:
  - rekursives Aufteilen der Daten (schon besprochen),
  - zwei Elemente vergleichen und ggf. tauschen (schon besprochen),
  - *vereinen von zwei sortierten Teillisten* ("merging").

Bildquelle: Sedgewick.

Mergesort

Vereinen von zwei sortierten Eingabelisten zu einer sortierten Ausgabeliste (sog. "two-way merging").

- Eine Methode, zwei sortierte Input-Listen mit N_1 bzw. N_2 Elementen zu einer sortierten Output-Liste mit $N = N_1 + N_2$ Elementen zu vereinen, lässt sich direkt angeben (z.B. bei *aufsteigender* Sortierung, o.B.d.A.):
  - Wir vergleichen das kleinste Elemente aus jeder der beiden Input-Listen miteinander.
  - Wir fügen das kleinere dieser beiden zur Output-Liste.
  - Wir nehmen aus der Liste, aus der zum Output hinzugefügt wurde, das nächste Element (da die beiden Input-Listen sortiert sind, ist das jeweils kleinste Element das nächste noch nicht zum Output hinzugefügte) und vergleichen.
  - Wir wiederholen die vorigen Schritte so lange, bis beide Input-Listen abgearbeitet sind.
- Die Implementierung ist einfach.
- Die Laufzeit ist linear $O(N)$
  - wenn sequentieller Elementzugriff konstant $O(1)$ ist.

Mergesort

Es wird i.Allg. zusätzlicher Speicherplatz für den Output benötigt.

- In dem beschriebenen Verfahren zum Vereinen zweier sortierter Inputs muss für den Output zusätzlich Speicherplatz zur Verfügung stehen.
- D.h. im Programm wird Speicherplatz für $2N$ Elemente benötigt:
  - $N_1 + N_2 = N$ Elemente aus den beiden Inputs,
  - N Elemente für den Output.
- Die Merge-Operation braucht also $O(N)$ zusätzlichen Speicherplatz.
- Es scheint auf den ersten Blick kein Problem zu sein, ein solches Merge-Verfahren auch *in situ* zu programmieren,
  - d.h. ohne den zusätzlichen Speicherplatz, nur mit einem Aufrufstack (dessen Höhe ungefähr logarithmisch mit N wächst).
  - Versuchen Sie es, wenn Ihnen einmal langweilig sein sollte: es sind tatsächlich Merge-Algorithmen beschrieben worden, die in situ und mit $O(N)$ Laufzeit funktionieren, aber sie sind nicht gerade einfach... (etwa Kronrud 1969)

Mergesort

Einige Eigenschaften von Mergesort.

- Mergesort benötigt auf zufälligen Inputs, aber auch im ungünstigsten Fall, nur ca. $N \log N$ Vergleiche (im günstigsten Fall $½\, N \log_2 N$).
  - Man kann sich das prinzipiell gut klar machen, wenn man sich die rekursiven Halbierungen als Entscheidungsbaum vorstellt.
  - Differenzengleichung: $C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + N; \quad C_1 = 1; \quad N \geq 2;$
  - Theoretisch möglich im ungünstigsten Fall sind bestenfalls $O(\, N \log_2 N\,)$ Vergleiche (siehe minimale Anzahl von Vergleichen).
  - Mergesort ist also "theoretisch optimal" selbst im ungünstigsten Fall,
    - im Unterschied zu Quicksort, welches $O(N^2)$ im ungünstigsten Fall benötigt und nur im durchschnittlichen Fall (zufällig gemischte Daten) theor. optimal ist.
    - Ein nach den Regeln der Technik implementiertes Quicksort ist aber in der Praxis ca. doppelt so schnell wie Mergesort und benötigt nur einen ca. halb so großen Stack.
- Mergesort arbeitet stabil (bei korrekter Implementierung),
  - im Unterschied zu Quicksort.
- Das Laufzeitverhalten von Mergesort ist kaum von der Struktur der Eingabedaten abhängig,
  - im Unterschied zu Quicksort.

Heapsort

Die *binäre Halde* (binary heap).

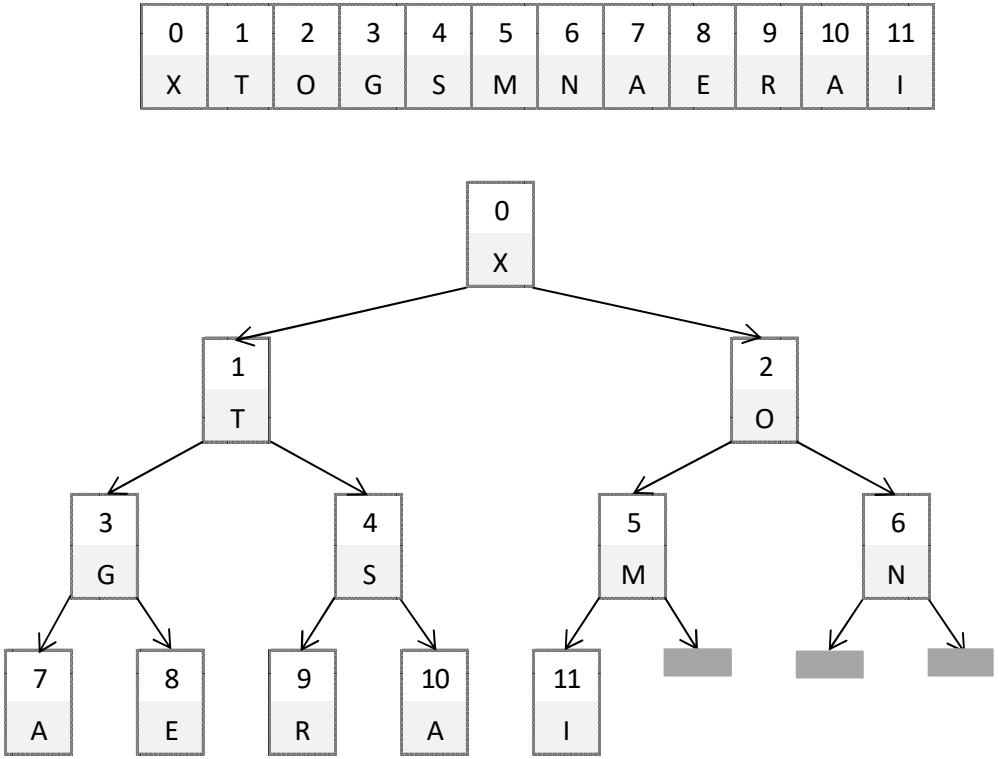
- Eine Halde ist ein Wurzelbaum, der die Halden-Eigenschaft besitzt.
  - *Halden-Eigenschaft:*

    Jeder Knotenwert ist kleiner oder gleich des Knotenwerts in seinem Vorgänger (gleichbedeutend: jeder Knotenwert ist größer oder gleich der Knotenwerte in seinem Teilbaum).
  - Es folgt, dass sich der größte Knotenwert immer in der Wurzel befinden muss.
- Jeder Wurzelbaum kann die Halden-Eigenschaft haben, man nutzt sie für gewöhnlich aber nur auf vollständigen Binärbäumen.
  - Sehen Sie sich ggf. nochmals an, wie der *vollständige Binärbaum* definiert wurde.
  - Anstelle von "vollständiger Binärbaum mit Halden-Eigenschaft" sagt man in der Informatik i.Allg. einfach "binäre Halde", manchmal auch nur "Halde".

Heapsort

Die *binäre Halde* (binary heap).

