Informatik II für Verkehrsingenieure
Sortieren (Kapitel 10)

Janis Voigtländer

Technische Universität Dresden

Sommersemester 2007

Überblick

Problemstellung

Insertsort

Quicksort

Heapsort

Problemstellung

Intern: Daten befinden sich im Hauptspeicher mit beliebigem Zugriff

Problemstellung

Intern: Daten befinden sich im Hauptspeicher mit beliebigem Zugriff

Gegeben: Feld mit zu sortierenden Zahlen:

```
int   a[n];             /* a[0] ... a[n-1] */
```

Problemstellung

Intern: Daten befinden sich im Hauptspeicher mit beliebigem Zugriff

Gegeben: Feld mit zu sortierenden Zahlen:

```
int   a[n];            /* a[0] ... a[n-1] */
```

Gesucht: die selben Zahlen in aufsteigender Reihenfolge, im selben Feld

Problemstellung

Intern: Daten befinden sich im Hauptspeicher mit beliebigem Zugriff

Gegeben: Feld mit zu sortierenden Zahlen:

```
int   a[n];            /* a[0] ... a[n-1] */
```

Gesucht: die selben Zahlen in aufsteigender Reihenfolge, im selben Feld

Beispiel:

12	7	9	8	4	6

Problemstellung

Intern: Daten befinden sich im Hauptspeicher mit beliebigem Zugriff

Gegeben: Feld mit zu sortierenden Zahlen:

```
int   a[n];           /* a[0] ... a[n-1] */
```

Gesucht: die selben Zahlen in aufsteigender Reihenfolge, im selben Feld

Beispiel:

12	7	9	8	4	6

$\mapsto$

4	6	7	8	9	12

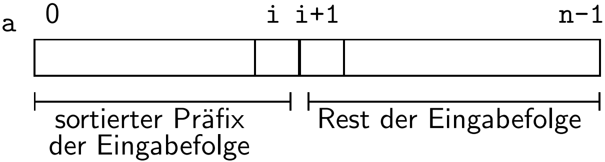
Insertsort

Idee: „direktes Einfügen“

Insertsort

Idee: „direktes Einfügen“

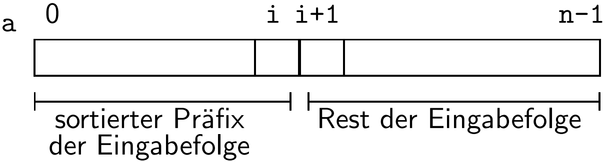
Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



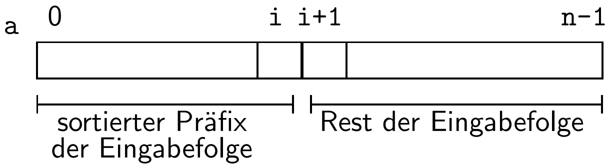
Beispiel:

12	7	9	8	4	6

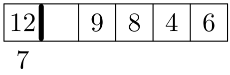
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



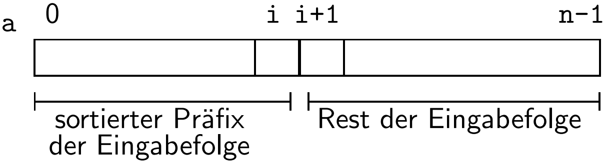
Beispiel:



Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



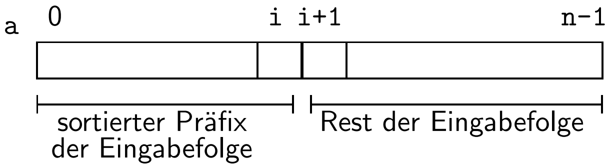
Beispiel:

7	12	9	8	4	6

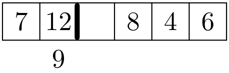
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



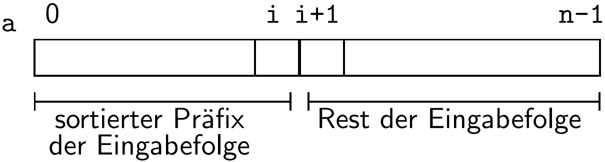
Beispiel:



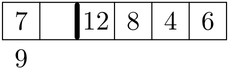
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



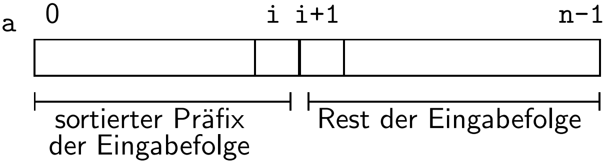
Beispiel:



Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



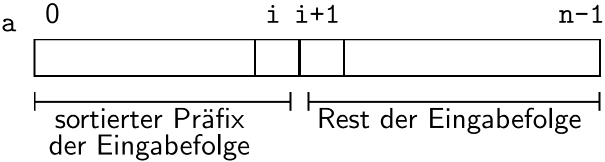
Beispiel:

7	9	12	8	4	6

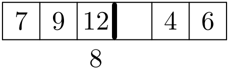
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



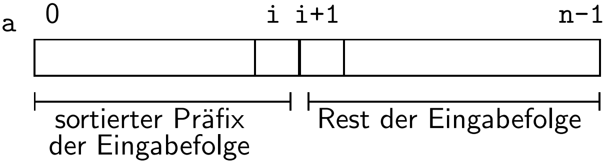
Beispiel:



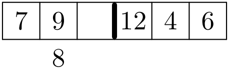
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



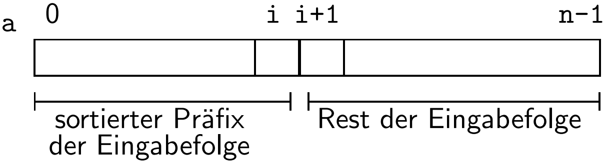
Beispiel:



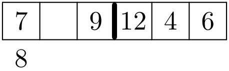
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index i mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



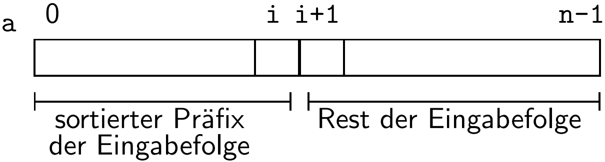
Beispiel:



Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



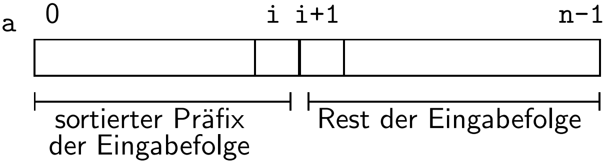
Beispiel:

7	8	9	12	4	6

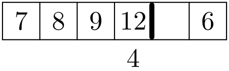
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



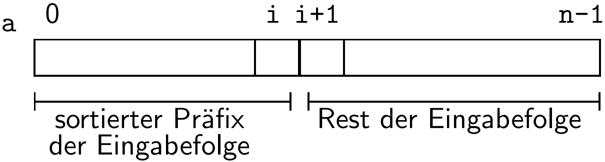
Beispiel:



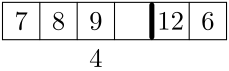
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



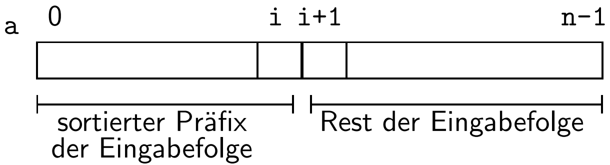
Beispiel:



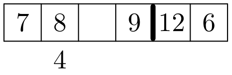
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



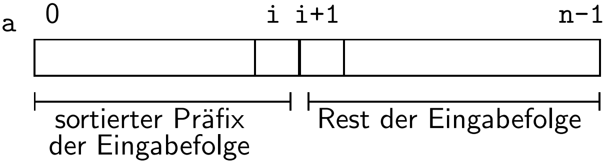
Beispiel:



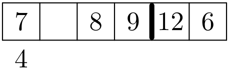
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



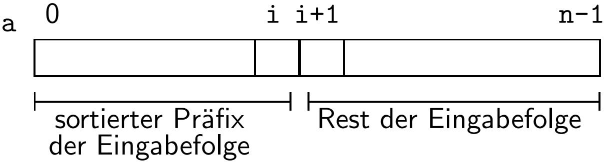
Beispiel:



Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \le \mathtt{i} \le \mathtt{n}-1$, so dass:



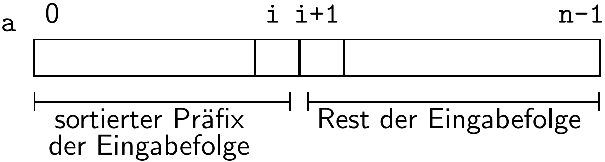
Beispiel:

4	7	8	9	12	6

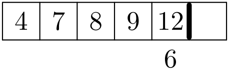
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



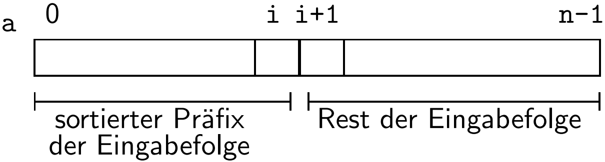
Beispiel:



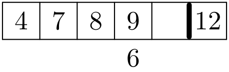
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



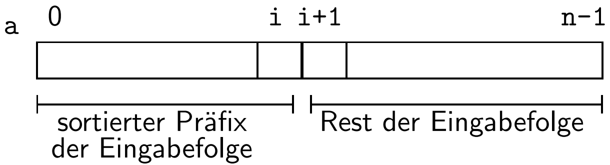
Beispiel:



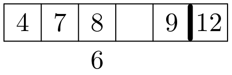
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



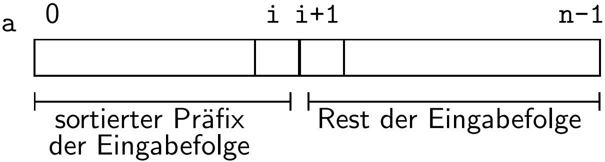
Beispiel:



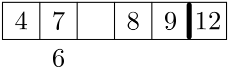
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



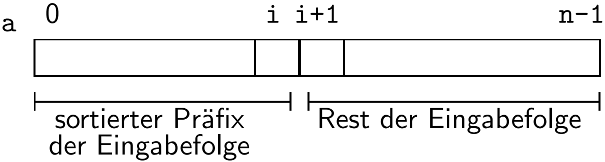
Beispiel:



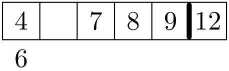
Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index $\mathtt{i}$ mit $0 \leq \mathtt{i} \leq \mathtt{n}-1$, so dass:



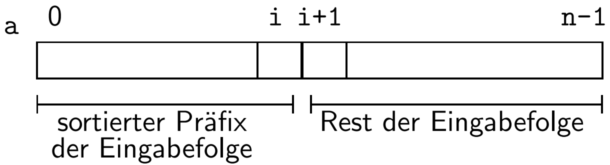
Beispiel:



Insertsort

Idee: „direktes Einfügen“

Invariante: Es gibt jederzeit einen Index `i` mit $0 \leq \texttt{i} \leq \texttt{n}-1$, so dass:



Beispiel:

4	6	7	8	9	12

Insertsort in C

```
int i,j,x;

for (i=0; i<n-1; i++)
  { x=a[i+1];
    j=i;
    while ((j >= 0) && (a[j] > x))
      { a[j+1]=a[j];
        j--;
      }
    a[j+1]=x;
  }
```

Insertsort in C

```
int i,j,x;

for (i=0; i<n-1; i++)
  { x=a[i+1];
    j=i;
    while ((j >= 0) && (a[j] > x))
      { a[j+1]=a[j];
        j--;
      }
    a[j+1]=x;
  }
```

Durchlaut:

7	12	9

Insertsort in C

```
int i,j,x;

for (i=0; i<n-1; i++)
  { x=a[i+1];
    j=i;
    while ((j >= 0) && (a[j] > x))
      { a[j+1]=a[j];
        j--;
      }
    a[j+1]=x;
  }
```

Durchlauf:

7		9

12

Insertsort in C

```
int i,j,x;

for (i=0; i<n-1; i++)
  { x=a[i+1];
    j=i;
    while ((j >= 0) && (a[j] > x))
      { a[j+1]=a[j];
        j--;
      }
    a[j+1]=x;
  }
```

Durchlauf:

7	12	9

Insertsort in C

```
int i,j,x;

for (i=0; i<n-1; i++)
  { x=a[i+1];
    j=i;
    while ((j >= 0) && (a[j] > x))
      { a[j+1]=a[j];
        j--;
      }
    a[j+1]=x;
  }
```

Durchlauf:

7	12	

9

Insertsort in C

```c
int i,j,x;

for (i=0; i<n-1; i++)
  { x=a[i+1];
    j=i;
    while ((j >= 0) && (a[j] > x))
      { a[j+1]=a[j];
        j--;
      }
    a[j+1]=x;
  }
```

Durchlauf:

7		12

9

Insertsort in C

```
int i,j,x;

for (i=0; i<n-1; i++)
  { x=a[i+1];
    j=i;
    while ((j >= 0) && (a[j] > x))
      { a[j+1]=a[j];
        j--;
      }
    a[j+1]=x;
  }
```

Durchlauf:

7	9	12

Insertsort — Komplexität

Best-case:

1	2	3	4	5

Insertsort — Komplexität

Best-case:

1		3	4	5

2

Insertsort — Komplexität

Best-case:

1	2	3	4	5

Insertsort — Komplexität

Best-case:

1	2		4	5
	3			

Insertsort — Komplexität

Best-case:

1	2	3	4	5

Insertsort — Komplexität

Best-case:

1	2	3		5

4

Insertsort — Komplexität

Best-case:

1	2	3	4	5

Insertsort — Komplexität

Best-case:

1	2	3	4	

5

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

5	4	3	2	1

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

5		3	2	1

4

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

4	5	3	2	1

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

4	5		2	1

3

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

4		5	2	1

3

Insertsort — Komplexität

Best-case:

1	2	3	4	5

⇝ linearer Aufwand

Worst-case:

3	4	5	2	1

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\leadsto$ linearer Aufwand

Worst-case:

3	4	5		1

2

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

3	4		5	1

2

Insertsort — Komplexität

Best-case:

1	2	3	4	5

⇝ linearer Aufwand

Worst-case:

3		4	5	1

2

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

2	3	4	5	1

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

2	3	4	5	

1

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

2	3	4		5

1

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

2	3		4	5

1

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

2		3	4	5

1

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\rightsquigarrow$ linearer Aufwand

Worst-case:

1	2	3	4	5

$\rightsquigarrow$ quadratischer Aufwand

Insertsort — Komplexität

Best-case:

1	2	3	4	5

$\leadsto$ linearer Aufwand

Worst-case:

1	2	3	4	5

$\leadsto$ quadratischer Aufwand

Average-case: ebenso quadratischer Aufwand

Mindestaufwand

Frage: Wie viele Operation sind unabhängig vom Algorithmus im Durchschnitt mindestens nötig, um die Reihenfolge von n Zahlen zu bestimmen?

Mindestaufwand

Frage: Wie viele Operation sind unabhängig vom Algorithmus im Durchschnitt mindestens nötig, um die Reihenfolge von n Zahlen zu bestimmen?

