Kapitel 13

Sortierverfahren

Sortierung ist eine wichtige Grundaufgabe der Programmierung.

Konkret werden wir die folgenden Sortierverfahren betrachten:

- Bubblesort
- Selectionsort
- Insertionsort
- Shellsort
- Quicksort
- Heapsort

Die verschiedenen Verfahren werden wir als Funktionen implementieren und mit einer einheitlichen Schnittstelle ausstatten, an der wir die Anzahl der Daten (`int n`) und den Array mit den Daten (`int *daten`) übergeben.

```
void XXXsort( int n, int *daten)
```

Damit sind wir in der Lage, einen einheitlichen Testrahmen für alle Sortierprogramme dieses Abschnitts zu erstellen.

Auch wenn wir hier nur Integer-Werte sortieren, sind die vorgestellten Algorithmen universell. Mit geringfügigen Änderungen können auch andere Datenwerte sortiert werden.

Bubblesort - Verfahrensbeschreibung

Durchlaufe die Daten in aufsteigender Richtung! Betrachte dabei immer zwei benachbarte Elemente. Wenn zwei benachbarte Elemente in falscher Ordnung sind, dann vertausche sie! Nach einem Durchlauf ist auf jeden Fall das größte Element am Ende der Daten.

Wiederhole den obigen Verfahrensschritt so lange, bis die Daten vollständig sortiert sind! Dabei muss jeweils das letzte Element des vorherigen Durchlaufs nicht mehr betrachtet werden, da es schon seine endgültige Position gefunden hat!

Bubblesort - Verfahrensablauf

	Array						i	k
	0	1	2	3	4	5		
Vor dem 1. Durchlauf	3	5	2	6	4	1	5	
								0
								1
								2
								3
								4
Nach dem 1. Durchlauf	3	2	5	4	1	6	4	
								0
								1
								2
								3
Nach dem 2. Durchlauf	2	3	4	1	5	6	3	
								0
								1
								2
Nach dem 3. Durchlauf	2	3	1	4	5	6	2	
								0
								1
Nach dem 4. Durchlauf	2	1	3	4	5	6	1	
								0
Nach dem 5. Durchlauf	1	2	3	4	5	6	0	

Bubblesort - Implementierung

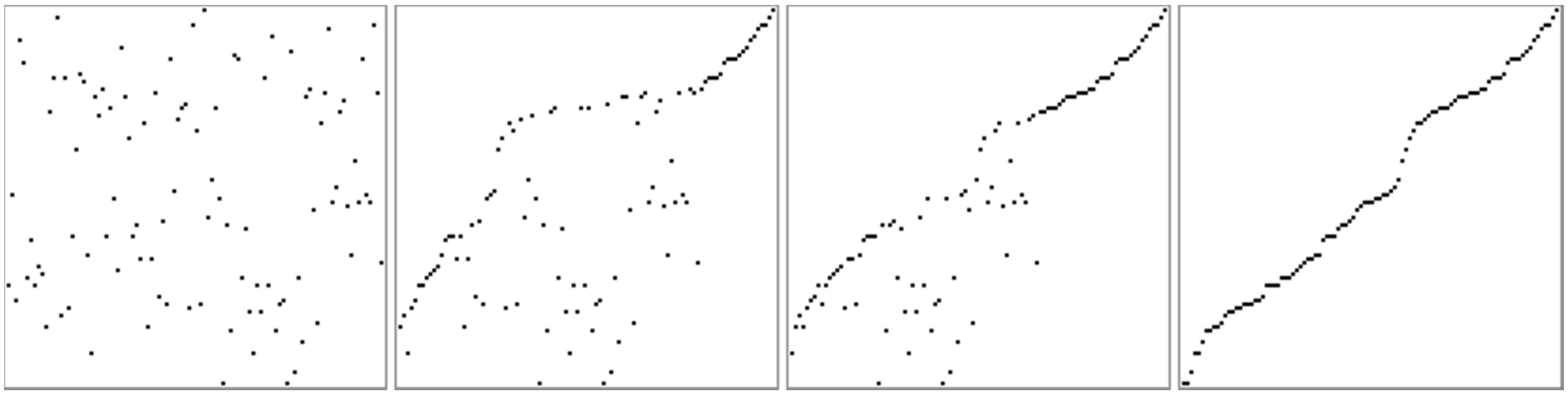
```
void bubblesort( int n, int *daten)
    {
    int i, k, t;

    for( i = n-1; i > 0; i--)
        {
        for( k = 0; k < i; k++)
            {
            if( daten[k] > daten[k+1])
                {
                t = daten[k];
                daten[k] = daten[k+1];
                daten[k+1] = t;
                }
            }
        }
    }
```

Am Anfang sind alle Elemente zu betrachten, dann immer eins weniger.

Durchlaufe den noch zu betrachtenden Bereich.

Vergleiche zwei benachbarte Elemente. Wenn sie in der falschen Reihenfolge sind, dann tausche sie.

Schnappschüsse



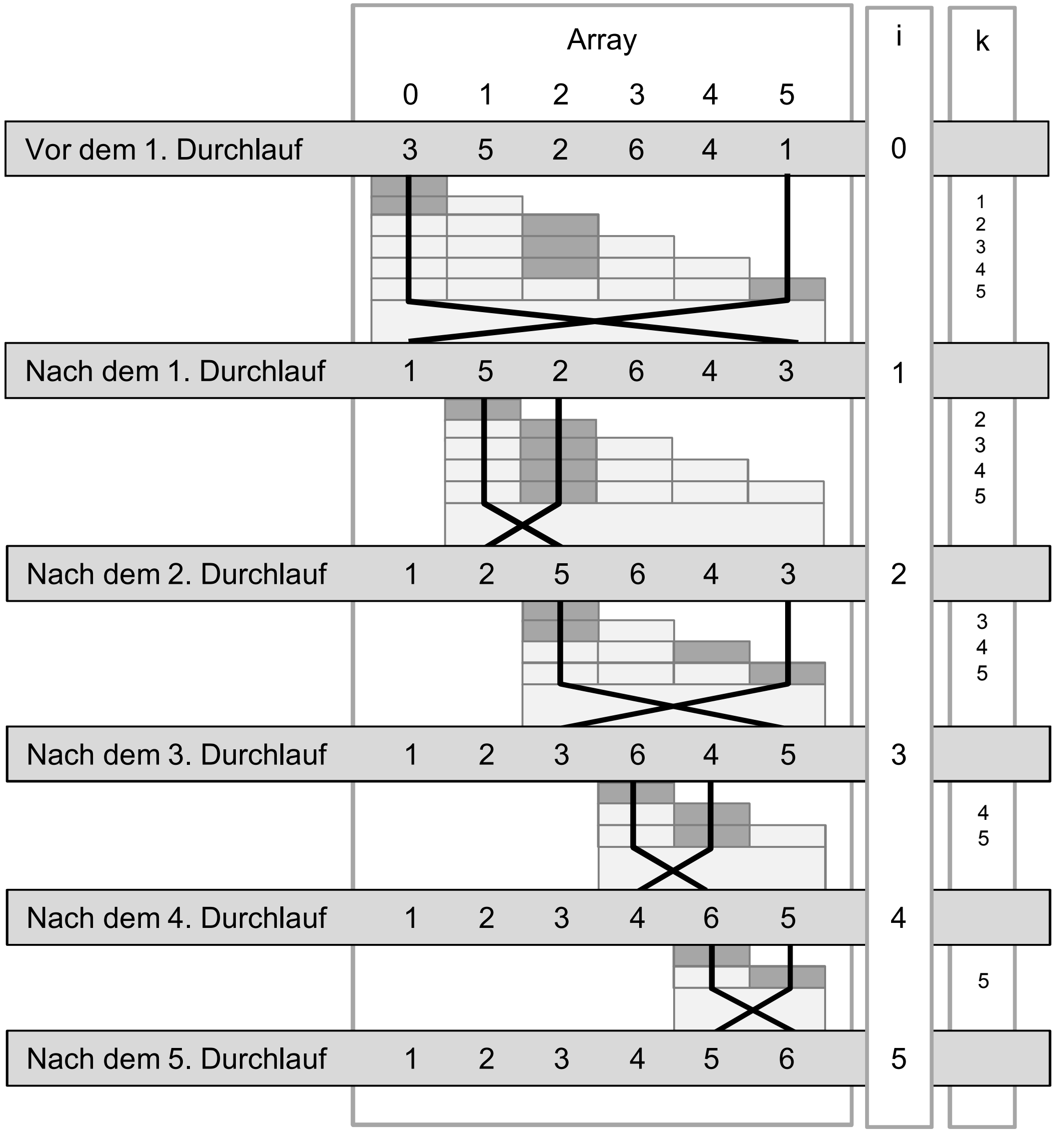
Selectionsort - Verfahrensbeschreibung

Durchlaufe den Array in aufsteigender Richtung und suche das kleinste Element! Vertausche das kleinste Element mit dem ersten Element! Das neue erste Element ist jetzt an der korrekten Position und muss im Weiteren nicht mehr betrachtet werden.

Durchlaufe den Array jetzt ab dem zweiten Element aufwärts und suche wieder das kleinste Element! Vertausche das gefundene Element mit dem zweiten Element! Jetzt sind die beiden ersten Elemente im Array in der richtigen Reihenfolge und müssen im Weiteren nicht mehr betrachtet werden.

Setze dies Verfahren fort, bis der gesamte Array sortiert ist!

Selectionsort - Verfahrensablauf



Selectionsort - Implementierung

```
void selectionsort( int n, int *daten)
    {
    int i, k, t, min;

    for( i = 0; i < n-1; i++)
        {
        min = i;
        for( k = i+1; k < n; k++)
            {
            if( daten[k] < daten[min])
                min = k;
            }
        t = daten[min];
        daten[min] = daten[i];
        daten[i] = t;
        }
    }
```

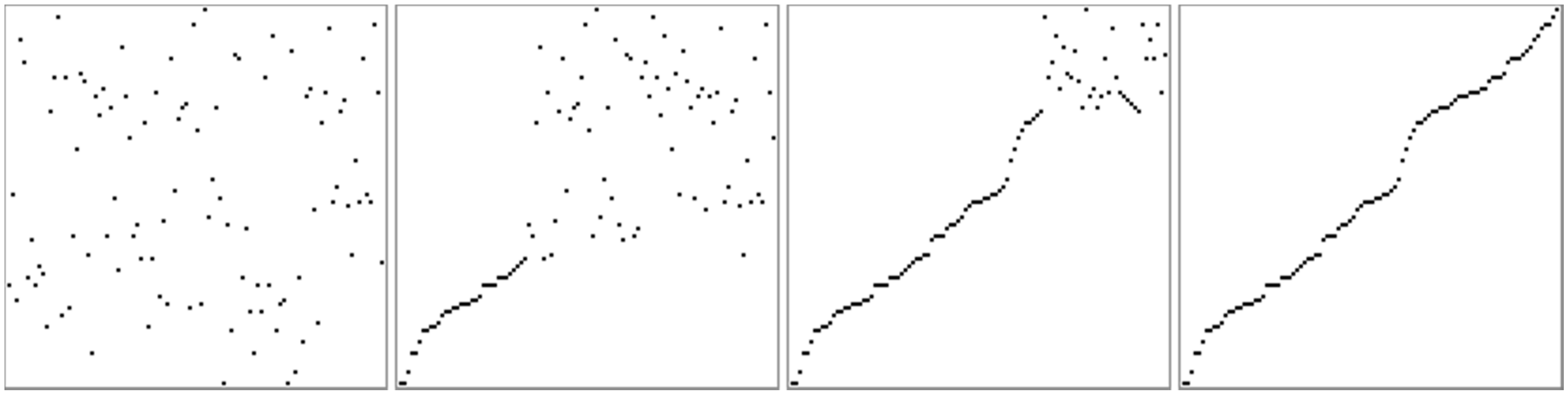
Es werden n-1 Verfahrensschritte durchgeführt.

