17 SORTIEREN IN VEKTOREN

17.1 ÜBERBLICK

- Wie kann man Daten sortieren?
- Wie kann man Daten *effizient* sortieren?
- Wie wählt man das effizienteste Sortierverfahren aus?
- Wie misst man die Schwierigkeit eines Problems?

17.2 BEGRIFFE

- Algorithmus: Endlich beschreibbar, effektiv ausführbar, eindeutig
- Effizienz:
  - Minimierung der Ressourcen: Zeit, Speicherplatz
  - Effizienzmessung: abstrakte Rechner mit standardisiertem Befehlssatz (Knuth[Knuth75]: MIX, Mehlhorn: RAM und RASP)
  - Zählen typischer Operationen:
    Vergleiche, Vertauschungen, Multiplikationen
- Komplexität eines Algorithmus

$$O(f) = \{g : \mathbb{N}_0 \to \mathbb{N}_0 \mid (\exists c > 0)(\exists n_0 \in \mathbb{N}_0) (\forall n \geq n_0)(g(n) \leq c f(n))\}$$

- Komplexitätsklassen

$O(1)$	konstant
$O(\log n)$	logarithmisch
$O(n)$	linear
$O(n^2)$	quadratisch
$O(n^p)$	polynomiell
$O(k^n)$	exponentiell

17.3 ENTWURFSPARADIGMEN

- Divide et impera
  - Zerlege das Problem in kleinere Teilprobleme
  - Löse diese Teilprobleme
  - Setze die Teillösungen zusammen.
  - Beispiele: Sortierverfahren
- Dynamische Programmierung
  - Zerlege das Problem in kleinere Teilprobleme und bilde alle möglichen Kombinationen
  - Löse diese Teilproblem-Kombinationen
  - Wähle die beste Kombination
  - Beispiele: Optimale Bäume, Wortproblem bei beliebigen kontextfreien Sprachen
- Iteration:
  - Verbessere eine Näherungslösung.
  - Beispiel: Maximaler Fluß in Netzen
- Exhaustive Suche (“brute force”)
  - Konstruiere der Reihe nach alle zulässigen Lösungen, und wähle die optimale.
  - Beispiele: Problem des Handlungsreisenden, Wortproblem bei kontextsensitiven Sprachen

17.4 SORTIEREN

- Problem
  - Grundmenge U (Universum)
  - lineare Ordnung $(S, \leq)$ auf einer Schlüsselmenge
  - $s : U \to S$ Schlüsselfunktion
  - $\{E_1, E_2, E_3, \ldots, E_n\} \subset U$
  - Finde Permutation $i_1, i_2, \ldots, i_n$ der Zahlen $1, 2, \ldots, n$ mit $(\forall 1 \leq j \leq n-1)(s(E_{i_j}) \leq s(E_{i_{j+1}}))$
- Anwendung der Verfahren

		Änderung	
		selten	häufig
Wahlfreier Zugriff	billig	Quicksort Heapsort	binäre Bäume
	teuer	Mergesort	Bayer-Bäume

- Vergleich einiger Verfahren

Verfahren	maximal	im Mittel
Heapsort	$2n\,\mathrm{ld}(n+1)$	$\approx 2n\,\mathrm{ld}\,n$
Quicksort	$\frac{1}{2}n^2$	$1.44n\,\mathrm{ld}\,n$
Mergesort	$n\,\mathrm{ld}\,n$	$n\,\mathrm{ld}\,n$
Max.ausw.	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$

- Stabiles Sortierverfahren:
  Seien zwei Schlüsselfunktionen $s_1 : U \to S_1$ und $s_2 : U \to S_2$ mit $i \leq j \Longrightarrow s_2(E_i) \leq s_2(E_j)$ gegeben. Ein Sortierverfahren heißt *stabil*, wenn es eine Permutation E_{i_j} liefert, so daß für $j \leq k$ gilt

$$s_1(E_{i_j}) \leq s_1(E_{i_k}) \wedge \Big(s_1(E_{i_j}) = s_1(E_{i_k}) \Rightarrow s_2(E_{i_j}) \leq s_2(E_{i_k})\Big)$$

