Einführung in die Informatik

Universität Stuttgart, Studienjahr 2005/06

Gliederung der Grundvorlesung

1. ~~Einführung in die Sprache Ada95~~
2. ~~Algorithmen und Sprachen~~
3. ~~Daten und ihre Strukturierung~~
4. ~~Begriffe der Programmierung~~
5. ~~Abstrakte Datentypen~~ Zurückgestellt
6. ~~Komplexität von Algorithmen und Programmen~~
7. ~~Semantik von Programmen~~
8. ~~Suchen~~
9. ~~Hashing~~
10. Sortieren
11. Graphalgorithmen
12. Speicherverwaltung

Stand: 14.7.06

10. Sortieren / Vollversion

Ziele des 10. Kapitels:

Um Dinge schneller wiederfinden zu können oder um den Überblick zu bewahren, legt man Dokumente in geordneter Form ab. Hierzu muss der Datenbestand sortiert werden.

In diesem Kapitel lernen Sie, welche Sortierverfahren es gibt und welche Eigenschaften, insbesondere welche Komplexität sie besitzen. Da man bei sequenziellem Vorgehen mindestens $n \cdot \log(n)$ Vergleiche benötigt, werden auch deutlich schnellere parallel arbeitende Sortierverfahren vorgestellt, die allerdings recht viele Bausteine erfordern.

Zugleich wiederholen wir aus Kapitel 7 den Nachweis der Korrektheit für einige dieser Verfahren (vor allem in den Übungen).

10.1 Überblick

Suchen und Sortieren machen einen Großteil aller Verwaltungstätigkeiten im Rechner aus, wo Daten ständig abgelegt und schnell wiedergefunden werden müssen. Das Sortieren nutzt die Anordnung der Schlüsselmenge direkt aus und ermöglicht ein schnelles Auffinden: $O(\log(n))$ durch Intervallsuche oder $O(\log(\log(n)))$ durch Interpolationssuche, siehe Abschnitt 6.5.1 und Abschnitt 8.1.

Wir klären als erstes den Begriff "Sortieren" und leiten dann eine *untere* Schranke für die Zahl der Vergleiche her, die bei einem Sortierverfahren, das ausschließlich auf Vergleichen basiert, erforderlich ist. Sodann geben wir einen Überblick über gängige Sortierverfahren und deren Komplexität.

Definition 10.1.1: Sortierte Folgen

Gegeben sei eine endliche oder unendliche Menge mit totaler Ordnung $A = \{a_1, ..., a_s\}$ oder $A = \{a_1, a_2, a_3, a_4, ...\}$ mit $a_1 < a_2 < a_3 < a_4 < ...$.

(1) Eine Folge $v = v_1v_2...v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt (aufsteigend) geordnet genau dann, wenn gilt $v_1 \le v_2 \le ... \le v_n$. (Speziell ist jede leere oder einelementige Folge geordnet.)

(2) Eine Folge $v = v_1v_2...v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt invers geordnet oder absteigend geordnet $\Leftrightarrow$ $v_n \le v_{n-1} \le ... \le v_1$. (Speziell ist jede leere oder einelementige Folge invers geordnet.)

Die Sortieraufgabe lautet: Ordne eine Folge von n Elementen so um, dass sie sortiert ist. Wir präzisieren dies nun.

Definition 10.1.2: Permutationen

Es sei n eine natürliche Zahl.
Eine bijektive Abbildung $\pi: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$ heißt Permutation der Ordnung n.

Erinnerung:
bijektiv = injektiv und surjektiv, d.h.,
injektiv: für je zwei Elementen $i \ne j$ gilt $\pi(i) \ne \pi(j)$ und
surjektiv: zu jedem j existiert ein i mit $\pi(i) = j$.
Eine Permutation ist also nur eine Umordnung der n Elemente.
Hinweis 1: Bekanntlich gibt es genau n! verschiedene Permutationen der Ordnung n.
Hinweis 2: Da π bijektiv ist, existiert auch die inverse Abbildung $\pi^{-1}: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$, definiert durch $[\pi^{-1}(i) = j \Leftrightarrow \pi(j) = i]$. π^{-1} ist ebenfalls eine Permutation.

Definition 10.1.3: Die Sortieraufgabe lautet:
Finde zu einer beliebigen Folge $v = v_1v_2...v_n \in A^*$ eine Permutation π der Ordnung n mit: $v_{\pi(1)}v_{\pi(2)}...v_{\pi(n)}$ ist sortiert (oder invers sortiert, je nach Anwendung).
Ein Algorithmus, welcher $v_1v_2...v_n$ in die sortierte Folge $v_{\pi(1)}v_{\pi(2)}...v_{\pi(n)}$ überführt, heißt Sortierverfahren. Hierbei wird das i-te Element v_i der Folge an die Position $\pi^{-1}(i)$ gesetzt.

Anschaulich gesprochen konstruieren Sortierverfahren die Permutation π^{-1}. Denn man gibt an, *wohin* ein Element der Folge verschoben werden muss (und nicht, *woher* es gekommen ist). Da mit π auch π^{-1} eine Permutation ist, ist es für den Formalismus egal, ob man das Sortieren über π oder über π^{-1} definiert. Für uns Menschen spielt dies aber eine wesentliche Rolle und Sie sollten sich bei konkreten Problemen stets im Klaren darüber sein, ob π oder π^{-1} berechnet wird!

Beispiel zum Formalismus:

Es sei $5, 8, 2, 5, 1, 8$ die gegebene Folge $v_1v_2...v_n \in A^*$ mit $A = \{1, 2, 5, 8\}$, $n = 6$ sowie $v_1 = 5$, $v_2 = 8$, $v_3 = 2$, $v_4 = 5$, $v_5 = 1$ und $v_6 = 8$.

Die geordnete Folge $v_{\pi(1)}v_{\pi(2)}...v_{\pi(n)}$ lautet $1, 2, 5, 5, 8, 8$; man kann also π folgendermaßen festlegen:
$\pi(1) = 5$, $\pi(2) = 3$, $\pi(3) = 1$, $\pi(4) = 4$, $\pi(5) = 2$, $\pi(6) = 6$;
denn dann gilt
$v_{\pi(1)}v_{\pi(2)}v_{\pi(3)}v_{\pi(4)}v_{\pi(5)}v_{\pi(6)} = v_5v_3v_1v_4v_2v_6 = 1, 2, 5, 5, 8, 8.$
(Die Kommata rechts müssen Sie sich wegdenken. Wir schreiben in Beispielen die Folgen der Deutlichkeit halber oft mit dem besser sichtbaren Trennzeichen "Komma" auf; aus Sicht der Definition ist dies aber nicht notwendig.)

$\pi^{-1}(1) = 3$, $\pi^{-1}(2) = 5$, $\pi^{-1}(3) = 2$, $\pi^{-1}(4) = 4$, $\pi^{-1}(5) = 1$, $\pi^{-1}(6) = 6$ gibt an, wohin das i-te Element der Folge verschoben wird.

■

Meist hängt ein Sortieralgorithmus ab von Fragen wie:

- Wie sind die Daten gegeben, zu welchen Mengen gehören sie?
- Wo liegen die Daten? (Hauptspeicher, Platten, Bänder, ...?)
- Zulässige Operationen? (Vergleichen, vertauschen ...?)
- Gibt es Elemente mit gleichem Schlüssel? ($\Rightarrow$ Stabilität.)
- Nur Zugriffsstruktur oder Gesamtdaten sortieren?
- Darf man zusätzlichen Speicher verwenden oder nicht?
- Sequentielles, paralleles, verteiltes Sortieren?
- Effizienz im Mittel oder auch im schlimmsten Fall?
- Erkennen oder Beachten von Vorsortierungen?

10.1.4: Jedes Element v_i der zu sortierenden Folge $v = v_1 v_2 ... v_n$ ist in der Praxis meist ein umfangreiches Dokument. Die Folge wird in der Regel nur nach *einem* Ordnungskriterium sortiert.

Fall 1: Dieses Kriterium wird durch eine Funktion $g: A \rightarrow M$ beschrieben, wobei M eine geordnete Menge ist (A braucht in diesem Fall gar nicht sortierbar zu sein). $g(v_i) = k_i$ heißt dann der Schlüssel von v_i, der in der Regel im record v_i enthalten ist. (Wir können also stets Fall 2 annehmen.)

Fall 2: Das Ordnungskriterium bezieht sich auf eine oder mehrere Komponenten des Dokuments. Den Vektor dieser Komponenten, nach denen zu sortieren ist, bezeichnen wir als Schlüssel. Man verlangt oft, dass dieser ein Element eindeutig beschreibt, doch lassen wir hier auch verschiedene Elemente mit gleichem Schlüssel zu.

In der Regel sortiert man nur die Schlüssel einer Folge.

Beispiel: Folgende Folge aus Name und Alter
(Beier, 23), (Zahn, 40), (Fuhr, 30), (Horn, 41), (Beier, 30), (Horn, 17)
wird zur Folge
(Beier, 23), (Beier, 30), (Fuhr, 30), (Horn, 41), (Horn, 17), (Zahn, 40),
sofern nach dem Namen sortiert wird. Selbstverständlich hat man bei *gleichen Schlüsseln* eine Wahlfreiheit. Statt dieser Reihenfolge hätte man auch die Folge
(Beier, 30), (Beier, 23), (Fuhr, 30), (Horn, 17), (Horn, 41), (Zahn, 40)
als korrektes Ergebnis erhalten können. Die erste Folge hat den Vorteil, dass dort die relative Anordnung von Elementen mit gleichem Schlüssel erhalten blieb (ein solches Verfahren nennt man "stabil").
Dagegen hätte man die Folge
(Horn, 17), (Beier, 23), (Beier, 30), (Fuhr, 30), (Zahn, 40), (Horn, 41),
erhalten können, falls nach dem Alter sortiert wird. ■

Wird also eine Folge v nach mehreren Komponenten (Schlüsseln) nacheinander geordnet, so kann man die Folge zunächst nach dem am wenigsten relevanten Kriterium (im Beispiel: nach dem Alter) und dann schrittweise nach dem nächstwichtigeren Kriterium sortieren. Hierfür muss ein Sortierverfahren "stabil" sein.

Definition 10.1.5:

Ein Sortierverfahren *Sort* heißt stabil, wenn *Sort* die Reihenfolge von Elementen mit gleichem Schlüssel nicht verändert, d.h.: Wenn $Sort(v_1 v_2 ... v_n) = v_{\pi(1)} v_{\pi(2)} ... v_{\pi(n)}$ ist, so gilt für alle $i < j$ mit $v_i = v_j$ stets $\pi^{-1}(i) < \pi^{-1}(j)$ [d.h., stand in der ursprünglichen Folge v_i vor v_j, so gilt dies auch für die sortierte Folge, vgl. 10.1.3].

Sort heißt invers stabil, wenn die Reihenfolge der Elemente mit gleichem Schlüssel von *Sort* gespiegelt wird.

Beispiel: Ein stabiles Sortierverfahren wird aus der Folge
(Beier, 23), (Zahn, 40), (Fuhr, 30), (Horn, 41), (Beier, 30), (Horn, 17)
beim Sortieren nach dem Alter die Folge
(Horn, 17), (Beier, 23), (Fuhr, 30), (Beier, 30), (Zahn, 40), (Horn, 41)
liefern, und das anschließende Sortieren nach dem Namen ergibt:
(Beier, 23), (Beier, 30), (Fuhr, 30), (Horn, 17), (Horn, 41), (Zahn, 40).

Wir erhalten also die Reihenfolge, die wir erwarten würden: Die Liste ist nach dem Namen sortiert und gleiche Namen sind aufsteigend nach dem Alter angeordnet.
Prüfen Sie Definition 10.1.5 nach, indem Sie die Permutation π präzise angeben.

■

Wir haben bereits einige Sortierverfahren kennen gelernt:

In 4.4.4 und 7.3.3: Sortieren durch Austauschen benachbarter, falsch stehender Elemente (*Bubble Sort*).

In 3.7.7 und 8.2.16: *Baumsortieren,* indem die Glieder der Folge nacheinander in einen binären Suchbaum eingefügt und anschließend in inorder-Reihenfolge ausgelesen werden.

In 7.3.3, 7.3.4 und 8.2.16: *Quicksort.* Man wählt ein "Pivot"-Element p aus und spaltet die in einem array gespeicherte Folge in zwei Teilfolgen, von denen alle Elemente der ersten Teilfolge kleiner oder gleich p und alle Elemente der zweiten Teilfolge größer oder gleich p sind. Danach: rekursiv weiter mit beiden Teilfolgen, sofern die jeweilige Teilfolge noch mindestens zwei Elemente besitzt.

Frage an Sie: Welche dieser drei Verfahren sind stabil?

Sortieren erfordert in der Praxis viele Umspeicherungs-operationen. Sind die Elemente sehr groß, so kostet dies viel Zeit. Man zieht daher die Schlüssel und den Verweis auf den jeweiligen Datensatz heraus und sortiert nur diese Schlüssel-Verweis-Tabelle. Wir werden also unseren Sortierverfahren Elemente des Datentyps

```
record  key: <Typ des Schlüssels>;
        zeiger: <Zeigertyp auf den Elementtyp>;
end record
```

zugrunde legen. Es genügt, sich nur auf die Sortierung der Schlüssel zu beschränken.

Definition 10.1.6:

Für eine Permutation $\pi: \{1, 2, ..., n\} \to \{1, 2, ..., n\}$ heißt
$$\Im(\pi) = \left| \{ (i,j) \mid i < j \text{ und } \pi(i) > \pi(j) \} \right|$$
die Inversionszahl (oder der Fehlstand) von π.

Analog: Für eine Folge $v = v_1 v_2 ... v_n \in A^*$ (A sei eine geordnete Menge) heißt $\Im(v) = \left| \{ (i,j) \mid i < j \text{ und } v_i > v_j \} \right|$
die Inversionszahl (oder der Fehlstand) von v.

Wegen $\Im(v_1 v_2 ... v_n) + \Im(v_n v_{n-1} ... v_1) = \left| \{ (i,j) \mid i < j \} \right|$ gilt der

Hilfssatz 10.1.7: $\Im(v_1 v_2 ... v_n) + \Im(v_n v_{n-1} ... v_1) = ½ \cdot n \cdot (n-1)$.

Wegen $\Im(1\ 2\ 3\ ...\ n) = 0$ folgt $\Im(n\ n\text{-}1\ ...\ 2\ 1) = ½ \cdot n \cdot (n-1)$.

Zeigen Sie: $\Im(\pi) = \Im(\pi^{-1})$ für alle Permutationen π.

Begriffsbeschreibung 10.1.8:

Ein Sortierverfahren *Sort* heißt ordnungsverträglich ⇔
Je geordneter die zu sortierende Folge bereits ist, umso schneller arbeitet *Sort*.

Genauer:
Wenn *Sort* für zwei Folgen $v = v_1 v_2 ... v_n$ und $w = w_1 w_2 ... w_n$ die Permutationen π_1 und π_2 realisiert und wenn die Inversionszahl von π_1 kleiner als die von π_2 ist, dann benötigt *Sort* zum Sortieren von v nicht mehr Zeit als zum Sortieren von w.

Hinweis: Eigentlich realisiert *Sort* die Permutationen π_1^{-1} und π_2^{-1}, siehe 10.1.3. Weil $\Im(\pi) = \Im(\pi^{-1})$ für alle Permutationen π gilt und weil im Folgenden nur die Absolutbeträge untersucht werden, kann man stattdessen auch π_1 und π_2 betrachten.

Wir betrachten die Operation "*benachbartes Vertauschen*". Diese Operation überführt eine Folge

$v_1 v_2 ... v_{i-1} v_i v_{i+1} v_{i+2} ... v_n$

in die Folge

$v_1 v_2 ... v_{i-1} v_{i+1} v_i v_{i+2} ... v_n$ (für ein $0<i<n$).

Hierfür gilt:

$\left| \Im(v_1 v_2 ... v_{i-1} v_i v_{i+1} v_{i+2} ... v_n) - \Im(v_1 v_2 ... v_{i-1} v_{i+1} v_i v_{i+2} ... v_n) \right| = 1.$

Wegen 10.1.7 haben wir somit gezeigt:

Hilfssatz 10.1.9:
Ein Sortierverfahren, das ausschließlich mit der Operation "benachbartes Vertauschen" arbeitet, benötigt im worst case mindestens $\frac{1}{2} \cdot n \cdot (n-1)$ Schritte.

Wir betrachten die Operation "*beliebiges Vertauschen*". Diese überführt eine Folge

(für $1 \le i \le j \le n$) $v_1 v_2 ... v_{i-1} v_i v_{i+1} ... v_{j-1} v_j v_{j+1} ... v_n$

in die Folge $v_1 v_2 ... v_{i-1} v_j v_{i+1} ... v_{j-1} v_i v_{j+1} ... v_n$.

Hierfür gilt:

$0 \le \left| \Im(v_1 ... v_i ... v_j ... v_n) - \Im(v_1 ... v_j ... v_i ... v_n) \right| \le 2(j-i)-1 \le 2n-3,$

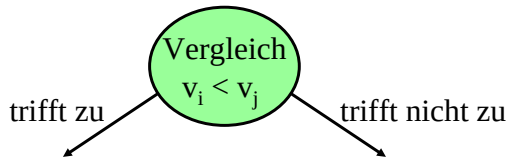
wobei der größte Wert 2n-3 nur erreicht wird, wenn das kleinste Element am Ende und das größte am Anfang stand und genau diese beiden vertauscht wurden. Es folgt:

Hilfssatz 10.1.10: Ein Sortierverfahren, das ausschließlich mit der Operation "Vertauschen" arbeitet, benötigt im worst case mindestens $n \cdot (n-1)/(4n-6) > \frac{1}{4} \cdot n$ Schritte.
(Man kann diese Schranke noch verschärfen. Versuchen Sie, die bestmögliche untere Schranke zu finden.)

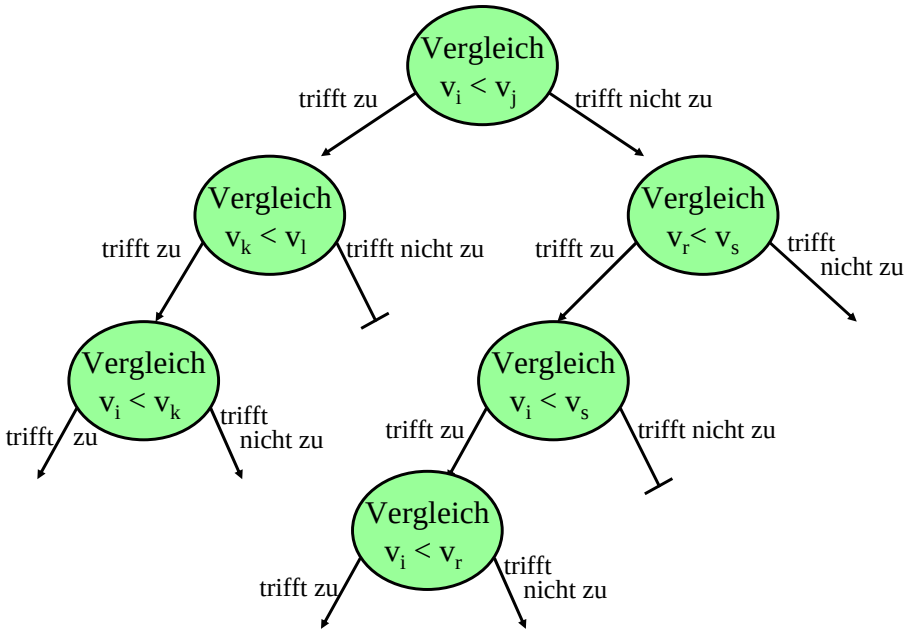
In der Regel weiß man nicht, ob man zwei Elemente einer Folge vertauschen soll. Diese Entscheidung wird durch einen Vergleich getroffen: Falls $v_i < v_j$ und $i > j$ ist, dann vertauscht man diese beiden Elemente in der Folge.