Zunächst ist das erste zu betrachtende Element das kleinste.

Dann wird im Rest des Arrays ein kleineres gesucht.

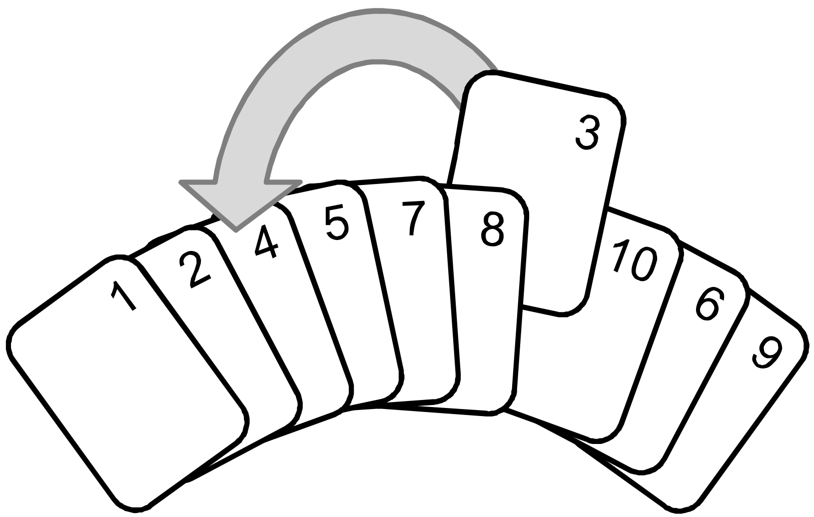
Das kleinste Element wird mit dem zuerst betrachteten getauscht.

Schnappschüsse



Insertionsort - Verfahrensbeschreibung

Insertionsort ist ein Sortierverfahren, das so arbeitet, wie wir Spielkarten auf der Hand sortieren.



Die erste Karte ganz links ist sortiert. Wir nehmen die zweite Karte und stecken sie, je nach Größe, vor oder hinter die erste Karte. Damit sind die beiden ersten Karten relativ zueinander sortiert.

Wir nehmen die dritte, vierte, fünfte ... Karte und schieben sie so lange nach links, bis wir an die Stelle kommen, an der sie hineinpasst. Dort stecken wir sie hinein.

In einem Array geht das Verschieben von Daten nicht so leicht wie bei einem Kartenspiel auf der Hand. Wir können im Array nicht einfach ein Element "dazwischenschieben". Dazu müssen zunächst alle übersprungenen Elemente nach rechts aufrücken, um für das einzusetzende Element einen Platz frei zu machen.

Insertionsort - Verfahrensablauf

	Array						i	k
	0	1	2	3	4	5		
Vor dem 1. Durchlauf	3	5	2	6	4	1	1	
								1
Nach dem 1. Durchlauf	3	5	2	6	4	1	2	
								2
								1
Nach dem 2. Durchlauf	2	3	5	6	4	1	3	
								3
Nach dem 3. Durchlauf	2	3	5	6	4	1	4	
								4
								3
								2
Nach dem 4. Durchlauf	2	3	4	5	6	1	5	
								5
								4
								3
								2
								1
Nach dem 5. Durchlauf	1	2	3	4	5	6	6	

Insertionsort - Implementierung

```
void insertionsort( int n, int *daten)
    {
    int i, k, v;

    for( i = 1; i < n; i++)
        {
        v = daten[i];

        for( k = i; (k >= 1)&&(daten[k-1] > v); k--)
          daten[k] = daten[k-1];

       daten[k] = v;
       }
   }
```

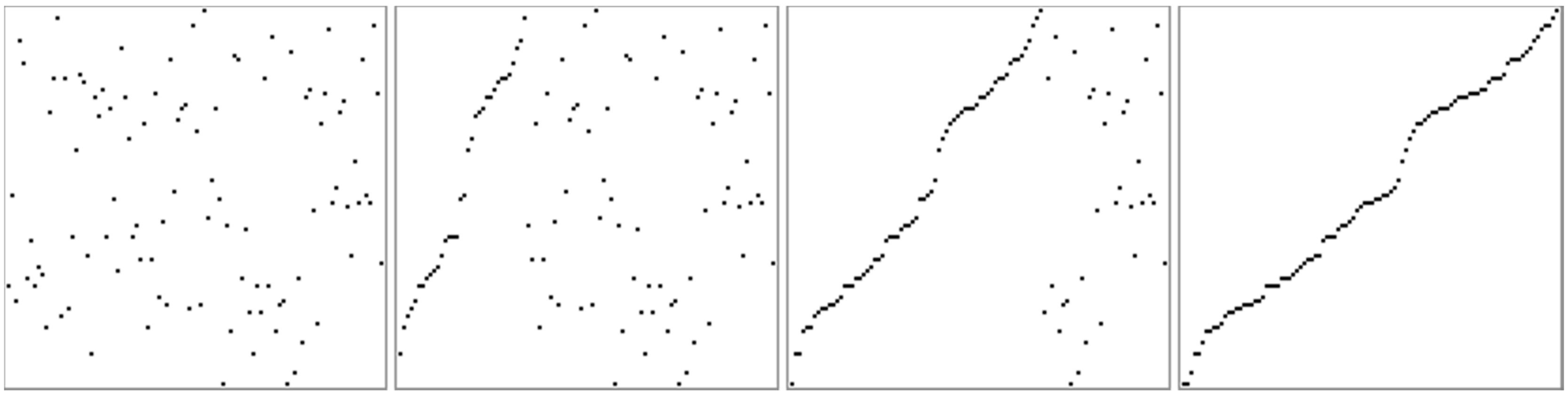
Es werden n-1 Verfahrensschritte durchgeführt.

Das betrachtete Element wird außerhalb des Arrays gesichert.

Größere Elemente rücken im Array auf.

Das gesicherte Element erhält seine korrekte Position.

Schnappschüsse



Modifikation von Insertionsort zu Insertion-h-sort

```
void insertionsort( int n, int *daten)
    {
    int i, k, v;

    for( i = 1; i < n; i++)
        {
        v = daten[i];

        for( k = i; (k >= 1)&&(daten[k-1] > v); k--)
          daten[k] = daten[k-1];

       daten[k] = v;
       }
   }
```

Schrittweite h

```
void insertion_h_sort( int n, int *daten, int h)
    {
    int i, k, v;

    for( i = h; i < n; i++)
        {
        v = daten[i];
        for( k = i; (k >= h) && (daten[k-h] > v); k -= h)
            daten[k] = daten[k-h];
        daten[k] = v;
        }
    }
```

Aus 1 wurde h

Für h = 1 ist das Insertionsort, aber was macht dieses Programm für h > 1?

Was macht Insertion-h-sort?

Beispiel eines Arrays mit 17 Elementen und Schrittweite h = 3

3	12	5	2	14	9	8	11	4	1	10	16	7	6	17	15	13
3			2			8			1			7			15	
	12			14			11			10			6			13
		5			9			4			16			17		

Insertion-3-sort betrachtet immer Elemente mit Abstand 3. Dadurch ergeben sich 3 ineinander verzahnte Teilarrays, die, für sich betrachtet sortiert werden:

1			2			3			7			8			15	
	6			10			11			12			13			14
		4			5			9			16			17		
1	6	4	2	10	5	3	11	9	7	12	16	8	13	17	15	14

Das Ergebnis nennen wir eine h-Sortierung (hier 3-Sortierung). Für h = 1 ist das eine vollständige Sortierung.

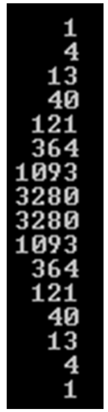
Shellsort - Implementierung

```
shellsort( int n, int *daten)
    {
    int h;

    for( h = 1; h <= n/9; h = 3*h+1)
        ;

    for( ; h > 0; h /= 3)
        insertion_h_sort( n, daten, h);
    }
```

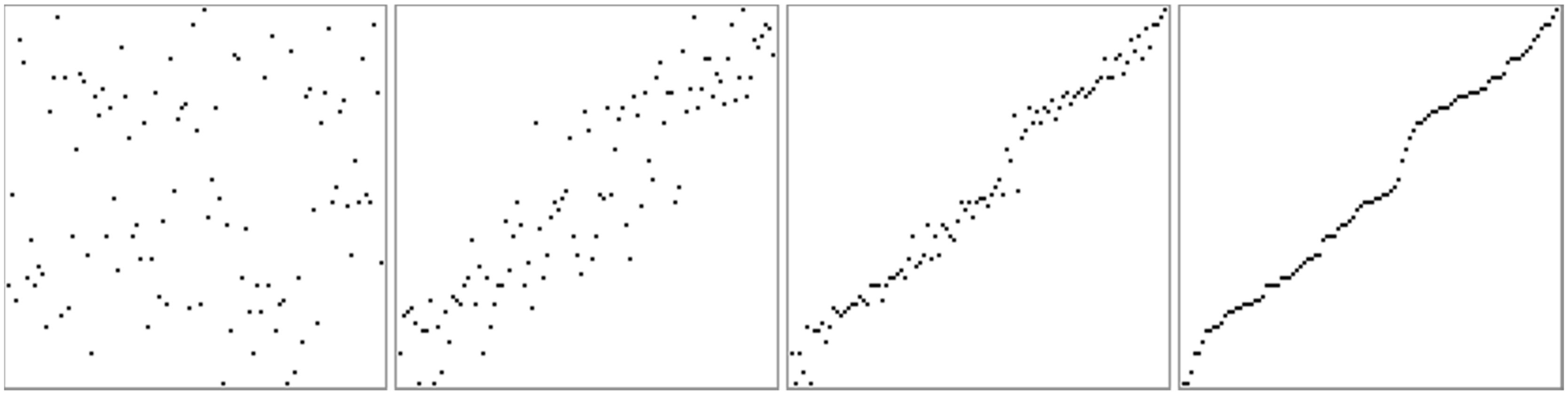
Hier wird eine Folge von h-Werten berechnet: $h = 1, 4, 13, \cdots$

Hier wird die gleiche Folge wieder rückwärts durchlaufen: $h = \cdots, 13, 4, 1$

Für jeden Wert von h wird `insertion_h_sort` gerufen.



Bei der Folge von h-Werten handelt es sich um die Folge $h_n = 1 + 3 + 9 + \cdots + 3^n = \frac{3^{n+1}-1}{2}$.
Warum gerade diese Folge gewählt wird, ist schwer zu begründen. Diese Folge ist jedenfalls keine schlechte Wahl, aber man könnte auch eine andere Folge wählen.
Die Folge wird vorwärts durchlaufen, um einen geeigneten Startwert zu ermitteln, von dem aus die Folge dann wieder rückwärts durchlaufen wird, um für jedes Element der Folge Insertion-h-sort auszuführen. Shellsort sortiert den Array, weil h = 1 als letztes Glied der absteigenden Folge vorkommt und damit Insertionsort ausgeführt wird.