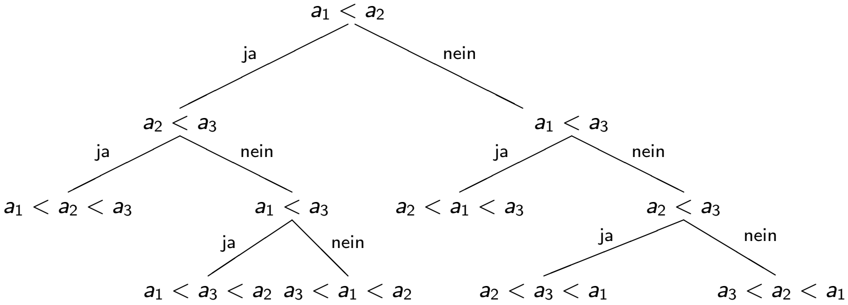
Ansatz: Entscheidungsbaum über notwendige Vergleiche

Mindestaufwand

Frage: Wie viele Operation sind unabhängig vom Algorithmus im Durchschnitt mindestens nötig, um die Reihenfolge von n Zahlen zu bestimmen?

Ansatz: Entscheidungsbaum über notwendige Vergleiche

$n = 3$:

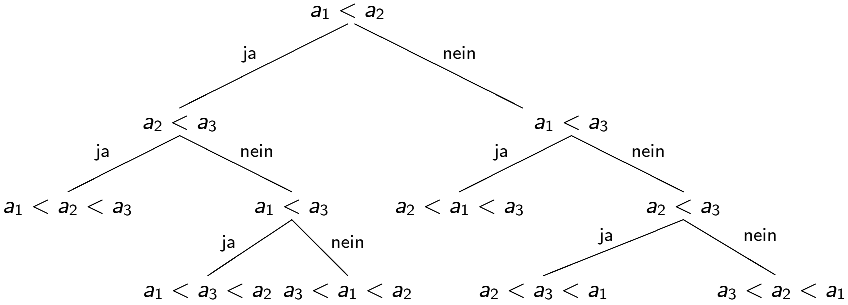


Mindestaufwand

Frage: Wie viele Operation sind unabhängig vom Algorithmus im Durchschnitt mindestens nötig, um die Reihenfolge von n Zahlen zu bestimmen?

Ansatz: Entscheidungsbaum über notwendige Vergleiche

$n = 3$:



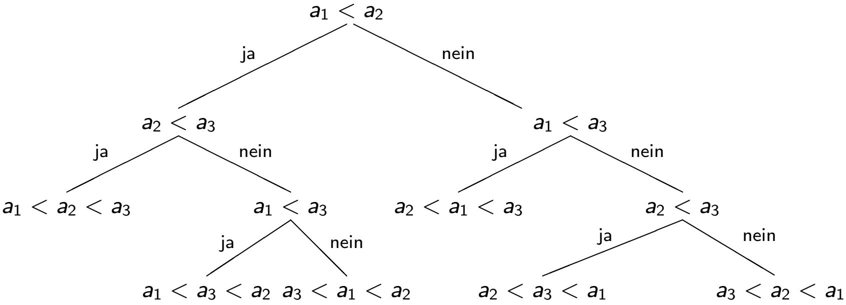
Allgemein: $n!$ mögliche Permutationen/Blätter

Mindestaufwand

Frage: Wie viele Operation sind unabhängig vom Algorithmus im Durchschnitt mindestens nötig, um die Reihenfolge von n Zahlen zu bestimmen?

Ansatz: Entscheidungsbaum über notwendige Vergleiche

$n = 3$:



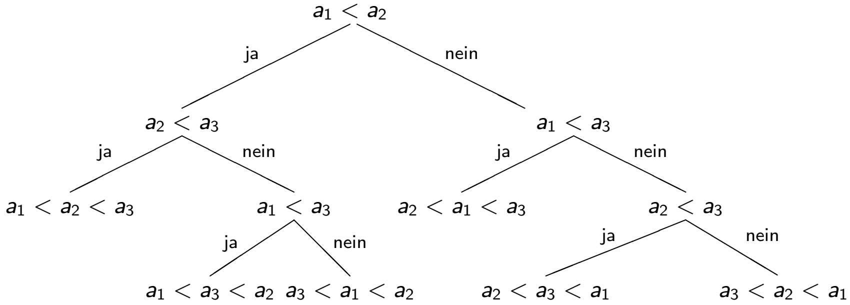
Allgemein: $n!$ mögliche Permutationen/Blätter
$\rightsquigarrow$ „mittlere Höhe“ entspricht $\log(n!)$

Mindestaufwand

Frage: Wie viele Operation sind unabhängig vom Algorithmus im Durchschnitt mindestens nötig, um die Reihenfolge von n Zahlen zu bestimmen?

Ansatz: Entscheidungsbaum über notwendige Vergleiche

$n = 3$:



Allgemein: $n!$ mögliche Permutationen/Blätter
$\rightsquigarrow$ „mittlere Höhe" entspricht $\log(n!)$
$\rightsquigarrow$ dies entspricht $n \cdot \log(n)$

Quicksort

Idee: „divide-and-conquer“ nach Zerlegen des Feldes bezüglich eines Teilungselements

Quicksort

Idee: „divide-and-conquer“ nach Zerlegen des Feldes bezüglich eines Teilungselements

1. Wähle ein Element `x=a[k]`, welches an einer mittleren Indexposition `k` von `a` steht (`x` heißt Pivotelement).

Quicksort

Idee: „divide-and-conquer" nach Zerlegen des Feldes bezüglich eines Teilungselements

1. Wähle ein Element `x=a[k]`, welches an einer mittleren Indexposition `k` von `a` steht (`x` heißt Pivotelement).
2. Suche von Position `i=0` beginnend nach rechts fortschreitend ein Element `a[i]`, welches größer oder gleich `x` ist.

Quicksort

Idee: „divide-and-conquer“ nach Zerlegen des Feldes bezüglich eines Teilungselements

1. Wähle ein Element `x=a[k]`, welches an einer mittleren Indexposition `k` von `a` steht (`x` heißt Pivotelement).
2. Suche von Position `i=0` beginnend nach rechts fortschreitend ein Element `a[i]`, welches größer oder gleich `x` ist.
3. Suche von Position `j=n-1` beginnend nach links fortschreitend ein Element `a[j]`, welches kleiner oder gleich `x` ist.

Quicksort

Idee: „divide-and-conquer“ nach Zerlegen des Feldes bezüglich eines Teilungselements

1. Wähle ein Element `x=a[k]`, welches an einer mittleren Indexposition `k` von `a` steht (`x` heißt Pivotelement).
2. Suche von Position `i=0` beginnend nach rechts fortschreitend ein Element `a[i]`, welches größer oder gleich `x` ist.
3. Suche von Position `j=n-1` beginnend nach links fortschreitend ein Element `a[j]`, welches kleiner oder gleich `x` ist.
4. Vertausche `a[i]` und `a[j]` (sofern nicht `i>j`).

Quicksort

Idee: „divide-and-conquer“ nach Zerlegen des Feldes bezüglich eines Teilungselements

1. Wähle ein Element `x=a[k]`, welches an einer mittleren Indexposition `k` von `a` steht (`x` heißt Pivotelement).
2. Suche von Position `i=0` beginnend nach rechts fortschreitend ein Element `a[i]`, welches größer oder gleich `x` ist.
3. Suche von Position `j=n-1` beginnend nach links fortschreitend ein Element `a[j]`, welches kleiner oder gleich `x` ist.
4. Vertausche `a[i]` und `a[j]` (sofern nicht `i>j`).
5. Wiederhole die Anweisungen 2., 3. und 4. jeweils mit dem um 1 inkrementierten `i` bzw. mit dem um 1 dekrementierten `j` beginnend solange, bis `i>j`.