- Man kann einen vollständigen Binärbaum sehr einfach auch sequentiell als Datenfeld bzw. `vector` implementieren:
  - Die Wurzel kommt an den Index 0,
  - die beiden Nachfolger der Wurzel kommen an die Indizes 1 und 2,
  - die beiden Nachfolger der 1 kommen an die Indizes 3 und 4,
  - die beiden Nachfolger der 2 kommen an die Indizes 5 und 6, usw.
- Vorgänger von i auf $\lfloor (i-1) / 2 \rfloor$,
- Nachfolger von i auf $2i+1$ und $2i+2$.
- Level-Order Prinzip.
- Vollständiger Binärbaum, also auch keine Lücken im Datenfeld bzw. `vector`.



Heapsort

Grundoperation auf binären Halden: beliebigen Wert als neuen Knoten einfügen.

- Ein neuer Knoten wird bei einer binären Halde zunächst einfach ans Ende der Halde angehängt (höchste Indexposition + 1).
- Ist nun der Wert des neuen Knotens größer als der seines Vorgängers, so ist die Halden-Eigenschaft verletzt und wird wieder hergestellt, indem der Knoten mit seinem Vorgänger vertauscht wird.
- Ist nach dem Tausch wieder der Vorgänger größer, ist die Halden-Eigenschaft immer noch verletzt und wird wieder hergestellt, indem der Knoten abermals mit seinem Vorgänger vertauscht wird.
- Das wiederholt sich so lange, bis der neue Knoten kleiner oder gleich seinem Vorgänger ist (oder die Wurzel geworden ist): man nennt den Vorgang oft "*Bottom-up Heapify*".
- Quellcode-Skizze:

```
void bu_heapify( vector<T>& vV, int k ) {
   while( k > 0 && vV[(k-1)/2] < vV[k] ) {
      swap( vV[k], vV[(k-1)/2] ); k = (k-1)/2;
   }
}
```

Heapsort

Grundoperation auf binären Halden: Wurzel (größter Wert) entfernen.

- Die Wurzel wird entfernt, indem sie mit dem Knoten am Ende der Halde (dem auf der höchsten Indexposition) überschrieben wird (der Knoten am Ende der Halde wird danach einfach gelöscht oder "genullt").
- Ist nun der Wert der neuen Wurzel kleiner als der eines Nachfolgers (oder beider Nachfolger), so ist die Halden-Eigenschaft verletzt und wird wieder hergestellt, indem der Knoten mit dem (größeren) Nachfolger vertauscht wird.
- Das wiederholt sich so lange, bis der Knoten größer oder gleich seiner beiden Nachfolger ist (oder ein Blatt geworden ist): man nennt den Vorgang oft "*Top-down Heapify*".
- Quellcode-Skizze:

```
void td_heapify( vector<T>& vV, int k, int N ) {
   int j{};
   while( (2*k)+1 <= N ) {
      j = (2*k)+1;
      if( j<N && vV[j]<vV[j+1] ) j++;
      if( !(vV[k] < vV[j]) ) break;
      swap( vV[k], vV[j] ); k = j;
   }
}
```

Heapsort

Ergebnis: sog. Prioritäts-Warteschlange.

- Mit der binären Halde und den beiden Grundoperationen haben wir bereits eine sog. *Prioritäts-Warteschlage* implementiert.
- Wir können beliebige Knotenwerte in die Warteschlange einfügen, `bu_heapify()` stellt sicher, dass die Halden-Eigenschaft erhalten bleibt.
- Verlässt der höchswertige Knoten (die Wurzel) die Warteschlange, stellt `td_heapify()` sicher, dass die Halden-Eigenschaft erhalten bleibt.
- Der Knoten mit dem größten Wert (der höchsten Priorität) findet sich immer an der Wurzel: die Suche nach dem größten Wert ist keine eigentliche Suche mehr und läuft $O(1)$ konstant.
- Einfügen eines beliebigen Knotens und Entfernen der Wurzel ist jeweils, wegen der Erhaltung der Halden-Eigenschaft, bei N Knoten $O(\log_2 N)$ komplex.
  - Das kann man sich klar machen, wenn man sich vorstellt, dass bei einem *heapify* ein Knoten auf- bzw. abwärts durch den vollständigen Binärbaum wandert.
- Mit solch einem Mechanismus bietet sich jetzt ein nahe liegendes Sortierverfahren an...

Heapsort

Es bietet sich ein jetzt nahe liegendes Sortierverfahren an.

- Heapsort ordnet die Daten in zwei Durchläufen:
  - 1. Durchlauf: aus den unsortierten Daten wird eine binäre Halde aufgebaut.
  - 2. Durchlauf: es wird sukzessiv der jeweils größte Wert (Wurzel) aus der Halde ausgelesen, bis alle Werte gelesen sind.
- Binäre Halde aufbauen (nicht stabil):
  - Statt die unsortierten Daten einfach nacheinander ans Ende vom Datenfeld / `vector` zu bringen und jeweils `bu_heapify()` aufzurufen kann man die binäre Halde auch effizienter aufbauen:
  - Man befüllt das Datenfeld / den `vector` zuerst komplett, Level-Order der Reihe nach mit den unsortierten Daten, und verwendet dann **`td_`**`heapify()` von unten nach oben auf allen Teilhalden.
  - Da man mit `td_heapify()` die letzte Ebene nicht zu betrachten braucht (die Knoten ohne Nachfolger, das können bis zu $\lfloor N/2 \rfloor+1$ der N Knoten sein), beginnt man am mittleren Index, auf der vorletzten Ebene, d.h. man muss weniger oft die Halden-Eigenschaft wiederherstellen, und die Teilhalden sind auch kleiner...
- Maxima auslesen, dabei die Wurzel *nicht* überschreiben sondern mit dem Knoten an der höchsten noch nicht behandelten Position vertauschen.
- Maximal $2N \log_2 N$ Vergleiche auch im ungünstigsten Fall ("theor. optimal"), und es wird kein zusätzlicher Speicher benötigt.

Einfache vs. fortgeschrittene Sortierverfahren

Die Leistungsunterschiede sind deutlich.

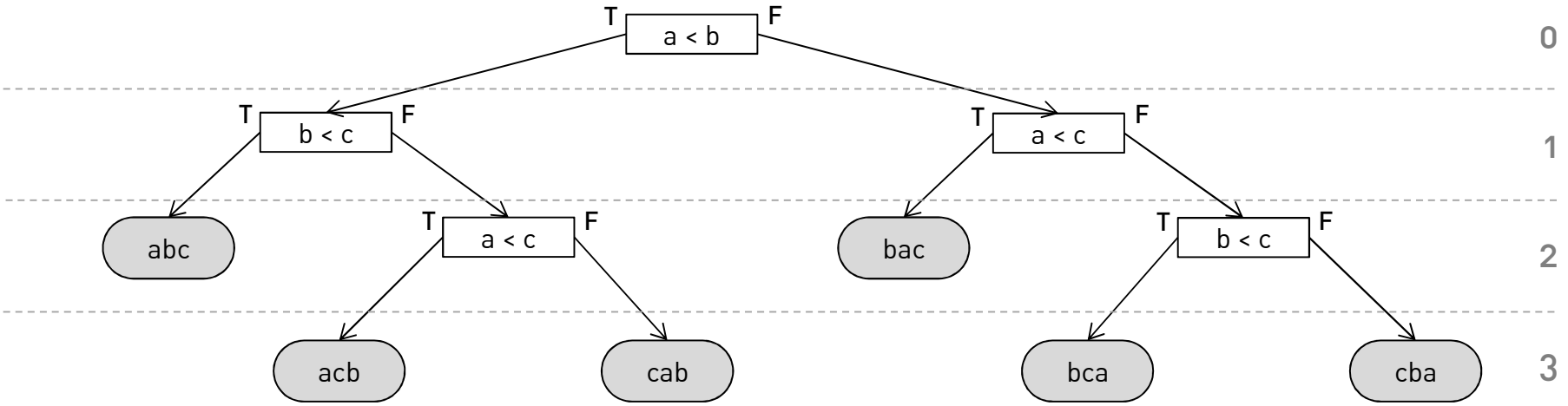
	Hardware 1	Hardware 2
Annahme: Vergleiche pro Sekunde	10^8	10^{12}
▶ Direkte Einfügung, ca. Laufzeit		
▪ 1'000 Elemente	keine	keine
▪ 1'000'000 Elemente	2,8 Stunden	1 Sekunde
▪ 1'000'000'000 Elemente	317 Jahre	1,6 Wochen
▶ Quicksort, ca. Laufzeit		
▪ 1'000 Elemente	keine	keine
▪ 1'000'000 Elemente	0,3 Sekunden	keine
▪ 1'000'000'000 Elemente	6 Minuten	keine