Shellsort - Optimierung

Um die Laufzeitkosten für den wiederholten Aufruf von Insertion-h-sort einzusparen, wird die Funktion an der Stelle des Funktionsaufrufes implementiert:

```
shellsort( int n, int *daten)
    {
    int h;

    for( h = 1; h <= n/9; h = 3*h+1)
        ;

    for( ; h > 0; h /= 3)
        insertion_h_sort( n
    }
```

```
void shellsort( int n, int *daten)
    {
    int i, k, h, v;

    for( h = 1; h <= n/9; h = 3*h+1)
        ;
    for( ; h > 0; h /= 3)
        {
        for( i = h; i < n; i++)
            {
            v = daten[i];
            for( k = i; (k >= h) && (daten[k-h] > v); k -= h)
                daten[k] = daten[k-h];
            daten[k] = v;
            }
        }
    }
```

Dies ist `insertion_h_sort`.

Quicksort - Verfahrensbeschreibung

Quicksort ist ein Sortierverfahren konstruieren, das auf dem Prinzip "Teile und herrsche" beruht und rekursiv arbeitet:

Zerlege den Array in zwei Teile, wobei alle Elemente des ersten Teils kleiner oder gleich allen Elementen des zweiten Teils sind. Die beiden Teile können jetzt unabhängig voneinander betrachtet werden, da beim Sortieren keine Elemente mehr von dem einen Teil in den andern bewegt werden müssen.

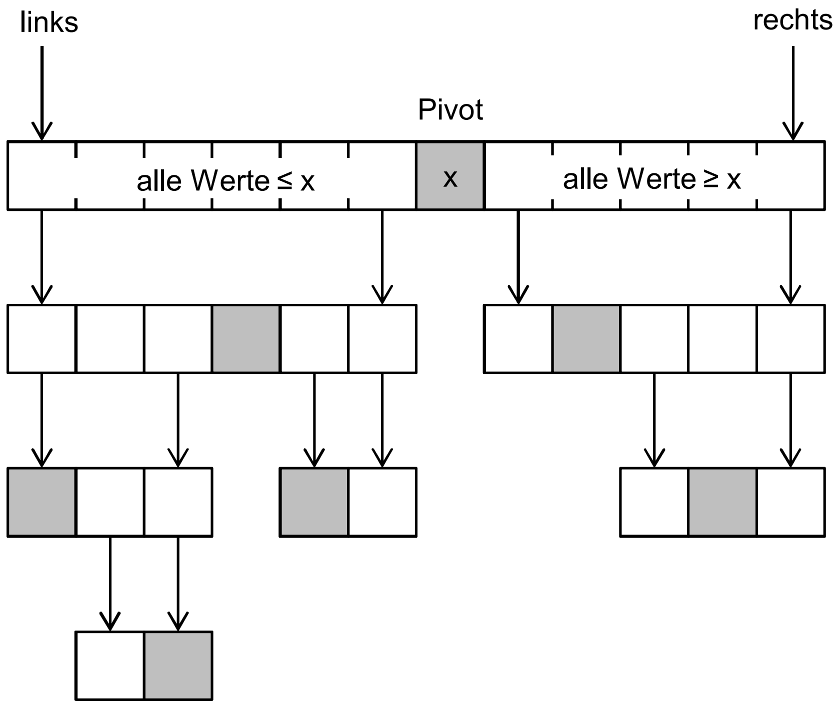
Zerlege jedes der beiden Teile aus dem vorherigen Schritt in gleicher Weise wieder in zwei Teile.

Setze den Prozess des Zerlegens fort, bis die Zerlegungsprodukte nur noch ein Element haben und damit sortiert sind.

Besonders effizient ist dieses Verfahren, wenn es gelingt die beiden Teile, in die wir den Array zerlegen, immer in etwa gleich groß zu halten.

Quicksort - Verfahrensablauf

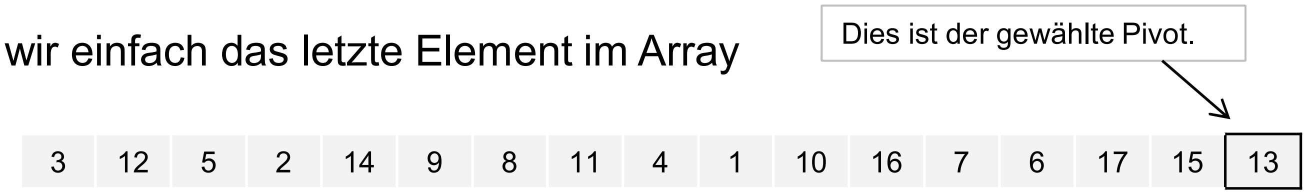
Zur Aufteilung wird ein sog. Pivot bestimmt, und der Array so umgeordnet, dass alle Werte links vom Pivot kleiner und rechts vom Pivot größer sind. Der Pivot selbst ist dann bereits an der richtigen Stelle und muss im weiteren nicht mehr betrachtet werden:



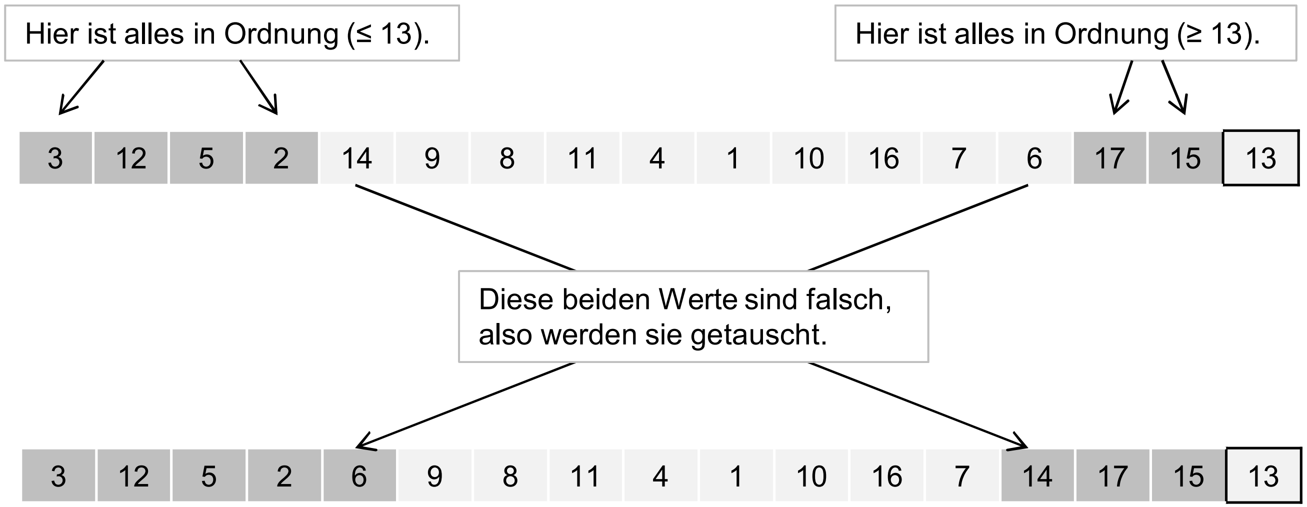
Offen bleibt allerdings noch die Frage, wie der Pivot zu wählen und wie die Umordnung durchzuführen ist.

Quicksort - Aufteilung des Arrays

Als Pivot wählen wir einfach das letzte Element im Array



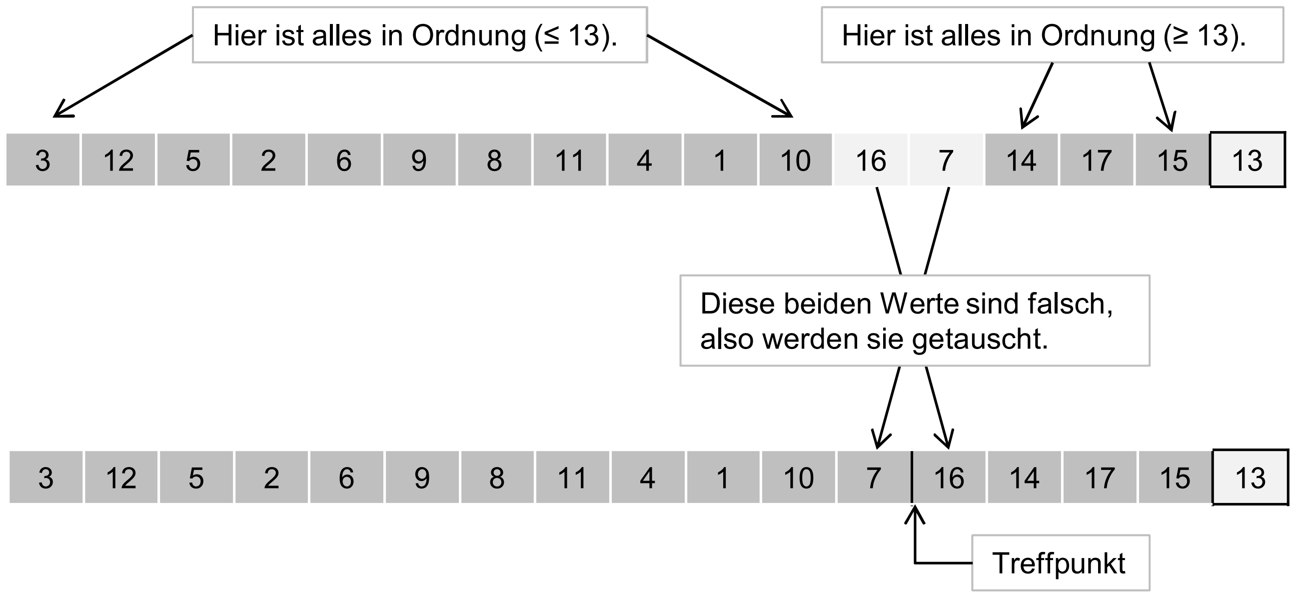
Jetzt gehen wir von den Rändern des Arrays zur Mitte, solange alle Elemente bezogen auf den Pivot korrekt positioniert sind:



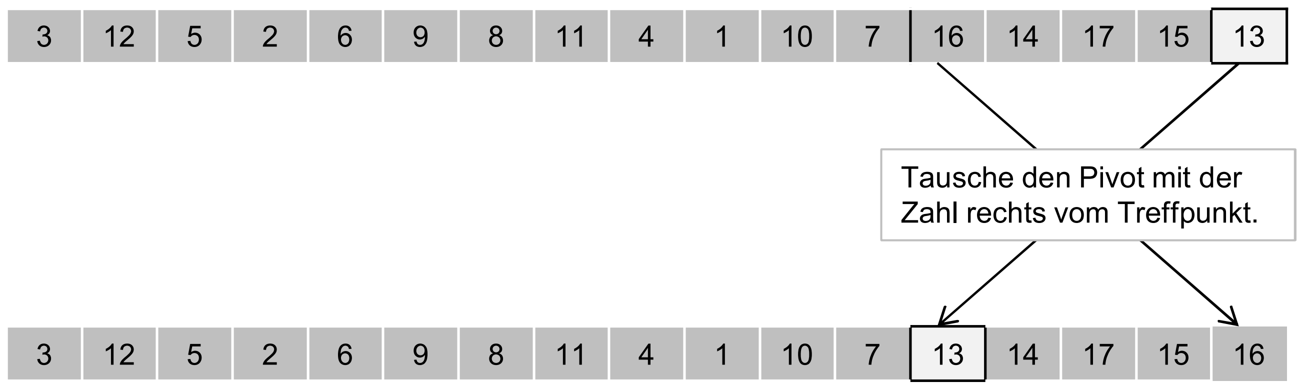
Sobald wir nicht mehr weiterkommen, werden die blockierenden Elemente getauscht.