Wird das Vertauschen durch vorher gehende Vergleiche gesteuert, so dauert das Sortierverfahren mindestens so lange wie die Anzahl der hierfür erforderlichen Vergleiche.

Eine Folge von Vergleichen bildet einen binären Baum, der aus den Knoten und Kanten

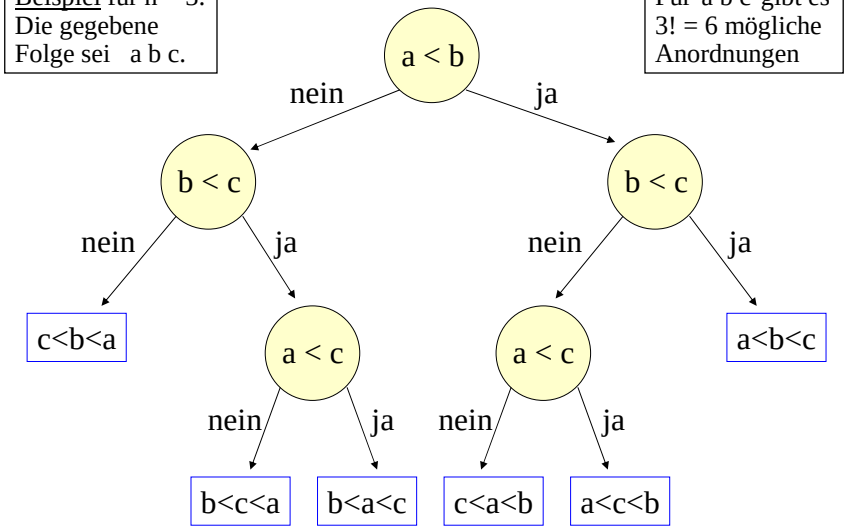


aufgebaut ist.



Beispiel für n = 3.
Die gegebene Folge sei a b c.

Für a b c gibt es 3! = 6 mögliche Anordnungen



Die sortierte Folge ist jeweils blau umrandet als Blatt angegeben.

Hat man alle Informationen zum Sortieren gewonnen, dann stellen genau die null-Zeiger in diesem Baum (= alle blau umrandeten Aussagen in obigem Beispiel) die Permutationen dar, die zur Sortierung gehören. Da es n! Permutationen der Ordnung n gibt, muss der "Baum der Vergleiche" daher mindestens n! null-Zeiger besitzen.

Ein binärer Baum mit m-1 Knoten besitzt genau m null-Zeiger. Also muss der Baum der Vergleiche

mindestens n!-1

Knoten besitzen.
Die Länge des längsten Weges, also die Tiefe dieses Baums gibt die Zahl der erforderlichen Vergleiche im worst case an. Die Tiefe eines binären Baums mit k Knoten ist aber mindestens log(k+1), siehe Folgerung 8.2.12. Daher gilt:

Satz 10.1.11: Ein Sortierverfahren, das ausschließlich auf Vergleichen zweier Elemente beruht, benötigt im worst case

mindestens $\log(n!) \approx n \cdot \log(n) - 1{,}4404 \cdot n + O(\log(n))$

Schritte.

Hinweis: Nach der *Stirlingschen Formel* gibt es zu jedem n ein d mit $0 < d < 1$, so dass gilt (mit $e = 2{,}718281828459...$):

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\ \ e^{\frac{1}{12}d}$$

Durch Logarithmieren erhält man hieraus:
$\log(n!) \approx n \cdot \log(n) - n \cdot \log(e) \approx n \cdot \log(n) - 1{,}4404 \cdot n$
und von $2\pi n$ verbleibt noch ein additiver Anteil $O(\log(n))$.

10.1.12 Überblick über die üblichen Sortiermethoden:

Austauschen:
- a. Benachbartes Austauschen (bubble sort, shaker sort)
- b. Shellsort
- c. Quicksort

Einfügen:
- a. Einfügen in Listen (Insertion sort)
- b. Baumsortieren (mit binären Bäumen, AVL-Bäumen, ...)
- c. Fachverteilen (und radix exchance)

Aussuchen / Auswählen:
- a. Minimumsuche (minimum sort)
- b. Heapsort (normal, bottom up, ultimativ)

Mischen:
- Merge sort und diverse Varianten

Streuen und Sammeln (bucket sort)

Sortiermethoden	Zeitaufwand	zusätzl. Platz
Austauschen		
a. Benachb. Austauschen	$\frac{1}{2}\cdot n^2$	konstant
b. Shellsort	$O(n^{\frac{3}{2}})$	$\leq \log(n)$
c. Quicksort (im Mittel !)	$1{,}3863\cdot n\cdot\log(n)$	$2\log(n)$
Einfügen		
a. Einfügen in Listen	$\frac{1}{2}\cdot n^2$	konstant
b. Baumsortieren (AVL-Bäume)	$\leq 1{,}4404\cdot n\cdot\log(n)$	$O(n)$
c. Fachverteilen (im Mittel)	$O(n\cdot\log(n))$	$O(n)$
Aussuchen / Auswählen		
a. Minimumsuche	$\frac{1}{2}\cdot n^2$	konstant
b. Heapsort	$\approx 2n\cdot\log(n)$	konstant
c. Bottom-up Heapsort	$n\cdot\log(n) + O(n)$	konstant
Mischen		
Verschmelzen (merge sort)	$O(n\cdot\log(n))$	n
Streuen und Sammeln	$O(n)$	$O(n)$

10.2 Sortieren durch Austauschen

Die beiden wichtigsten Vertreter Bubble Sort und Quicksort wurden bereits vorgestellt; sie sind auf den nächsten Folien nochmals als Programm ausformuliert.

Aufwandsabschätzung für Bubble Sort:
Platzbedarf: konstant (3 zusätzliche Variablen)
Zeitbedarf: worst case: $\frac{1}{2}\cdot n\cdot(n-1)$ Vergleiche,
best case: n, falls das Feld bereits sortiert ist,
im Mittel sind $\frac{1}{4}\cdot n\cdot(n-1)$ Vergleiche zu erwarten **(wirklich?)**.

Man kann das Feld abwechselnd aufwärts und abwärts durchlaufen (dies nennt man Shaker Sort). Diese Variante bringt aber in der Praxis keine Verbesserung des Laufzeitverhaltens.

10.2.1 Bubble Sort für ein Integer-Feld A mit dem Indextyp 1..n (das Feld A sei vom Feld-Typ Vektor):

```
procedure BubbleSort (A: in out Vektor) is
Weiter: Boolean := True;  H: Integer;
begin
   while Weiter loop
      Weiter := False;
      for i in 1..n-1 loop
         if A(i) > A(i+1) then Weiter := True;
            H := A(i); A(i) := A(i+1); A(i+1) := H;
         end if;
      end loop;
   end loop;
end BubbleSort;
```

Beachte: Durch die Variable "Weiter" wird Bubble Sort ordnungsverträglich.

Oft programmiert man Buble Sort auch "abwärts", also:
for i in revers 1..n-1 loop ...

10.2.2 Quicksort: Aus 7.3.4 übernehmen wir mit den Datentypen
type Index is 1..n; ... A: array (Index) of Integer; ... das Programm:

```
procedure Quicksort(L, R: Index) is     -- A ist global, A(L..R) wird sortiert
i, j: Index; p, h: Integer;             -- p wird das Pivot-Element
begin
  if L < R then
     i := L; j := R; p := A((L+R)/2);   -- man kann p auch anders wählen
     while i <= j loop                  -- die Indizes i und j laufen aufeinander zu
          while A(i) < p loop i := i+1; end loop;
          while A(j) > p loop j := j-1; end loop;
          if i <= j then h:=A(i); A(i):=A(j); A(j):=h;
                         i := i+1; j := j-1;  end if;
     end loop;                          -- auch bei Gleichheit A(i)=p oder A(j)=p vertauschen!
     if (j-L) < (R-i) then Quicksort(L, j); Quicksort(i, R);
     else Quicksort(i, R); Quicksort(L, j); end if;  -- Vorsicht; siehe unten!
  end if;
end Quicksort;

... Quicksort(1,n); ...                 -- Aufruf des Sortierverfahrens
```


10.2.3 Platzbedarf von Quicksort

In 7.3.4 wird erläutert, warum mit einem hohen Platzbedarf bei der rekursiven Formulierung zu rechnen ist. Man muss das Quicksort-Programm so abändern, dass man den Stack für die Rekursion selbst verwaltet und nutzlos gewordene Information sofort entfernt und nicht mehr im Stack stehen lässt. Es ist klar, dass man auf diese Weise die Tiefe des Stacks auf log(n) Paare beschränken kann.

Aufgabe: Versuchen Sie, eine Lösung für dieses Problem anzugeben, d.h., Sie sollen die Prozedur Quicksort so abwandeln, dass die rekursive Tiefe des Aufruf-Stacks durch $2 \cdot \log(n)$ beschränkt bleibt.
(Hinweis: Eine Lösung finden Sie im Buch Ottmann/Widmaier.)

10.2.4 Zeitbedarf von Quicksort (siehe 8.2.16):

Zeitbedarf: worst case: $\approx \frac{1}{2} \cdot n^2$ Vergleiche, wenn das Pivot-Element stets das kleinste oder das größte Element des Teilfelds ist.
best case: $n \cdot \log(n)$, wenn das Pivot-Element stets das mittelste der Elemente des Teilfelds ("Median") ist.
average case: Es ist mit $1{,}3863 \cdot n \cdot \log(n) - 1{,}8456 \cdot n$ Vergleichen im Mittel zu rechnen. Begründung:

Man kann das Aufteilen des Feldes in zwei Teilfelder mit dem Aufbau eines binären Suchbaums vergleichen, wobei das Pivot-Element in die Wurzel kommt und aus den beiden Teilfeldern rekursiv der linke und rechte Unterbaum aufgebaut werden. Quicksort verhält sich daher im Mittel genau wie das Baumsortieren mit binären Suchbäumen (Satz 8.2.15). Die formalen Berechnungen liefern das gleiche Ergebnis, siehe Lehrbücher.

Zeitbedarf von Quicksort im Mittel (Fortsetzung):

Für den Zeitbedarf ist die "gute Wahl" des Pivot-Elements p entscheidend. Gebräuchlich ist folgende Variante: Statt das Element p irgendwie fest zu wählen (z.B.: p:=A((L+R)/2) oder p:=A(L) oder ...) nimmt man das mittlere von drei Elementen, z.B. von A(L), A(R) und A((L+R)/2).
Mit dieser "Mittelwert aus 3"-Variante erhält man einen deutlich besseren mittleren Zeitbedarf, nämlich im Mittel höchstens

$1.188 \cdot n \cdot \log(n) - 2{,}255 \cdot n$ Vergleiche.

Diese Variante wird daher gern in der Praxis verwendet.

Aufgabe: Erweitern Sie unsere Prozedur um diese Variante. Machen Sie Messungen mit zufällig erzeugten Daten und verifizieren Sie hiermit den Laufzeitgewinn, der ab einem gewissen n eintritt. (Bestimmen Sie dieses n experimentell.)

Zeitbedarf von Quicksort im Mittel (Fortsetzung):

Eine andere Variante besteht darin, die Zerlegung in drei Teilfelder vorzunehmen (sog. "Dreiwege-Split"):
Alle Elemente in A(L..j) sind echt kleiner als p,
alle Elemente in A(i..R) sind echt größer als p und
alle Elemente in A(j+1..i-1) sind gleich p, wobei dieser Bereich nie leer ist. Da man ihn nicht bei der Rekursion berücksichtigen muss, verringert sich die Rekursionstiefe und damit die Laufzeit. Treten in einer Folge viele Schlüssel mehrfach auf, so lässt sich hierdurch das Sortieren beschleunigen.

Aufgabe: Erweitern Sie unsere Prozedur auch um diese Variante. Die Schwierigkeit liegt darin, alle Schlüssel, die gleich p sind, ohne Zusatzaufwand in der Mitte des zu sortierenden Teilfelds zu platzieren. Auch hier sollten Sie dann Messungen mit zufällig erzeugten Daten vornehmen und prüfen, ab welchem Prozentsatz gleicher Schlüssel sich dieses Vorgehen lohnt.

Hinweise:

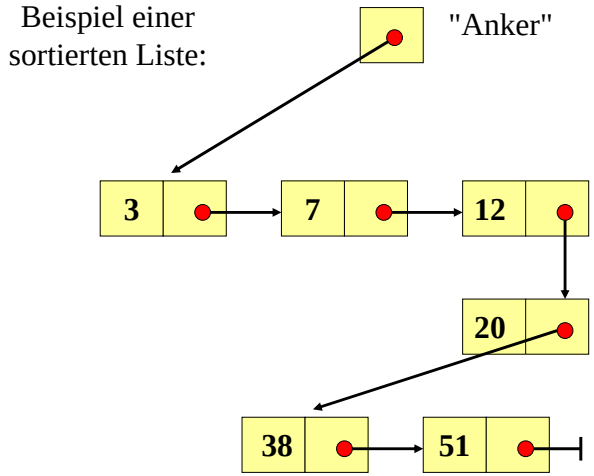
Historisch gesehen gibt es weitere Sortierverfahren, die auf dem Austauschen beruhen. Am bekanntesten ist Shellsort, bei dem die Folge in äquidistante Teilfolgen unterteilt wird und als erste Phase in diesen eine Sortierung erfolgt (z.B. ein Bubble Sort Durchlauf). Die äquidistanten Abstände werden dann für eine zweite Phase verringert und diese Verringerung wird fortgesetzt, bis der Abstand gleich 1 ist, d.h. eine Sortierung der bis dahin entstandenen schon gut vorsortierten Folge stattfindet. Dieses Vorgehen setzt in den einzelnen Phasen ordnungsverträgliche Sortierverfahren voraus.
Denken Sie sich selbst "schnelle" heuristische Verfahren aus. Der Fantasie sind hier kaum Grenzen gesetzt. Führen Sie aber auf jeden Fall Messungen durch (denn fast nie erfüllen sich die erhofften Beschleunigungen).

10.3 Sortieren durch Einfügen

Wenn die Elemente einer Folge durch Zeiger (sortierte Listen, Suchbäume) dargestellt werden, so wird man die einzelnen Elemente der Folge nacheinander in diese Struktur einfügen und dabei die Struktur stets wiederherstellen.

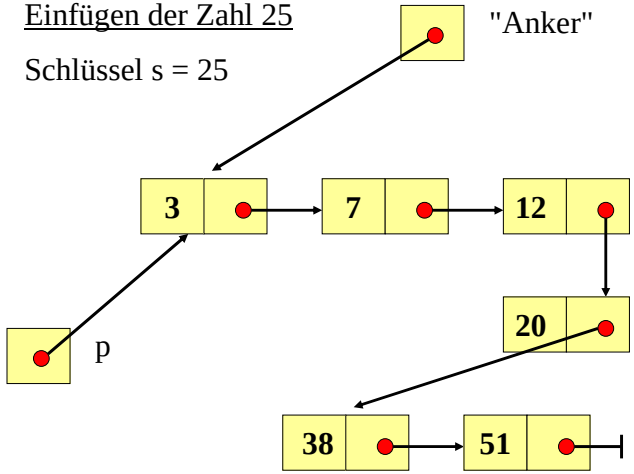
Einfachster Fall: Einfügen in eine sortierte Liste.
Eine Liste heißt sortiert, wenn für alle Elemente der Liste gilt: p.Inhalt ≤ p.next.Inhalt. Hierbei ist p ein Zeiger, der auf das jeweilige Element der Liste verweist.

Erinnerung: Datenstruktur für eine Liste (vgl. z. B. 3.5.2.0):
```
type Zelle;
type Ref_Zelle is access Zelle;
type Zelle is record Inhalt: Integer; Next: Ref_Zelle; end record;
```

Einfügen der Zahl 25

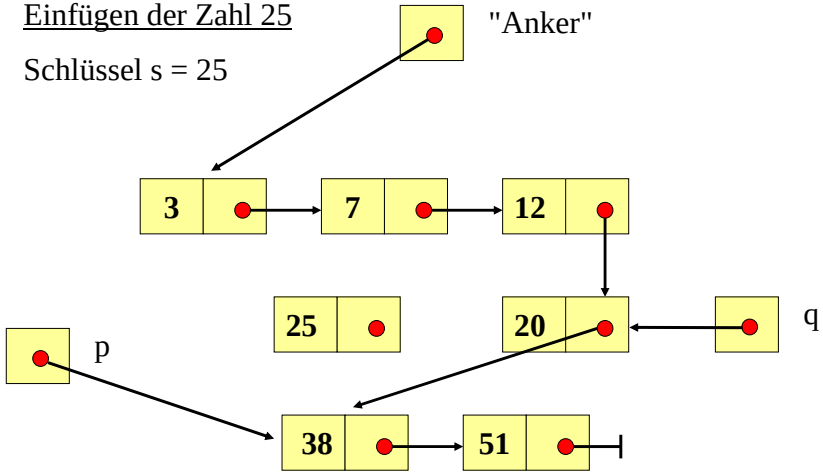
Schlüssel s = 25



p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;

Einfügen der Zahl 25

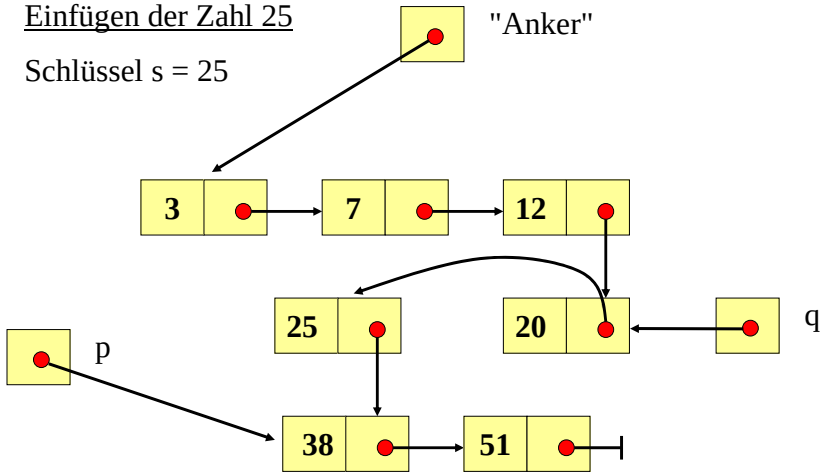
Schlüssel s = 25



p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;

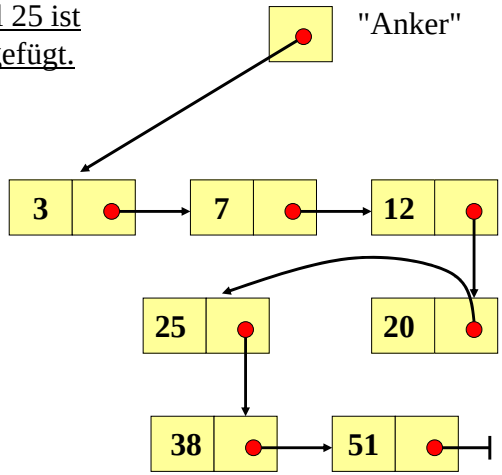
Einfügen der Zahl 25

Schlüssel s = 25



p := Anker;
while p /= null and then s > p.Inhalt loop p:=p.Next; end loop;

Die Zahl 25 ist nun eingefügt.