▶ Fazit: gute Algorithmen bringen mehr als zehntausendmal schnellere Hardware

Die minimale Anzahl von Vergleichen

Sortieren: minimale Anzahl von Vergleichsoperationen im ungünstigsten Fall.

- Jeder Sortieralgorithmus, der über paarweise Vergleiche der zu sortierenden Elemente arbeitet, benötigt bei N Elementen im ungünstigsten Fall mindestens $\lceil \log_2 (N!) \rceil$ Vergleiche.
- Man kann sich das wie folgt klar machen:
  - paarweise Vergleiche als Knoten eines binären Baums,
  - $N!$ Permutationen als Blätter, d.h. externe Knoten,
  - minimale Höhe mit den externen Knoten (ausgeglichener Binärbaum) dann $\lceil \log_2 (N!) \rceil$.
- Z.B. im einfachen Fall von $N = 3$ Elementen $\{ a,b,c \}$, $\log_2 (3!) \approx 2{,}585$:



Exkurs: $N!$ für große N

Es gibt eine gute Näherungsformel.

- Für große N kann man $N!$ so bestimmen
  (J. Stirling, Methodus Differentialis, 1730):

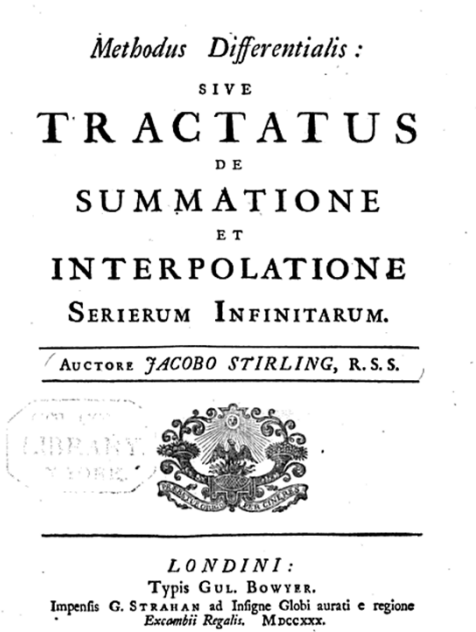
$$N! \approx \sqrt{2\pi N}\left(\frac{N}{e}\right)^N$$

$$\log_2\left[\sqrt{2\pi N}\left(\frac{N}{e}\right)^N\right] = \log_2\sqrt{2\pi N} + \log_2\left[\left(\frac{N}{e}\right)^N\right]$$

$$= \frac{1}{2}\log_2(2\pi N) + N\log_2\left(\frac{N}{e}\right) =$$

$$= \frac{1}{2}\log_2(2\pi) + \frac{1}{2}\log_2 N + N\log_2 N - N\log_2 e$$

$$= \left(\frac{2N+1}{2}\right)\log_2 N - N\underbrace{\log_2 e}_{\sim 1{,}4427} + \underbrace{\frac{1}{2}\log_2(2\pi)}_{\sim 1{,}3257}$$


Methodus Differentialis :
SIVE
TRACTATUS
DE
SUMMATIONE
ET
INTERPOLATIONE
SERIERUM INFINITARUM.
AUCTORE JACOBO STIRLING, R. S. S.
LONDINI:
Typis GUL. BOWYER.
Impensis G. STRAHAN ad Insigne Globi aurati e regione Excambii Regalis. MDCCXXX.

Bildquelle: public domain

LEERE SEITE

Vergleicher und Sortiernetze

Sortiernetze sind Sortiermethoden, die unabhängig von den Werten der zu sortierenden Elemente immer die selben Vergleiche durchführen.

- Sortiernetze sind *nicht-adaptive* Methoden, d.h. der Programmablauf hängt nicht von den Elementwerten und ihrer Verarbeitungsreihenfolge ab (aber von der maximalen Anzahl der zu sortierenden Elemente).
- Man setzt in Sortiernetzen nur eine Art von Operation ein: vertausche x und y falls x größer als y ist.
  - `void swap_if( T x, T y );`
  - Vorteile: einfach zu parallelisieren, als Hardware umsetzbar.
- Sortieren bedeutet (auch für nicht-adaptive Verfahren) eine beliebige Folge $a = a_0, a_1, \ldots, a_{n-1}$ mit n Elementen so umzuordnen, dass $a_i \leq a_j$ für $i < j$.
  - Etwas formaler ausgedrückt: es wird die Indexmenge $\{ 0, 1, \ldots, n-1 \}$ der $a_i \in A$ so permutiert, dass $a_{\varphi(i)} \leq a_{\varphi(j)}$ für alle i, j und $i < j$, wobei φ eine Permutation der Indexmenge ist.
  - Die Ordnungsrelation $\leq$ muss auf der Menge A der Folgenelemente definiert sein (`operator<=()`).

Vergleicher und Sortiernetze

Vergleicher kann man formal als Abbildung definieren, die auf eine Elementfolge angewandt werden.

- Bezeichnet man mit A^n die Menge aller Folgen mit n Elementen aus A, kann ein Vergleicher $[i{:}j]$ formal definiert werden als Abbildung
- $[i,j] : A^n \rightarrow A^n$ mit $i, j \in \{ 0, ..., n{-}1 \}$
  - Dabei gilt für alle $a \in A^n$

$$[i,j](a)_i = \min(a_i, a_j)$$

$$[i,j](a)_j = \max(a_i, a_j)$$

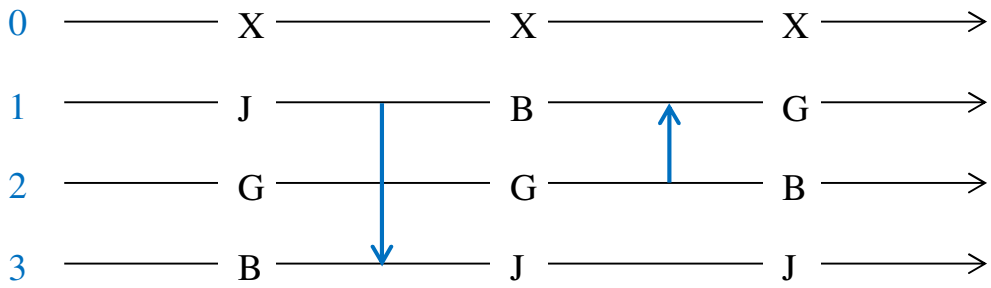
$$[i,j](a)_k = a_k \text{ falls } k \neq i,\ k \neq j$$

- D.h.: der Vergleicher $[i,j]$ bringt das i-te und das j-te Element einer Folge in aufsteigende Reihenfolge (und macht ansonsten nichts).

Vergleichernetze

Vergleichernetze lassen sich grafisch anschaulich und intuitiv darstellen.