Dieses Vorgehen setzen wir fort, bis wir in der Mitte ankommen.

Quicksort - Aufteilung des Arrays (Fortsetzung)



Zum Abschluss tauschen wir das Element rechts vom Treffpunkt mit dem Pivot:



Die gewünschte Aufteilung ist hergestellt.

Quicksort - Implementierung der Aufteilung

```
int aufteilung( int links, int rechts, int *daten)
    {
    int pivot, i, j, t;

    pivot = daten[rechts];
    i = links-1;
    j = rechts;

    for(;;)
        {
        while( daten[++i] < pivot)
            ;
        while( (j > i) && (daten[--j] > pivot))
            ;
        if( i >= j)
            break;
        t = daten[i];
        daten[i] = daten[j];
        daten[j] = t;
        }

    daten[rechts] = daten[i];
    daten[i] = pivot;
    return i;
    }
```

Der Wert ganz rechts ist der Pivot.

Starte mit zwei Fingern links und rechts vom aufzuteilenden Bereich.

Solange links alles in Ordnung ist, gehe nach rechts.

Solange rechts alles in Ordnung ist, gehe nach links.

Die Finger haben sich getroffen. Hier geht's weiter.

Tausche die blockierenden Elemente.

Tausche den Pivot mit dem Element rechts vom Treffpunkt.

Gib den Index des Pivot zurückt.

Quicksort - Implementierung

Die Daten in diesem Bereich werden sortiert.

```
void qcksort( int links, int rechts, int *daten)
    {
    int i;



    if( links < rechts)
        {
        i = aufteilung( links, rechts, daten);
        qcksort( links, i-1, daten);
        qcksort( i+1, rechts, daten);
        }
    }
```

Wenn `links` < `rechts` ist, dann muss weiter aufgeteilt werden

Bestimme den Index des Pivots und teile den Array dadurch in zwei Teile.

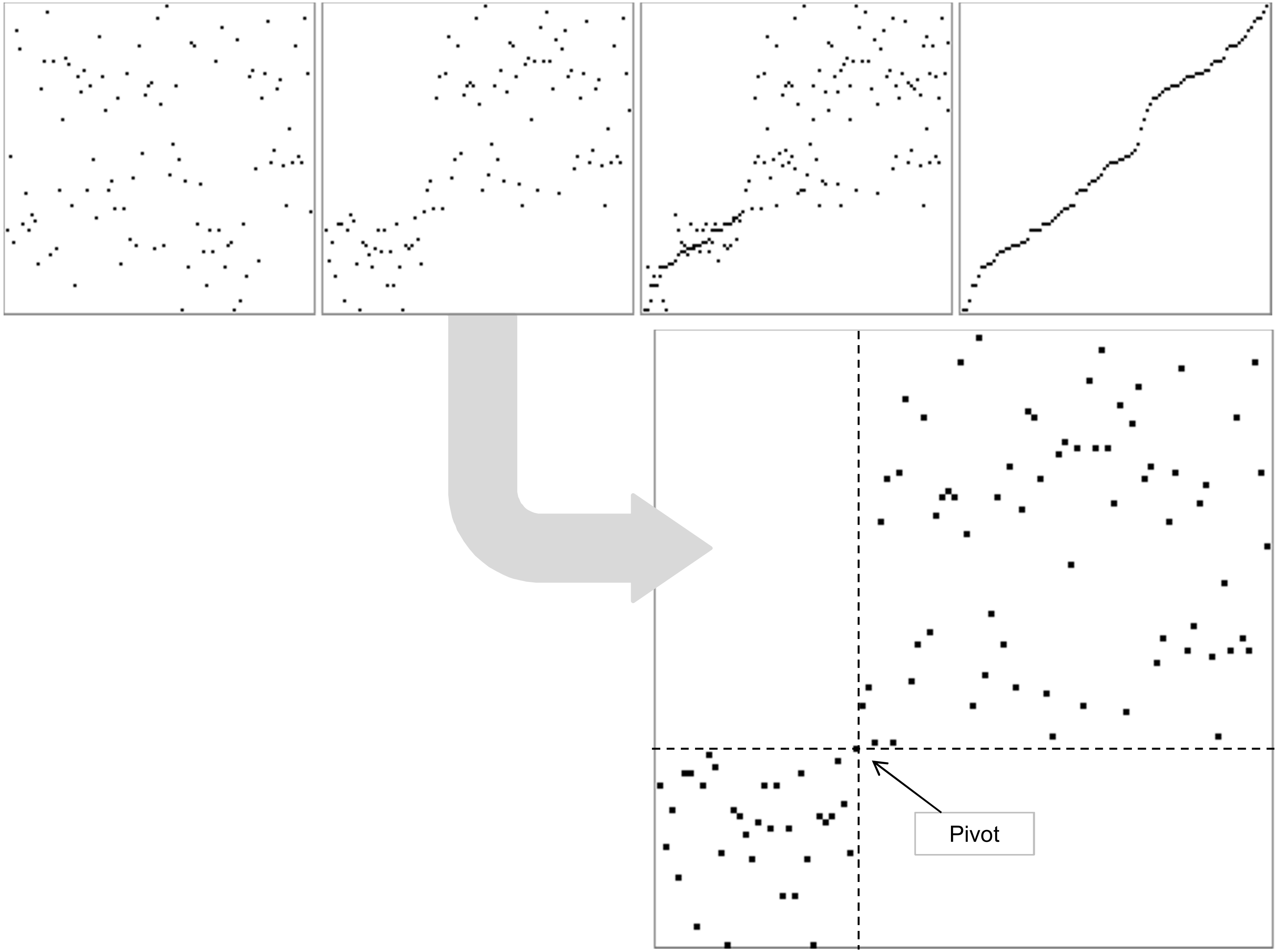
Sortiere links vom Pivot

Sortiere rechts vom Pivot

Wegen der Rekursion konnte Quicksort nicht mit der gleichen Schnittstelle wie die anderen Sortierverfahren erstellt werden. Die einheitliche Schnittstelle wird durch eine Funktionsschale hergestellt:

```
void quicksort( int n, int *daten)
    {
    qcksort( 0, n-1, daten);
    }
```

Quicksort - Schnappschüsse



Quicksort - Optimierung durch direkte Implementierung der Aufteilung

Um die Laufzeitkosten für den wiederholten Aufruf von `aufteilung` einzusparen, wird die Funktion an der Stelle des Funktionsaufrufes implementiert:

```
void qcksort( int links, int rechts, int *daten)
    {
    int i;



    if( links < rechts)
        {
        i = aufteilung( links, rechts, daten);
        qcksort( links, i-1, daten);
        qcksort( i+1, rechts, daten);
        }
    }
```

```
void qcksort( int links, int rechts, int *daten)
    {
    int pivot, i, j, t;

    if( rechts > links)
        {
        pivot = daten[rechts];
        i = links-1;
        j = rechts;
        for( ; ;)
            {
            while( daten[++i] < pivot)
                ;
            while((j > i) && (daten[--j] > pivot))
                ;
            if( i >= j)
                break;
            t = daten[i];
            daten[i] = daten[j];
            daten[j] = t;
            }
        daten[rechts] = daten[i];
        daten[i] = pivot;
        qcksort( links, i-1, daten);
        qcksort( i+1, rechts, daten);
        }
    }
```

Dies ist `aufteilung`.

Quicksort - Optimierung durch Vermeidung der Rekursion

Rekursion arbeitet so, dass die lokalen Variablen einer Funktion auf den Stack gelegt werden und somit für jede Aufrufinstanz der Funktion separat zur Verfügung stehen. Ist das Unterprogramm beendet, werden die Variablen des rufenden Programms wiederhergestellt und es kann weiterarbeiten, als wäre nichts geschehen.

Wenn wir einen Stack nachbilden und dort die Werte für links und rechts zwischenspeichern, können wir die Rekursion in Quicksort vermeiden.

Ein Stack ist ein Stapel auf oben etwas gelegt und von oben wieder etwas entnommen werden kann.. Was zuletzt auf den Stapel gelegt wurde, kommt als erstes wieder herunter. Man spricht deswegen auch von einem Last-In-First-Out- oder kurz LIFO-Speicher.

Zur Implementierung eines Stacks benötigen wir einen Array und einen Zeiger (Stackpointer) auf das oberste Element des Stapels (Stacktop).

Ein Stack für 100 Integer-Zahlen.

Der Stackpointer zeigt immer auf die nächste freie Position.

```
int stack[100];
int pos = 0;
int i;

printf( "Push: ");
for( i = 0; i < 8; i++)
     {
     printf( "%d " , i);
     stack[pos++] = i;
     }
printf( "\nPop:  ");
while( pos)
     {
     i = stack[--pos];
     printf( "%d ", i);
     }
```

Eine Zahl wird auf den Stack gelegt und der Stackpointer wird inkrementiert. Diese Stackoperation heißt **push** (Teller auf Stapel legen).

Test, ob noch etwas auf dem Stack liegt.

Der Stackpointer wird Dekrementiert und eine Zahl wird vom Stack entfernt. Diese Stackoperation heißt **pop** (Teller vom Stapel nehmen).

```
Push: 0 1 2 3 4 5 6 7
Pop:  7 6 5 4 3 2 1 0
```

Quicksort - Rekursionsfreie Implementierung

Rekursive Version.

```
void qcksort( int links, int rechts, int *daten)
    {
    int i;

    if( links < rechts)
        {
        i = aufteilung( links, rechts, daten);
        qcksort( links, i-1, daten);
        qcksort( i+1, rechts, daten);
        }
    }
```

Nicht rekursive Version.

```
void qcksortiter(int links, int rechts, int *daten)
    {
    int i;
    int stack[256];
    int pos = 0;

    stack[pos++] = links;
    stack[pos++] = rechts;


    while( pos)
        {
        rechts = stack[--pos];
        links = stack[--pos];
        if( links < rechts)
            {
            i = aufteilung( links, rechts, daten);
            stack[pos++] = links;
            stack[pos++] = i-1;
            stack[pos++] = i+1;
            stack[pos++] = rechts;
            }
        }
    }
```

Stack und Stackpointer.

Der erste Auftrag.

Solange noch Aufträge in der Warteschlange sind.

Lies den nächsten Auftrag vom Stack.

Bearbeite den Auftrag.

Erzeuge zwei neue Aufträge auf dem Stack.

Zusätzlich sollte, wie oben gezeigt, die Funktion `aufteilung` eliminiert werden.