Quicksort

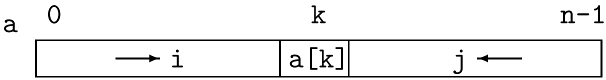
Idee: „divide-and-conquer“ nach Zerlegen des Feldes bezüglich eines Teilungselements

1. Wähle ein Element `x=a[k]`, welches an einer mittleren Indexposition `k` von `a` steht (`x` heißt Pivotelement).
2. Suche von Position `i=0` beginnend nach rechts fortschreitend ein Element `a[i]`, welches größer oder gleich `x` ist.
3. Suche von Position `j=n-1` beginnend nach links fortschreitend ein Element `a[j]`, welches kleiner oder gleich `x` ist.
4. Vertausche `a[i]` und `a[j]` (sofern nicht `i>j`).
5. Wiederhole die Anweisungen 2., 3. und 4. jeweils mit dem um 1 inkrementierten `i` bzw. mit dem um 1 dekrementierten `j` beginnend solange, bis `i>j`.
6. Wende den gesamten Algorithmus auf die (nicht-trivialen) Teilfelder `a[0]...a[j]` und `a[i]...a[n-1]` an.

Quicksort

Idee: „divide-and-conquer“ nach Zerlegen des Feldes bezüglich eines Teilungselements

1. Wähle ein Element `x=a[k]`, welches an einer mittleren Indexposition `k` von `a` steht (`x` heißt Pivotelement).
2. Suche von Position `i=0` beginnend nach rechts fortschreitend ein Element `a[i]`, welches größer oder gleich `x` ist.
3. Suche von Position `j=n-1` beginnend nach links fortschreitend ein Element `a[j]`, welches kleiner oder gleich `x` ist.
4. Vertausche `a[i]` und `a[j]` (sofern nicht `i>j`).
5. Wiederhole die Anweisungen 2., 3. und 4. jeweils mit dem um 1 inkrementierten `i` bzw. mit dem um 1 dekrementierten `j` beginnend solange, bis `i>j`.
6. Wende den gesamten Algorithmus auf die (nicht-trivialen) Teilfelder `a[0]...a[j]` und `a[i]...a[n-1]` an.



Quicksort am Beispiel

2	15	7	9	12	4	11

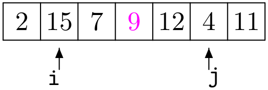
Quicksort am Beispiel

2	15	7	9	12	4	11
↑						↑
i						j

Quicksort am Beispiel

2	15	7	9	12	4	11
	i					j

Quicksort am Beispiel



Quicksort am Beispiel

2	4	7	9	12	15	11
	↑				↑	
	i				j	

Quicksort am Beispiel

2	4	7	9	12	15	11
		i		j		

Quicksort am Beispiel

2	4	7	9	12	15	11
			↑ i	↑ j		

Quicksort am Beispiel

2	4	7	9	12	15	11

i j

Quicksort am Beispiel

2	4	7	9	12	15	11
		j		i		

Quicksort am Beispiel

2	4	7	9	12	15	11

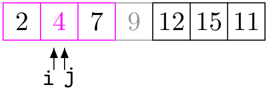
Quicksort am Beispiel

2	4	7	9	12	15	11
↑		↑				
i		j				

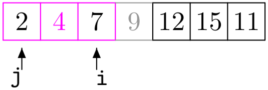
Quicksort am Beispiel

2	4	7	9	12	15	11
	i	j				

Quicksort am Beispiel



Quicksort am Beispiel



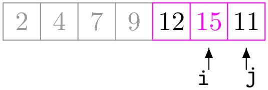
Quicksort am Beispiel

2	4	7	9	12	15	11

Quicksort am Beispiel

2	4	7	9	12	15	11
				i		j

Quicksort am Beispiel



Quicksort am Beispiel

2	4	7	9	12	11	15
					↑ i	↑ j

Quicksort am Beispiel

2	4	7	9	12	11	15

j i

Quicksort am Beispiel

2	4	7	9	12	11	15

Quicksort am Beispiel

2	4	7	9	12	11	15
				↑	↑	
				i	j	

Quicksort am Beispiel

2	4	7	9	11	12	15
				i	j	

Quicksort am Beispiel

2	4	7	9	11	12	15
				j	i	

Quicksort am Beispiel

2	4	7	9	11	12	15

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}
```

7	21	9	5	2	3	14

```
sort(0,n-1);
```

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

7	21	9	5	2	3	14
i						j

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

7	21	9	5	2	3	14
i					j	

Quicksort in C

```c
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

3	21	9	5	2	7	14
↑ i					↑ j	

Quicksort in C

```c
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

3	21	9	5	2	7	14
	i			j		

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

3	2	9	5	21	7	14
	i			j		

Quicksort in C

```c
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

3	2	9	5	21	7	14
		i	j			

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

3	2	5	9	21	7	14
		i	j			

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

3	2	5	9	21	7	14
		j	i			

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}
```

3	2	5	9	21	7	14

```
sort(0,n-1);
```

Quicksort in C

```
void sort(int L,int R)
{ int i,j,k,x,w;
  i=L; j=R; k=(L+R)/2; x=a[k];
  do
    { while (a[i]<x) i++;
      while (a[j]>x) j--;
      if (i<=j)
        { w=a[i]; a[i]=a[j]; a[j]=w;
          i++; j--;
        }
    } while (i<=j);
  if (L<j) sort(L,j);
  if (R>i) sort(i,R);
}