Prozedur zum Einfügen eines *Elementes in eine sortierte Liste:*

```
procedure Einf (Anker: in out Ref_Zelle; s: in Integer) is
p, q: Ref_Zelle;
begin  p := Anker; q := null;
       while p /= null and then s > p.Inhalt loop
           q := p;  p := p.Next;  end loop;
       if  q = null  then
           if p = null then Anker := new Zelle'(s, null);
           else Anker := new Zelle'(s, Anker); end if;
       else  q.Next := new Zelle'(s, p);  end if;
end Einf;
```


Diese Prozedur kann man vereinfachen zu (bitte selbst nachprüfen!):

```
procedure Einf (Anker: in out Ref_Zelle; s: in Integer) is
p, q: Ref_Zelle;
begin  p := Anker; q := new Zelle'(s, Anker);
           while p /= null and then s > p.Inhalt loop
               q := p;  p := p.Next;  end loop;
           q.Next := new Zelle'(s, p);
end Einf;
```

10.3.1 Sortieren von n Elementen durch Einfügen in eine Liste:

```
while not End_of_File loop Get(v); Einf(Anker, v); end loop;
```

Zahl der Vergleiche im Mittel und im schlechtesten Fall: $O(n^2)$.

10.3.2 Sortieren durch Einfügen in einen Baum: siehe 8.2.16.

Wenn man beliebige binäre Suchbäume verwendet, so können diese im schlechtesten Fall zu einer Liste entarten und daher beträgt im worst case die Zeitkomplexität $O(n^2)$.

Benutzt man aber anstelle eines beliebigen Suchbaums AVL-Bäume, so ist deren Höhe durch $1.4404 \cdot n \cdot \log(n)$ nach Satz 8.4.8 beschränkt. Daher ist die Anzahl der Vergleiche für das Sortieren mit Bäumen auch im schlechtesten Fall durch $1.4404 \cdot n \cdot \log(n) + O(n)$ beschränkt.

In der Praxis kann man bei der Verwendung von beliebigen Suchbäumen im Mittel mit $1.3863 \cdot n \cdot \log(n) + O(n)$, bei der Verwendung von AVL-Bäumen im Mittel mit $n \cdot \log(n) + O(n)$ rechnen (siehe Satz 8.2.15 und Hinweis nach Satz 8.4.8).

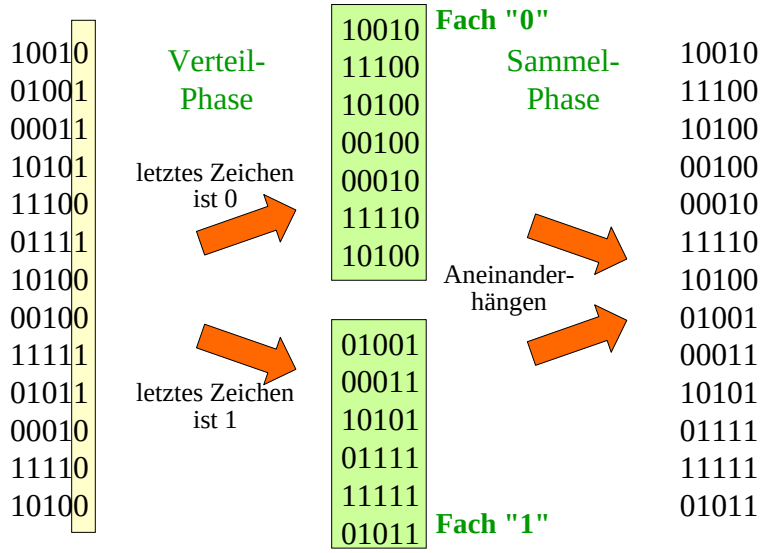
10.3.3 Sortieren durch Fachverteilen

Sind die Schlüssel Wörter über einem endlichen Alphabet $A = \{a_1, a_2, ..., a_s\}$, kann man die zu sortierenden Elemente zunächst bzgl. des letzten Zeichens an s Listen anfügen. Diese Listen hängt man aneineinander und fügt nun alle Elemente bzgl. des vorletzten Zeichens an s Listen an usw. Liegt die Länge jedes Schlüssels in der Größenordnung von $\log(n)$, dann ergibt sich ein $O(n \cdot \log(n))$-Sortierverfahren.

Das Anhängen an die s Listen entspricht dem Ablegen in s verschiedene Fächer, weshalb dieses Verfahren als "Fach-verteilen" bezeichnet wird.

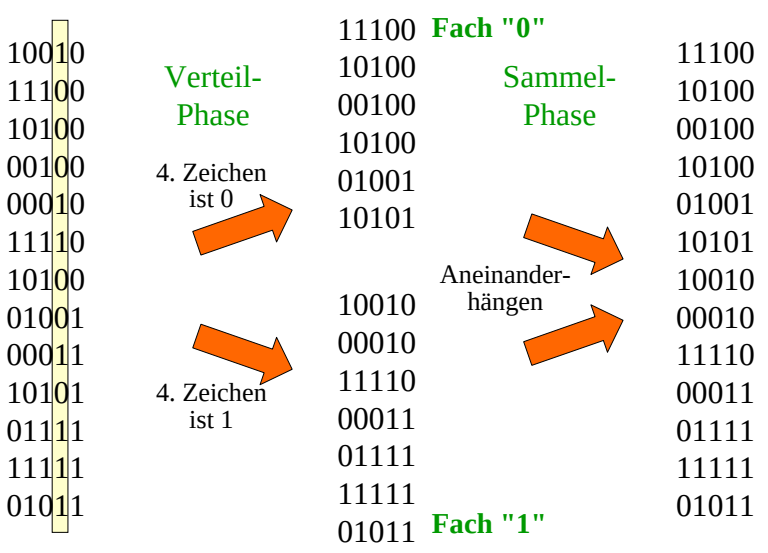
Wir erläutern das Verfahren nur an einem Beispiel mit $s = 2$. Die Programmierung ist nicht schwierig.

Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



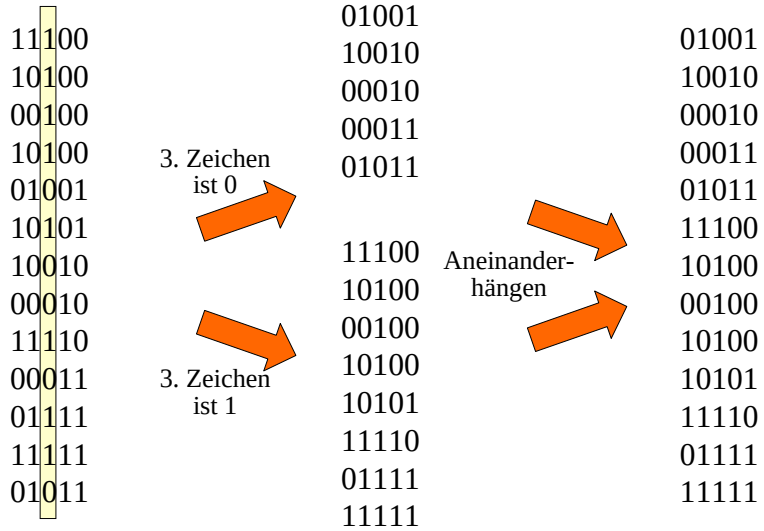
Nun iterativ weiter durch alle Stellen der Schlüssel.

Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5

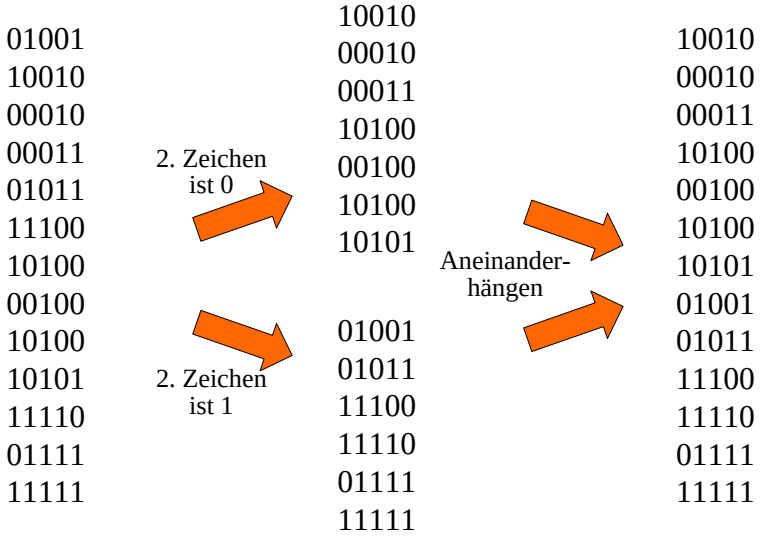


Nun iterativ weiter durch alle Stellen der Schlüssel.

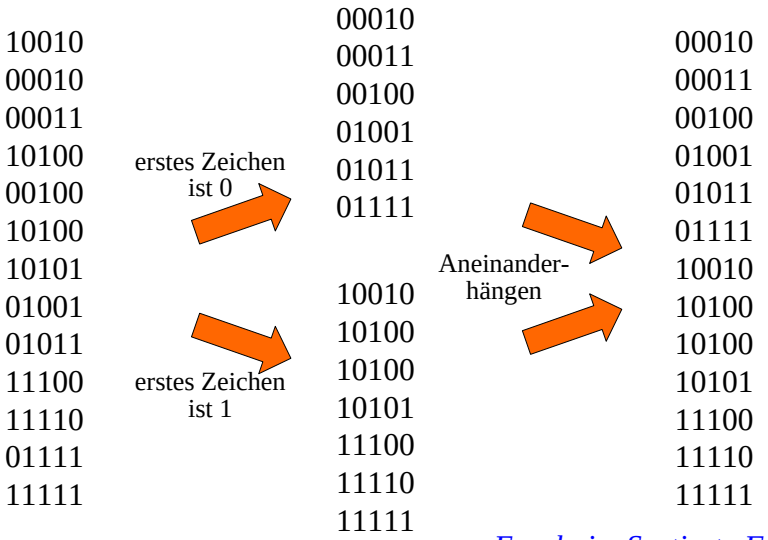
Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Gegeben seien 13 Schlüssel als 0-1-Darstellungen der Länge 5



Ergebnis: Sortierte Folge.

Man beachte: Die Sortierung jeder Phase muss stabil sein. Hat man s Zeichen (z.B. $s = 26$ für das Alphabet oder $s = 128$ für den ASCII-Zeichensatz), dann muss man s solche "Fächer" bereitstellen. In der Regel organisiert man die Fächer als Listen, an die man hinten (in einem Schritt) den jeweils nächsten gelesenen Schlüssel anhängt; danach werden die s Listen (in s Schritten) aneinandergehängt und die nächste Verteilphase kann beginnen.
Das Sortieren erfordert $s \cdot n$ Vergleiche. Es eignet sich besonders gut für das Sortieren von binärer Information (z.B. in der Systemprogrammierung) und in allen Fällen, in denen $s \leq \log(n)$ ist. Nachteilig ist, dass man in der Regel das Doppelte an Speicherplatz benötigt. Man kann aber auch einen Austausch wie bei Quicksort vornehmen, so dass die 0-en oben und die 1-en unten zu stehen kommen (Vorsicht wegen der erforderlichen Stabilität!); im Falle $s > 2$ bietet sich auch ein bucket sort an, siehe 10.6.

10.4 Sortieren durch Aussuchen/Auswählen

Vorgehen:
Wähle das kleinste Element aus, stelle es an die erste Stelle und mache genauso mit den restlichen Elementen weiter.

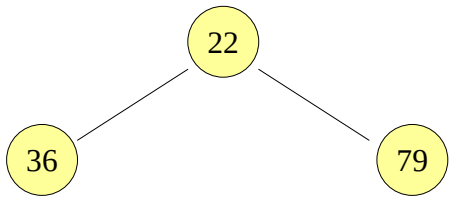
10.4.1 Sortieren durch "Minimum sortieren":

```
for i in 1..n-1 loop
    min := A(i); pos := i;          -- finde das kleinste Element von A(i) bis A(n)
    for j in i+1..n loop
        if A(j) < min then min:=A(j); pos := j; end if;
    end loop;                       -- das kleinste Element steht an Position pos
    A(pos) := A(i); A(i) := min;    -- nun steht das kleinste Element an Position i
end loop;
```

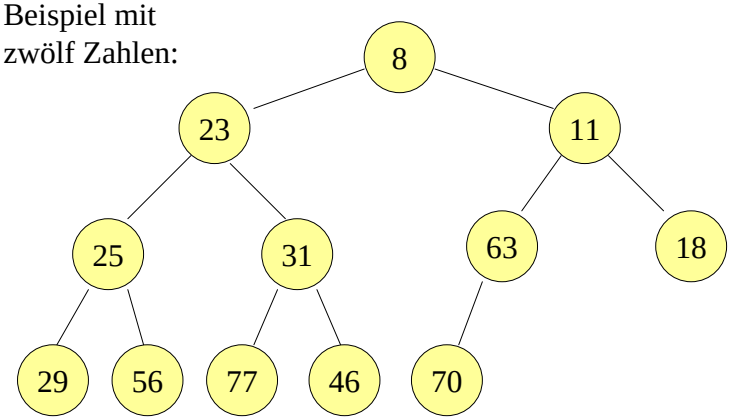
Zahl der Vergleiche stets $\frac{1}{2} \cdot n \cdot (n-1)$ Schritte: $\Theta(n^2)$
Platzaufwand 4 zusätzliche Speicherplätze: $O(1)$

Überlegung:
Könnte man die Information "Minimum sein" besser anordnen?

Ja, in einem binären Baum. Betrachte einen Konten mit zwei Nachfolgern. Schreibe in den Elternknoten das Minimum der drei Knoten.



Beispiel mit zwölf Zahlen:



Als Feld levelweise aufgeschrieben:

8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

Besonderheit dieses Baums: Auf jedem Pfad von der Wurzel zu einem Blatt sind die Elemente aufsteigend geordnet.

8	23	11	25	31	63	18	29	56	77	46	70
1	2	3	4	5	6	7	8	9	10	11	12

Die Bedingung "Der Inhalt eines Knotens ist nicht größer als der Inhalt jedes Nachfolgeknotens" lässt sich präzisieren durch $A(i) \le A(2i)$ und $A(i) \le A(2i+1)$. Folgen oder Felder mit dieser Eigenschaft nennen wir "Heap" (meist übersetzt mit "Haufen", es handelt ich aber um gut geordnete Haufen; sie haben nichts mit der Halde zu tun, die die mit new eingeführten Daten aufnimmt und die im Englischen ebenfalls "heap" heißt).

Definition 10.4.2: Heap-Eigenschaft

Eine Folge oder ein array A(1), A(2), A(3), ..., A(n) heißt ein (aufsteigender) **Heap**, wenn für jedes i gilt:

$A(i) \le A(2i)$ und $A(i) \le A(2i+1)$,

wobei natürlich nur solche Ungleichungen betrachtet werden, bei denen 2i bzw. 2i+1 nicht größer als n sind.

Eine Folge oder ein array A(1), A(2), A(3), ..., A(n) heißt ein absteigender **Heap**, wenn für jedes i gilt:

$A(i) \ge A(2i)$ und $A(i) \ge A(2i+1)$,

wobei natürlich nur solche Ungleichungen betrachtet werden, bei denen 2i bzw. 2i+1 nicht größer als n sind.

10.4.3 Heapsort:

Gegeben sei ein Feld A(1), A(2), A(3), ..., A(n) mit Elementen aus einer geordneten Menge.

1. Wandle dieses Feld in einen absteigenden Heap um, so dass anschließend gilt: $A(i) \ge A(2i)$ und $A(i) \ge A(2i+1)$ für alle i (sofern $2i \le n$ bzw. $2i+1 \le n$ ist).
2. Für j von n abwärts bis 2 wiederhole:
   Vertausche A(1) und A(j).
   (Nun verletzt A(1) in der Regel die Heapeigenschaft.)
   Wandle das Feld A(1..j-1) ausgehend von der Wurzel so um, dass wieder ein absteigender Heap entsteht.

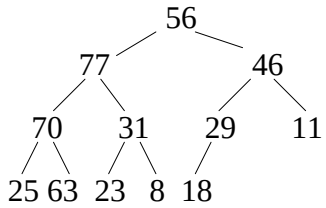
Im Folgenden beschreiben wir das Umwandeln in einen Heap (1.) und die Wiederherstellung der Heap-Eigenschaft (2.).

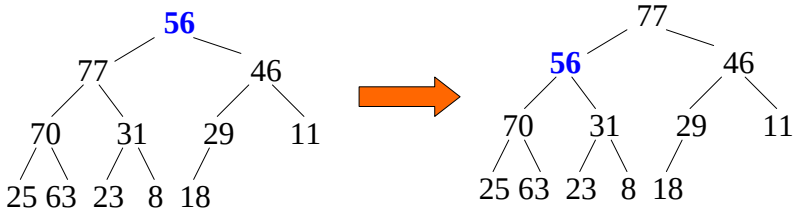
Die zentrale Prozedur ist die Herstellung der Heap-Eigenschaft in dem Teil des Feldes A, das mit dem Index "links" beginnt und mit dem Index "rechts" endet, unter der Annahme, dass höchstens beim Index "links" die Heap-Eigenschaft verletzt ist.

Hierfür vergleicht man den Inhalt des Elements A(links) mit den Inhalten der beiden Nachfolgeknoten und lässt gegebenenfalls den Inhalt von A(links) durch Vertauschen mit dem kleineren der beiden Nachfolger-Inhalten "absinken".

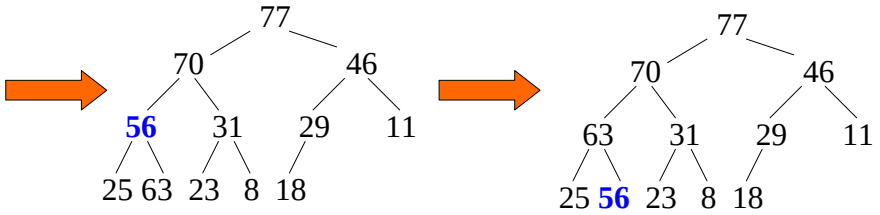
Betrachte ein Beispiel (mit links = 1 und rechts = 12) :

Die Heap-Eigenschaft ist hier nur bei "56" verletzt.





Vorgehen: Vergleiche 56 mit 77 und 46. Das Maximum ist 77, daher werden 77 und 56 vertauscht und bei 56 weitergemacht.