- Aus Vergleichern kann man Vergleichernetze zusammensetzen, ein Vergleichernetz $V = [i_1, j_1]\cdot[i_2, j_2]\cdot \ \ldots \ \cdot[i_m, j_m]$ ist eine definierte Abfolge von Vergleichern.
- Vergleichernetze können grafisch so dargestellt werden:

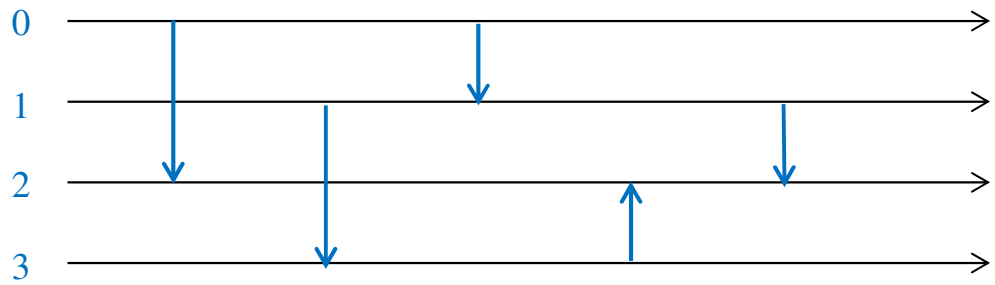


  - Man stellt sich vor, dass die Elementfolgen von links nach rechts entlang der waagerechten Linien laufen.
  - Elemente, die dabei auf einen Vergleicher treffen, werden in Pfeilrichtung verglichen und ggf. vertauscht.
  - Im Beispiel trifft die Folge X-J-G-B erst auf den Vergleicher [1,3] und dann auf den Vergleicher [2,1].

Vergleichernetze

Vergleichernetze sind Vergleicher, die hintereinander ausgeführt werden.

- Weiteres Beispiel, das Vergleichernetz $[0,2]\cdot[1,3]\cdot[0,1]\cdot[3,2]\cdot[1,2]$:



- Zwei Vergleicher $[i, j]$ und $[k,l]$ sind *unabhängig,* wenn $[i, j]\cdot[k,l] = [k,l]\cdot[i, j]$ (d.h. wenn sie im Diagramm keine waagerechte Linie gemeinsam haben).
- Unabhängige Vergleicher, die hintereinander ausgeführt werden, können beliebige Reihenfolge haben (d.h. auch parallel laufen), man spricht von einer *Vergleicherstufe*.
- Zwei Vergleichernetze V und V' heißen *äquivalent*, wenn sie jede Eingabefolge in die selbe Ausgabefolge überführen:
  $V = V' \Leftrightarrow \forall a : V(a) = V'(a)$

Sortiernetze

Ein Sortiernetz ist ein Vergleichernetz, das alle Eingabefolgen sortiert.

- Die beiden Beispiele sind keine Sortiernetze.
  - Das erste Vergleichernetz setzt für vier Elemente nur zwei Vergleiche ein, es sind aber im ungünstigsten Fall mindestens $\lceil \log_2 4! \rceil = 5$ Vergleiche zum Sortieren von vier Elementen erforderlich.
  - Beim zweiten Vergleichernetz gibt es Gegenbeispiele, es wird z.B. T-E-A-M nicht richtig sortiert.
- Ob ein bestimmtes Vergleichernetz ein Sortiernetz ist, lässt sich allgemein nicht leicht beantworten.
  - ("NP-schwere" Frage, d.h. es ist kein deterministisches, polynomial komplexes Lösungsverfahren bekannt – und es ist fraglich, ob man jemals eines finden wird.)
- Es lassen sich aber Sortiernetze systematisch konstruieren und beweisen,
  - z.B. sind Sortiernetze für Mergesort (Batcher 1968) oder für Shellsort (Pratt 1972) beschrieben worden.

Sortiernetze

Das 0-1-Prinzip für Sortiernetze.

- Das 0-1-Prinzip (nach D. Knuth) lautet:

  Ein Vergleichernetz, welches alle Folgen von Nullen und Einsen sortiert, ist ein Sortiernetz.

  - Ein solches Vergleichernetz sortiert auch alle Folgen von *beliebigen* Werten.

- Daraus folgt u.a.:

  Ob ein Vergleichernetz ein Sortiernetz ist oder nicht, hängt *nur von seiner Struktur* ab, und nicht von der Eingabe.

- Beweis vgl. z.B. bei D. Knuth.

  - (Der Beweis ist nicht kompliziert, zieht sich aber, weil mehrere Definitionen und Hilfssätze gebraucht werden...)

LEERE SEITE

Indexsortieren

Die Grundidee der direkten Schlüsselindizierung.

- Unter bestimmten Bedingungen lassen sich Schlüsselwerte als Indizes für Speicherstrukturen verwenden.
  - Wir haben dieses Prinzip schon bei den Streuwert-Tabellen angesprochen.
  - Vorteil u.a.: man braucht geeignete Schlüssel nicht mehr durch abstrakte Operationen ggf. aufwändig untereinander vergleichen.
- Das bekannte Lehrbeispiel: "Sortieren Sie N Elemente, deren Schlüssel unterschiedliche Ganzzahlen zwischen 0 und $N–1$ sind".
  (Kommt in der Praxis so wohl kaum vor, verdeutlicht aber besonders klar die Grundidee...)
  - Ist zusätzlicher Speicherplatz für eine Kopie der N Elemente verfügbar, kann man das Problem $O(N)$ zeiteffizient mit einem einzigen Schleifendurchlauf lösen:

```
for( int i{}; i < N; ++i ) sorted_copy[ data[i] ] = data[i];
// z.B. wuerde 1,4,3,0,2 so sortiert:
// data[0] ist 1, sorted_copy[data[0]] wird 1 (d.h. sorted_copy[1]==1)
// data[1] ist 4, sorted_copy[data[1]] wird 4 (d.h. sorted_copy[4]==4)
// data[2] ist 3, sorted_copy[data[2]] wird 3 (d.h. sorted_copy[3]==3)
// data[3] ist 0, sorted_copy[data[3]] wird 0 (d.h. sorted_copy[0]==0)
// data[4] ist 2, sorted_copy[data[4]] wird 2 (d.h. sorted_copy[2]==2)
```

- Man kann die Methode als *direkte Schlüsselindizierung* bezeichnen.

Indexsortieren

Indextabellen und Indexsortierung.

- Beobachtung bei der direkten Schlüsselindizierung:
  - Die ursprünglichen Daten bleiben unsortiert, es werden aber zusätzlich sortierte Indizes gespeichert.
- Diese Grundidee ist recht nützlich.
  - Man kann Schlüsselwerte als Indizes interpretieren und diese Indizes abspeichern (wie im vorigen, illustrierenden Beispiel).
  - Man kann dann die indiziert abgespeicherten Indizes weiter verarbeiten und nutzen.
- Man spricht vom *Indexsortieren*, falls man statt der Ursprungsdaten die Indextabelle(n) sortiert,
  - bzw. vom *Zeigersortieren*, falls statt ganzzahliger Indizes unmittelbar die Speicheradressen zur Sortierung abgespeichert sind.
- Man spricht bei indiziert gespeicherten Indizes von *Indextabellen*.
  - Ein wichtiger Vorteil von Indextabellen: man kann damit das laufende Umkopieren zu sortierender Elemente während einer Sortierung vermeiden,
    - etwa weil die Elemente groß sind und Kopieroperationen daher aufwändig.

Indexsortieren

Die Vorgehensweise beim Indexsortieren (bzw. Zeigersortieren).