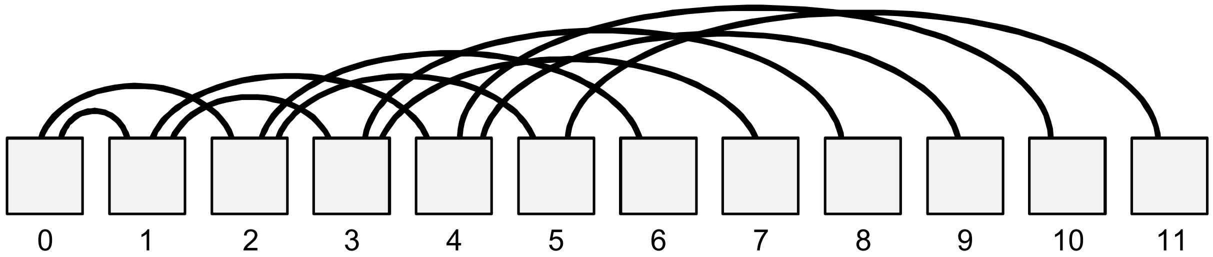
Quicksort - Rekursionsfreie Implementierung und Eliminierung der Funktion `aufteilung`

```
void quicksort(int n, int *daten)
    {
    int pivot, i, j, t;
    int links, rechts;
    int stack[256];
    int pos = 0;

    stack[pos++] = 0;
    stack[pos++] = n-1;

    while( pos)
        {
        rechts = stack[--pos];
        links = stack[--pos];

        if( links < rechts)
            {
            i = aufteilung( links, rechts, daten);
            stack[pos++] = links;
            stack[pos++] = i-1;
            stack[pos++] = i+1;
            stack[pos++] = rechts;
            }
        }
    }
```

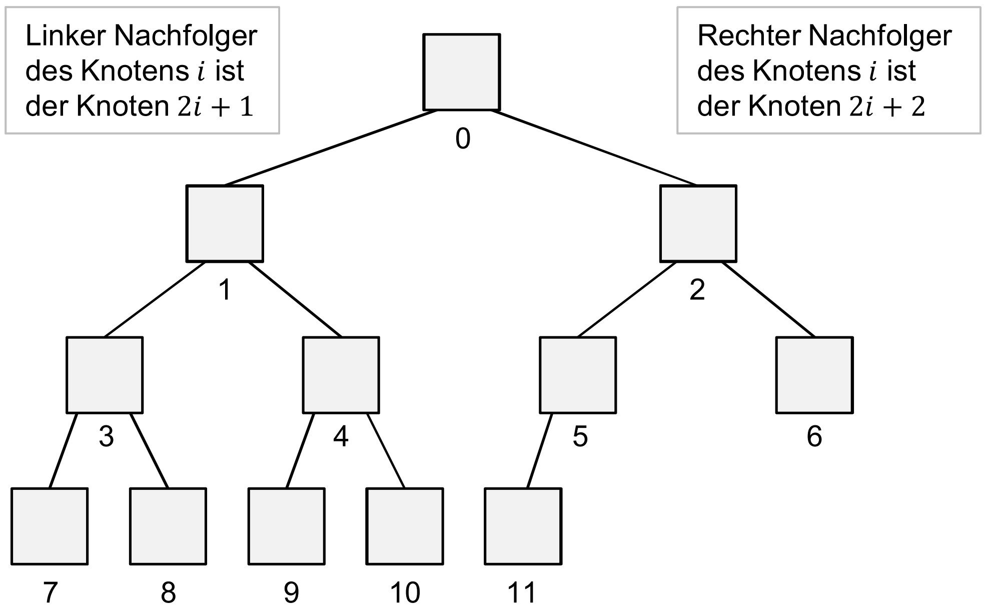
```
i = links-1;
j = rechts;
pivot = daten[rechts];
for(;;)
    {
    while( daten[++i] < pivot)
        ;
    while((j > i) && (daten[--j] > pivot))
        ;
    if( i >= j)
        break;
    t = daten[i];
    daten[i] = daten[j];
    daten[j] = t;
    }
daten[ rechts] = daten[i];
daten[ i] = pivot;
```

Arrays mit Baumstruktur

Wir können uns die Elemente eines Arrays wie in einem Baum* angeordnet denken.



Die Verweise von Knoten auf Folgeknoten bzw. Blätter sind nicht explizit, sondern nur gedanklich vorhanden. Auseinandergeschüttelt sieht das so aus:



In den Knoten des Baums, also im Array, können beliebige Zahlenwerte stehen.

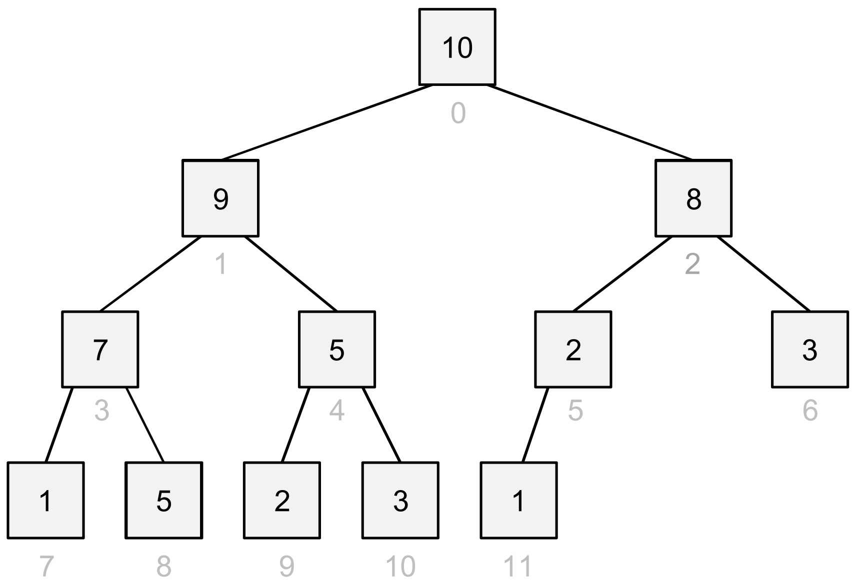
*Die Begriffe "Baum", "Knoten" und "Blätter" werden hier intuitiv verwendet. Später werden wir uns intensiver mit Baumstrukturen beschäftigen.

Heaps

Wir sagen, dass ein Baum die sogenannte **Heapbedingung** erfüllt, wenn für jeden Knoten des Baumes gilt, dass der Wert des Knotens größer oder gleich den Werten seiner Nachfolgerknoten ist

Einen Baum, der die Heapbedingung erfüllt, nennen wir einen **Heap**.

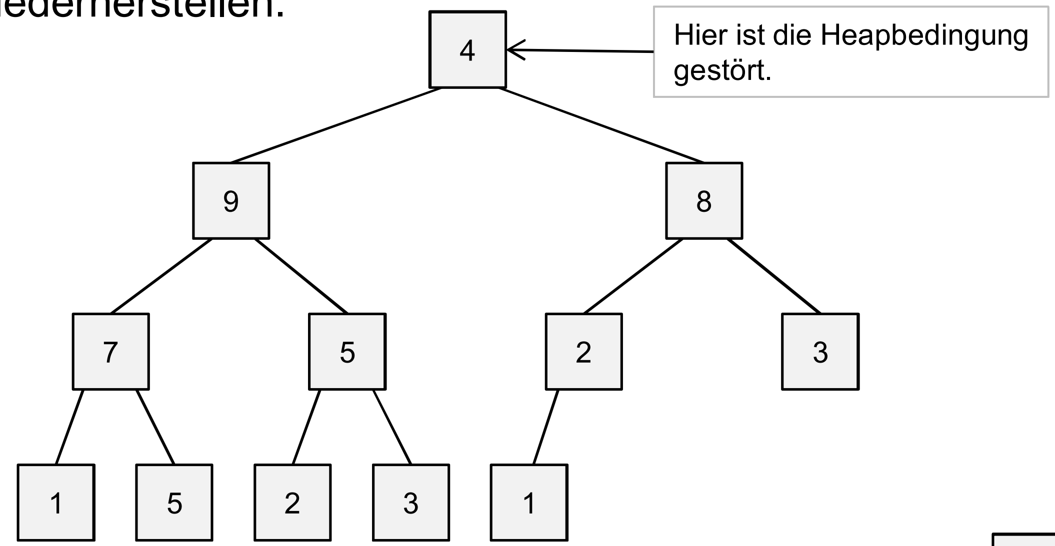
Beispiel



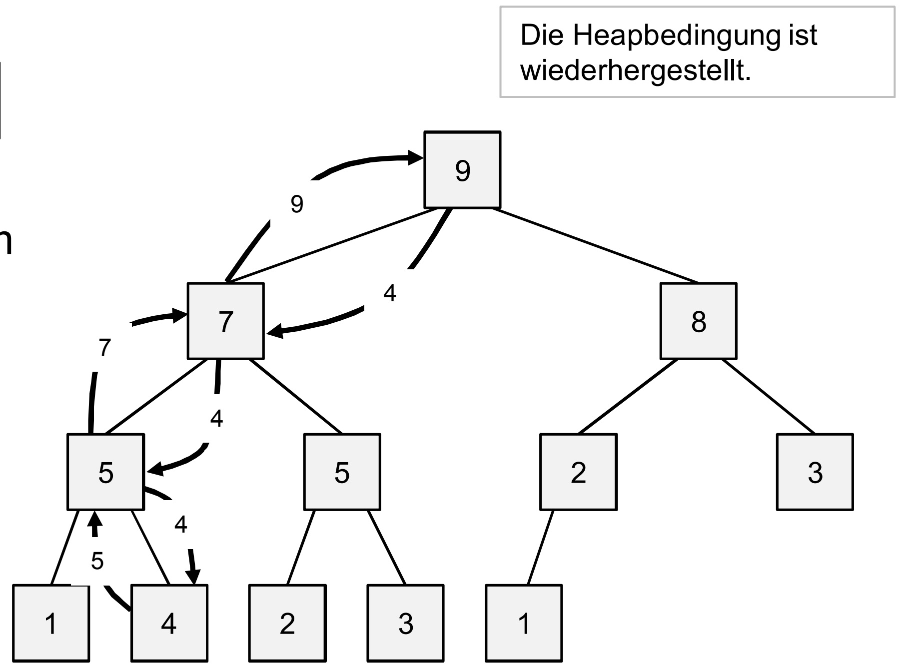
Aufgrund der Heapbedingung steht an der Wurzel des Baums (und jeden Teilbaums) das größte Element im Baum (Teilbaum).

Wiederherstellung der Heapbedingung

Wenn die Heapbedingung an einer (und nur einer) Stelle im Baum gestört ist, so kann man sie sehr einfach wiederherstellen.



Man tauscht den Störenfried so lange mit seinem größten Nachfolger, bis die Störung nach unten aus dem Baum herausgewachsen ist:



Funktion zur Wiederherstellung der Heapbedingung

Die Größe des Heaps,
der Heap,
der Index des Störenfrieds.

```
void adjustheap( int n, int *daten, int k)
    {
    int j, v;

    v = daten[k];
    while( k < n/2)
        {
        j = 2*k+1;
        if( (j < n-1) && (daten[j] < daten[j+1]))
            j++;
         if( v >= daten[j])
            break;
        daten[k] = daten[j];
        k = j;
        }
    daten[k] = v;
    }
```

Nimm den Störenfried aus dem Heap.

Fahre fort, solange der betrachtete Knoten noch mindestens einen Nachfolger hat.