17.5 SORTIEREN DURCH MAXIMUMAUSWAHL

512	087	503	061	908	170	897
						↑
				↑		
512	087	503	061	897	170	**908**
					↑	**908**
				↑		**908**
512	087	503	061	170	**897**	**908**
				↑	**897**	**908**
		↑			**897**	**908**
↑					**897**	**908**
170	087	503	061	**512**	**897**	**908**
			↑	**512**	**897**	**908**
		↑		**512**	**897**	**908**
170	087	061	**503**	**512**	**897**	**908**
		↑	**503**	**512**	**897**	**908**
	↑		**503**	**512**	**897**	**908**
↑			**503**	**512**	**897**	**908**
061	087	**170**	**503**	**512**	**897**	**908**
	↑	**170**	**503**	**512**	**897**	**908**
061	**087**	**170**	**503**	**512**	**897**	**908**

17.6 SORTIEREN DURCH MAXIMUMAUSWAHL

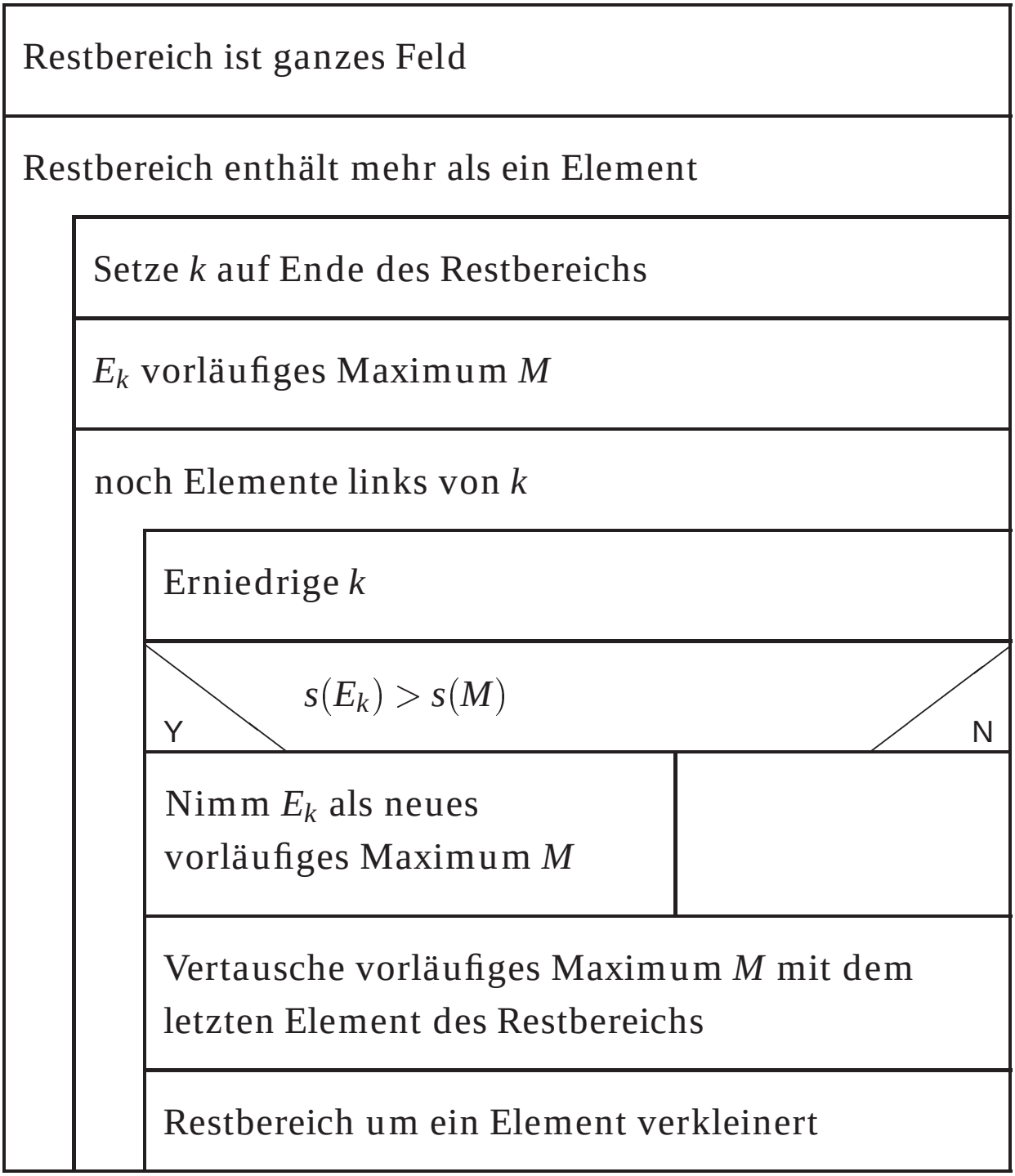
- Aufteilung in zwei Teilmengen L und R
- Schleifeninvariante:

$$(\forall e \in L)(\forall e' \in R)(e \leq e') \wedge R \text{ sortiert}$$

- Initialisierung: $R = \emptyset$
- Abbruch: $|L| = 1$
- Schleifenrumpf
  - Suche $\max(L)$
  - Entferne es aus L
  - Füge es als erstes Element in R ein.

- Algorithmus:

Maximumauswahl —



17.7 SORTIEREN MIT MERGESORT

- Sortieren durch Verschmelzen
- Idee:
  Divide-et-impera-Verfahren liefern optimales Zeitverhalten, wenn man in zwei gleichgroße Teilprobleme teilt. Mergesort halbiert einfach die zu sortierende Menge, muß dann aber die sortierten Teilmengen unter Beachtung des Sortierkriteriums „verschmelzen“.
- Rekursive Zerlegung:

512 170 061 897 908 087 503 415							
512 170 061 897	908 087 503 415						
512 170	061 897	908 087	503 415				
512	170	061	897	908	087	503	415

- Iterative Verschmelzung:

512	170	061	897	908	087	503	415
170 512	061 897	087 908	415 503				
061 170 512 897	087 415 503 908						
061 087 170 415 503 512 897 908							

- Das Verfahren mit Hintergrundspeicher benutzt nur den iterativen Verschmelzungsvorgang.

17.8 REKURSIVES MERGESORT

```
public class MergeSortRekursiv{
    int anzahlElemente = 20;
    int[] zuSortieren = new int[anzahlElemente];
    int[] hilfsFeld = new int [anzahlElemente];

    void feldBelegen(){
        int lfv;
        for(lfv=0;lfv<zuSortieren.length;lfv++){
            zuSortieren[lfv] =
                (int) (Math.random()*100);}
    }

    void feldAusgeben(){
        int lfv;
        for(lfv=0;lfv<zuSortieren.length;lfv++){
            System.out.print
              (zuSortieren[lfv]+"␣␣");}
        System.out.println("\n");
    }
```

```
void mergesort(int links, int rechts)
{int i,j,k,mitte;

if ( rechts-links >0 )
    {   mitte = (int)(rechts+links)/2;

        mergesort(links, mitte);
        mergesort(mitte+1,rechts);

        for (i=mitte;i>=links; i--)
          {hilfsFeld[i] =
            zuSortieren[i];}

        for (j=mitte+1;j<=rechts;j++)
           {hilfsFeld[rechts+mitte+1-j] =
             zuSortieren[j];}

        i=links;j=rechts;
        for (k=links;k<=rechts;k++){
            if(hilfsFeld[i]<hilfsFeld[j]){
                zuSortieren[k]=hilfsFeld[i];
                i++;
            }
            else {zuSortieren[k]=hilfsFeld[j];
                  j--;
            }
        }
    }
}//mergesort
```

```
    public static void main(String[] args) {

        MergeSortRekursiv merge =
          new MergeSortRekursiv();

        System.out.println("Unsortiertes Feld: ");
        merge.feldBelegen();
        merge.feldAusgeben();

        System.out.println("Sortiertes Feld:   ");
        merge.mergesort
            (0,merge.zuSortieren.length-1);

        merge.feldAusgeben();
    }//main

}//MergeSortRekursiv
```

17.9 ITERATIVES MERGESORT

MergeSort iterativ: —

Datei A = Datei 1; Datei B = Datei 2;
Solange Quelldatei nicht abgearbeitet
Vergleiche die beiden nächsten Elemente
Schreibe kleineres in Datei A
Schreibe größeres in Datei A
Vertausche Dateien A und B