10.4.4: Prozedur für das Absinken

```
procedure sink (links, rechts: 1..n) is          -- A und n sind global
i, j: Natural;  weiter: Boolean:=true;  v: <Elementtyp>;
begin  v := A(links);  i := links;  j := i+i;
    while  (j <= rechts) and weiter  loop
        if  j = rechts  then
            if A(j) > v then A(i):=A(j); i:=j; end if;
            weiter:=false;
        elsif A(j) < A(j+1) then
                if v < A(j+1) then  A(i) := A(j+1); i := j+1;
                else weiter:=false; end if;
        else    if v < A(j) then A(i) := A(j); i := j;
                else weiter:=false; end if;
        end if;
        j := i+i;                    -- v wird erst am Ende explizit eingetragen.
    end loop;                        -- i gibt am Ende die aktuelle Position an, an
    A(i):=v;                         -- der v einzufügen ist.
end sink;                            -- Im Inneren der Schleife werden bis zu zwei
                                     --Vergleiche zwischen Elementen durchgeführt.
```


10.4.5: Prozedur für Heapsort

```
procedure heapsort is                          -- A und n sind global
  procedure sink ... begin .... end sink;      -- siehe oben 10.4.4
  h: Natural;  x: <Elementtyp>;
  begin
     h := n div 2;                             -- wandle A in einen Heap um
     for k in reverse 1..h loop sink(k, n); end loop;
     for k in reverse 2..n loop
         x := A(1); A(1) := A(k); A(k) := x;   -- vertausche A(1) und A(k)
         sink(1, k-1);  end loop;              -- Wurzel absinken lassen
  end heapsort;
```

Hinweis: Es lassen sich noch einige Umspeicherungen vermeiden, indem man das Wechselspiel zwischen v (in 'sink') und x optimiert. Dies ändert aber nichts an der Zahl der (Element-) Vergleiche.

Wie viele Vergleiche benötigt Heapsort?

1. Aufbau des Heaps (for k in reverse 1..h loop sink (k, n); end loop;)

Für k von h bis h/2: maximal 2 Vergleiche
für k von h/2 bis h/4: maximal 4 Vergleiche
für k von h/4 bis h/8: maximal 6 Vergleiche ....
für k von $h/2^{i-1}$ bis $h/2^i$ maximal 2i Vergleiche (i=1, 2, ..., log(n))

Aufsummieren ergibt maximal 2·n Vergleiche (beachte h = n/2):
$2 \cdot h/2 + 4 \cdot h/4 + 6 \cdot h/8 + 8 \cdot h/16 + ... + 2 \cdot \log(n) \cdot 1$
$= 2h \cdot (2 - 2 \cdot (\log(n)+1)/n) = 2 \cdot n - 2 \cdot \log(n) - 2 \leq 2 \cdot n$ Vergleiche.
Der Aufbau des Heaps erfolgt also in linearer Zeit.

Dies beweist man genauso wie die entsprechende Formel $\sum_{j=1}^{k} j \cdot 2^{j-1}$ in Abschnitt 6.5.1. Es gilt:
$1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + ... + m/2^m = 2 - (m+1)/2^{(m-1)}$.

2. Sortierphase (for k in reverse 2..n loop ... sink(1, k-1); end loop;)

Für k von n bis n/2: maximal 2·log(n) Vergleich
für k von n/2 bis n/4: maximal 2·log(n)-1 Vergleiche
für k von n/4 bis n/8: maximal 2·log(n)-2 Vergleiche ....
für k von $n/2^{i-1}$ bis $n/2^i$ maximal 2·i Vergleiche (i=1, 2, ..., log(n))

Aufsummieren ergibt maximal 2·n·log(n) Vergleiche: Sei n > 1:

$\log(n) \cdot n + (\log(n)-1) \cdot n/2 + (\log(n)-2) \cdot n/4 + (\log(n)-3) \cdot n/8 + ... + 2 =$
$2 \cdot n \cdot (\log(n)/2 + \log(n)/4 + ... + 1/2^{\log(n)} - 1/4 - 2/8 - 3/16 - ... - (\log(n)-1)/2^{\log(n)}) =$
$2 \cdot n \cdot \log(n) \cdot (1-1/2^{\log(n)}) - n \cdot (1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + ... + (\log(n)-1)/2^{\log(n)-1}) =$
$2 \cdot n \cdot \log(n) - 2 \cdot \log(n) - n \cdot (2 - \log(n)/2^{\log(n)-2}) = 2 \cdot n \cdot \log(n) - 2 \cdot n + 2 \cdot \log(n)$
$< 2 \cdot n \cdot \log(n)$

(Wir benutzten hier erneut die Formel von der vorherigen Folie ganz unten.)

Insgesamt ergeben sich für Heapsort im worst case maximal
$(2 \cdot n - 2 \cdot \log(n) - 2) + (2 \cdot n \cdot \log(n) - 2 \cdot n + 2 \cdot \log(n)) =$
$2 \cdot n \cdot \log(n) - 2$ Vergleiche (für n>1).
Im best case kann n·log(n) erreicht werden, da durch spezielle Folgen das Absinken beschränkt werden kann. Der Mittelwert allerdings wird ebenfalls dicht bei 2·n·log(n) liegen, da Heapsort keine Vorsortierungen oder günstigen Konstellationen ausnutzt und da das in die Wurzel vertauschte Element klein ist, also meist weit nach unten im Baum absinkt. Daher folgt:

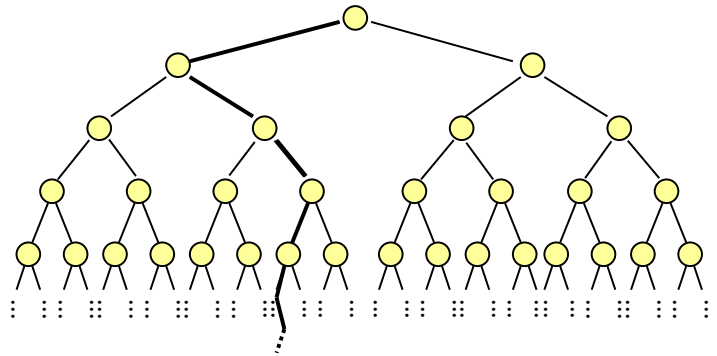
Satz 10.4.6:
Dieses normale Heapsort (aus dem Jahre 1962) benötigt im schlechtesten Fall höchstens 2·n·log(n) Vergleiche (für n>1).
(Im besten Fall kann man höchstens den Faktor 2 sparen. Der average case liegt recht nahe beim schlechtesten Fall.)

Geht es nicht doch noch besser?

Ja, man kann den Faktor 2 noch verkleinern (allerdings kann er nicht kleiner als "1" werden, wie in Satz 10.1.11 gezeigt wurde.)

Beobachtung: Beim eigentlichen Sortieren wird das letzte Element mit dem ersten vertauscht. Die Prozedur "sink" wird dieses letzte Element in der Regel sehr weit absenken, da es ja zu den leichten Elementen gehört hat, die sich ganz unten im Baum befinden. Man sollte daher das Element nicht von oben nach unten absenken, sondern es von unten nach oben (also "bottom-up") aufsteigen lassen. Hierzu muss man aber die Stelle, an der es einzufügen ist, kennen. Genau diese Stelle ermittelt man durch die Berechnung des "Einsinkpfads".

10.4.7 Einsinkpfad: Dies ist der Weg, den ein Element, das in die Wurzel gesetzt wurde, nehmen muss, damit die Heap-Eigenschaft wiederhergestellt wird (vgl. 10.4.4).
Dieser Pfad ist unabhängig vom einzusortierenden Element. Er endet in einem Blatt des Baumes. Es genügt, den Index dieses Blattes zu bestimmen.



Ermittlung des Einsinkpfads:

starte mit der Wurzel;
while noch nicht Blatt erreicht loop
    gehe zum Nachfolgeknoten mit dem größeren Inhalt;
end loop;

Bei der Darstellung mit Feldern muss man nur den Index j des letzten Elements auf dem Einsinkpfad kennen. Die anderen Knoten auf diesem Pfad besitzen die Indizes j div 2, (j div 2) div 2, ((j div 2) div 2) div 2, ..., 1. (In binärer Darstellung muss man also nur die letzte Ziffer streichen.)

Die folgende Funktion berechnet den Index j des letzten Elements des Einsinkpfads, wobei man den Fall, dass der letzte Knoten keinen Geschwisterknoten besitzt, gesondert berücksichtigt ("if m = rechts then ...").

Ermittlung des Einsinkpfads:

```
function einsinkpfad (rechts: 1..n) return 1..n is
j: 1..n := 1; m: Natural := 2;
begin
    while  m < rechts  loop
        if  A(m) < A(m+1)  then  j := m+1;  else  j:= m;  end if;
        m := j+j;
    end loop;
    if  m = rechts  then  j := rechts;  end if;
    return j;
end;
```


10.4.8 Bottom-up-Heapsort: (Carlsson 1987, Wegener 1993)

1. Ermittle den Index j des letzten Knotens auf dem Einsinkpfad.
2. Suche von j aus rückwärts entlang des Einsinkpfads die Stelle, wo das einzusortierende Element hingehört.
3. Füge es dort ein und schiebe alle darüber stehenden Elemente entlang des Einsinkpfads um eine Position in Richtung zur Wurzel.

Für die Programmierung benutzen wir die obige Funktion "einsinkpfad", die wir jedoch direkt in den Algorithmus integrieren. Weiterhin führen wir die Verschiebung von Punkt 3 bereits beim Berechnen von j durch, da man in der Regel den Einsinkpfad nur wenige Schritte zurücklaufen muss und hierdurch im Mittel ein doppeltes Durchlaufen vermieden wird.

```
procedure bottumupheapsort is                -- A und n sind global
procedure sink ... begin .... end sink;      -- wie früher
h, j, m: Natural; x: <Elementtyp>;
begin  h := n div 2;                         -- wandle A in einen Heap um
    for k in reverse 1..h loop  sink (k, n); end loop;
    for k in reverse 2..n loop               -- x = A(k) einsinken lassen
        x := A(k); A(k) := A(1);             -- rette A(1) nach A(k)
        j := 1; m:= 2;                       -- Suche den Index j
        while  m < k-1  loop
            if  A(m) < A(m+1)  then  A(j) := A(m+1); j := m+1;
            else  A(j) := A(m); j:= m;  end if;
            m := j+j;
        end loop;
        if  m = k-1  then  A(j) := A(k-1); j := k-1;  end if;
-- Nun ist der Index j (=Ende des Einsinkpfads) bekannt und alle Inhalte auf dem
-- Einsinkpfad sind um eine Position in Richtung der Wurzel verschoben worden.
```



```
-- Die Elemente des Einsinkpfads müssen nun zurückgeschoben werden,
-- solange die Stelle, an die x gehört, noch nicht erreicht ist.
        while  (j > 1) and then (A(j) < x)  loop
            i := j div 2; A(j) := A(i); j := i;
        end loop;
-- Die Stelle j, an die x gehört, ist nun erreicht.
    A(j) := x;
    end loop k;
end bottomupheapsort;
```


Wie viele Vergleiche benötigt Bottom-up-Heapsort?

1. Aufbau des Heaps: Genauso wie beim normalen Heapsort
maximal $2 \cdot n - 2 \cdot \log(n) - 2 \leq 2 \cdot n$ Vergleiche.

2. Sortierphase

Hier benötigt man maximal für jedes k so viele Vergleiche, wie die doppelte Länge des Einsinkpfads ist, also rund $2 \cdot \log(k)$.
Dies führt zunächst nur auf genau die gleiche Abschätzung wie beim normalen Heapsort.
Aber: In den meisten Fällen wird man bereits nach etwas mehr als log(k) Vergleichen fertig sein.
Experimente bestätigen, dass der Faktor "2" vom normalen Heapsort im Mittel auf "1" sinkt; dies lässt sich auch beweisen.
Man kann aber Folgen konstruieren, bei denen man nur auf den Faktor 1,5 kommt. Daher gilt im worst case nur:

Bottom-up-Heapsort benötigt im worst case maximal $1{,}5 \cdot n \cdot \log(n)$ Vergleiche. Dieser Fall tritt aber fast nie auf, so dass man in der Praxis von $n \cdot \log(n) + O(n)$ Vergleichen ausgehen kann. Carlsson konnte zeigen, dass im Mittel höchstens $n \cdot \log(n) + 0.67n$ Vergleiche benötigt werden.

10.4.9 Satz: Bottom-up-Heapsort benötigt
im schlimmsten Fall höchstens $1{,}5 \cdot n \cdot \log(n)$,
im Mittel höchstens $n \cdot \log(n) + 0.67 \cdot n$ Vergleiche.

Beachte: Heapsort und seine Varianten sind
garantierte $n \cdot \log(n)$ - Verfahren.

Hinweis: Es gibt weitere Varianten, z.B. von McDiarmid and Reed 1998 oder von Katajainen 1998. Ziel ist es, die untere theoretische Schranke von Satz 10.1.11 zu erreichen. Der Faktor 1 (statt 1,5) wurde mit dem "ultimativen" Heapsort erreicht, das aber (wegen eines zu hohen linearen Anteils) bisher noch ungeeignet ist (siehe Algorithmik-Vorlesung).

Diskussion dieses Ergebnisses:

Mit Bottomup-Heapsort haben wir ein Verfahren gefunden, das schneller als Quicksort sein müsste. Dies trifft vor allem dann zu, wenn im Algorithmus der Vergleich zwischen zwei Elementen, also die Abfragen "A(m) < A(m+1)" und "A(j) < x", viel mehr Zeit kosten als alle anderen elementaren Anweisungen; denn wir haben ja nur die Anzahl dieser Vergleiche gezählt. Diese Annahme gilt sicher, wenn die Schlüssel besonders lang (z.B. lange Wörter) sind.

Sind die Schlüssel jedoch Zahlen vom Typ Integer, so dauert der Vergleich nicht länger als eine Zuweisung der Form "A(j) := A(i)". Vor allem die Zugriffe auf den Hauptspeicher kommen nun zum Tragen. Bei Quicksort erfordert das Umspeichern 4 Zugriffe, da zwei Werte ausgetauscht werden, bei Heapsort braucht man nur 2, da ein Wert verschoben wird. Allerdings werden bei Heapsort bei jedem Absinken mindestens log(k), insgesamt also ungefähr n·log(n) Verschiebungen ausgeführt, während Quicksort mit weniger als ½·n·log(n) Vertauschungen auskommt. Insgesamt ist bei Heapsort also mit 2·n·log(n), bei Quicksort dagegen mit weniger als 2·n·log(n) Zugriffen auf den Hauptspeicher zu rechnen. Daher wird Quicksort in der Praxis trotzdem oft etwas schneller als Heapsort sein.

Auch die weiteren elementaren Anweisungen spielen eventuell noch eine Rolle. Für eine genaue (nicht uniforme) Abschätzung muss man nun die Implementierung und die Bearbeitungszeiten, die der verwendete Computers für die verschiedenen Anweisungen benötigt, kennen. Erfahrene Informatiker(innen) können dies gut einschätzen, oder sie ermitteln - aufbauend auf den theoretischen Resultaten - die tatsächlichen Laufzeiten mit Hilfe von Testläufen oder aus Statistiken des Betriebssystems.

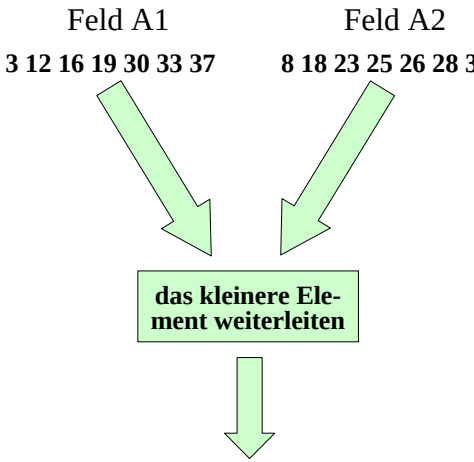
10.5 Mischen

Die bisherigen Sortierverfahren haben einzelne Elemente verglichen, die oft weit voneinander entfernt in der zu sortierenden Folge standen. Sie sind daher für riesige Datenbestände, die auf Hintergrundspeichern stehen, nur bedingt einsetzbar.

Daten werden von Hintergrundspeichern "stromartig" eingelesen, vergleichbar dem Lesen von Magnetbändern. Deshalb muss man immer möglichst viele Daten, die zusammenhängend verglichen und sortiert werden können, zusammenfassen und verarbeiten.

Hierfür ist vor allem das Zusammen-Mischen ("Verschmelzen") zweier bereits sortierter Datenströme geeignet.

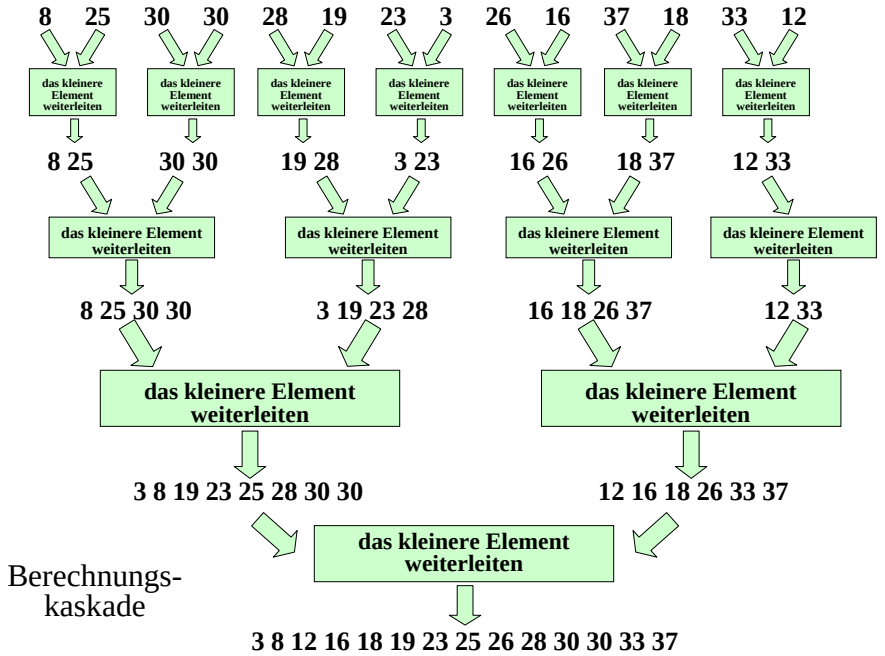
(Hinweis: Ob auf "≤" oder auf "<" abgefragt wird, ist wichtig, sofern man Elemente innerhalb eines Feldes vergleicht. Fragt man auf "<" ab, so ist das "Mischen" nicht stabil.)



Verschmelzen zweier Folgen

```
i:= 1; j := 1; k := 0;
while i≤n and j≤n loop
  k := k+1;
  if A1(i) ≤ A2(j)
  then B(k) := A1(i); i := i+1;
  else  B(k) := A2(j); j := j+1;
  end if;
end loop;
"Füge den Rest an B an"
```

Der gesamte Sortierprozess ist auf der nächsten Folie dargestellt.



10.5.1: Verschmelzen zweier Folgen vom Typ Vektor

```
procedure Verschmelzen (A1, A2: in Vektor; B: in out Vektor;
                        LA1, RA1, LA2, RA2, LB: in Integer;
                        RB: out Integer) is
i, j, k: Integer;
begin i:= LA1; j := LA2; k := LB-1;
  while i≤RA1 and j≤RA2 loop
    k := k+1;
    if A1(i) ≤ A2(j) then B(k) := A1(i); i := i+1;
                     else  B(k) := A2(j); j := j+1; end if;
  end loop;                  -- nun muss der Rest einer Folge noch angefügt werden
  if i ≤ RA1  then
        for m in i..RA1 loop k:=k+1; B(k):=A1(m); end loop;
  else  for m in j..RA2 loop k:=k+1; B(k):=A2(m); end loop; end if;
  RB := k;                   -- RB dient nur für Kontrollzwecke, kann entfallen
end Verschmelzen;
```


10.5.2: Sortieren durch Mischen

Es soll ein Feld A(1..n) durch Mischen sortiert werden.
Zuerst die "*Bottom-Up*-Denkweise", die zu einem Iterationsverfahren führt: Man verschmilzt zunächst je zwei Folgen der Länge 1 zur sortierten Folgen der Länge 2 (man muss hierfür n/2 mal "Verschmelzen" aufrufen),
dann verschmilzt man je zwei sortierte Folgen der Länge 2 zu sortierten Folgen der Länge 4 (hierfür muss man n/4 mal "Verschmelzen" aufrufen),
danach das Gleiche für sortierte Folgen der Länge 4, 8, 16 usw., bis zwei Folgen der Länge n/2 zu einer sortierten Folge der Länge n verschmolzen wurden.
Sofern n eine Zweierpotenz war, funktioniert dieses Verfahren bereits; im allgemeinen Fall muss man beim Verschmelzen unterschiedliche Längen von Folgen berücksichtigen (vgl. Beispiel).
Dieses Mischen heißt in der Literatur "straight mergesort".