Wenn es einen rechten Nachfolger gibt und dieser größer ist, als der linke, dann gehe zum rechten Nachfolger (`j++`).

Gehe zunächst zum linken Nachfolger.

Der Störenfried ist größer als sein größter Nachfolger. Es muss nicht weiter abgestiegen werden.

Ziehe den größten Nachfolger eine Ebene hoch und mache an dessen Knoten weiter.

Lege den Störenfried in den frei gewordenen Knoten

Heapsort - Implementierung

Baue einen Heap im zu sortierenden Array auf.

Tausche das erste (größte) Element des Heaps mit dem letzten. Anschließend repariere den um ein Element verkleinerten Heap. Fahre so fort, bis der Heap leer und der Array sortiert ist.

```
void heapsort( int n, int *daten)
    {
    int k, t;

    for( k = n/2; k;)
        adjustheap( n, daten, --k);

    while( --n)
        {
        t = daten[0];
        daten[0] = daten[n];
        daten[n] = t;
        adjustheap( n, daten, 0);
        }
    }
```

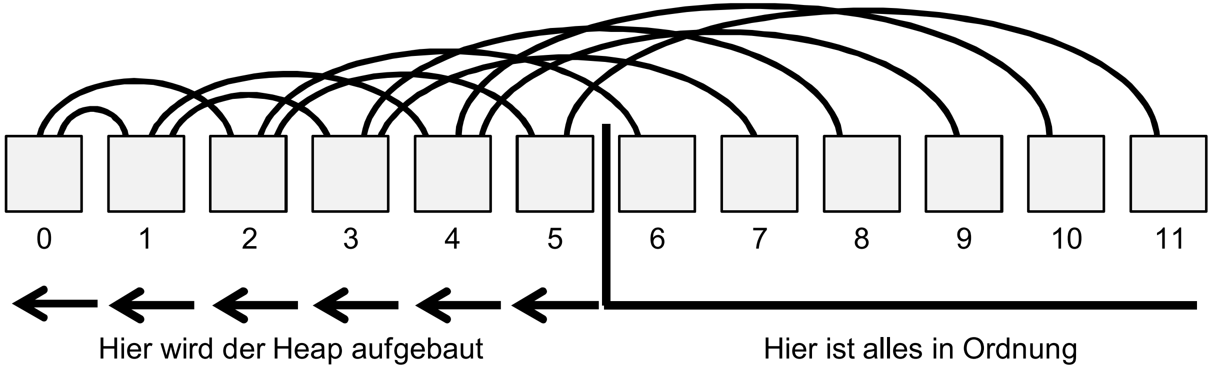
Hier wird von hinten nach vorn im Array ein Heap aufgebaut (Erklärung s.u.).

Solange sich der Heap durch Abtrennen des letzten Elements verkleinern lässt.

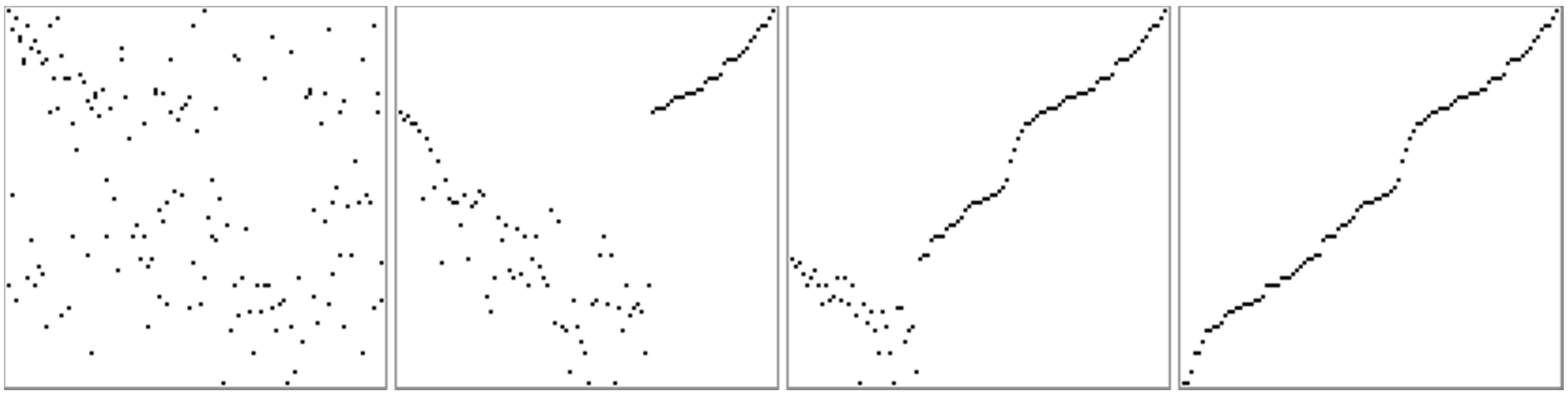
Tausche das erste (größte) Element mit dem letzten und repariere den um ein Element verkleinerten Heap.

Aufbau des Heaps

Durch Rückzug bei fortlaufender Adjustierung wird der Heap aufgebaut:



Danach beginnt die eigentliche Sortierung durch Tauschen und Reparieren (Adjustieren) bis der links im Array aufgebaute Heap leer ist:



Welches ist das beste Sortierverfahren?

Wir kennen jetzt sechs Sortierverfahren (es gibt übrigens noch viel mehr):

- Bubblesort
- Selectionsort
- Insertionsort
- Shellsort
- Quicksort
- Heapsort

Bezüglich ihres Speicherverbrauchs sind alle Verfahren gleich gut. Alle Verfahren sortieren den Array "in place" ohne zusätzlichen Speicher zu verwenden.

Bezüglich der Laufzeit gibt es aber erhebliche Unterschiede.

Analyse von Bubblesort

Bubblesort macht in der inneren Schleife zunächst n-1, dann n-2, … Durchläufe. Das sind insgesamt $(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$ Durchläufe.

Die Überdeckungsanalyse für n = 1000 Zahlen bestätigt dies:

$n = 1000$

```
              void bubblesort( int n, int *daten)
     1            {
                  int i, k, t;

   999            for( i = n-1; i > 0; i--)
                      {
499500                for( k = 0; k < i; k++)
                          {
                          if( daten[k] > daten[k+1])
                              {
239518                        t = daten[k];
                              daten[k] = daten[k+1];
                              daten[k+1] = t;
                              }
                          }
                      }
                  }
```

$n-1$

$\frac{n(n-1)}{2}$

$\approx \frac{n(n-1)}{4}$

c_{bub}

Bubblesort hat also quadratische Laufzeit: $t_{bub}(n) = c_{bub}\frac{n(n-1)}{2} \approx n^2$,

wobei die Konstante c_{bub} für die mittlere Laufzeit des Codes in der inneren Schleife steht. Bei zufälligen Daten ist in etwa der Hälfte der Fälle mit einer Tauschoperation zu rechnen.

Analyse von Selectionsort

Selectionsort macht $n-1$ Durchläufe in der äußeren und $\frac{n(n-1)}{2}$ Durchläufe in der inneren Schleife:

$n = 1000$

$n-1$

$\frac{n(n-1)}{2}$

```
            void selectionsort( int n, int *daten)
     1          {
                int i, k, t, min;

   999          for( i = 0; i < n-1; i++)
                    {
                    min = i;
499500              for( k = i+1; k < n; k++)
                        {
                        if( daten[k] < daten[min])
  5419                      min = k;
                        }
   999              t = daten[min];
                    daten[min] = daten[i];
                    daten[i] = t;
                    }
                }
```

c_{sel2}

c_{sel1}

Selectionsort hat wie Bubblesort quadratische Laufzeit: $t_{sel}(n) = c_{sel1}\frac{n(n-1)}{2} + c_{sel2}(n-1) \approx n^2$

Für kleine Werte von n mag Selectionsort wegen des Terms $c_{sel2}(n-1)$ langsamer sein als Bubblesort. Für große Werte ist Selectionsort aber wegen der offensichtlich besseren Laufzeit im Kern ($c_{sel1} < c_{bub}$) sicherlich schneller als Bubblesort – vielleicht 3-5 mal so schnell.

Analyse von Insertionsort

Bei Insertionsort wird die innere Schleife über eine zusätzliche Bedingung (`daten[k-1] > v`) kontrolliert, und gegebenenfalls vorzeitig abgebrochen. Bei zufällig verteilten Daten können wir davon ausgehen, dass diese Bedingung im Mittel bei der Hälfte des zu durchlaufenden Indexbereichs erfüllt ist, die Schleife also im Durchschnitt auf halber Strecke abgebrochen werden kann. Das bedeutet $\frac{n(n-1)}{4}$ Durchläufe in der inneren Schleife:

		`void insertionsort( int n, int *daten)`	$n = 1000$
		`{`	
	1	`int i, k, v;`	
$n-1$			
	999	`for( i = 1; i < n; i++)`	
		`{`	
		`v = daten[i];`	c_{ins2}
		`for( k = i; (k>=1)&& (daten[k-1] > v); k--)`	
	239518	`daten[k] = daten[k-1];`	c_{ins1}
$\approx \frac{n(n-1)}{4}$		`daten[k] = v;`	c_{ins2}
		`}`	
		`}`	

Selectionsort hat ebenfalls quadratische Laufzeit: $t_{ins}(n) = c_{ins1}\frac{n(n-1)}{4} + c_{ins2}(n-1) \approx n^2$

Wegen $c_{ins1} \approx c_{sel1}$ könnte Insertionsort doppelt so schnell wie Selectionsort sein.

Analyse von Shellsort

Die Überdeckungsanalyse von Shellsort zeigt deutlich weniger Schleifendurchläufe:

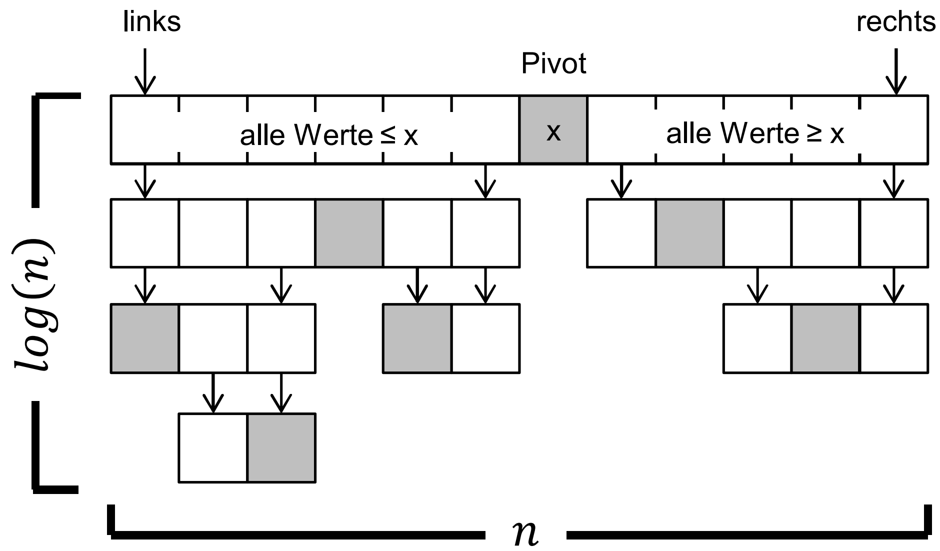
$n = 1000$

```
        void shellsort( int n, int *daten)
   1    {
        int i, k, h, v;

   5    for( h = 1; h <= n/9; h = 3*h+1)
            ;
        for( ; h > 0; h /= 3)
            {
4821        for( i = h; i < n; i++)
                {
                v = daten[i];
                for( k = i; (k >= h) && (daten[k-h] > v); k -= h)
9690                daten[k] = daten[k-h];
                daten[k] = v;
                }
            }
        }
```

Wie oft die inneren Schleifen durchlaufen werden, konnte bisher nicht allgemein berechnet werden, zumal ja auch noch die spezielle Wahl der Distanzenfolge (hier 1, 4, 13, ...) eine wichtige Rolle spielt. Für das obige Programm wird ein asymptotisches Verhalten wie $n\left(log(n)\right)^2$ oder $n^{\frac{5}{4}}$ vermutet. Sicher ist, dass Shellsort für die hier gewählte Distanzenfolge besser als Bubblesort, Insertionsort und Selectionsort ist. Auch in der Praxis zeigt Shellsort eine deutlich bessere Performance als die zuvor diskutierten Verfahren.