MergeSort iterativ (Fort.): —

- Datei C = Datei 3; Datei D = Datei 4;
- Solange weder Datei A noch Datei B leer
  - Solange weder Datei A noch Datei B leer
    - Betrachte die nächste Gruppe aus A
    - Betrachte die nächste Gruppe aus B
    - Solange beide Gruppen nicht leer
      - Betrachte Anfangselemente
      - Schreibe das kleinere nach C
      - Streiche es aus seiner Gruppe
    - Kopiere nichtleere Gruppe nach C
    - Vertausche Dateien C und D
  - Ggf. Kopiere letzte Gruppe nach C
  - Vertausche Datei A mit C und B mit D

17.10 Sortieren durch Zerlegen (Quicksort, Hoare)

- Vorgehen nach Hoare: Sortiere den Bereich $[l \ldots r]$
  1. Wähle ein *beliebiges* (!) Element E_x.
  2. Wiederhole, bis sich i und k getroffen haben.
     Suche von links (l) nach einem Element $s(E_i) \geq s(E_x)$.
     Suche von rechts (r) nach einem Element $s(E_k) \leq s(E_x)$.
     Vertausche die Elemente E_i und E_k miteinander.
  3. Sortiere die beiden Teilbereiche $[l \ldots k-1]$ und $[i+1 \ldots r]$.
- Beispiel:

170	512	087	503	061	908	170	897	415
	↑					↑		
170	170	087	503	061	908	512	897	415
			↑	↑				
170	170	087	061	503	908	512	897	415
			↑	↑				
	170	087	061	$\leq$	170			
		170	$\leq$	503	908	512	897	415

- Vorgehen nach Knuth:

512	512	087	503	061	908	170	897	415
								↑
512	415	087	503	061	908	170	897	**512**
					↑			
512	415	087	503	061	**512**	170	897	908
						↑		
512	415	087	503	061	170	**512**	897	908
	415	087	503	061	170	–	897	908

17.11 QUICKSORT

```
public class QuickSortRekursiv{
    int anzahlElemente = 20;
    int[] zuSortieren = new int[anzahlElemente];


    void feldBelegen(){
        int lfv;
        for(lfv=0;lfv<zuSortieren.length;lfv++){
            zuSortieren[lfv] =
                (int) (Math.random()*100);}
    }

    void feldAusgeben(){
        int lfv;
        for(lfv=0;lfv<zuSortieren.length;lfv++){
            System.out.print
                (zuSortieren[lfv]+"  ");}
        System.out.println("\n");
    }
```