- Ausgangslage: es befinden sich N zu sortierenden Datenelemente an den Indizes 0 bis $N–1$ in einer Speicherstruktur (z.B. namens `data`), die den indizierten Zugriff (bzw. Zeigerzugriff) auf ihre Elemente ermöglicht.
- Man legt eine geeignete Indextabelle der Größe N an (z.B. namens `idx`) und initialisiert sie mit den Indizes (bzw. Adressen) der Elemente:
  - `idx[0]` wird auf den Index (bzw. die Adresse) des *ersten* Elements in `data` gesetzt (d.h. `data[0]`, der erste Index ist `0`),
  - `idx[1]` wird auf den Index (bzw. die Adresse) des *zweiten* Elements `data[1]` gesetzt,
  - usw. bis `idx[N–1]`.
- Ziel der Indexsortierung (bzw. der Zeigersortierung):
  - `idx[0]` soll am Ende den Index (bzw. die Adresse) des Elements mit dem *kleinsten* Schlüsselwert aus `data` enthalten,
  - `idx[1]` soll am Ende den Index (bzw. den Adresse) des Elements mit dem *zweitkleinsten* Schlüsselwert aus `data` enthalten,
  - usw.

Indexsortieren

Die Vorgehensweise beim Indexsortieren (bzw. Zeigersortieren).

- Zur Ermittlung der richtigen Sortierreihenfolge in `idx` wird in den Vergleichsoperationen während des Sortiervorgangs *lesend* auf die Werte in `data` zugegriffen.
- Die ursprünglichen Daten werden nicht bewegt, nur ihre Indizes in der Indextabelle werden umgeordnet.
  - Nicht die Elemente in `data` werden sortiert, sondern die Elemente in `idx`.
  - Auch schreibgeschützte Daten können so sortiert werden.
  - Es kann gleichzeitig mehrere Indextabellen geben,
    - falls z.B. die ursprünglichen Daten nach mehreren Schlüsseln sortiert werden müssen.
- Nach der Sortierung der Indextabelle ist der Zugriff auf die ursprünglichen Elemente in `data` in sortierter Weise möglich, indem man indirekt über die sortierte Indextabelle zugreift.
  - Sog. *Indirektion*, man kann sich vorstellen, dass man "durch die Indextabelle hindurch" zugreift, wodurch die Zugriffe umgeleitet werden.

Indexsortieren

Das Programm zeigt, wie Sortieren durch direkte Einfügung auf Indextabellen implementiert werden kann.

```
template<class T>
void insSortidx( const vector<T>& data, int ui, int oi, vector<int>& idx ) {
   int i{}, j{}, tmp{};
   for( i=ui; i<=oi; ++i ) idx.push_back(i); // Indextabelle initialisieren
   // Durchlauf vorab fuer das kleinste Element:
   for( i=oi; ui<i; --i ) {
      if( data[idx[i]] < data[idx[i-1]] ) swap( idx[i], idx[i-1] );
   }
   // sortieren der Indextabelle durch direkte Einfuegung:
   for( i=ui+2; i<=oi; ++i ) {
      tmp = idx[i];
      for( j = i; data[tmp] < data[idx[j-1]]; --j ) idx[j] = idx[j-1];
      idx[j] = tmp;
   }
}
```

- Dieses Programm greift auf die ursprünglichen Daten nur zu, um je zwei Schlüsselwerte zu vergleichen,
- daher kann es leicht für beliebige andere Container als `vector` angepasst werden.

Indexsortieren mit der StdLib

Man kann Indexsortierung recht einfach auch für Container und Algorithmen der StdLib umsetzen.

- Beispiel `std::vector` Container, `std::sort` Algorithmus, und Container-Elemente vom selbst definierten Typ `Zyx`:

```
// Daten im vector Container der StdLib
std::vector<Zyx> dat{};
// Container fuellen...

// Indextabelle vorbereiten
std::vector<size_t> ind( dat.size() );
// Indextabelle initialisieren...

// entweder mit Funktor nach Indizes sortieren
std::sort( ind.begin(), ind.end(), idxComp_vector<Zyx>{ dat } );

// oder alternativ mit Funktionsadapter std::bind nach Indizes sortieren
std::sort( ind.begin(), ind.end(),
           std::bind( compare<Zyx>,
                      std::placeholders::_1,
                      std::placeholders::_2,
                      dat ) );

// sortierte Ausgabe durch Indirektion
idxPrint_vector( ind, dat );
```

Indexsortieren mit der StdLib

Man kann Indexsortierung recht einfach auch für Container und Algorithmen der StdLib umsetzen.

- Beispiel `std::vector` Container, `std::sort` Algorithmus, und Elemente vom selbst definierten Typ `Zyx`:

```
// Templatefunktion fuer indirekte Ausgaben der Container-Elemente
using citer = std::vector<size_t>::const_iterator;
template<class T> void idxPrint_vector( const std::vector<size_t>& idx,
                                        const std::vector<T>& data ) {
  for( citer i{ idx.begin() }; i != idx.end(); ++i )
    std::cout << data[*i] << std::endl;
}

// Definition des Funktors als dritter Parameter fuer std::sort
template<class T> class idxComp_vector {
  public:
    idxComp_vector( const std::vector<T>& v ) : dataref{ v } {}
    bool operator() ( const size_t& first, const size_t& second ) const {
      return dataref[first].comp_value( dataref[second] );
    }
  private:
    const std::vector<T>& dataref;
};
```

Indexsortieren mit der StdLib

Man kann Indexsortierung recht einfach auch für Container und Algorithmen der StdLib umsetzen.

- Beispiel `std::vector` Container, `std::sort` Algorithmus, und Elemente vom selbst definierten Typ `Zyx`:

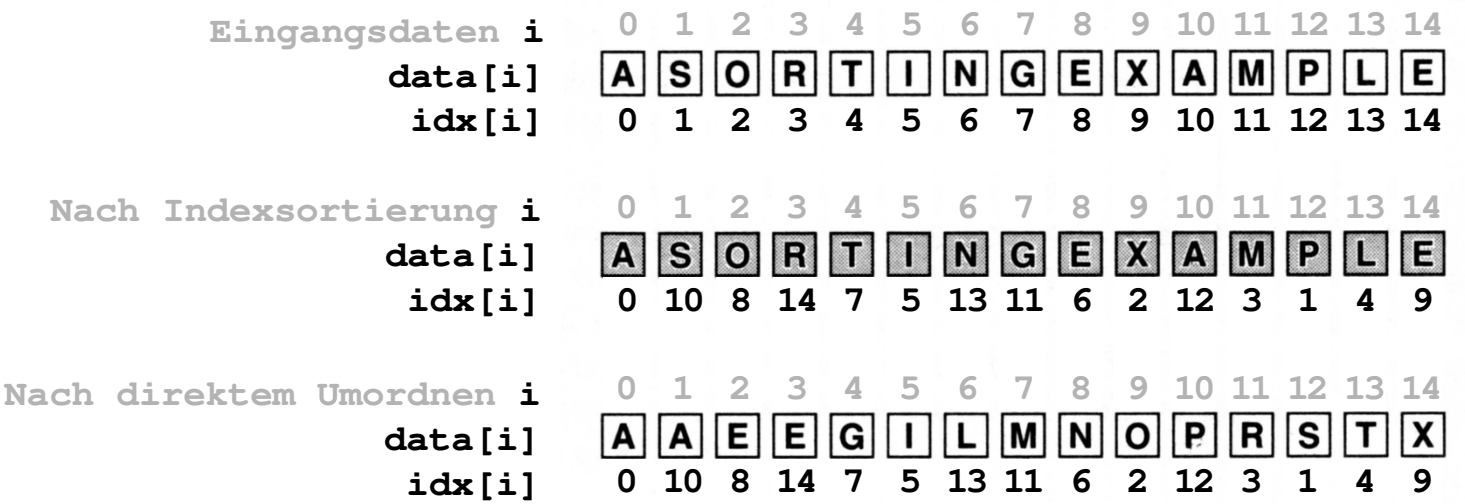
```
// Vergleichsfunktion fuer den std::bind Funktionsadapter
template<class T> bool compare( const size_t& first, const size_t& second,
                                const std::vector<T>& data )
  { return data[first].comp_name( data[second] ); }

// Skizze fuer den beispielhaften Elementtyp namens Zyx
class Zyx {
    std::string name;
    double value;
    //…
  public:
    //…
    bool comp_name(  const Zyx& other ) const { return name  < other.name; }
    bool comp_value( const Zyx& other ) const { return value < other.value; }
    //…
    friend ostream& operator<<( ostream& os, const Zyx& z )
      { return os << z.name << ' ' << z.value /*…*/ ; }
    //…
};
```

Indexsortieren

Direktes Umordnen.