sort(0,n-1);
```

3	2	5	9	21	7	14
i		j				

Quicksort — Komplexität

„Worst-case“:

5	4	3	2	1

Quicksort — Komplexität

„Worst-case“:

5	4	3	2	1
↑				↑
i				j

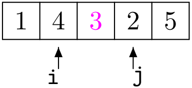
Quicksort — Komplexität

„Worst-case“:

1	4	3	2	5
↑				↑
`i`				`j`

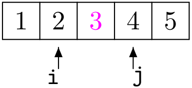
Quicksort — Komplexität

„Worst-case“:



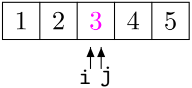
Quicksort — Komplexität

„Worst-case“:



Quicksort — Komplexität

„Worst-case“:



Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5
	j		i	

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	5	3	2

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	5	3	2
i				j

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	5	3	2
	↑ i			↑ j

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	5	3	2
		↑ i		↑ j

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	2	3	5
		i		j

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	2	3	5

i j

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	2	3	5
			↑ j	↑ i

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	2	3	5

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	2	3	5

`i` `j`

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	4	2	3	5
	↑ i		↑ j	

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	3	2	4	5

i j

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	3	2	4	5
		i j		

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	3	2	4	5
		j	i	

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	3	2	4	5

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	3	2	4	5
i		j		

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\leadsto$ Aufwand $n \cdot \log(n)$

Worst-case:

1	3	2	4	5
	$\uparrow$ `i`	$\uparrow$ `j`		

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	2	3	4	5
	i	j		

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	2	3	4	5
	↑	↑		
	j	i		

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	2	3	4	5

$\rightsquigarrow$ Aufwand n^2

Quicksort — Komplexität

„Worst-case“:

1	2	3	4	5

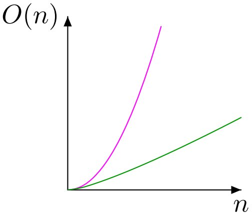
$\rightsquigarrow$ Aufwand $n \cdot \log(n)$

Worst-case:

1	2	3	4	5

$\rightsquigarrow$ Aufwand n^2

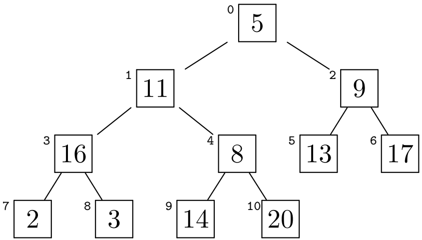
Average-case: Aufwand $n \cdot \log(n)$



Heapsort

Idee: Interpretation des Feldes a als Binärbaum:

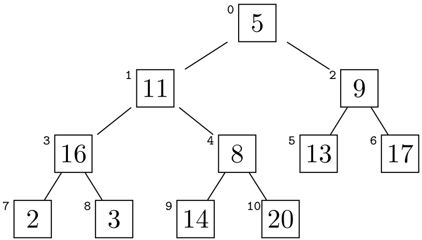
5	11	9	16	8	13	17	2	3	14	20



Heapsort

Idee: Interpretation des Feldes a als Binärbaum:

5	11	9	16	8	13	17	2	3	14	20

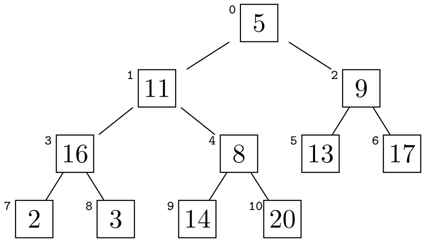


1. Phase: Heap-Eigenschaft herstellen: Nachfolger nie mit größerer Zahl beschriftet als ein Knoten selbst

Heapsort

Idee: Interpretation des Feldes a als Binärbaum:

5	11	9	16	8	13	17	2	3	14	20



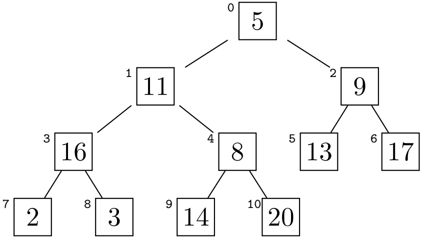
1. Phase: Heap-Eigenschaft herstellen: Nachfolger nie mit größerer Zahl beschriftet als ein Knoten selbst

2. Phase:
   - Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende

Heapsort

Idee: Interpretation des Feldes a als Binärbaum:

5	11	9	16	8	13	17	2	3	14	20



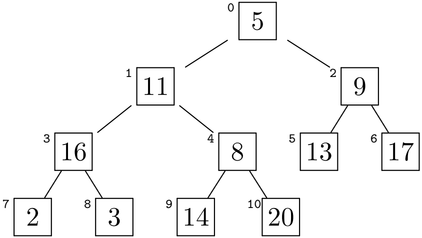
1. Phase: Heap-Eigenschaft herstellen: Nachfolger nie mit größerer Zahl beschriftet als ein Knoten selbst

2. Phase:
- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft

Heapsort

Idee: Interpretation des Feldes a als Binärbaum:

5	11	9	16	8	13	17	2	3	14	20