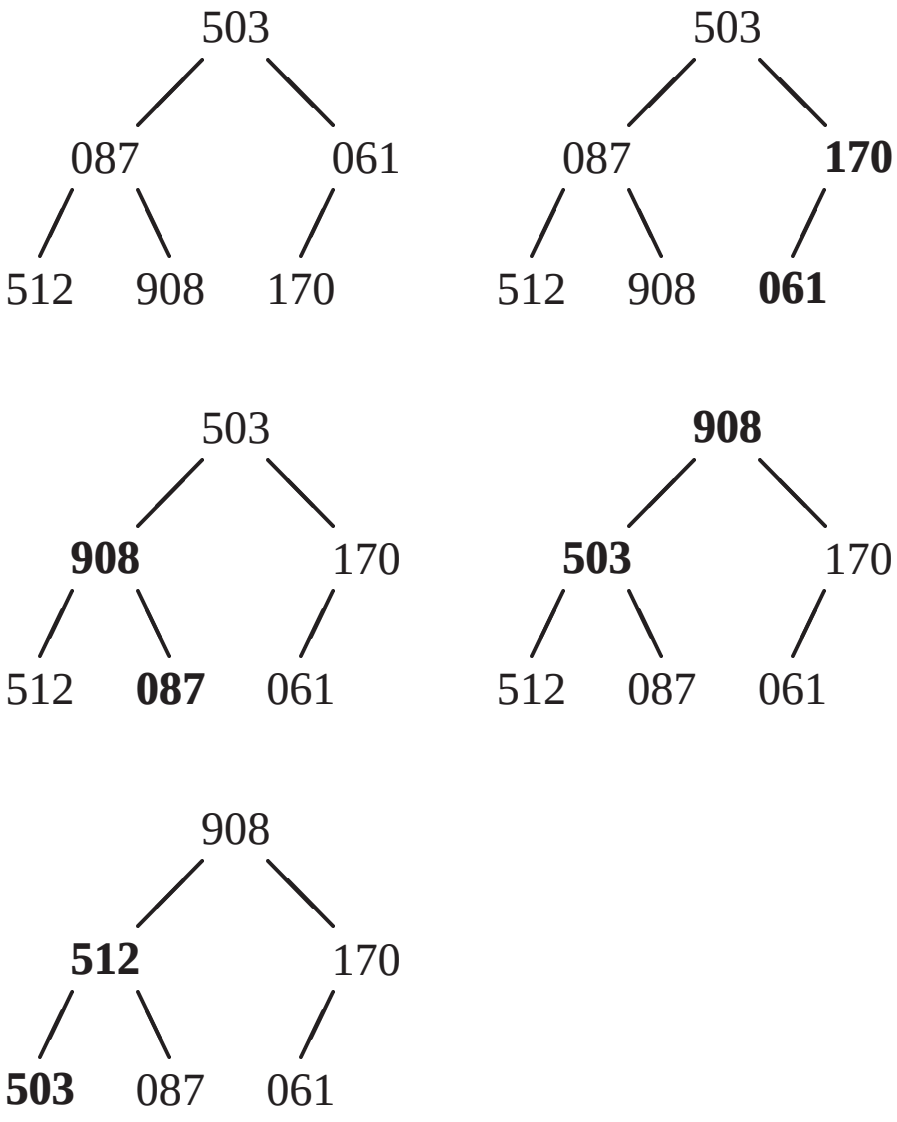
```
void quicksort(int links, int rechts)
{int vergleichsElem,hilf,i,j;
if ( links < rechts  )
    {
        vergleichsElem = zuSortieren[rechts];
        i=links-1;
        j=rechts;
        do {
            do{i++;}
              while (zuSortieren[i]
                    <vergleichsElem);
            do{j--;}
              while (zuSortieren[j]>
                     vergleichsElem);
            hilf=zuSortieren[i];
            zuSortieren[i]=zuSortieren[j];
            zuSortieren[j]=hilf;
        } while (j>i);

        zuSortieren[j]=zuSortieren[i];
        zuSortieren[i]=zuSortieren[rechts];
        zuSortieren[rechts]= hilf;

        quicksort(links, i-1);
        quicksort(i+1,rechts);
    }
}//quicksort
```

```
    public static void main(String[] args) {

        QuickSortRekursiv quick =
            new QuickSortRekursiv();

        System.out.println("Unsortiertes Feld: ");
        quick.feldBelegen();
        quick.feldAusgeben();

        System.out.println("Sortiertes Feld:   ");
        quick.quicksort
            (0,quick.zuSortieren.length-1);

        quick.feldAusgeben();
    }//main

}
```

17.12 HEAPSORT

Idee: Man merke sich das Ergebnis einmal durchgeführter Vergleiche!