Sortieren durch Mischen (Fortsetzung)

Nun die "*Top-Down*-Denkweise": Um ein Feld zu sortieren, sortiert man zuerst die linke Hälfte, dann die rechte Hälfte und verschmilzt die beiden sortierten Folgen. Das Ergebnis ist eine rekursive Prozedur.
Da dieses Vorgehen leichter zu verstehen und aufzuschreiben ist, wird die rekursive Version im Folgenden realisiert.
Der Ansatz ist einfach. Die Hauptschwierigkeit besteht hier in der präzisen Angabe der jeweiligen Teilfeld-Grenzen.
Wir verzichten auf volle Allgemeinheit und wollen "nur" das Feld A(L..R) sortieren. Sortierte Teilfelder A(x..y) und A(u..v) mischen wir in ein Hilfsfeld B(L..R) und schreiben danach das Ergebnis wieder nach A zurück.
Beachten Sie, dass die Abfrage "A(i) ≤ A(j)" die Stabilität des Mischens sichert (im Gegensatz zu "A(i) < A(j)").

10.5.3: Programm zum Sortieren durch Mischen ("Mergesort")

```
procedure Mergesort (L, R: in Integer) is
Mitte: Integer;  i, j, k: Integer;        -- Die Felder A und B sind global.
begin                                     -- Es wird A(L..R) sortiert.
if R > L then  Mitte := (L+R)/2;          -- Sortiere rekursiv zwei Hälften der Folge
   Mergesort(L, Mitte);  Mergesort(Mitte+1, R);
   i := L; j := Mitte+1; k := L-1;        -- Nun mischen von A nach B, wie in 10.5.1
   while i≤Mitte and j≤R loop  k := k+1;
      if A(i) ≤ A(j)  then B(k) := A(i); i := i+1;
                      else  B(k) := A(j); j := j+1; end if;
   end loop;                              -- nun muss der Rest einer Folge noch angefügt werden
   if i ≤ Mitte  then
      for m in i..Mitte loop k:=k+1; B(k):=A(m); end loop;
   else for m in j..R loop k:=k+1; B(k):=A(m); end loop; end if;
   for m in L..R loop A(m) := B(m); end loop;    -- zurückkopieren nach A
end if;
end Mergesort;
```

Wir haben das Verfahren mit Feldern realisiert.

Es ist aber klar, dass man es auch leicht mit linearen Listen implementieren kann, wobei nur Zeiger umgesetzt werden müssen. Hierbei kann man alle Umspeicherungen vermeiden. Vor allem der Teil im Programm, der mit

-- nun muss der Rest einer Folge noch angefügt werden

beginnt, kann durch eine einzige Zeigersetzung erledigt werden.
Durchdenken Sie die Vor- und Nachteile eines solchen Verfahrens und entwerfen Sie ein Programm, für das die zu sortierenden Daten als einfach verkettete lineare Liste vorliegen.

10.5.4 Aufwandsabschätzung für das Mischen:

Wenn V(n) die maximale Zahl der Vergleiche zum Sortieren von n Elementen ist, dann gilt:

V(1) = 0 und für alle n > 1: $V(n) = 2 \cdot V(n/2) + n-1$.

Die maximale Rekursionstiefe ist beim Mischen log(n), da in jedem Rekursionsschritt halbiert wird. Also muss die maximale Zahl der Vergleiche durch $n \cdot \log(n) - n/2 - n/4 - n/8 - \ldots - 1 = n \cdot \log(n) - n + 1$ beschränkt sein (das sieht man unmittelbar am Beispiel auf Folie 75). Dies ist auch die Lösung der obigen Gleichung für Zweierpotenzen n:

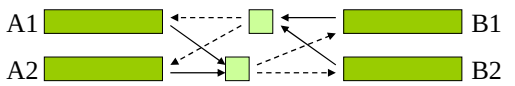
$V(n) = n \cdot \log(n) - n + 1$ (Beweis durch Einsetzen).

Mergesort ist also ein garantiertes n·log(n)-Verfahren.
Die Zahl seiner Vergleiche kommt sehr dicht an die untere theoretische Grenze heran, siehe Satz 10.1.11.

Bei Mergesort sind *viele Umspeicherungen* erst beim Verschmelzen von A nach B und dann zurück nach A erforderlich. Wie hoch ist die Zahl der Speichervorgänge?
In jeder Rekursionstiefe werden alle Daten genau einmal nach B und wieder zurück transportiert. Folglich werden insgesamt stets $4 \cdot n \cdot \log(n)$ Speicherzugriffe durchgeführt. (Dies setzt allerdings voraus, dass das Programm leicht modifiziert wird, dass also "if $A(i) \le A(j)$ then" nicht zu zusätzlichen Speicherzugriffen führt.)
Dies scheint viel zu sein. Man mache sich aber klar, dass bei Quicksort je Rekursionstiefe bis zu n/2 Vertauschungen stattfinden, die jeweils 4 Speicherzugriffe erfordern, weshalb Quicksort auch im günstigsten Falle, dass die Rekursionstiefe durch log(n) beschränkt bleibt, bis zu $2 \cdot n \cdot \log(n)$ Umspeicherungen ausführen kann.

10.5.5 Varianten des Sortierens durch Mischen

Das obige Vorgehen bezeichnet man als "*Zwei-Phasen-Mischen*", da die Daten von Feld A nach B verschmolzen (erste Phase) und anschließend zurück nach A (zweite Phase) kopiert werden. Man spricht auch von "*2-Wege-Mischen*", da ein Weg nach B und einer zurück nach A führt.
Natürlich kann man die Rolle der Ziel-Felder in jedem Durchgang ändern, d.h.: Man verschmilzt zwei sortierte Folgen von A1 und A2 abwechselnd auf die Felder B1 und B2 und anschließend zwei sortierte Folgen von B1 und B2 zurück nach A1 bzw. A2 usw. So spart man die Hälfte der Umspeicheroperationen. Man muss sich hierbei die jeweiligen Grenzen merken und bis zu 2n zusätzliche Speicherplätze bereitstellen.



Ein-Phasen-Mischen, wobei 4 Wege möglich sind.

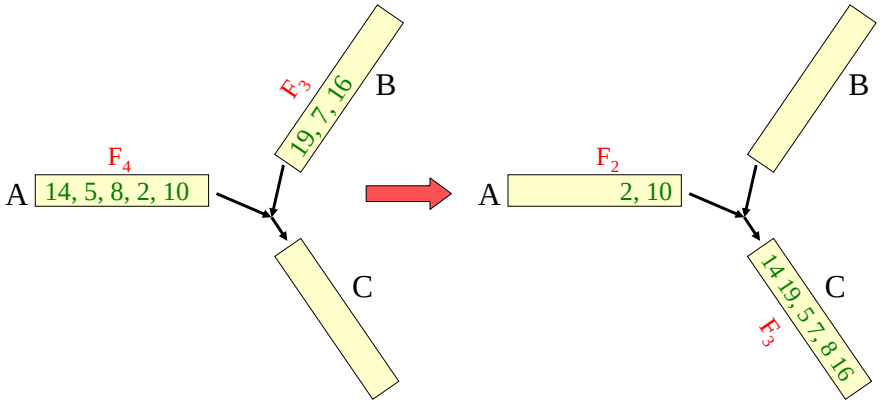
Die durchgezogenen Pfeile bezeichnen das Zusammenführen zweier sortierter Folgen. Das sortierte Ergebnis des Verschmelzens wird abwechselnd über einen der gestrichelten Pfeile weitergeleitet.

3-Wege-Mischen:

Man kann auch mit drei Feldern A, B und C arbeiten. Zuerst wird die zu sortierende Folge auf zwei Bänder A und B verteilt und zwar F_k sortierte Teilfolgen auf Band A und F_{k-1} sortierte Teilfolgen auf Band B (F_k ist die k-te Fibonaccizahl, siehe 8.4.7; falls die Anzahlen nicht "aufgehen", so fülle man mit "dummy"-Folgen auf die nächste Fibonaccizahl auf). Nun werden sortierte Teilfolgen von A und B nach C gemischt, bis das Feld B keine sortierte Folge mehr besitzt. Dann besitzen C genau F_{k-1} und A genau $F_k - F_{k-1} = F_{k-2}$ sortierte Teilfolgen. Nun werden die sortierten Teilfolgen von A und C nach B gemischt, bis das Feld A keine sortierte Teilfolge mehr besitzt (auf den Feldern B und C befinden sich nun F_{k-2} bzw. F_{k-3} sortierte Teilfolgen). Nun wird A das Ziel-Feld, d.h., es werden sortierte Teilfolgen von B und C nach A gemischt usw., bis am Ende auf einem der Bänder die sortierte Gesamtfolge steht. (Man wählt hier Fibonacci-Zahlen, weil man dann nicht ständig die Folgenlängen abfragen muss.)

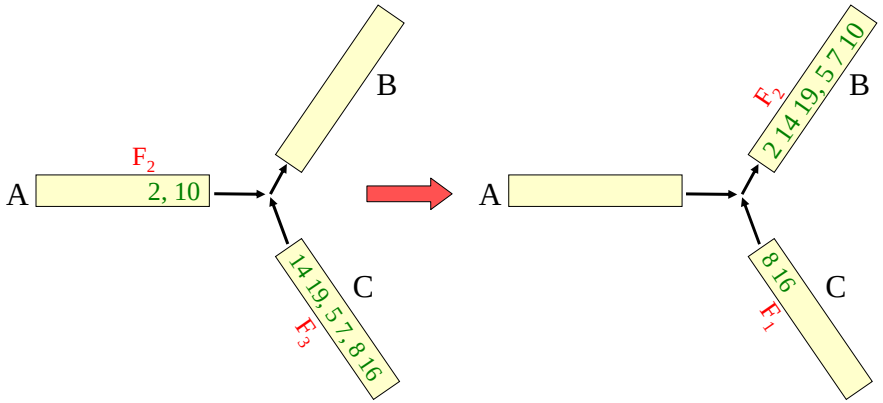
Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16

Dies sind $F_6 = 8$ Elemente. Verteile also $F_5 = 5$ und $F_4 = 3$ Elemente auf zwei Bänder A und B und mische diese auf Band C:



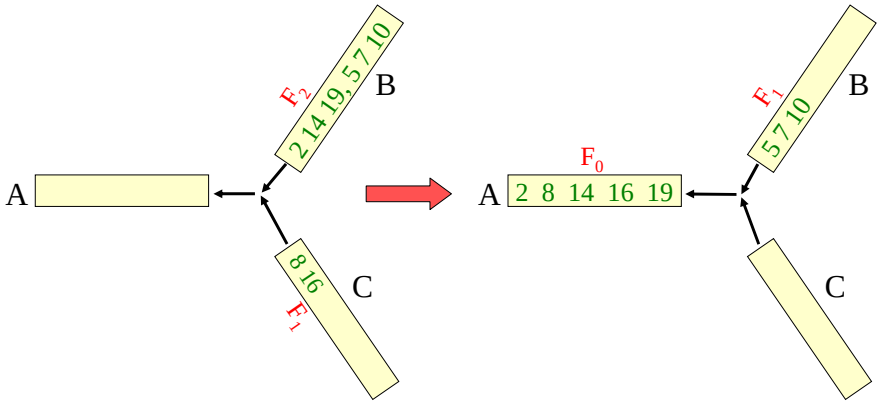
Verschmelze nun $F_4 = 3$ sortierte Teilfolgen von A mit $F_4 = 3$ sortierten Teilfolgen von B. Das Band B wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



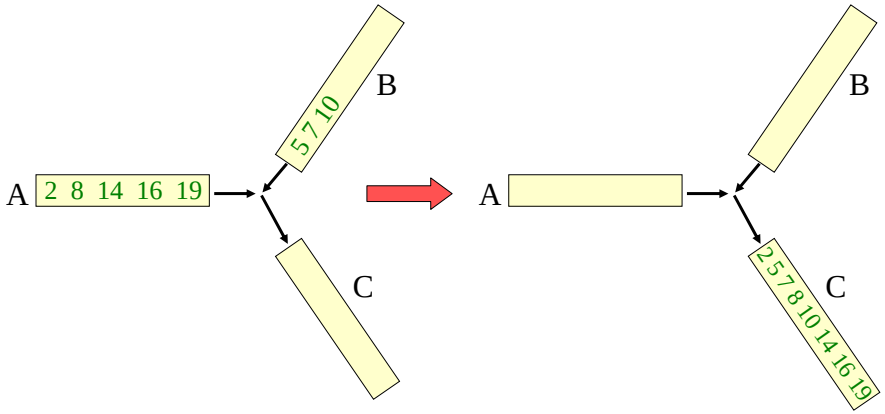
Verschmelze $F_3 = 2$ sortierte Teilfolgen von A mit $F_3 = 2$ sortierten Teilfolgen von C. Das Band A wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



Verschmelze $F_2 = 1$ sortierte Teilfolge von B mit $F_1 = 1$ sortierten Teilfolge von C. Das Band C wird hierbei leer.

Beispiel: Zu sortierende Folge: 14 5 8 2 10 19 7 16



Verschmelze $F_1 = 1$ sortierte Teilfolge von A mit $F_0 = 1$ sortierten Teilfolge von B auf Band C. Fertig.

10.5.6 Natürliches Mischen:

Eine Teilfolge $a_i a_{i+1} \dots a_j$ der Folge $a_1 a_2 \dots a_n$ heißt ein Lauf, wenn dies eine maximal lange nicht fallende Teilfolge ist, d.h., wenn $a_{i-1} > a_i \le a_{i+1} \le \dots \le a_j > a_{j+1}$ gilt (wenn $i = 1$ oder $j = n$ ist, so entfallen die entsprechenden Ungleichungen am Rande).

Statt Teilfolgen der Länge 1, dann der Länge 2, 4, ... usw. zu mischen, kann man in jedem Durchgang die vorhandenen Läufe mischen, wodurch sich die Rekursionstiefe und damit die Zahl der Umspeicherungen verringert, die Zahl der Abfragen aber wegen des Tests, wo ein Lauf endet, insgesamt nicht geringer wird. Ein solches Ausnutzen der zufällig vorhandenen Teil-Sortierungen bezeichnet man als *natürliches Mischen*. Dadurch wird das Mischen mit seiner garantierten $n \cdot \log(n)$-Laufzeit auch zu einem ordnungsverträglichen Sortierverfahren.

Aufgabe: Programmieren Sie das "natürliche 3-Wege-Mischen".

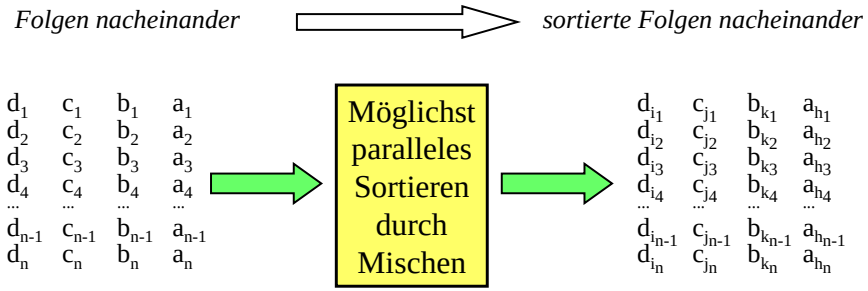
10.5.7 Parallelisieren

Hat man einen Baustein, der das Verschmelzen zweier Folgen vornimmt, so kann man in jeder Rekursionstiefe alle Operationen parallel ausführen (siehe die Kaskade zu Beginn von 10.5; in der obersten Zeile kann man alle n/2 Vergleiche parallel zueinander durchführen).

Man benötigt dann n/2 solcher Bausteine zum Mischen von Teilfolgen der Länge 1, man braucht n/4 dieser n/2 Bausteine für das Verschmelzen aller Teilfolgen der Länge 2 usw. Durch geschicktes Zusammenschalten kann man also die Kaskade mit n/2 solcher Bausteine realisieren.

Wie lange dauert dann das Sortieren? Die erste Schicht benötigt genau 2 "Takte", die nächste 4, die nächste 8 usw. bis zur letzten Schicht mit n Takten. Folglich kann man wegen $2+4+8+\dots+n/4+n/2+n = 2n-2$ die n Daten mit diesem Parallelverfahren in $2n-2$ Schritten sortieren. Der "Preis" hierfür sind die n/2 Bausteine und ihr Verschaltungsnetz.

Bringt dieses Verfahren trotzdem Vorteile, z.B. im Falle, dass man viele Zahlenfolgen a, b, c, d, ... sortieren möchte?



Faktisch bringt unser Mischverfahren leider kaum etwas, weil ein "Pipelining" zwar denkbar ist, aber wegen der sequenziellen Abarbeitung des letzten Schritts mit mindestens n/2 Schritten zu entsprechend langen Wartezeiten führt. Ein schnelles Verfahren, das das ständige Einschleusen der nächsten Folge in den Verarbeitungsprozess erlaubt, stellen wir in 10.7 vor.

10.6 Streuen und Sammeln

Manchmal liegen die zu sortierenden Werte $a_1\, a_2 \ldots a_n$ in einem festen Intervall [UNT, OB]: UNT ≤ a_i ≤ OB.
Der Einfachheit halber nehmen wir an, die a_i seien natürliche Zahlen und es seien UNT = 0 und OB = m-1.

Bucket sort: Verteile ("streue") die n Elemente A(1..n) auf m nacheinander angeordnete Fächer bucket(0..m-1) ("Eimer" genannt, daher "bucket sort"), hole sie anschließend in der Reihenfolge der Fächer wieder heraus und lege sie im Feld A ab.

Aufwand: O(n+m) sowohl für die Zeit als auch für den Platz.

Einsatz: Vor allem, wenn m in der Größenordnung von n liegt.

Programmstück (fügen Sie die Listenbearbeitung selbst hinzu):


```
declare A: array (1..n) of Integer; k: 0..n;     -- n ist global
bucket: array (0..m-1) of "liste von Integer"; ...
begin
  for j in 0..m-1 loop bucket(j) := "leer"; end loop;
  for i in 1..n loop                              -- streuen
    "hänge A(i) an bucket(A(i)) an";
  end loop;
  k := 0;
  for j in 0..m-1 loop                            -- sammeln
    while "bucket(j) nicht leer" loop
        k := k+1; A(k) := "erstes Element von bucket(j)";
        "Entferne aus bucket(j) das erste Element"; end loop;
  end loop;
end;
```


10.7 Paralleles Sortieren

Das Sortieren von n Elementen kostet mindestens n·log(n) Vergleiche, wenn nur ein Prozessor vorhanden ist. Ein guter Rechner kann heute etwa 10 Millionen Vergleiche pro Sekunde durchführen, sofern man 32-stellige Zahlen als Schlüssel verwendet. Setzt man für die Umspeicherungen usw. den Faktor 10 an, so lassen sich mit einem Programm 1 Million Vergleiche einschl. der übrigen Operationen pro Sekunde durchführen. Will man maximal eine Minute auf das Ergebnis warten, so lassen sich also $60 \cdot 10^6$ Vergleiche ausführen. Berechne n so, dass $n \cdot \log(n) = 60 \cdot 10^6$ gilt, d.h., es lassen sich knapp 3.000.000 Schlüssel in einer Minute sortieren.