1. Phase: Heap-Eigenschaft herstellen: Nachfolger nie mit größerer Zahl beschriftet als ein Knoten selbst

2. Phase:
   - Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
   - Wiederherstellen der Heap-Eigenschaft
   - Wiederholung bis gesamtes Feld sortiert

Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten

Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend

Herstellen der Heap-Eigenschaft

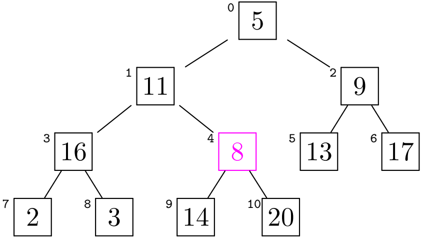
- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

5	11	9	16	8	13	17	2	3	14	20

Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

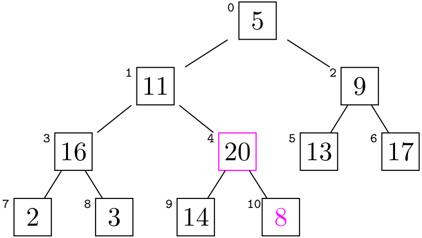
5	11	9	16	8	13	17	2	3	14	20



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

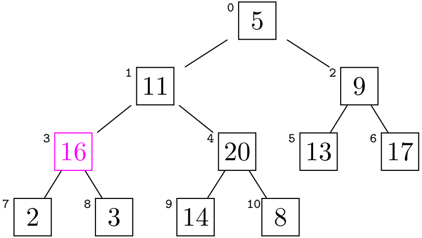
5	11	9	16	20	13	17	2	3	14	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

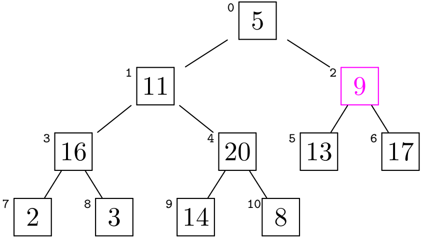
5	11	9	16	20	13	17	2	3	14	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

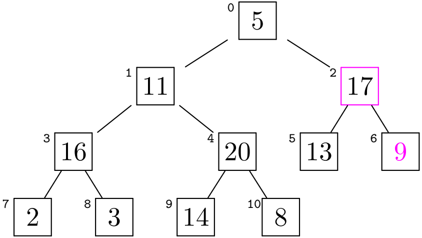
5	11	9	16	20	13	17	2	3	14	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

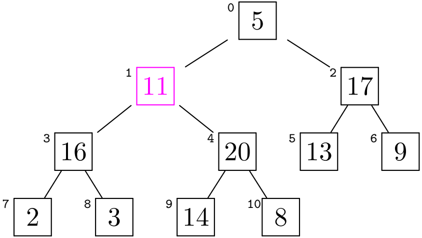
5	11	17	16	20	13	9	2	3	14	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

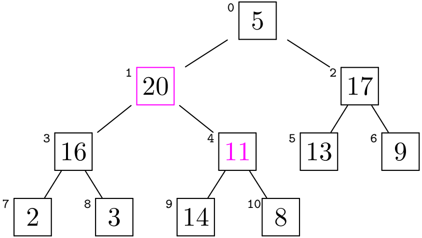
5	11	17	16	20	13	9	2	3	14	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

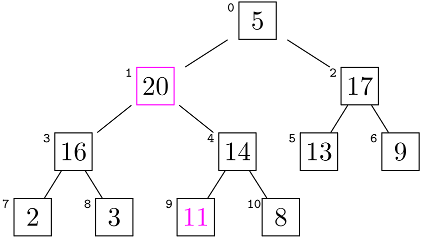
5	20	17	16	11	13	9	2	3	14	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

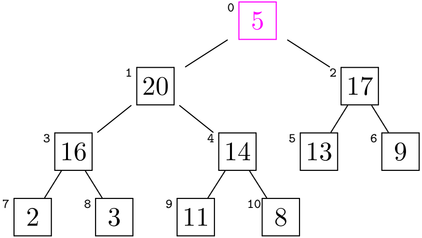
5	20	17	16	14	13	9	2	3	11	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

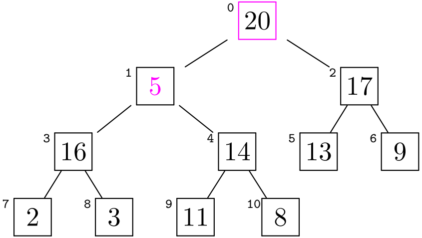
5	20	17	16	14	13	9	2	3	11	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

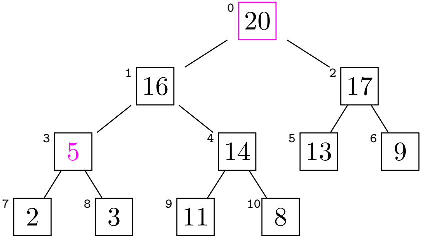
20	5	17	16	14	13	9	2	3	11	8



Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

20	16	17	5	14	13	9	2	3	11	8



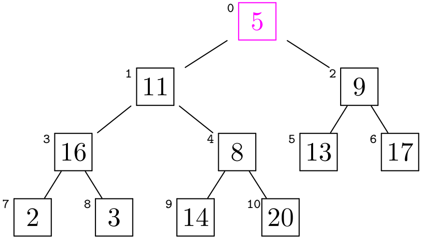
Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!

Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!
- sonst:

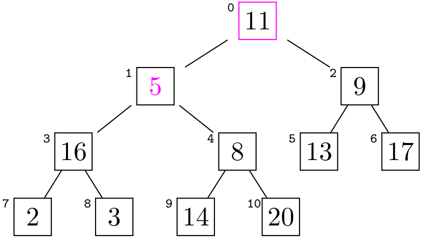
5	11	9	16	8	13	17	2	3	14	20



Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!
- sonst:

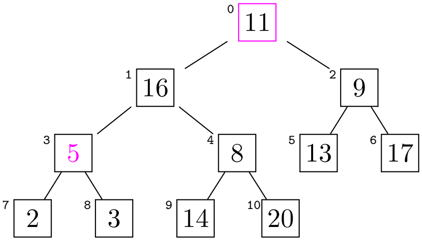
11	5	9	16	8	13	17	2	3	14	20



Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen" nicht an der Wurzel beginnen lassen!
- sonst:

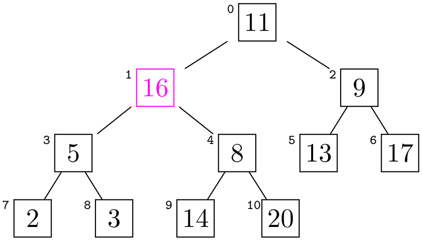
11	16	9	5	8	13	17	2	3	14	20



Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!
- sonst:

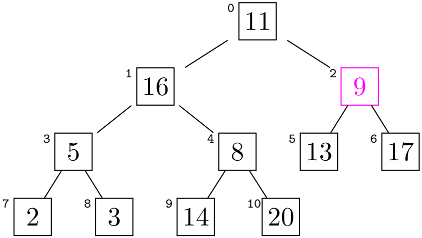
11	16	9	5	8	13	17	2	3	14	20



Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!
- sonst:

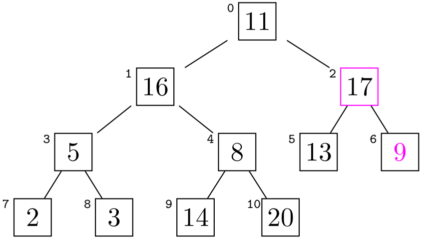
11	16	9	5	8	13	17	2	3	14	20



Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!
- sonst:

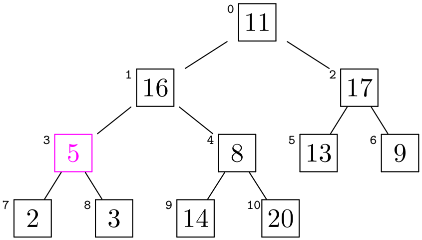
11	16	17	5	8	13	9	2	3	14	20



Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!
- sonst:

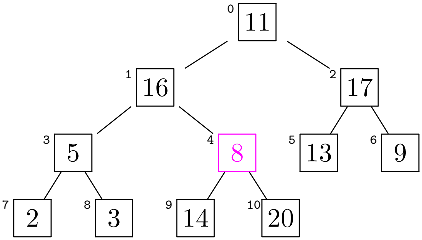
11	16	17	5	8	13	9	2	3	14	20



Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!
- sonst:

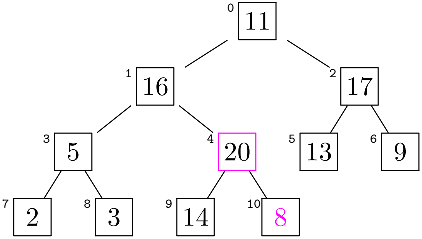
11	16	17	5	8	13	9	2	3	14	20



Herstellen der Heap-Eigenschaft — FALSCH!

- „Sinkenlassen“ nicht an der Wurzel beginnen lassen!
- sonst:

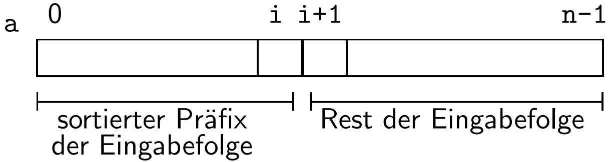
11	16	17	5	20	13	9	2	3	14	8



Wiederholung — Insertsort

Wiederholung — Insertsort

Grundidee:

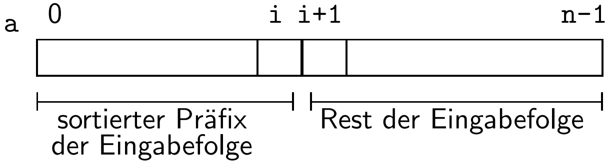


Beispiel:

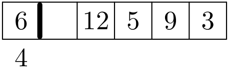
6	4	12	5	9	3

Wiederholung — Insertsort

Grundidee:

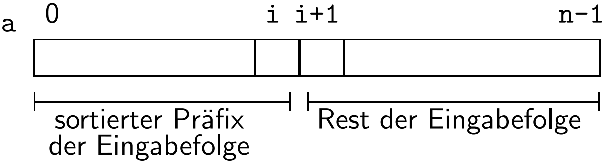


Beispiel:



Wiederholung — Insertsort

Grundidee:

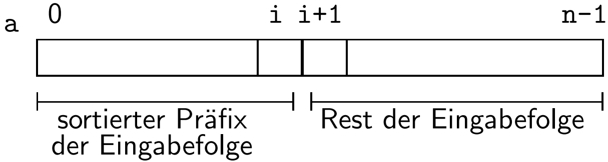


Beispiel:

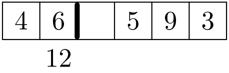
4	6	12	5	9	3

Wiederholung — Insertsort

Grundidee:

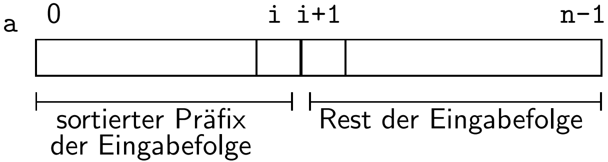


Beispiel:



Wiederholung — Insertsort

Grundidee:

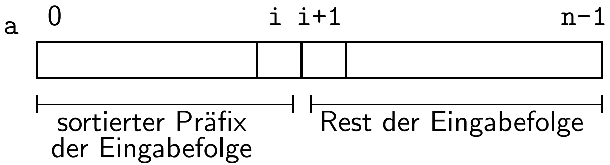


Beispiel:

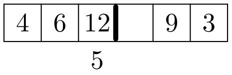
4	6	12	5	9	3

Wiederholung — Insertsort

Grundidee:

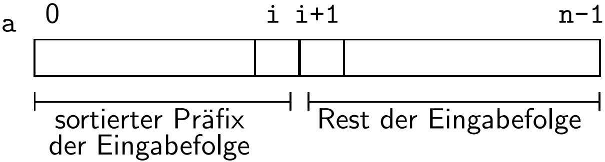


Beispiel:

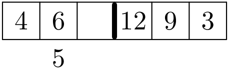


Wiederholung — Insertsort

Grundidee:

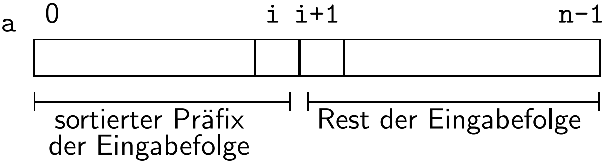


Beispiel:

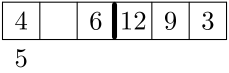


Wiederholung — Insertsort

Grundidee:

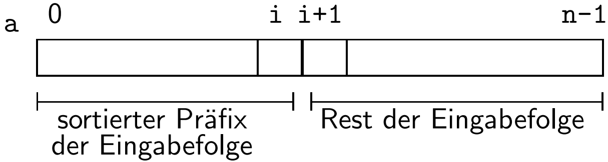


Beispiel:



Wiederholung — Insertsort

Grundidee:

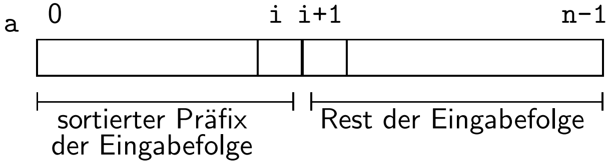


Beispiel:

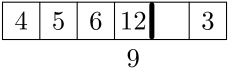
4	5	6	12	9	3

Wiederholung — Insertsort

Grundidee:

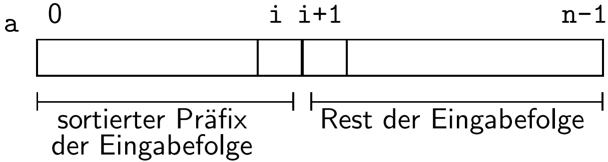


Beispiel:

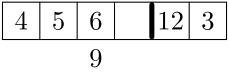


Wiederholung — Insertsort

Grundidee:

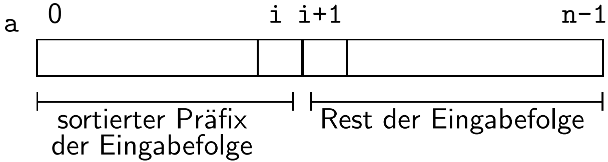


Beispiel:



Wiederholung — Insertsort

Grundidee:

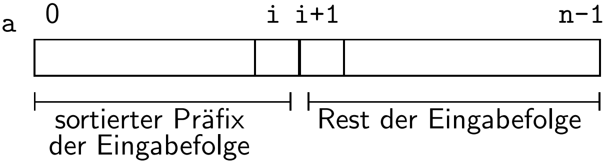


Beispiel:

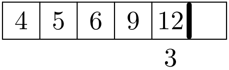
4	5	6	9	12	3

Wiederholung — Insertsort

Grundidee:

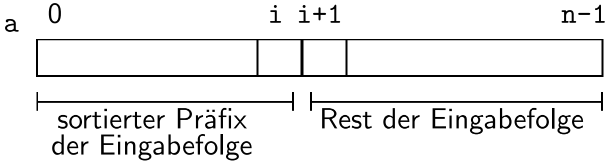


Beispiel:

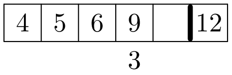


Wiederholung — Insertsort

Grundidee:

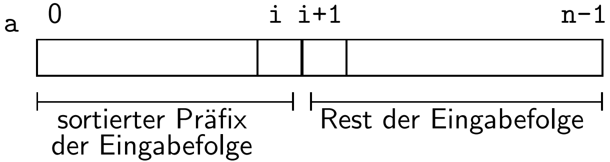


Beispiel:

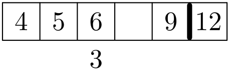


Wiederholung — Insertsort

Grundidee:

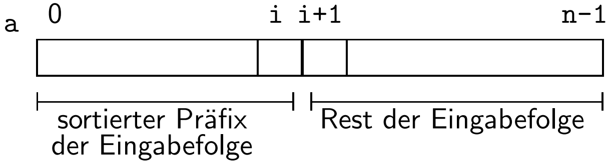


Beispiel:



Wiederholung — Insertsort

Grundidee:



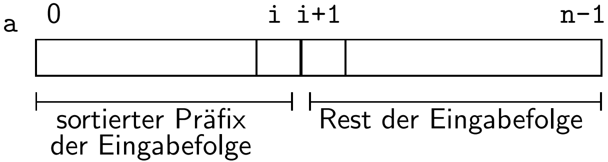
Beispiel:

4	5		6	9	12

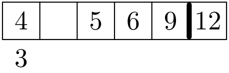
3

Wiederholung — Insertsort

Grundidee:

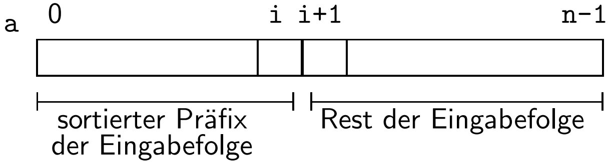


Beispiel:



Wiederholung — Insertsort

Grundidee:

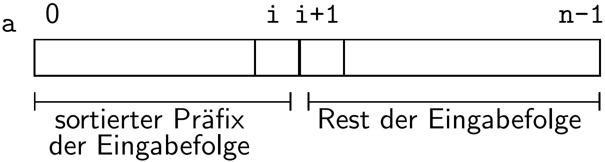


Beispiel:

3	4	5	6	9	12

Wiederholung — Insertsort

Grundidee:



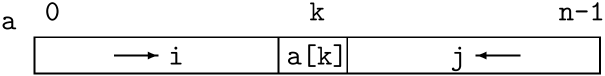
Beispiel:

3	4	5	6	9	12

Aufwand: n^2 in Worst- und Average-case

Wiederholung — Quicksort

Grundidee:

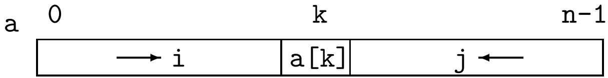


Beispiel:

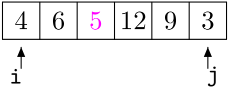
4	6	5	12	9	3

Wiederholung — Quicksort

Grundidee:

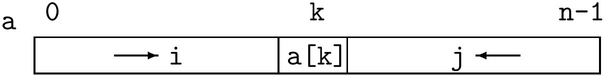


Beispiel:

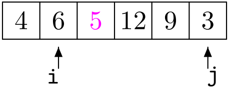


Wiederholung — Quicksort

Grundidee:

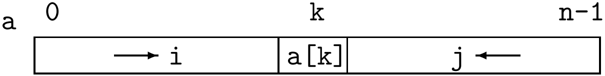


Beispiel:



Wiederholung — Quicksort

Grundidee:

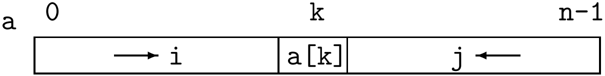


Beispiel:

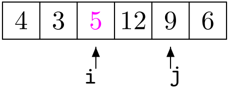
4	3	5	12	9	6

Wiederholung — Quicksort

Grundidee:

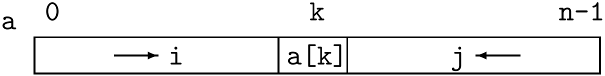


Beispiel:

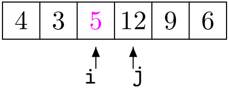


Wiederholung — Quicksort

Grundidee:

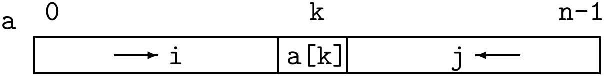


Beispiel:



Wiederholung — Quicksort

Grundidee:

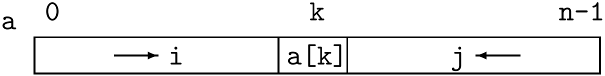


Beispiel:

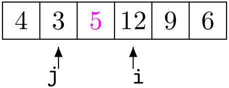
4	3	5	12	9	6

Wiederholung — Quicksort

Grundidee:

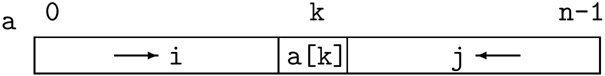