Analyse von Quicksort

Wenn man eine in etwa zentrierte Lage des Pivot unterstellt, hat Quicksort wegen der fortlaufenden Halbierung der zu betrachtenden Teilbereiche eine Rekursionstiefe von $log(n)$.



Auf jedem Teilbereich arbeitet das Unterprogramm `aufteilung`. Dieses Programm läuft linear über den Teibereich. Selbst wenn bei jedem Schritt eine Vertauschung erforderlich wäre, käme dabei nicht mehr als eine linear wachsende Laufzeit heraus. Da auf jeder Rekursionsebene über alle Teilbereiche hinweg (maximal) n Elemente zu betrachten sind und das Unterprogramm `aufteilung` in jedem dieser Teilbereiche mit linearer Zeitkomplexität arbeitet, ergibt sich für Quicksort das Laufzeitverhalten:

$$t_{qck}(n) = n \cdot log(n)$$

Quicksort ist damit deutlich besser als alle zuvor diskutierten Verfahren, sofern der Pivot bei jeder Teilung zentral gewählt wird. Bei ungünstiger Pivotwahl kann Quicksort auf quadratische Laufzeit zurückfallen, da der Array "worst case" nur um ein Element verkleinert wird.

Überdeckungsanalyse von Quicksort

```
        void qcksort(int links, int rechts, int *daten)
1333        {
            int pivot, i, j, t;

 666        if( links < rechts)
                {
                i = links-1;
                j = rechts;
                pivot = daten[rechts];
                for(;;)
                    {
2407                while( daten[++i] < pivot)
                        ;
2407                while( (j > i) && (daten[--j] > pivot))
                        ;
2407                if( i >= j)
 666                    break;
1741                t = daten[i];
                    daten[i] = daten[j];
                    daten[j] = t;
                    }
 666            daten[ rechts] = daten[i];
                daten[ i] = pivot;
                qcksort( links, i-1, daten);
                qcksort( i+1, rechts, daten);
                }
            }
```

In der Überdeckungsanalyse zeigt Quicksort die besten Werte aller bisherigen Verfahren.

Analyse von Heapsort: In der Funktion Heapsort wird $n - 1 + \frac{n}{2}$ mal `adjustheap` gerufen.
Das zeigt auch die Überdeckungsanalyse:

$n = 1000$

$\frac{n}{2}$

$n - 1$

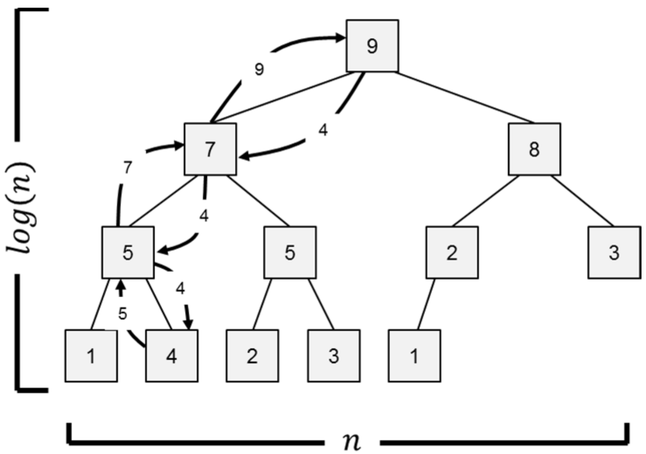
```
         void heapsort( int n, int *daten)
   1         {
             int k, t;

             for( k = n/2; k;)
 500             adjustheap( n, daten, --k);
             while( --n)
                 {
                 t = daten[0];
                 daten[0] = daten[n];
                 daten[n] = t;
 999             adjustheap( n, daten, 0);
                 }
             }
```

Die Funktion `adjustheap` benötigt maximal $log(n)$ (Tiefe des Heaps) Schritte, um den Heap zu reparieren.

Damit hat Heapsort die gleiche Laufzeitkomplexität wie Quicksort:

$$t_{heap}(n) = n \cdot log(n)$$

Obwohl Heapsort im asymptotischen Verhalten Quicksort entspricht, erwarten wir aufgrund der aufwändigeren inneren Schleife ein schlechteres Laufzeitverhalten als Quicksort.



Der Testrahmen für die Sortierprogramme

Testobjekt

```
void XXXsort( int n, int *daten)
    {
    ...
    }
```

Testrahmen

```
# define ANZAHL 100
# define SEED 4711

void main()
    {
    int daten[ANZAHL];

    srand( SEED);
    testdaten( ANZAHL, daten);

    XXXsort( ANZAHL, daten);

    if( pruefen( ANZAHL, daten))
        printf( "ok\n");
    else
        printf( "nicht ok\n");
    }
```

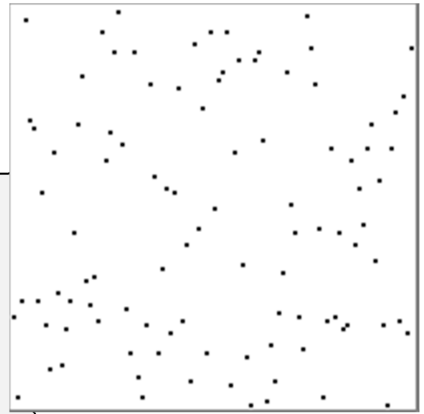
```
void testdaten( int n, int *daten)
    {
    int i;

    for( i = 0; i < n; i++)
        daten[i] = rand() % n;
    }
```

```
int pruefen( int n, int *daten)
    {
    int i;

    for( i = 0; i < n-1; i++)
        {
        if( daten[i] > daten[i+1])
            return 0;
        }
    return 1;
    }
```

Im Testrahmen werden Testdaten generiert, mit der zu testenden Funktion sortiert und anschließend auf korrekte Sortierung geprüft. In diesem Testrahmen werden mit wechselnden Funktionen zur Testdatengenerierung Laufzeitmessungen durchgeführt.

Laufzeitmessung mit Zufallsdaten

```
void testdaten1( int n, int *daten)
    {
    int i;

    for( i = 0; i < n; i++)
        daten[i] = rand()%n;
    }
```



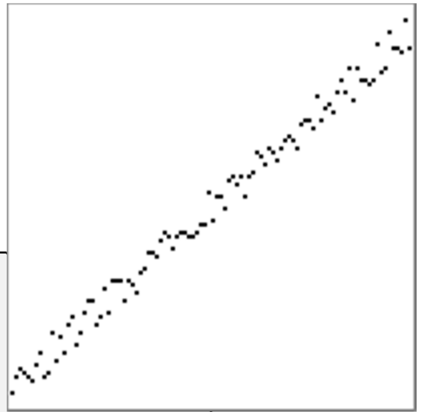
Zufallszahlen zwischen 0 und n-1.

Es ergeben sich folgende Messwerte in Millisekunden für bis zu 10 Millionen Daten:

testdaten1	Bubblesort	Selectionsort	Insertionsort	Shellsort	Quicksort rekursiv	Quicksort iterativ	Heapsort
100	0,01	0,01	0,00	0,00	0,00	0,00	0,00
1000	0,86	0,50	0,22	0,07	0,07	0,05	0,07
10000	150,26	39,72	18,90	1,05	0,86	0,66	0,88
100000	18159,78	3898,27	1938,95	14,60	9,10	7,59	11,29
1000000	ca. 30 min	ca. 6 min	3 min		456,57	85,17	150,36
10000000	ca. 2 Tage	ca. 12 std	ca. 6 std		4442,62	863,08	2507,95

Wie erwartet sind die $n \cdot log(n)$-Verfahren deutlich schneller als die n^2-Verfahren. Iteratives Quicksort ist durchweg das schnellste Programm. Quicksort benötigt weniger als eine Sekunde wenn Bubblesort bereits mehrere Tage rechnet.

Laufzeitmessung mit "gestreut" vorsortierten Daten



```
void testdaten2( int n, int *daten)
    {
    int i;

    for( i = 0; i < n; i++)
        daten[i] = (9*i)/10 + rand()%(n/10);
    }
```

10% Abweichung von aufsteigender Ordnung.

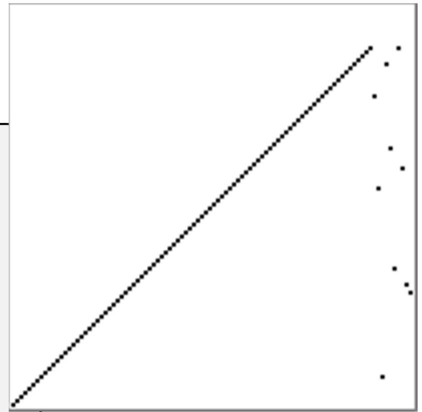
Getestet werden nur noch Insertionsort, iteratives Quicksort und Heapsort

testdaten2	Insertionsort	Quicksort iterativ	Heapsort
100	0,00	0,00	0,00
1000	0,03	0,05	0,07
10000	1,48	0,60	0,81
100000	142,68	7,34	9,84
1000000	4632,99	97,25	120,40
10000000		3796,46	1279,30

Insertionsort verbessert sich deutlich ohne jedoch Quicksort oder Heapsort zu erreichen.
Quicksort verschlechtert sich und fällt für große Datenmengen hinter Heapsort zurück.

Laufzeitmessungen für sortierte Daten mit 10% Ausreißern am Ende

```
void testdaten3( int n, int *daten)
    {
    int i;

    for( i = 0; i < (9*n)/10; i++)
        daten[i] = i;
    for( ; i < n; i++)
        daten[i] = rand()%n;
    }
```



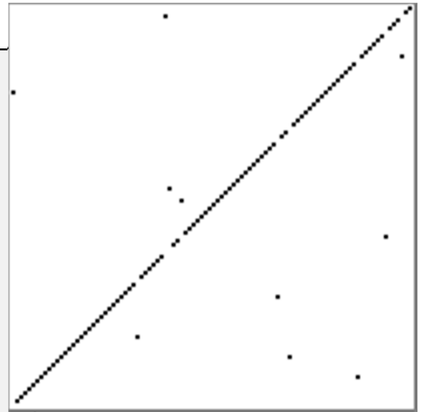
Die letzten 10% der Daten sind unsortiert.

testdaten3	Insertionsort	Quicksort iterativ	Heapsort
100	0,00	0,00	0,00
1000	0,03	0,04	0,06
10000	3,64	0,49	0,68
100000	591,22	99,74	7,04
1000000	71322,23	15651,39	76,21
10000000			884,36

Quicksort verschlechtert sich noch einmal und ist Heapsort jetzt deutlich unterlegen. Heapsort ist offensichtlich sehr robust, was vorsortierte Daten angeht. Insertionsort verschlechtert sich, bleibt aber besser als bei Zufallsdaten. Das liegt daran, dass hier großräumigere Verschiebungen zu machen sind als im vorherigen Testszenario.