Solche Größenordnungen sind selten, so dass im Prinzip das Sortieren heute kein Problem mehr darstellen sollte. Allerdings entstehen Probleme, wenn eine Sortierung sehr schnell erfolgen muss, weil sicherheitskritische Systeme hiervon abhängen oder andere Gründe vorliegen.

Will man also schneller sortieren, so kann man entweder auf noch schnellere Rechner warten oder man kann eine Beschleunigung des Sortierens durch paralleles Vorgehen erhoffen. Wir stellen hierzu zwei „einfache“ Verfahren vor. (Weitere Verfahren finden Sie z.B. in dem Buch von Ingo Wegener, "Effiziente Algorithmen für grundlegende Funktionen", Teubner-Verlag.)

Verfahren 1:

Lineare Kette mit n Prozessoren.

Verfahren 2:

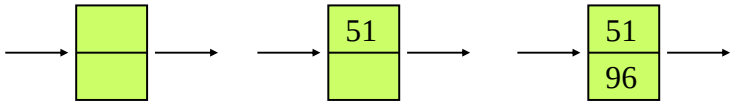
Divide and Conquer Verfahren mit $(1/4) \cdot n \cdot \log^2(n)$ Prozessoren.

10.7.1 Verfahren 1: Lineare Kette

Hierzu betrachten wir einen Prozessor, der

zwei Speicherplätze für Zahlen,
einen Eingang (links) und
einen Ausgang (rechts)

besitzt:



Grundbaustein, leer

Grundbaustein mit einer Zahl

Grundbaustein mit zwei Zahlen

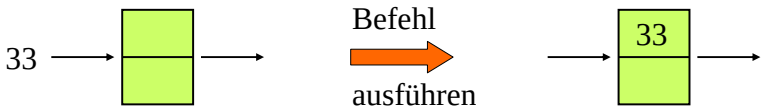
(Die Zahl darf auch unten stehen.)

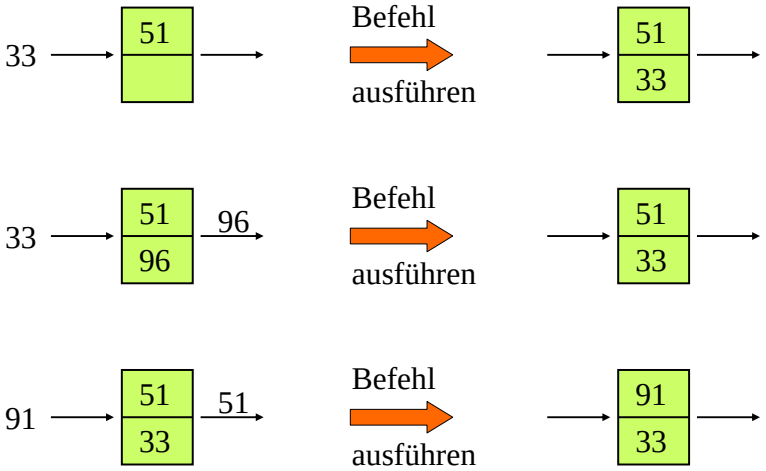
Der Grundbaustein kann nur folgenden Befehl ausführen:

1. Falls er zwei Zahlen besitzt, gibt er die größere der beiden nach rechts aus.
2. Falls eine Zahl von links kommt, legt er sie in einen seiner freien Speicherplätze.

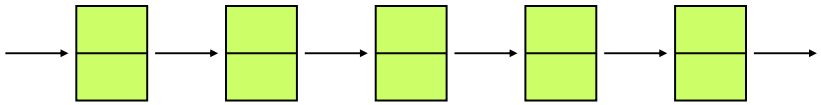
Die Befehlsteile 1. und 2. werden gleichzeitig ausgeführt!

Wenn die Bedingung in 1. und/oder in 2. zutrifft, dann muss der Befehl auch ausgeführt werden.

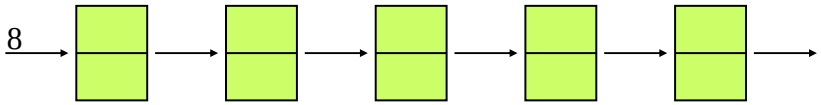


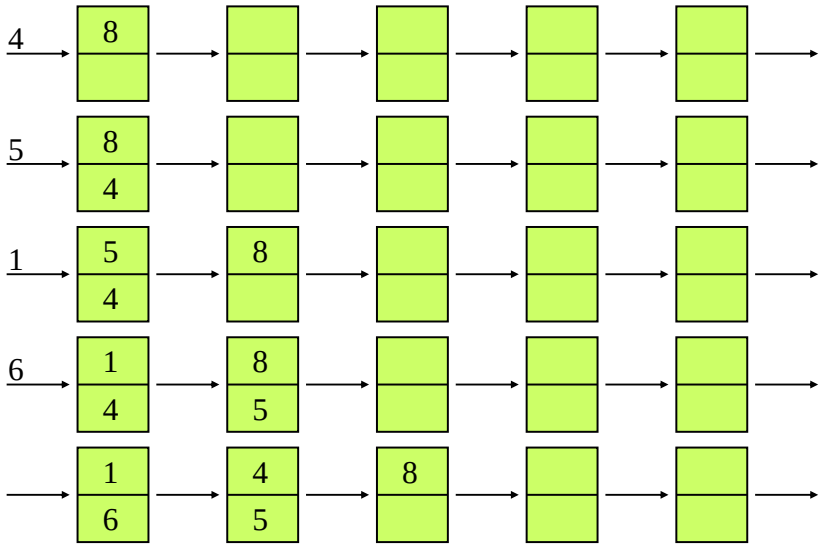


Nun koppeln wir n=5 solche Grundbausteine zu einer Kette aneinander:

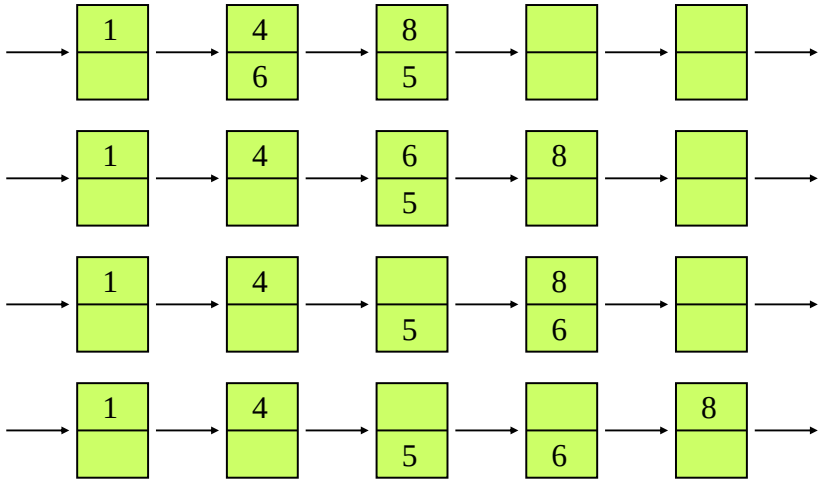


Wir verlangen, dass alle n Bausteine synchron arbeiten. Nun geben wir in n=5 Takten die Zahlen 8, 4, 5, 1, 6 von links ein:





Die Eingabe ist beendet. Die Kette arbeitet aber noch weiter, bis in jedem Baustein genau eine Zahl steht.



Nach 2n-1 Takten endet das Verfahren und jeder Baustein gibt im 2n-ten Takt seine Zahl aus. Diese Folge ist sortiert.

10.7.2 Wie viele Schritte benötigt dieses Verfahren?

Offensichtlich sind n Werte nach **2n-1** Schritten sortiert.

Hierfür benötigt man n Prozessoren.
Für große Werte von n ist dies eine deutliche Verbesserung gegenüber den bisherigen O(n·log(n))-Verfahren.

Hinweis: In 10.5.7 haben wir ein Verfahren skizziert, wie man n Werte schon mit n/2 Prozessoren parallel in 2n-2 Schritten sortieren kann. Jene Prozessoren hatten aber zwei Eingänge und waren komplexer verschaltet.

Gibt es bessere Verfahren? Möglichst solche, die viel schneller als in O(n) Schritten arbeiten - dafür dürfen sie dann auch mehr als O(n) Prozessoren besitzen ...

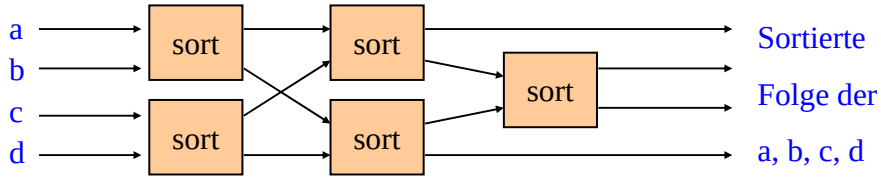
10.7.3 Verfahren 2: Divide and Conquer Ansatz

Der Divide-and-Conquer-Ansatz lautet:
Zerlege das Problem in z.B. zwei Teilprobleme, löse diese und setze aus den Teillösungen die Gesamtlösung zusammen.

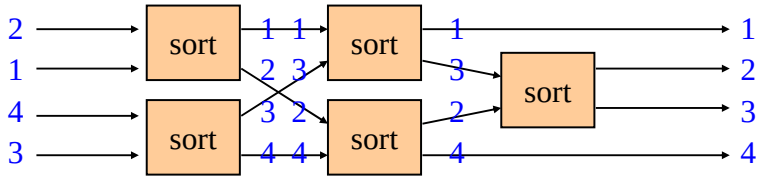
Wir gehen nun von einer allgemeinen Struktur aus, bei der die zu sortierenden Elemente nicht nacheinander, sondern parallel zueinander eingegeben werden. Dies bedeutet, dass wir mit mindestens O(n) Grundbausteinen rechnen müssen, um n Elemente zu sortieren.

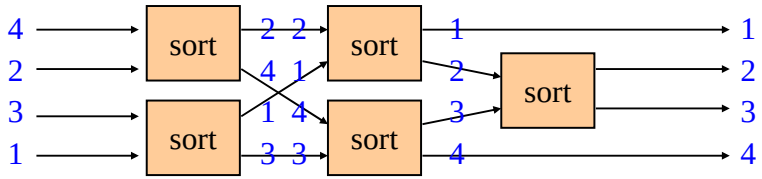
10.7.4 Als Grundbaustein verwenden wir einen elementaren Sortierbaustein **sort** für zwei Werte:

Hiermit kann man beispielsweise n = 4 Werte wie folgt sortieren:



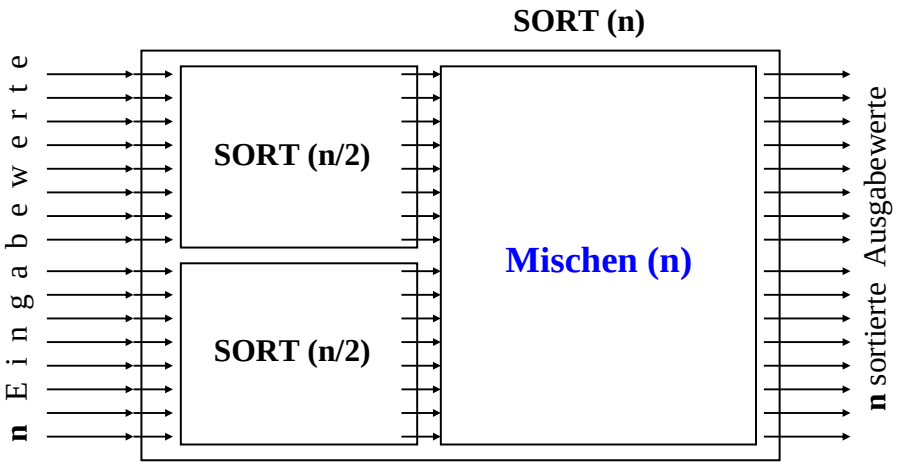
Beispiele:





4 Elemente werden in 3 Schritten mit 5 Bausteinen sortiert.

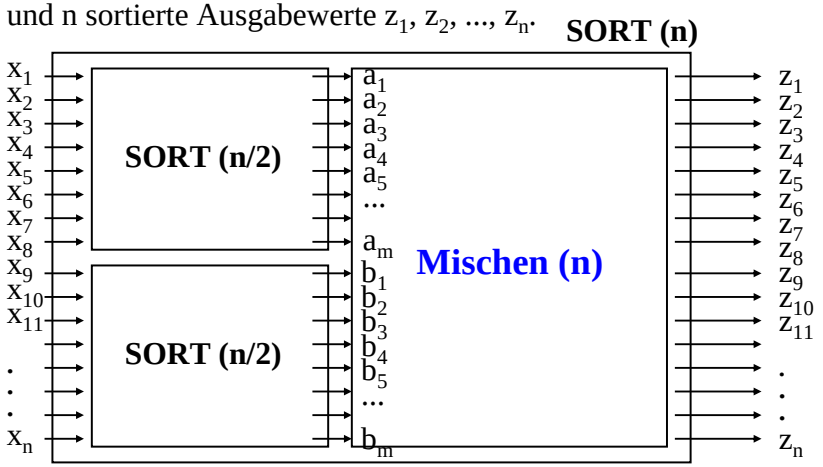
10.7.5 Divide and Conquer Ansatz für n Eingabewerte:



Genauer: Gegeben sind n Eingabewerte $x_1, x_2, ..., x_n$.
Wir erhalten 2 sortierte Folgen mit je m Elementen:

$a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ $(m = n/2)$

und n sortierte Ausgabewerte $z_1, z_2, ..., z_n$.



Die Rekursion sorgt dafür, dass die Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ sortiert sind (m=n/2).
Wenn es einen "guten" Baustein **Mischen(n)** gibt, dann lässt sich hieraus auch ein "gutes" Sortierwerk für $n = 2^s$ Eingabewerte, für jede natürliche Zahl s, konstruieren.

In der Tat gibt es eine Technik, um zwei sortierte Folgen *parallel* in relativ wenigen Schritten zu einer gemeinsamen sortierten Folge zu mischen.

Diese Technik nennt sich "odd-even-merge" (dtsch.: gerade-ungerad-Mischen) und wird ebenfalls rekursiv definiert.

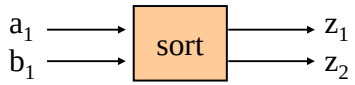
Definition 10.7.6: odd-even-merge "Mischen(2m)"
Gegeben: zwei sortierte Folgen (für $m = 2^k$ für eine natürliche Zahl $k \geq 0$) $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$.
Ergebnis: die zugehörige sortierte Folge $z_1, z_2, ..., z_{2m}$.
Vorgehen zur Durchführung von "Mischen (2m)":
m = 1: Vergleiche mit dem Baustein "**sort**"; fertig.
m > 1:
(1) Mische rekursiv die ungeraden Glieder der a- und der b-Folge sowie die geraden Glieder der a- und der b-Folge, d.h., mische die sortierten Folgen $a_1, a_3, a_5, ..., a_{m-1}$ und $b_1, b_3, b_5, ..., b_{m-1}$ zur sortierten Folge $u_1, u_2, u_3, ..., u_m$ und mische die sortierten Folgen $a_2, a_4, a_6, ..., a_m$ und $b_2, b_4, b_6, ..., b_m$ zur sortierten Folge $v_1, v_2, v_3, ..., v_m$.
(2) $z_1 = u_1$, $z_{2m} = v_m$ und führe parallel m-1 Vergleiche durch: $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für i=1,2,..,m-1.

Beispiel 10.7.7 für m = 1, 2, 4.

Wir konstruieren zunächst das Mischen für einige Werte von m.

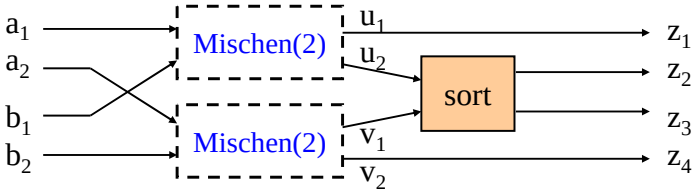
Der Fall m=1 benötigt genau einen Sortierbaustein:

m = 1:

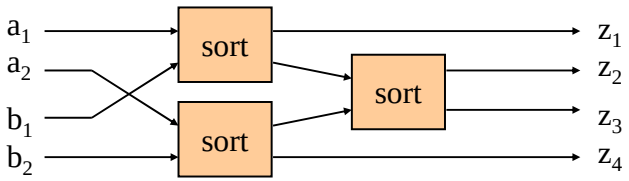


Dies ist der Baustein Mischen(2).

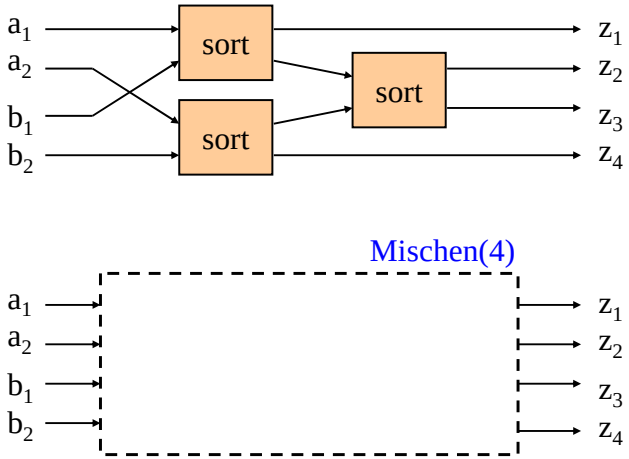
m = 2: (beachte, dass a_1, a_2 bzw. b_1, b_2 sortiert sind.)