- Was ist zu tun, wenn umfangreiche Daten, die über eine Indextabelle geordnet zugreifbar sind, auch tatsächlich selbst umsortiert werden sollen, aber kein Speicherplatz zur Verfügung steht?
  - Eine klassische Programmierübung:
    "Sortieren Sie N Elemente nach einer Indextabelle um, *ohne* zusätzlichen Speicherplatz für eine Kopie der Daten zu verwenden".
  - Einfach `data[i] = data[idx[i]]` geht nicht, da man damit Werte überschreibt.



Bildquelle: Sedgewick.

Indexsortieren

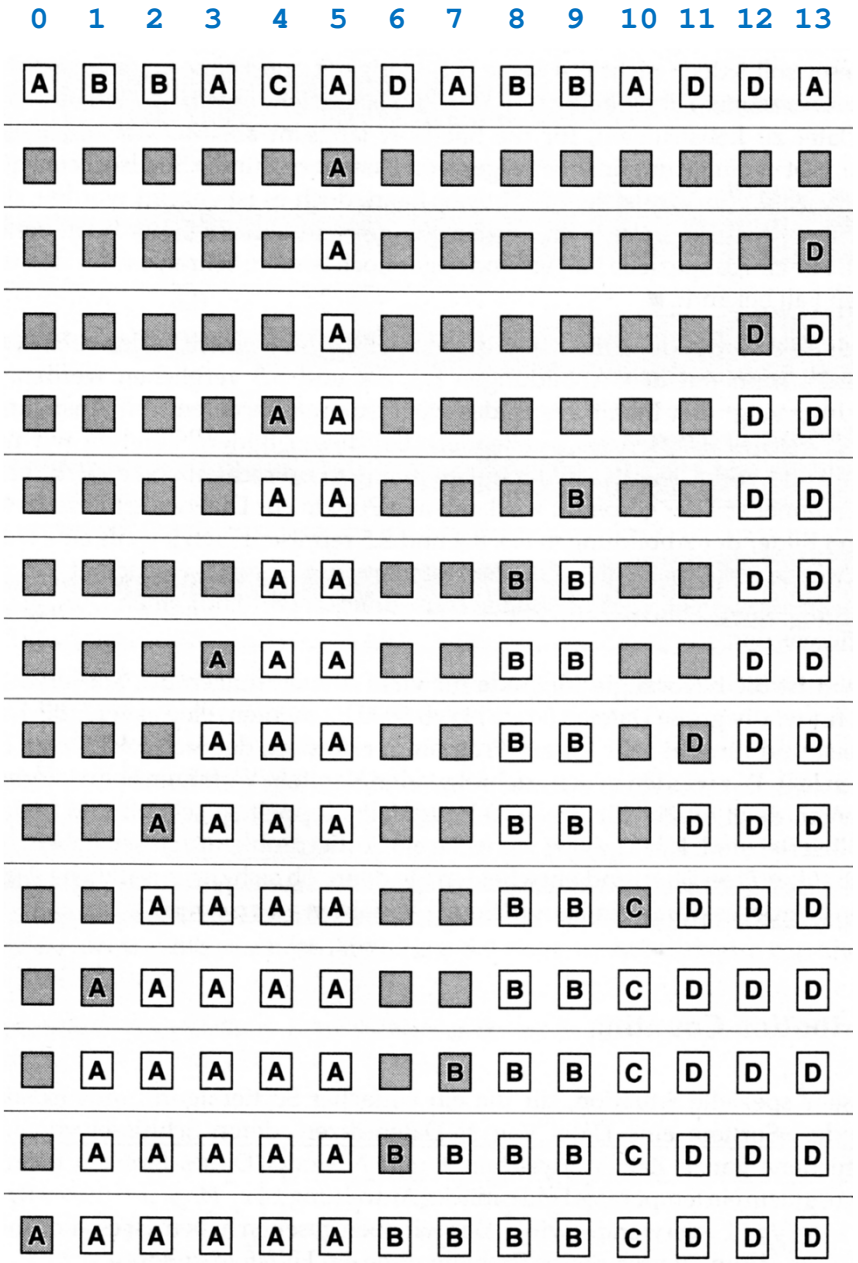
Vorgehensweise beim direkten Umordnen.

- Um das Element an der ersten Position zu seiner richtigen Position zu verschieben, muss man *vorher* das Element, das sich an dieser Zielposition befindet und sie blockiert, zu *seiner* richtigen Position verschieben.
- Bevor man dies tun kann, muss man wiederum vorher das Element, das diese zweiten Zielposition blockiert, an dessen richtige Position bringen.
- Usw. bis man schließlich auf das Element trifft, welches an die erste Position verschoben werden muss,
  - womit ein geschlossener Zyklus von Verschiebungen vorliegt, der ohne Überschreibung von Werten ausgeführt werden kann.
- Das wiederholt sich mit dem zweiten Element und seinem Zyklus usw.
- Elemente mit `data[i]==i` sind bereits an der richtigen Position (d.h. auf einem Zyklus der Länge eins) und werden ignoriert.
- Ein solches Sortieren wird auch "direktes Umordnen" genannt.

Verteilungszählen

Veranschaulichung.

- Die Eingangsdaten mit $N = 14$ Elementen und $M = 4$ Fächern (Schlüsselwerten):
  `{A,B,B,A,C,A,D,A,B,B,A,D,D,A}`
  - Sechs mal `A`,
    d.h. `A` an den Indizes `0` bis `5`
  - Vier mal `B`,
    d.h. `B` an den Indizes `6` bis `9`
  - Ein `C` am Index `10`
  - Drei mal `D`,
    d.h. `D` an den Indizes `11` bis `13`



Bildquelle: Sedgewick.

Verteilungszählen

Das sog. Verteilungszählen (engl. "distribution counting") ist die Grundidee für digitales Fachverteilen.

- Beim Verteilungszählen bestimmt man erst die Anzahl der Schlüssel pro Wert.
- Die Ergebnisse dieser Zählung verwendet man, um in einem zweiten Durchlauf die Daten richtig zu positionieren.
- Eine Implementierung:

```
void dcount( int M, int* data, int ui, int oi ) {
   // dyn. Zaehlerfeld, alles mit 0 initialisiert:
   int* counter = new int[M]{};
   // Anzahl der Schluesselwerte zaehlen:
   for( int i{ui}; i<=oi; ++i ) { counter[data[i]] += 1; }
   // Anzahlen kumulieren:
   for( int j{1}; j<M; ++j ) { counter[j] += counter[j-1]; }
   // dyn. Hilfsfeld zum stabilen Umordnen initialisieren:
   int* N = new int[oi-ui+1]{};
   // umsortieren (sehen Sie sich die folgende Quellcodezeile genau an):
   for( int i{ui}; i<=oi; ++i ) N[ --counter[data[i]] ] = data[i];
   // sortiert zurueckkopieren und dyn. Hilfsspeicher freigeben
   for( int i{ui}; i<=oi; ++i ) data[i] = N[i];
   delete[] counter; counter = nullptr;
   delete[] N; N = nullptr;
}
```

Verteilungszählen

Einige Eigenschaften.