503 087 061 512 908 170



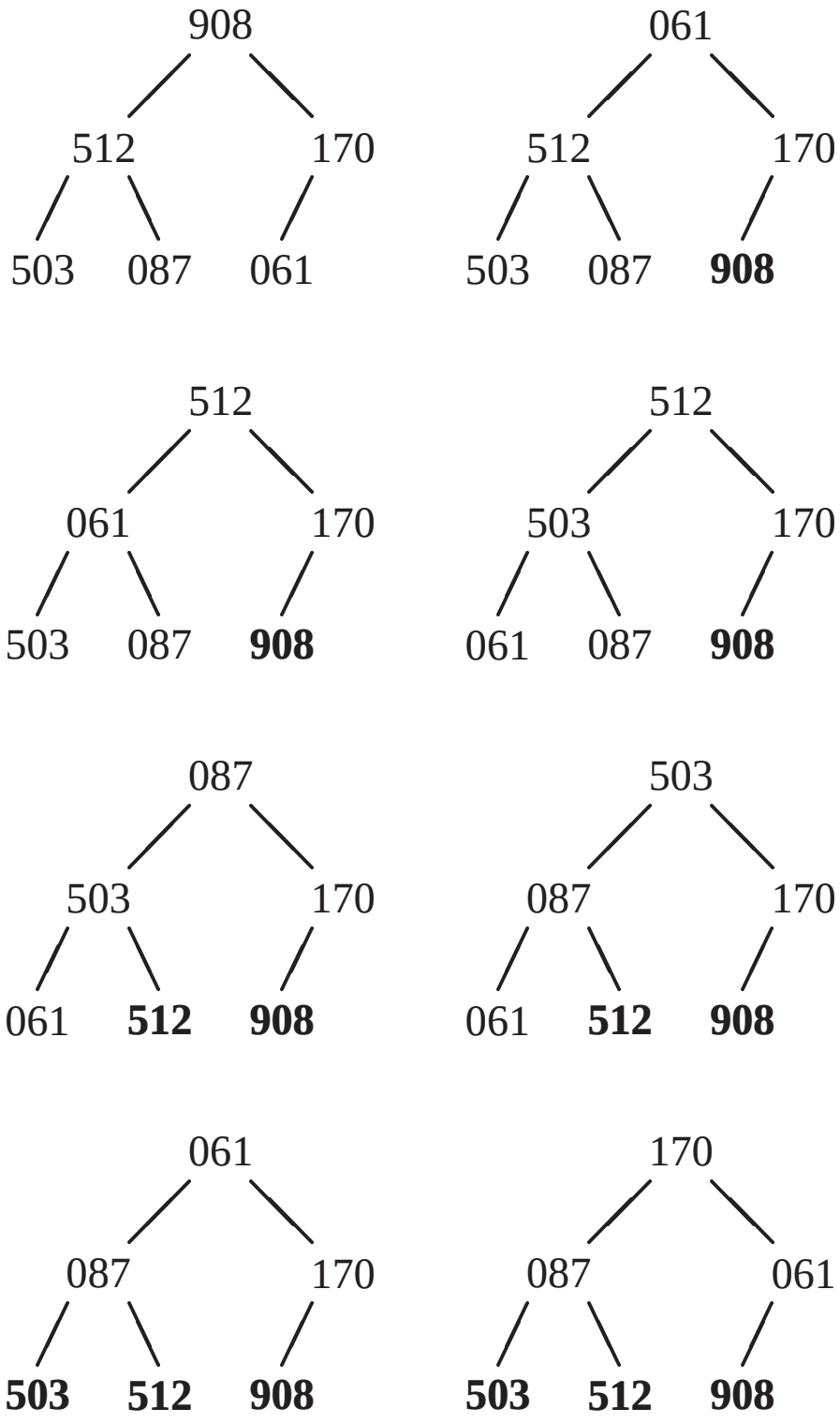
Jedes Element ist größer als seine beiden Nachfolger.