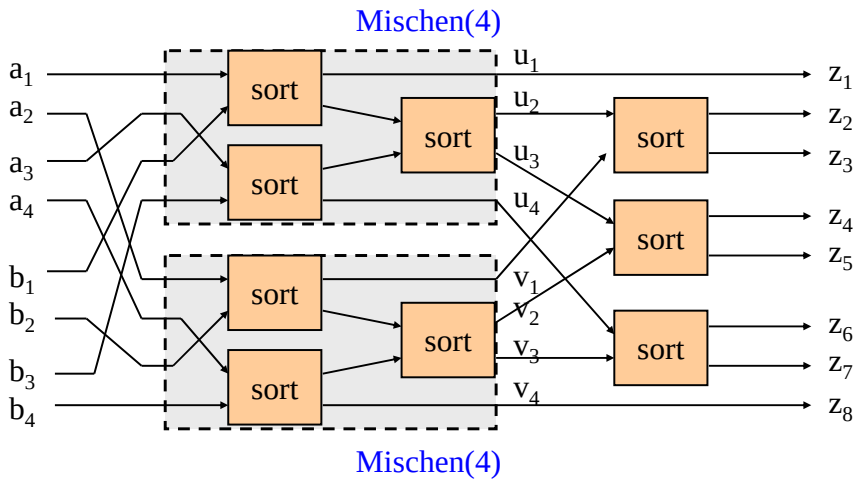
Einsetzen von Mischen(2) ergibt den Baustein Mischen(4):



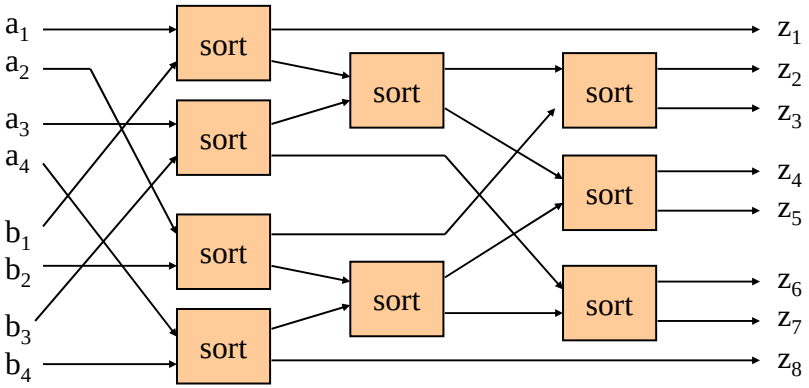
Wir kürzen diesen Baustein durch Mischen(4) ab:



m = 4:

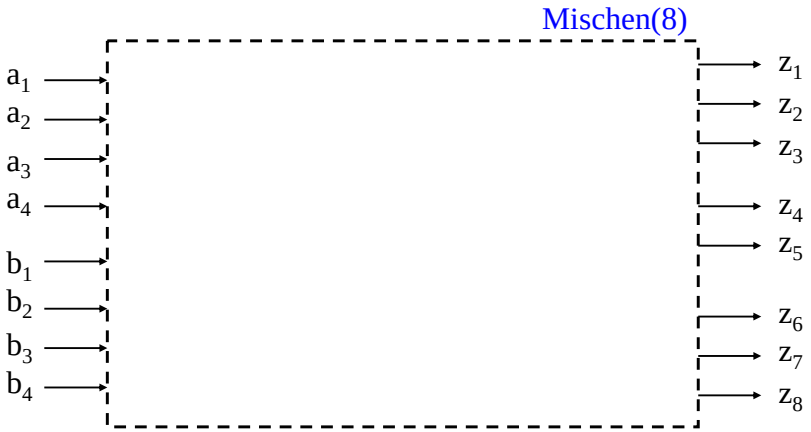


m = 4: Bereinigung der Verbindungen ergibt:

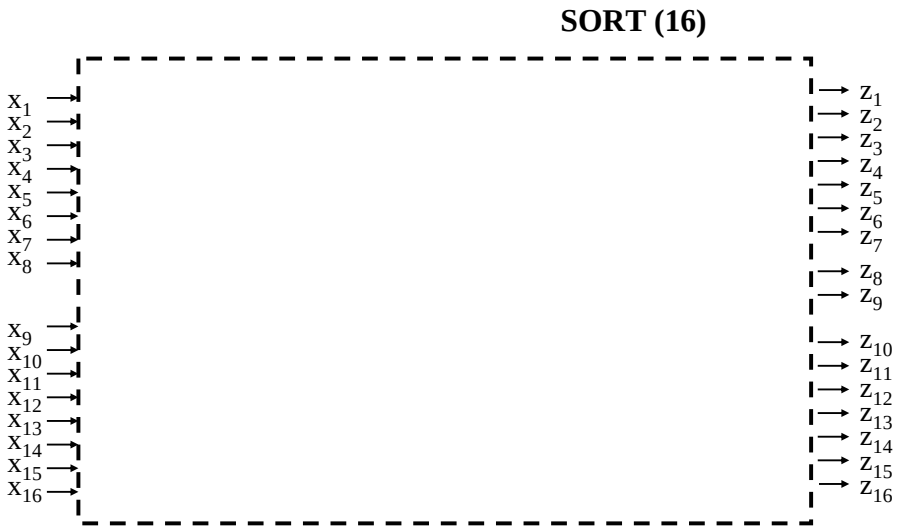


Beachte: a_1, a_2, a_3, a_4 bzw. b_1, b_2, b_3, b_4 sind sortierte Folgen.

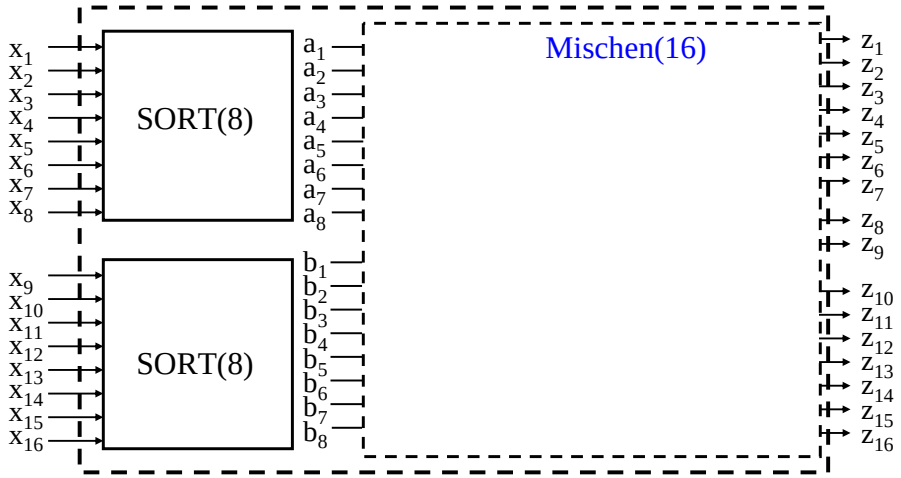
m = 4: Dies kürzen wir erneut ab durch



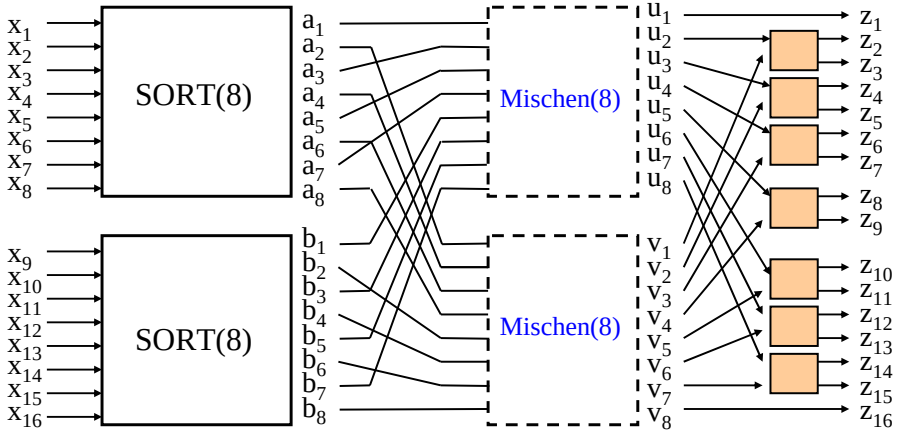
10.7.8 Betrachte nun den Gesamtalgorithmus SORT für n=16:



Rekursives Ersetzen für SORT(16) ergibt:

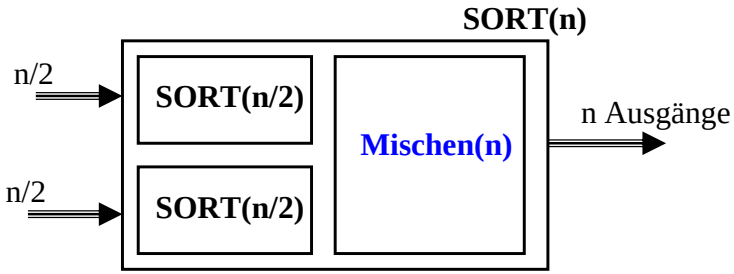


Einsetzen von Mischen(16) ergibt die Struktur:



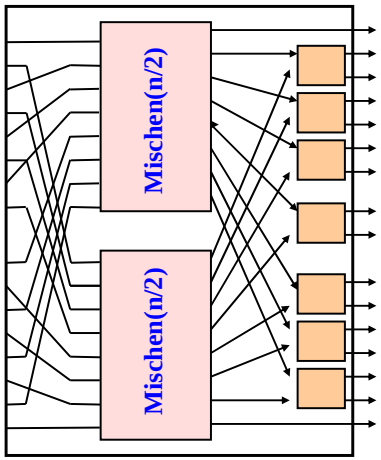
Setzt man alle kleineren schon konstruierten Bausteine ein, so erhält man den ausführbaren Algorithmus (selbst durchführen, vgl. dann letzte Folie in Kap. 10).

10.7.9 Die rekursive Struktur des parallelen Sortierens:



Anzahl G(n) der Bausteine **sort**: G(2) = 1 und für $n=2^k$, $k>1$:
$G(n) = 2 \cdot G(n/2) + M(n)$,
wobei M(n) die Anzahl **sort** in **Mischen(n)** ist.

10.7.10 Mischen(n) ist ebenfalls rekursiv definiert:



Mischen(n)

Anzahl M(n) der Bausteine **sort** im Algorithmus Mischen(n):
M(2) = 1 und für n>2: M(n) = 2·M(n/2) + n/2-1.

Nachdem der Aufbau des parallelen Sortieralgorithmus klar ist, müssen wir beweisen, dass die Technik "odd-even-merge" die beiden geordneten Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ tatsächlich zu der geordneten Folge $z_1, z_2, ..., z_{2m-1}, z_{2m}$ zusammenmischt.

Satz 10.7.11:

odd-even-merge sortiert die geordneten Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$ zur geordneten Folge $z_1, z_2, ..., z_{2m-1}, z_{2m}$.

Beweis:
Wenn m=1 ist, dann werden zwei Zahlen durch den Sortierbaustein **sort** geordnet und der Satz ist richtig.

Sei daher m>1. Gegeben sind zwei sortierte Folgen $a_1, a_2, ..., a_m$ und $b_1, b_2, ..., b_m$. Nach Definition von odd-even-merge werden die sortierten Teilfolgen mit jeweils m/2 Elementen $a_1, a_3, a_5, ..., a_{m-1}$ und $b_1, b_3, b_5, ..., b_{m-1}$ zur sortierten Folge $u_1, u_2, u_3, ..., u_m$ bzw. $a_2, a_4, a_6, ..., a_m$ und $b_2, b_4, b_6, ..., b_m$ zur sortierten Folge $v_1, v_2, v_3, ..., v_m$ rekursiv gemischt.

Die Ergebnisfolge ist definiert durch $z_1 = u_1$, $z_{2m} = v_m$ und $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für i=1,2,..,m-1. Wir zeigen, dass die z-Folge hierdurch korrekt sortiert ist.

u_1 muss das Minimum der beiden Folgen sein, da a_1 und b_1 die Minima der a- bzw. b-Folge und beide Elemente in der u-Folge sind. Analog gilt, dass v_m das Maximum der Ergebnisfolge sein muss. Also sind z_1 und z_{2m} richtig bestimmt worden.

Um zu zeigen, dass $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ richtig festgelegt wurden, genügt es zu zeigen, dass sich das Element u_{i+1} in der sortierten Ergebnisfolge an der Position 2i oder 2i+1 befinden muss (analog muss man dies für das Element v_i nachweisen). Wir beweisen dies in acht Schritten (a) bis (h).

Wir betrachten u_{i+1} für ein i zwischen 1 und m-1. O.B.d.A. nehmen wir an, dass dieses Element aus der a-Folge stammt und das j-te Element der Teilfolge mit den ungeraden Indizes ist, d.h., es gibt ein j mit $1 \le j \le m/2$ mit $u_{i+1} = a_{2j-1}$.

(a) Wie viele der Elemente der a-Folge sind kleiner als u_{i+1}?
Genau 2j-2 Elemente;
denn weil $u_{i+1} = a_{2j-1}$ ist, müssen die Elemente $a_1, a_2, ..., a_{2j-2}$ der geordneten a-Folge kleiner als u_{i+1} sein.

(b) Wie viele der Elemente der b-Folge sind kleiner als u_{i+1}?
Mindestens 2i-2j+1 Elemente. Beweis hierfür:
Wegen $u_{i+1} = a_{2j-1}$ und wegen $a_1 \le a_3 \le a_5 \le ... \le a_{2j-3}$ müssen sich unter den ersten i Elementen $u_1, u_2, ..., u_i$ der u-Folge genau diese (j-1) Elemente aus der a-Folge befinden. Folglich müssen unter diesen ersten i Elementen der u-Folge genau i-(j-1) = i-j+1 Elemente der b-Folge sein. Da in der u-Folge aber nur die Elemente mit ungeradem Index sind, müssen dies genau die Elemente $b_1, b_3, b_5, ..., b_{2(i-j+1)-1}$ sein. Da die b-Folge sortiert ist, müssen daher mindestens die Elemente $b_1, b_2, b_3, ..., b_{2(i-j+1)-1}$ kleiner als u_{i+1} sein. Ihre Anzahl ist 2i-2j+1.

(c) Folglich stehen in der sortierten Ergebnisfolge mindestens
2j-2 + 2i-2j+1 = 2i-1 Elemente vor u_{i+1}, d.h., in der Ergebnisfolge kann u_{i+1} frühestens das Element z_{2i} sein.

(d) Wie viele der Elemente der a-Folge sind größer als u_{i+1}?
Genau m-2j+1 Elemente,
denn wegen $u_{i+1} = a_{2j-1}$ müssen $a_{2j}, a_{2j+1}, ..., a_m$ größer als u_{i+1} sein.

(e) Wie viele der Elemente der b-Folge sind größer als u_{i+1}?
Mindestens m-2i+2j-2 Elemente. Beweis hierfür:
Wegen (b) muss $u_{i+1} \le b_{2(i-j+1)+1}$ sein; denn sonst wäre u_{i+1} nicht das (i+1)-te, sondern ein späteres Element der u-Folge. Also müssen mindestens die Elemente $b_{2(i-j+1)+1}, b_{2(i-j+1)+2}, ..., b_m$ größer als u_{i+1} sein. Ihre Anzahl ist m - (2i-2j+3) + 1 = m-2i+2j-2.

(f) Folglich stehen in der sortierten Ergebnisfolge mindestens
m-2j+1 + m-2i+2j-2 = 2m-2i-1 Elemente hinter u_{i+1}, d.h.,
in der sortierten Ergebnisfolge muss u_{i+1} spätestens das Element $z_{2m-(2m-2i-1)} = z_{2i+1}$ sein.

(g) Nun führe man genau die gleiche Untersuchung für v_i durch.
Auch hier erhält man, dass v_i frühestens das Element z_{2i} und spätestens das Element z_{2i+1} in der Ergebnisfolge sein kann.
(Führen Sie diesen Beweis selbst analog zu (a) bis (f).)

(h) Aus (f) und (g) folgt nun, dass die Elemente u_{i+1} und v_i sich an den Positionen 2i und 2i+1 der z-Folge befinden müssen.
Daher muss $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$ für die sortierte Ergebnisfolge $z_1, z_2, ..., z_{2m}$ gelten.

Damit ist der Satz bewiesen.

Skizze zum Beweis: O.B.d.A. sei $u_{i+1} = a_{2j-1}$.

Es sind kleiner als u_{i+1}		Es sind größer als u_{i+1}
$a_1 \le a_2 \le ... \le a_{2j-2}$		$a_{2j} \le a_{2j+1} \le ... \le a_m$
$b_1 \le b_2 \le ... \le b_{2(i-j+1)-1}$	$\le u_{i+1} \le$	$b_{2(i-j+1)+1} \le b_{2(i-j+1)+2} \le ... \le b_m$

$z_1, z_2, ..., z_{2i-2}, z_{2i-1}$ z_{2i} z_{2i+1} $z_{2i+2}, z_{2i+3}, ..., z_{2m}$

[Wenn $u_{i+1} = a_{2j-1}$ ist, so ist $v_i = b_{2(i-j+1)}$. Beide müssen an den Positionen 2i oder 2i+1 in der sortierten z-Ergebnisfolge stehen. Es ist nur noch zu prüfen, ob a_{2j-1} kleiner oder größer als $b_{2(i-j+1)}$ ist. Folglich gilt $z_{2i} = \text{Min}(u_{i+1}, v_i)$, $z_{2i+1} = \text{Max}(u_{i+1}, v_i)$.]

10.7.12 Zur Komplexität (Größe, Tiefe, Breite):
Wie groß, wie breit und wie tief ist dieser parallele Sortieralgorithmus?

"Größe" soll ein Maß für die "Herstellungskosten" sein, wenn man den Algorithmus hardwaremäßig realisiert:
G(n) = Anzahl der Bausteine **sort** in **SORT(n)**.

"Tiefe" soll die Laufzeit des Sortierens angeben:
T(n) = Länge des längsten gerichteten Weges durch Bausteine **sort** in **SORT(n)**.

"Breite" soll die Anzahl der Mikroprozessoren angeben, die für eine Softwarelösung benötigt werden:
B(n) = Minimale Zahl der Bausteine **sort**, die parallel zueinander liegen müssen, ohne die Tiefe zu verändern.

Wir gehen nun von 10.7.9 und 10.7.10 aus:

Anzahl G(n) der Bausteine **sort**: G(2) = 1 und für $n=2^k$, k>1:
G(n) = 2·G(n/2) + M(n),
wobei M(n) die Anzahl **sort** in Mischen(n) ist:

M(2) = 1 und für $n=2^k$, k>1: M(n) = 2·M(n/2) + n/2-1.

Für die Tiefe gilt: T(n) = T(n/2) + "Tiefe von Mischen(n)".

Die Breite beträgt B(n) = n/2. Denn von jedem Eingang x_i muss ein längster Weg ausgehen und die Anzahl der Bausteine, die unmittelbar hinter den Eingängen liegen, beträgt n/2. Im weiteren Verlauf der Rekursion kommt man mit dieser Zahl auch aus, siehe die Formeln für SORT(n) und für Mischen(n). Dies lässt sich auch formal beweisen, worauf wir hier verzichten.

Einige Größen der Mischen-Bausteine: M(2) = 1, M(4) = 3, M(8) = 9, M(16) = 25, M(32) = 65. Aus M(n) = 2·M(n/2) + n/2-1 gewinnt man durch Einsetzen rasch die Gleichungen:

$$
\begin{aligned}
M(n) &= 2\cdot M(n/2) + n/2-1\\
&= 2\cdot(2\cdot M(n/4) + n/4-1) + n/2-1\\
&= 4\cdot M(n/4) + 2\cdot n/2 - 1 - 2\\
&= 4\cdot(2\cdot M(n/8) + n/8 -1) + 2\cdot n/2 - 1 - 2\\
&= 8\cdot M(n/8) + 3\cdot n/2 - 1 - 2 - 4\\
&= \ldots\\
&= 2^{\log(n)-1}\cdot M(2) + (\log(n)-1)\cdot n/2 - (2^{\log(n)-1} - 1)\\
&= n/2 + n/2\cdot\log(n) - n/2 - n/2 + 1\\
&= n/2 \cdot (\log(n) - 1) + 1
\end{aligned}
$$

Es gilt also $M(n) = n/2 \cdot (\log(n) - 1) + 1$.
(Vgl. die ähnliche Gleichung in 10.5.4 für V(n).)