- Verteilungszählen sortiert N Elemente, von denen bekannt ist, dass die Schlüssel Ganzzahlen zwischen 0 und $M-1$ sind*, in $O(N)$ Laufzeit.
  - Jedes Element wird zweimal bewegt: das erste Mal, um es zu verteilen, und das zweite Mal, um es zurück zu kopieren.
  - Jeder Schlüssel wird zweimal gelesen: das erste Mal, um die Anzahl der Schlüsselwerte zu ermitteln, und das zweite Mal, um die Verteilung korrekt durchzuführen.
  - Die anderen Schleifen ermitteln die Zählerwerte und tragen nicht wesentlich zur Laufzeit bei (solange die Anzahl der Zähler* nicht die Anzahl der Daten überschreitet).
- Es wird eine Hilfstabelle der Größe M für die Zählergebnisse benötigt.
- Es wird eine weitere Hilfstabelle der Größe N für das stabile Umordnen der Daten benötigt.
  - Diese Hilfstabelle kann bei großen Datenmengen problematisch werden, eine mögliche Lösung wäre dann der Einsatz des bereits besprochenen direkten Umordnens.
  - Da direktes Umordnen *nicht* stabil ist, ist das Verteilungszählen mit direktem Umordnen ebenfalls nicht mehr stabil.

* M darf "nicht zu groß" sein.

Verteilungszählen

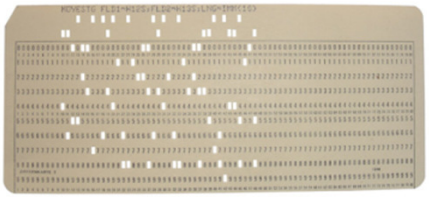
Verteilungszählen geht von einem eingeschränkten Wertebereich der Schlüssel aus.

- Beobachtung beim Verteilungszählen:
  - Man geht davon aus, dass die Sortierschlüssel aus einem beschränkten, vorher bekannten Wertebereich stammen, und nutzt die Vorteile aus, die sich daraus für das Sortieren ergeben.
- Diese Grundidee kann überaus nützlich sein...
  - Die auf einem Computer normal darstellbaren Werte sind durch die Rechnerarchitektur beschränkt, deshalb kann die Annahme eines beschränkten Zahlenbereichs für Sortierschlüssel bei bestimmten Anwendungen in der Informatik sehr vernünftig sein.
  - In Computern sind Binärzahlen (mit den Ziffern `0` und `1`) in beschränkten Zahlenbereichen (die von der Systemarchitektur gegeben sind) dargestellt.
  - Man hat entsprechende Sortierverfahren entwickelt, die genau diese Eigenschaften ausnutzen, man spricht von *digitalen Sortierverfahren* (engl. "radix sorts").

Digitales Fachverteilen

Digitales Fachverteilen ist eine nahe liegende Verallgemeinerung des Verteilungszählens.

- Digitale Sortierverfahren beruhen nicht mehr auf abstrakten Vergleichsoperationen zwischen Werten aus Schlüsseltypen, sondern verarbeiten direkt die Bits aus der Binärdarstellung der Schlüssel.
- Man unterscheidet zwei grundlegende Ansätze.
  - MSD-Verfahren, die zuerst die höchstwertigen Bits verarbeiten (MSD: most significant digit), d.h. "von links nach rechts" durch die Binärziffern gehen,
  - LSD-Verfahren, die zuerst die niederwertigsten Bits verarbeiten (LSD: least significant digit), d.h. "von rechts nach links" durch die Binärziffern gehen.
    - Digitales Fachverteilen ist ein LSD-Verfahren.
    - Digitales Fachverteilen ist das früher von den Lochkartenmaschinen verwendete Verfahren.



- Beide Ansätze beruhen auf dem Verteilungszählen,
  - d.h. man kann w-Bit lange Schlüssel in linearer Laufzeit sortieren, solange man die Hilfstabellen zur Verfügung hat,
    - eine für 2^w "Fächer" (zum Zählen der Schlüsselwerte pro "Fach"),
    - eine weitere für eine Kopie der N Indizes / Schlüssel (zum stabilen Umordnen).

Bildquelle: public domain

Digitales Fachverteilen

Digitales Fachverteilen ist eine nahe liegende Verallgemeinerung des Verteilungszählens.

- Digitales Fachverteilen verwendet die Binärdarstellung der Sortierschlüssel und betrachtet deren einzelne Bits.
  - Die Funktion iteriert, beim *LSD* beginnend, über alle Bitpositionen aus der Binärdarstellung der Schlüsselwerte.
  - In jeder Iteration werden die Elemente per Verteilungszählen in eines von zwei Fächern verteilt,
    - je nachdem, ob die Bitdarstellung des Schüssels an der Stelle, die gerade in der Iteration ist, den Wert 0 oder 1 hat (d.h. `M` ist 2 pro Iteration im digitalen Fachverteilen).
  - Der für das Verteilungszählen vorgestellte Quellcode kann recht einfach entsprechend angepasst werden.

Digitales Fachverteilen

Digitales Fachverteilen benötigt die Binärdarstellung des Schlüsselwerts.

- Eine wichtige Operation für digitales Fachverteilen ist die Extraktion des *LSD* aus einem digital repräsentierten Schlüsselwert.
  - In C++ gibt es gute Möglichkeiten, Zugriff auf die Binärdarstellung von Werten zu erhalten.
    - (Im Unterschied zu vielen anderen Programmiersprachen…)
  - Z.B. über einen `bitset`-Container:

```
bool digit( int value, int position ) {
  return bitset<8*sizeof(int)>(value)[position];
}

bool lsd( int val ) { return digit( val, 0 ); }
```

Digitales Fachverteilen

Veranschaulichung.

Start					
d	0	0	1	0	0
o	0	1	1	1	1
n	0	1	1	1	0
a	0	0	0	0	1
u	1	0	1	0	1
w	1	0	1	1	1
e	0	0	1	0	1
l	0	1	1	0	0
l	0	1	1	0	0
e	0	0	1	0	1

1. Iteration:					
d	0	0	1	0	**0**
n	0	1	1	1	**0**
l	0	1	1	0	**0**
l	0	1	1	0	**0**
o	0	1	1	1	**1**
a	0	0	0	0	**1**
u	1	0	1	0	**1**
w	1	0	1	1	**1**
e	0	0	1	0	**1**
e	0	0	1	0	**1**

2. Iteration:					
d	0	0	1	**0**	0
l	0	1	1	**0**	0
l	0	1	1	**0**	0
a	0	0	0	**0**	1
u	1	0	1	**0**	1
e	0	0	1	**0**	1
e	0	0	1	**0**	1
n	0	1	1	**1**	0
o	0	1	1	**1**	1
w	1	0	1	**1**	1

3. Iteration:					
a	0	0	**0**	0	1
d	0	0	**1**	0	0
l	0	1	**1**	0	0
l	0	1	**1**	0	0
u	1	0	**1**	0	1
e	0	0	**1**	0	1
e	0	0	**1**	0	1
n	0	1	**1**	1	0
o	0	1	**1**	1	1
w	1	0	**1**	1	1

4. Iteration:					
a	0	**0**	0	0	1
d	0	**0**	1	0	0
u	1	**0**	1	0	1
e	0	**0**	1	0	1
e	0	**0**	1	0	1
w	1	**0**	1	1	1
l	0	**1**	1	0	0
l	0	**1**	1	0	0
n	0	**1**	1	1	0
o	0	**1**	1	1	1

5. Iteration:					
a	**0**	0	0	0	1
d	**0**	0	1	0	0
e	**0**	0	1	0	1
e	**0**	0	1	0	1
l	**0**	1	1	0	0
l	**0**	1	1	0	0
n	**0**	1	1	1	0
o	**0**	1	1	1	1
u	**1**	0	1	0	1
w	**1**	0	1	1	1

Digitales Fachverteilen

Einige Eigenschaften.