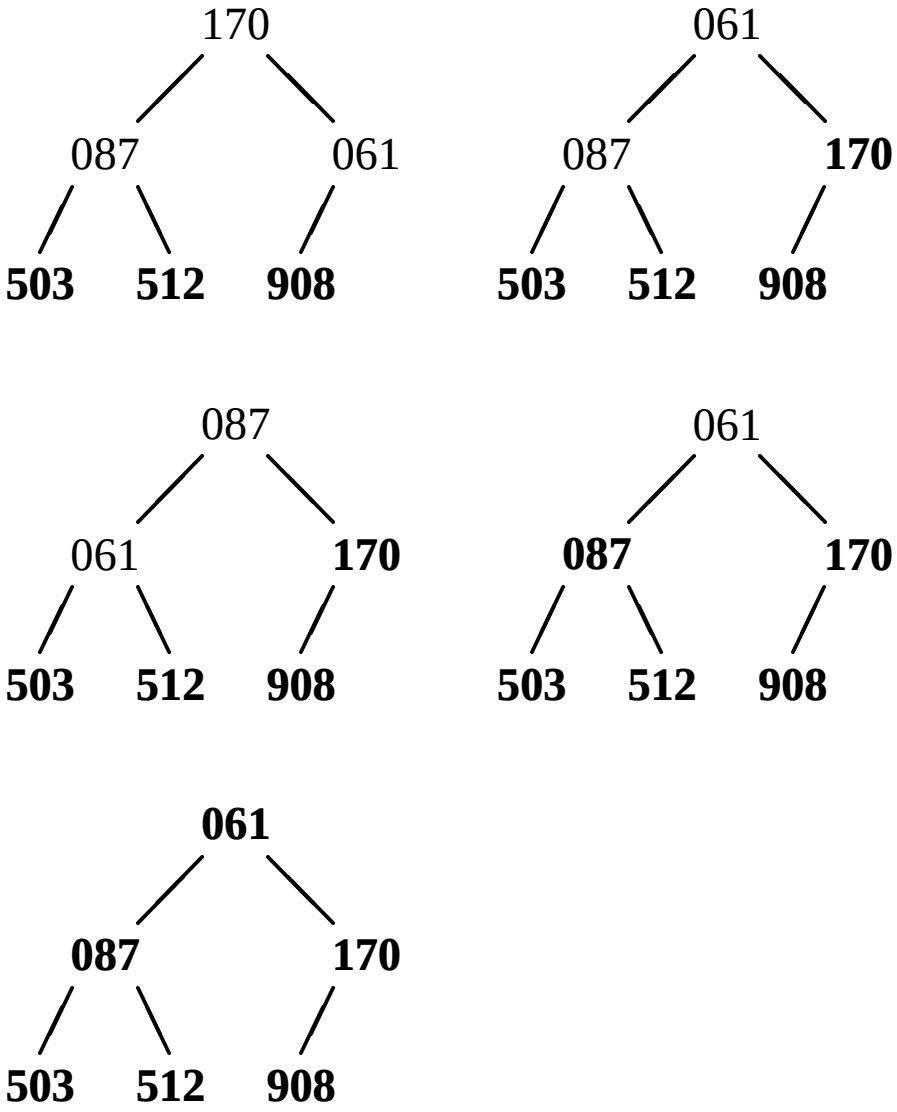
- Ordne (zumindest gedanklich) die Elemente als vollständigen Baum an.
- Heap-Eigenschaft:
  - Ein Feld $E_1, E_2, \ldots, E_n$ erfüllt ab Position l die Heap-Eigenschaft, wenn gilt:

$$(\forall k)(l \leq k/2 < k \leq n \Longrightarrow s(E_{k/2}) \geq s(E_k))$$

  - Ein Feld ist ein Heap, wenn es die Heap-Eigenschaft ab Position 1 erfüllt.
- Stelle Heap-Eigenschaft her:
  - Beginne auf der letzten vollständigen Ebene (rückwärts bis zum Anfang)
  - Vergleiche das Element mit seinen (beiden) Nachfolgern
  - Ist der größere größer als das Element, so stelle von da an Heap-Eigenschaft her
- Auswahlphase
  - Vertausche Wurzel mit letztem Element
  - Stelle für den Baum ohne das letzte Element die Heap-Eigenschaft wieder her
  - Wiederhole, bis nur noch ein Element

17.13 HEAPSORT: AUSWAHLPHASE





Ergebnis:

061 087 170 503 512 908

17.14 ZUSAMMENFASSUNG

- Wie kann man Daten *effizient* sortieren? Sortierverfahren
  - Maximumauswahl
  - Mergesort (Verschmelzen)
  - Quicksort (Zerlegen)
  - Heapsort
- Wie wählt man das effizienteste Sortierverfahren aus? Auswahl aufgrund der Eigenschaften des Datenbestandes und der Zugriffe
- Wie misst man die Schwierigkeit eines Problems? Komplexitätsklassen

17.15 LITERATUR

- Algorithmen allgemein: [Knuth75, Sedgewick72]
- Quicksort Knuth [Knuth75]
- Mergesort Hoare [Hoare62]

Literaturverzeichnis

[Hoare62] Hoare, C. A. R. *Quicksort*. *Computer Journal*, 5(1), 1962.

[Knuth75] Knuth, D. E. *The Art of Computer Programming*, Band Vol. 3, Sorting and Searching. Addison-Wesley, 1975.

[Sedgewick72] Sedgewick, Robert. *Algorithmen*. Addison-Wesley, 1972.