Hiermit können wir nun die "Größe" G(n) ausrechnen:

$$
\begin{aligned}
G(n) &= 2\cdot G(n/2) + M(n)\\
&= 2\cdot G(n/2) + n/2 \cdot (\log(n) - 1) + 1\\
&= 2\cdot(2\cdot G(n/4) + n/4 \cdot (\log(n/2) - 1) + 1) + n/2 \cdot (\log(n) - 1) + 1\\
&= 4\cdot G(n/4) + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2\\
&= 4\cdot(2\cdot G(n/8) + n/8 \cdot (\log(n/4) - 1) + 1)\\
&\qquad + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2\\
&= 8\cdot G(n/8) + n/2 \cdot (\log(n) - 3)\\
&\qquad + n/2 \cdot (\log(n) - 2) + n/2 \cdot (\log(n) - 1) +1 +2 + 4\\
&= \ldots\\
&= 2^{\log(n)-1}\cdot G(2) + n/2\cdot(1+2+\ldots+(\log(n) - 1)) +1+2+\ldots+2^{\log(n)-2}\\
&= 2^{\log(n)-1} + n/4\cdot\log(n)\cdot(\log(n)-1)) + 2^{\log(n)-1} - 1\\
&= (n/4)\cdot\log(n)\cdot(\log(n)-1)) + n - 1.
\end{aligned}
$$

Ergebnis: $\mathbf{G(n) = (n/4)\cdot\log(n)\cdot(\log(n)-1)) + n - 1} \in O(n\cdot\log^2(n))$.

Entscheidend für den praktischen Einsatz ist die Tiefe T(n).
Hierzu betrachten wir 10.7.10:
Die Tiefe von Mischen(n) ist die Tiefe von Mischen(n/2) + 1,
woraus sofort die Tiefe log(n) für den Teil Mischen(n) folgt.

Nach Folie 118 gilt dann:

$$
\begin{aligned}
T(n) &= T(n/2) + \log(n) \\
&= T(n/4) + \log(n/2) + \log(n) = T(n/4) + \log(n)-1 + \log(n) \\
&= T(n/8) + \log(n/4) + \log(n/2) + \log(n) \\
&= \ldots \\
&= T(2) + \log(n/2^{\log(n)-2}) + \log(n/2^{\log(n)-3}) + \ldots + \log(n)-1 + \log(n) \\
&= 1 + 2 + 3 + \ldots + \log(n) \\
&= (1/2)\cdot\log(n)\cdot(\log(n) + 1).
\end{aligned}
$$

Ergebnis: **$T(n) = (1/2)\cdot\log(n)\cdot(\log(n) + 1) \in O(\log^2(n))$.**

10.7.13 Resultat:

$G(n) = (n/4)\cdot\log(n)\cdot(\log(n)-1)) + n - 1$

$T(n) = (1/2)\cdot\log(n)\cdot(\log(n) + 1)$

Faustformel:

$G(n) \approx n/2 \cdot T(n)$.

n Eingänge	G(n) Größe	T(n) Tiefe
2	1	1
4	5	3
8	19	6
16	63	10
32	191	15
64	543	21
128	1471	28
256	3839	36
512	9727	45
1024	24063	55

n Eingänge	G(n) Größe	T(n) Tiefe
16	63	10
128	1471	28
1024	24063	55
16384	1490957	105

Mit 24.063 Bausteinen **sort** kann man also in 55 Schritten 1024 Zahlen sortieren.

Mit etwa 100 Millionen Bausteinen **sort** kann man in nur 210 Schritten rund 1 Million Zahlen sortieren.

Solche Algorithmen sind technisch durchaus realisierbar. Sie können in Zukunft die Leistungsfähigkeit beim Sortieren deutlich steigern.

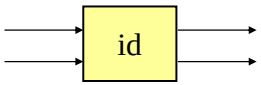
10.7.14 Durchdenken Sie eine mögliche Realisierung weiter, z.B.:

- Gibt es Probleme bei der Hardware-Realisierung? Lassen sich die vielen Leitungen problemlos verschalten / auf Platten drucken? ... Jede Leitung in unserer Skizze besitzt eine "Breite", z.B. 64 Bits zuzüglich Kontrollbits.
- Wie kann man den Sortierbaustein **sort** realisieren? Nehmen Sie an, dass die zu sortierenden Schlüssel k-stellige 0-1-Folgen sind, wobei man wegen der vielfältig möglichen Schlüssel von k=64 ausgehen sollte. Was ist dann die minimale Tiefe (=längster Weg über Gatter von einem Eingang zu einem Ausgang) von **sort**?
- Wie kann man Millionen von Daten *gleichzeitig* an alle Eingänge legen? Welche Strukturen müssen die Speicher hierfür besitzen?

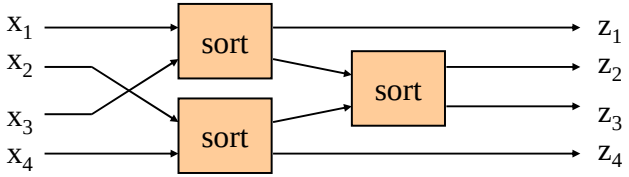
10.7.15 Wie sehen softwaremäßige Realisierungen aus?

Bei n Eingabewerten kann man n/2 Prozeduren definieren, die jeweils "den nächsten Gesamtschritt" (= alle parallel durchführbaren Sortierschritte) ausführen. Hierfür greifen sie immer wieder auf das Feld der zu sortierenden Daten x: array(1..n) of <Elementtyp> zu, aus dem in jedem Schritt gelesen und in das in jedem Schritt geschrieben wird.

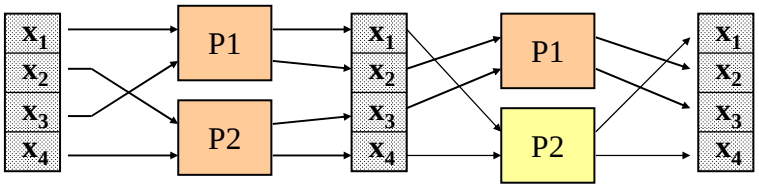
Im Algorithmus gibt es Teile, in denen ein Wert nur weitergereicht wird. Hierfür führen wir den zusätzlichen Baustein "id" (Identität) ein, der zwei Werte einliest und unverändert wieder ausgibt:

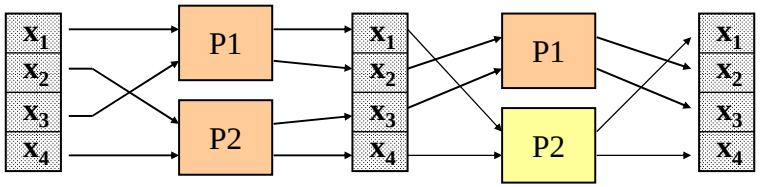


Der Mischbaustein Mischen (4) (siehe 10.7.4)



wird dann realisiert durch zwei Prozeduren P1 und P2:

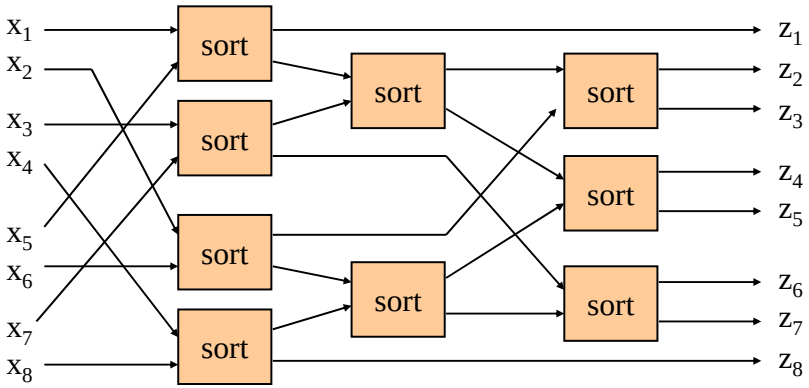




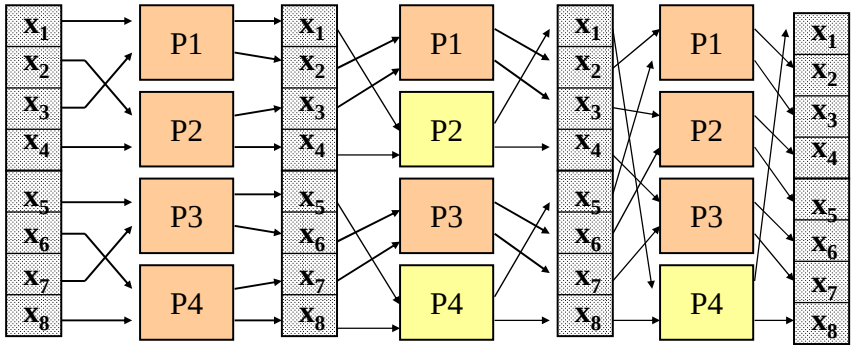

```
procedure P1 is                    -- x: array(1..n) of <Elementtyp> ist global
h1, h2: <Elementtyp>;
begin (h1, h2) := sort (x(1), x(3));  x(1) := h1; x(2) := h2; $
      (h1, h2) := sort (x(2), x(3));  x(2) := h1; x(3) := h2;
end;
procedure P2 is                    -- x: array(1..n) of <Elementtyp> ist global
h1, h2: <Elementtyp>;
begin (h1, h2) := sort (x(2), x(4));  x(3) := h1; x(4) := h2; $
      (h1, h2) := id (x(1), x(4));    x(1) := h1; x(4) := h2;
end;
```

$ steht für "Synchronisation"

Der Mischbaustein Mischen(8)



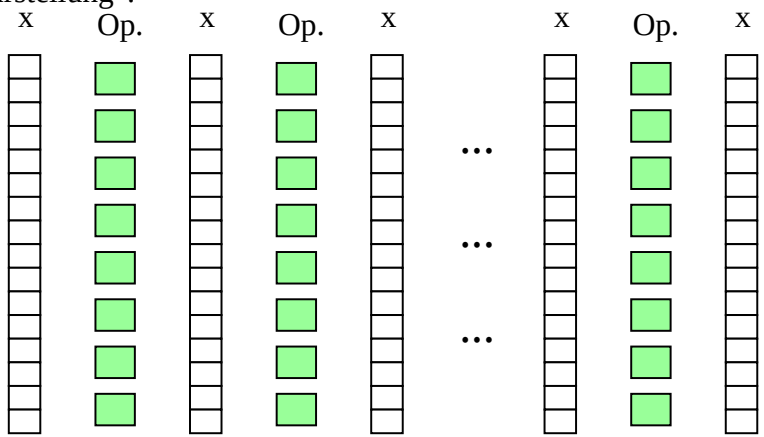
wird dann realisiert durch vier Prozeduren, die ihre Arbeitsweise synchronisieren müssen:




```
procedure Mischen8 is
    procedure P1 is begin ... end;
    procedure P2 is begin ... end;
    procedure P3 is begin ... end;
    procedure P4 is begin ... end;
begin  P1; P2; P3; P4; end;
```

Die Synchronisation ist innerhalb der Prozeduren P1 bis P4 sicherzustellen.

10.7.16: Man erhält auf diese Weise eine "normierte Darstellung":



x = Datenschicht, Op. = Operationsschicht. Jeder Operationsbaustein **sort** oder **id** hat zwei einlaufende und zwei ausgehende Verbindungen.

Jede Operationsschicht lässt sich durch n/2 Prozeduren softwaremäßig beschreiben; diese Prozeduren können wir ebenfalls für die nächste Operationsschicht verwenden usw., so dass wir aus Software-Sicht folgendes Ergebnis erhalten (die Operationen **id** kann man natürlich im Programm weglassen, nicht aber die Synchronisierung):

Mit n/2 Prozeduren lassen sich n Elemente in
$T(n) = (1/2) \cdot \log(n) \cdot (\log(n) + 1)$
Schritten sortieren, wobei jede der n/2 Prozeduren aus einem sequentiellen Programmstück mit bis zu T(n) einzelnen Sortieroperationen **sort** besteht und T(n)-1 Synchronisationsanweisungen besitzt.

Diese Prozeduren können automatisch durch ein Programm erzeugt werden; sie hängen nur vom Parameter n ab.

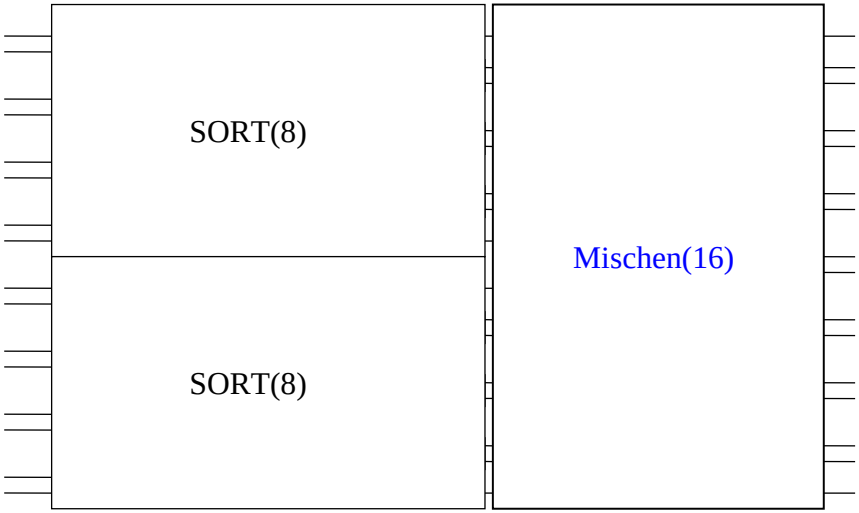
Wie müssen Programmiersprachen beschaffen sein, damit der synchrone Ablauf und die Verzögerungen gesichert werden?

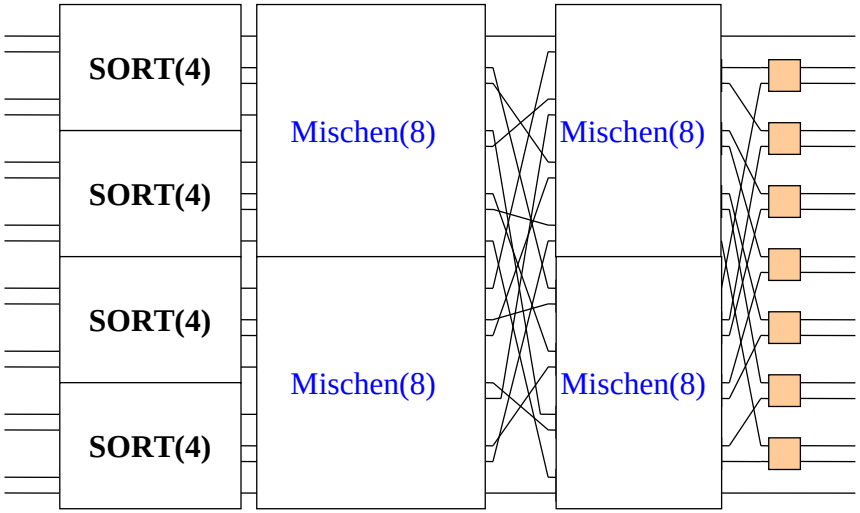
Diese Sprachen müssen in der Regel mehrere Anforderungen erfüllen:

- Teile eines Programms müssen unabhängig voneinander ablaufen können (Nebenläufigkeit, "tasks").
- Es muss Synchronisationsmechanismen geben.
- Man muss Zeitvorgaben (innerhalb von ... Mikrosekunden führe durch ....) machen können und es muss Verzögerungselemente (delay) geben.
- Es muss der parallele Zugriff auf Daten möglich sein.
- Es muss ...

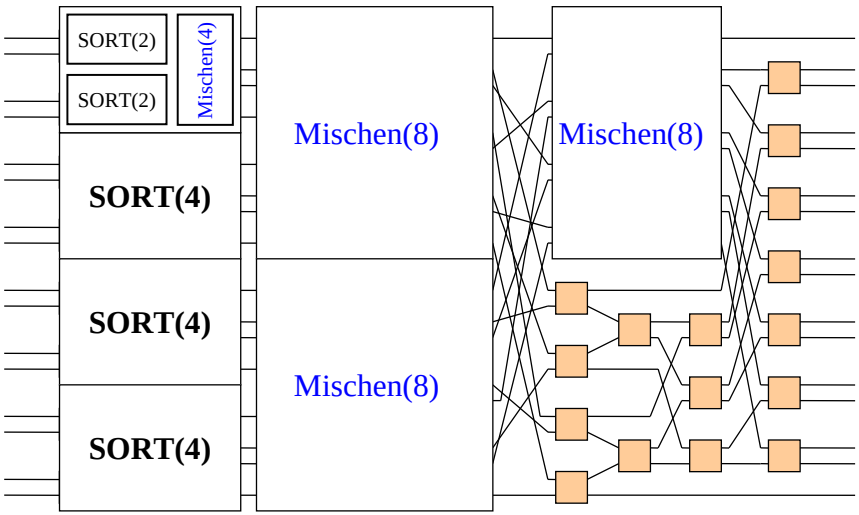
Siehe weiteres Studium, Praktika, Projekte .....

10.7.17 Zum Abschluss das vollständige Beispiel SORT(16)

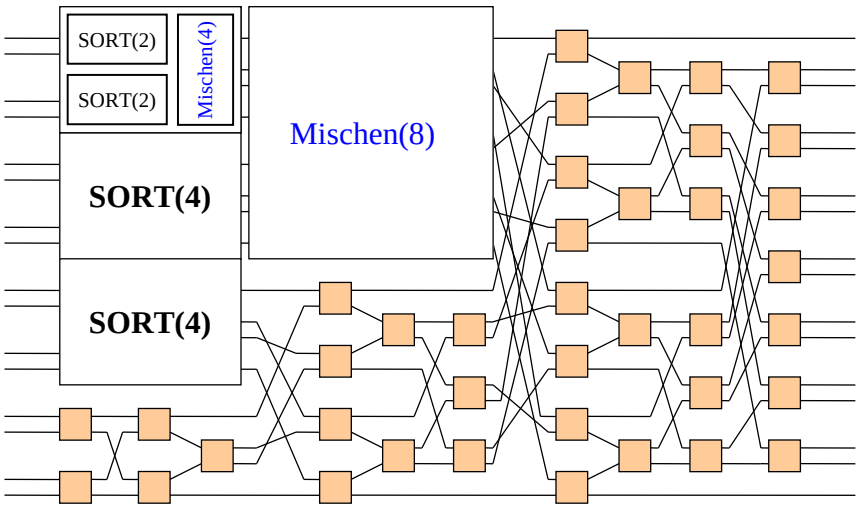




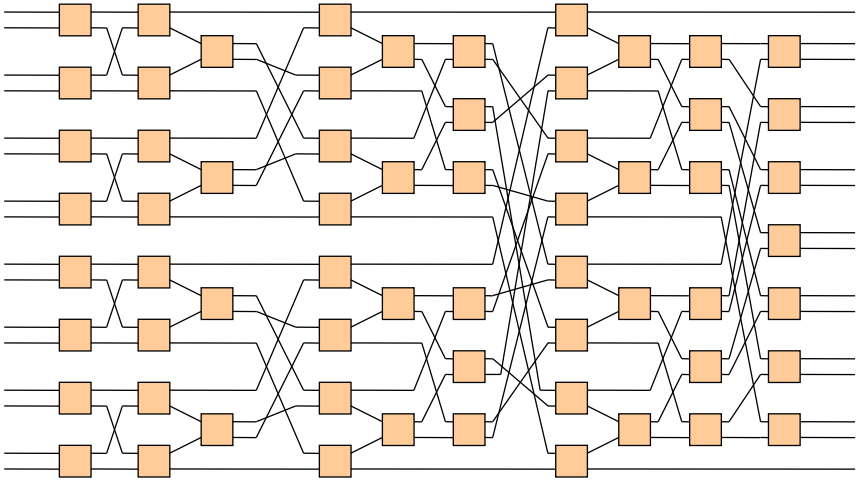
SORT(16)



SORT(16)



SORT(16)



63 Bausteine **sort**, größte Tiefe: 10, Breite: 8.