- LSD Verfahren scheinen allgemein etwas unlogisch zu sein.
  - Sie fangen beim LSD an, d.h. sie verwenden Ressourcen für die Sortierung von Ziffern, die unter Umständen das Endergebnis der Sortierung gar nicht beeinflussen.
- Digitales Fachverteilen basiert auf festen Schlüssellängen und kann daher nicht einfach auf, beispielsweise, unterschiedlich lange Zeichenketten als Schlüssel übertragen werden.
- Für das Sortieren von N Schlüsseln mit w-Bit Länge werden im ungünstigsten Fall maximal wN Bits (alle Bits in allen Schlüsseln) untersucht.
  - Das Verfahren führt über alle N Schlüssel je höchstens w Durchgänge aus,
  - d.h. die Laufzeit des Verfahrens ist linear proportional zur Anzahl der Bits in den Schlüsseln.
  - Diese Eigenschaft folgt direkt aus dem Verfahren: kein Bit muss mehr als einmal betrachtet werden.

Digitales Fachverteilen

Einige Eigenschaften.

- Die Leistung des digitalen Fachverteilens hängt fast gar nicht von den Eingangsdaten ab.
  - Eine bekannte Faustregel aus der Praxis besagt aber, dass digitale Sortierverfahren bei Daten mit vielen gleichen Schlüsselwerten ihre Stärken nicht ausspielen können.
- N Elemente, deren Schlüssel w Bit lang sind, können in w Iterationen sortiert werden, wenn zusätzlich Speicher für die Zähler und für das stabile Umordnen der Elemente verfügbar sind.
  - Allgemein: N Elemente mit w-Bit Schlüsseln können in $w / \log_2 R$ Durchgängen sortiert werden, wenn zusätzlich Speicher für R Zähler und für das stabile Umordnen der Elemente verfügbar sind.
    - Werden nicht nur $R = 2$ Fächer sondern $R = 2^{w/4}$ gewählt, erfolgen im digitalen Fachverteilen vier Durchgänge des Verteilungszählens.
  - Statt einzelner Bits kann man auch besser gleich ganze Bytes extrahieren (was auf vielen Computern sowieso effizienter ist als die Extraktion einzelner Bits).
    - Ist die Rechnerarchitektur z.B. in 32 Bit zu je vier 8-Bit-Bytes organisiert, dann ist das gesamte Verfahren mit vier Durchgängen des Verteilungszählens in linear proportionaler Laufzeit beendet.
  - Es geht aber auch mit nur zwei Durchgängen (*sublineares Sortieren*): verwendet man nur die führenden $w/2$ Bits der w-Bit Schlüssel, sind die Daten danach *fast sortiert*, und man kann den Gesamtvorgang per Sortierung durch direktes Einfügen beenden...

Einige Beispielfragen

Fortgeschrittenes Sortieren.

- Welche Gemeinsamkeiten sehen Sie zwischen Shellsort und dem Sortieren durch direkte Einfügung? Welchen entscheidenden Unterschied sehen Sie?
- Erklären Sie, welche Schwachstelle des Sortierens durch direkte Einfügung von Shellsort behoben wird.
- Was verstehen Sie unter h-sortierten Daten? Geben Sie ein eigenes Beispiel zur Illustration.
- Erklären Sie die Arbeitsweise von Shellsort mit eigenen Worten.
- Welchen Vorteil hat es, dass Shellsort h-sortierte Daten mit immer kleiner werdenden h erzeugt?
- Geben Sie ein Beispiel für eine Abstandsfolge, die für Shellsort ungünstig ist.
- Warum wird in unserem Shellsort Quellcode h mal das kleinste Element innerhalb aller h-sortierten Teildaten an den Anfang (den untersten Index innerhalb der h unterschiedlichen Teile) gebracht? Welcher Vorteil ergibt sich daraus? Welchen Fachbegriff kennen Sie für solchen Elemente?

Einige Beispielfragen

Fortgeschrittenes Sortieren.

- Warum bezeichnet man Quicksort als rekursiven Algorithmus?
- Erklären Sie die Arbeitsweise von Quicksort mit eigenen Worten.
- Welche Invarianten muss die Zerlegung der Daten in zwei Teile bei Quicksort erfüllen?
- Was wissen Sie über das Leistungsverhalten von Quicksort, Mergesort, Heapsort?
- Zeigen Sie Schritt für Schritt, wie Quicksort die Zeichenkette QUICKSORT sortiert.
- Zeigen Sie Schritt für Schritt, wie Mergesort die Zeichenkette MERGESORT sortiert.
- Zeigen Sie Schritt für Schritt, wie Heapsort die Zeichenkette HEAPSORT sortiert.
- Geben Sie ein Sortiernetz an, das drei Elemente mit drei Vergleichen sortiert.

Einige Beispielfragen

Fortgeschrittenes Sortieren.

- Was ist schlüsselindiziertes Zählen?
- Wie sortieren Sie N Elemente, deren Schlüssel unterschiedliche Ganzzahlen zwischen 0 und N-1 sind?
- Was verstehen Sie unter einer Indextabelle?
- Welche praktischen Anwendungsvorteile haben Indextabellen beim Sortieren von Daten? Welche Nachteile sehen Sie?
- Wie sortieren Sie N Elemente nach einer Indextabelle um, ohne zusätzlichen Speicherplatz für eine Kopie der Daten zu verwenden?
- Sortieren Sie (A,B,C,D,E,F,G,H,J) nach der Indextabelle (3,7,1,6,4,9,2,8,5).
- Geben Sie die Indextabelle für die alphabetisch aufsteigende Sortierung der Zeichenkette K L A B A U T E R M A N N an.
- Geben Sie die Zyklen der Zeichenkette K L A_1 B A_2 U T E R M A_3 N_1 N_2 hinsichtlich der Permutation zur alphabetisch aufsteigenden Sortierung an (setzen Sie mehrfach vorkommende Buchstaben systematisch von niedrigeren zu höheren Indizes hintereinander – achten Sie darauf, mehrfach vorkommende Buchstaben nicht zu verwechseln).

Einige Beispielfragen

Fortgeschrittenes Sortieren.

- Erklären Sie die Arbeitsweise des Verteilungszählens an einem *eigenen* Beispiel.
- Was wissen Sie über die Leistungseigenschaften des Verteilungszählens?
- Würden Sie das Verteilungszählen zu den stabilen Sortierverfahren rechnen?
- Welche grundlegenden Ansätze für das digitale Sortieren kennen Sie?
- Was ist der wesentliche Unterschied zwischen den digitalen Sortierverfahren und den "klassischen" Methoden?
- Wie kann man in C++ auf einzelne Bits aus der Binärdarstellung eines Werts zugreifen?
- Welche Leistungseigenschaften des digitalen Fachverteilens kennen Sie?
- Erklären Sie das Funktionsprinzip des digitalen Fachverteilens.
- Was bedeutet "sublineares Sortieren"?

Nächste Einheit:
Grafische Benutzeroberflächen