Laufzeitmessungen für sortierte Daten mit 10% zufälligen Ausreißern

```
void testdaten4( int n, int *daten)
    {
    int i, k;

    for( i = 0; i < n; i++)
        daten[i] = i;
    for( i = 0; i < n/10; i++)
        {
        k = rand()%n;
        daten[k] = rand()%n;
        }
    }
```



10% zufällig gewählte Daten sind unsortiert.

testdaten4	Insertionsort	Quicksort iterativ	Heapsort
100	0,00	0,00	0,00
1000	0,03	0,06	0,06
10000	2,60	0,63	0,68
100000	75,01	1751,50	6,67
1000000	203,31		70,88
10000000	221,15		795,65

Quicksort verschlechtert sich noch einmal, während Heapsort unverändert bleibt. Das schnellste Programm ist jetzt aber Insertionsort.

Fazit aus der Analyse und den Messungen

- Bei zufällig verteilten Daten ist Quicksort das beste Sortierprogramm. Quicksort kann aber empfindlich einbrechen, wenn die Daten teilsortiert sind.
- Insertionsort arbeitet am schnellsten, wenn zur Sortierung nur wenige Daten über kleine Stecken bewegt werden müssen.
- Heapsort ist sehr robust gegenüber verschiedenen Vorsortierungen und erreicht fast die Performance von Quicksort.

Man könnte versuchen, Quicksort durch eine bessere Wahl des Pivots robuster zu machen. Dazu gibt es verschiedene Ansätze. Man könnte den Pivot zufällig wählen oder drei zufällige Werte aus dem Array betrachten und den mittleren der drei auswählen. Aber keiner der Ansätze verbessert Quicksort in allen denkbaren Situationen, zumal solche Erweiterungen auch zusätzliche Rechenzeit verbrauchen.

Für welches Verfahren soll man sich entscheiden, wenn man keine Informationen über die Verteilung der zu sortierenden Daten hat?

Die Antwort auf diese Frage lautet Introsort.

Introsort ist ein hybrides Verfahren, dass die Vorteile von Quicksort, Heapsort und Insertionsort zu kombinieren versucht. Introsort startet als Quicksort und beobachtet dabei die Tiefe des Abstiegs. Wenn dabei ein bestimmter Wert (z.B. $2 \cdot log(n)$) überschritten wird, schaltet das Verfahren auf Heapsort um (stop loss). Wenn am Ende nur noch kleine Teilbereiche zu sortieren sind wird die Feinarbeit mit Insertionsort gemacht.

Grenzen der Optimierung von Sortierverfahren

Ein Sortierverfahren erzeugt eine ganz bestimmte Permutation der zu sortierenden Daten. Wir wissen, dass es n! solcher Permutationen gibt. Ein Sortierverfahren, das einen beliebigen Array mit n Elementen sortieren kann, muss prinzipiell in der Lage sein, alle möglichen Permutationen zu erzeugen, da es ja eine beliebig vorgegebene Permutation rückgängig machen muss. Wenn das Verfahren dabei seine Informationen aus Einzelvergleichen zweier Elemente zieht, kann man folgendes feststellen:

Mit einem Vergleich können maximal zwei Permutationen erzeugt werden. Man kann aufgrund des Vergleichs alles so lassen wie es ist oder eine ganz bestimmte Vertauschung vornehmen. Mit k Vergleichen können maximal doppelt so viele Permutationen erzeugt werden, wie mit k-1 Vergleichen, da man auch hier wieder zwei Möglichkeiten hat. Man kann alles so lassen oder eine möglicherweise neue Permutation erzeugen.

Insgesamt kann man also sagen, dass man mit k Vergleichen maximal 2^k Permutationen erzeugen kann. Die Anzahl k der Vergleiche muss also mindestens so groß sein, dass $2^k \geq n!$ ist. Es muss also gelten: $k = log\left(2^k\right) \geq log(n!) \geq log\left(n^{\frac{n}{2}}\right) = \frac{n}{2} log(n)$. Damit hat das Verfahren mindestens die Laufzeitkomplexität $n \cdot log(n)$.

Wir fassen dieses wichtige theoretische Ergebnis noch einmal zusammen:

Sortierverfahren, die auf Einzelvergleichen und Vertauschungen basieren, haben mindestens die Laufzeitkomplexität $n \cdot log(n)$.

In diesem Sinn sind Quicksort und Heapsort optimal.

Sortieren mit linearer Zeitkomplexität

In speziellen Situationen kann man Verfahren konstruieren, die nicht auf Einzelvergleichen beruhen und effizienter als Quicksort sind.

Beispiel: Sortieren von Briefen nach Postleitzahl

Beispiel: Sortieren von Briefen (Radixsort bzw. Distributionsort)

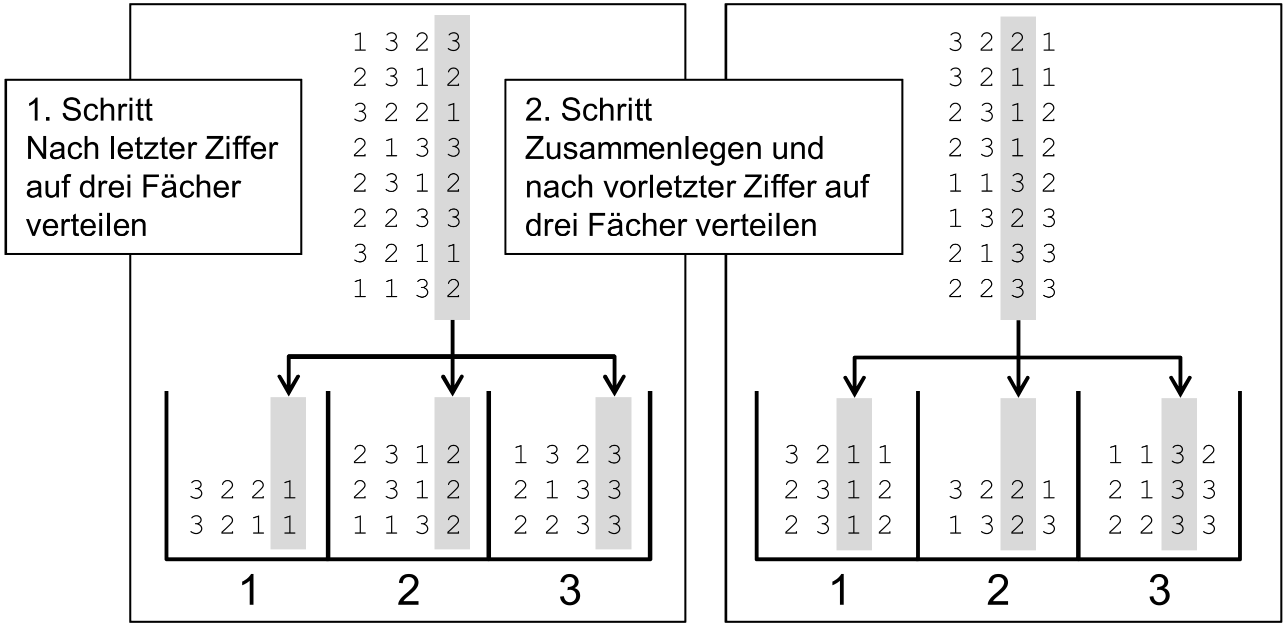
Sortiere 4-stellige Postleitzahlen mit Ziffern 1, 2 und 3

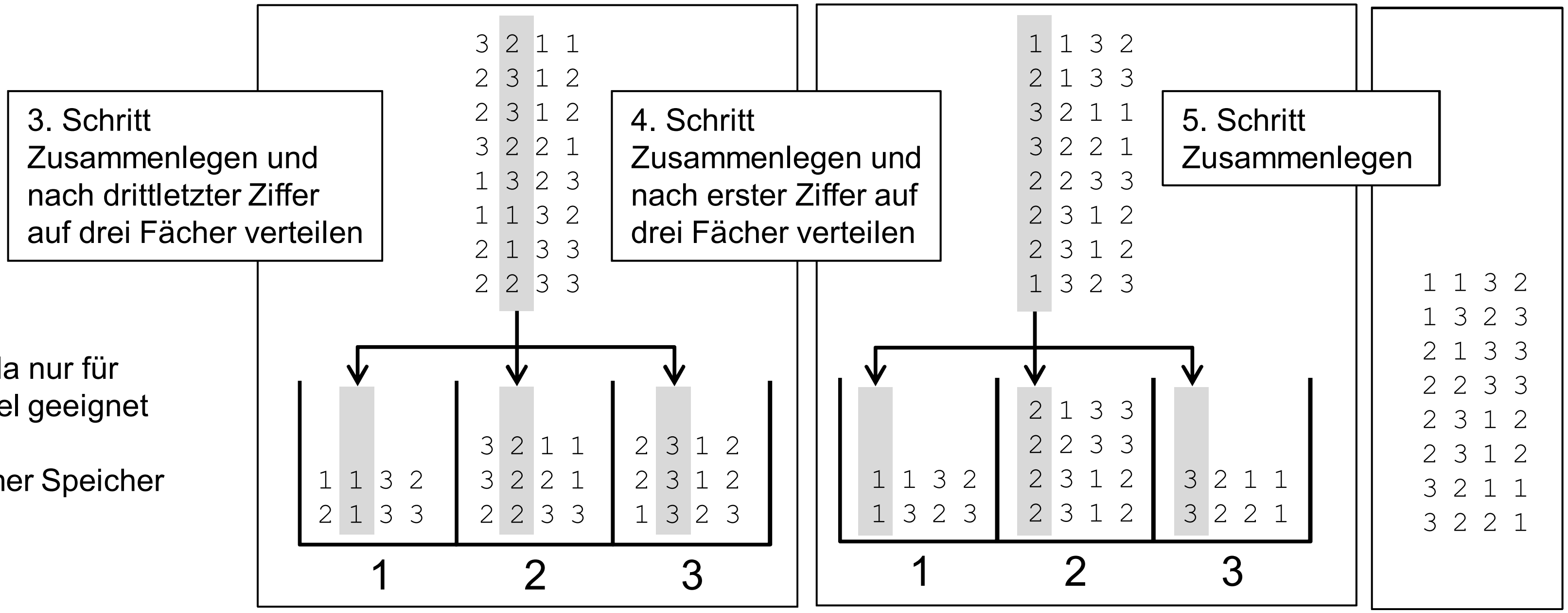
Jeder Schritt ist linear.

Es werden max. 5 (Stellenzahl+1) Durchläufe gemacht.

Das Verfahren ist linear.

Zu keinem Zeitpunkt werden zwei Postleitzahlen miteinander verglichen.





Nachteile:

Nicht universell, da nur für spezielle Schlüssel geeignet

Es wird zusätzlicher Speicher benötigt.