Beispiel:



Wiederholung — Quicksort

Grundidee:

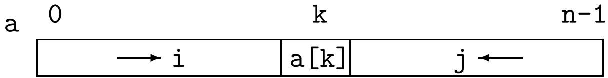


Beispiel:

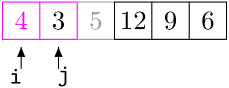
4	3	5	12	9	6

Wiederholung — Quicksort

Grundidee:

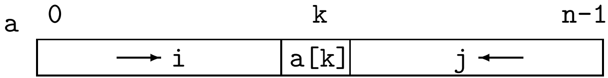


Beispiel:

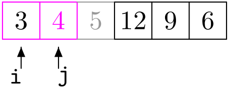


Wiederholung — Quicksort

Grundidee:

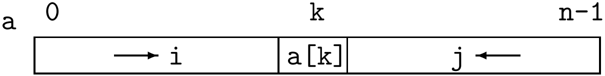


Beispiel:

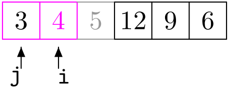


Wiederholung — Quicksort

Grundidee:

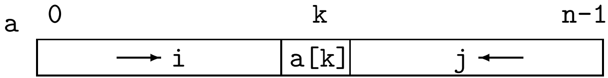


Beispiel:



Wiederholung — Quicksort

Grundidee:

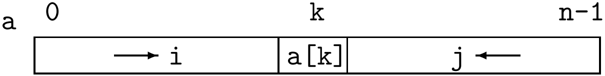


Beispiel:

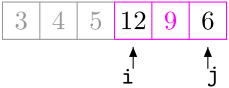
3	4	5	12	9	6

Wiederholung — Quicksort

Grundidee:

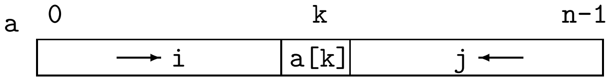


Beispiel:

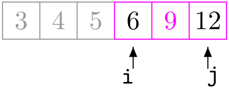


Wiederholung — Quicksort

Grundidee:

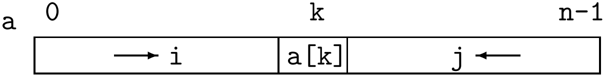


Beispiel:

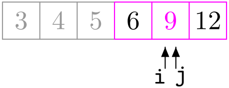


Wiederholung — Quicksort

Grundidee:

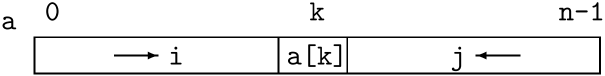


Beispiel:

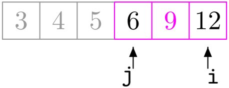


Wiederholung — Quicksort

Grundidee:

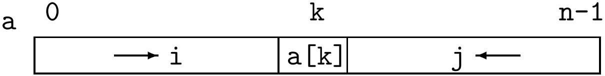


Beispiel:



Wiederholung — Quicksort

Grundidee:

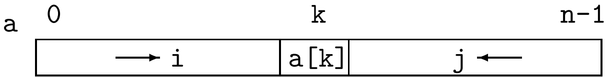


Beispiel:

3	4	5	6	9	12

Wiederholung — Quicksort

Grundidee:



Beispiel:

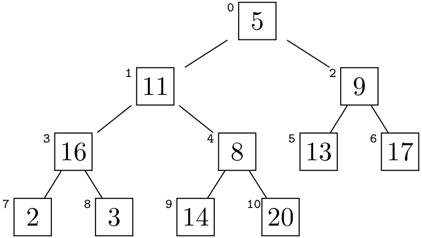
3	4	5	6	9	12

Aufwand: n^2 in Worst- und $n \cdot \log(n)$ in Average-case

Wiederholung — Heapsort

Idee: Interpretation des Feldes a als Binärbaum:

5	11	9	16	8	13	17	2	3	14	20



1. Phase: Heap-Eigenschaft herstellen: Nachfolger nie mit größerer Zahl beschriftet als ein Knoten selbst

2. Phase:
- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

Wiederholung — Herstellen der Heap-Eigenschaft

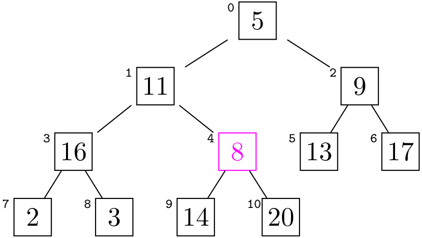
- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

5	11	9	16	8	13	17	2	3	14	20

Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

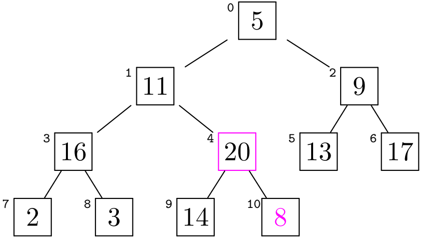
5	11	9	16	8	13	17	2	3	14	20



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

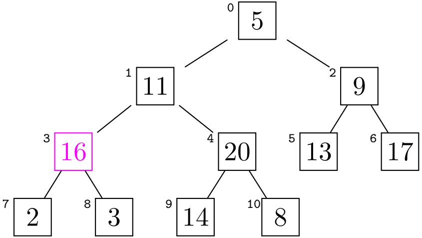
5	11	9	16	20	13	17	2	3	14	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

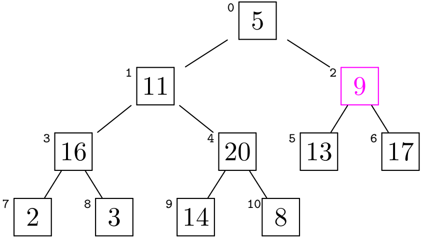
5	11	9	16	20	13	17	2	3	14	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

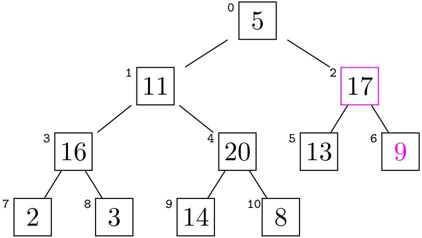
5	11	9	16	20	13	17	2	3	14	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

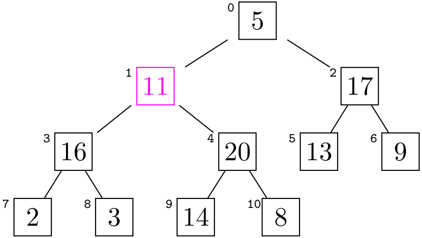
5	11	17	16	20	13	9	2	3	14	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

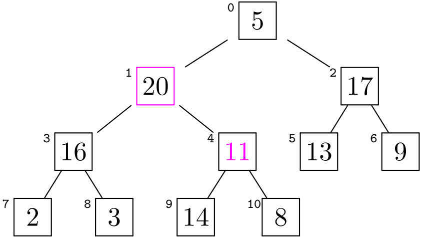
5	11	17	16	20	13	9	2	3	14	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

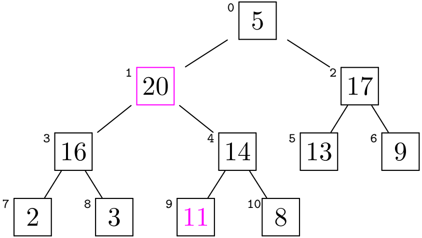
5	20	17	16	11	13	9	2	3	14	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

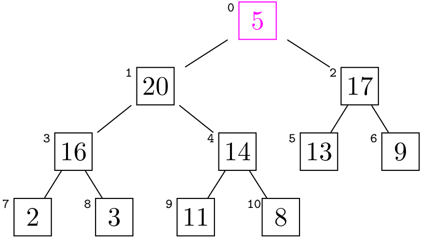
5	20	17	16	14	13	9	2	3	11	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

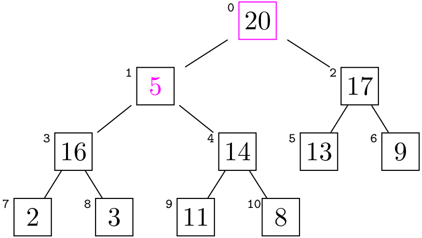
5	20	17	16	14	13	9	2	3	11	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

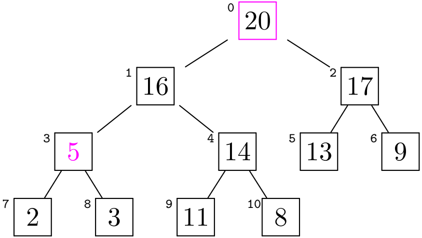
20	5	17	16	14	13	9	2	3	11	8



Wiederholung — Herstellen der Heap-Eigenschaft

- durch „Sinkenlassen“ von Knoten
- am letzten Knoten mit mindestens einem Nachfolger beginnend, zur Wurzel fortschreitend
- am Beispiel:

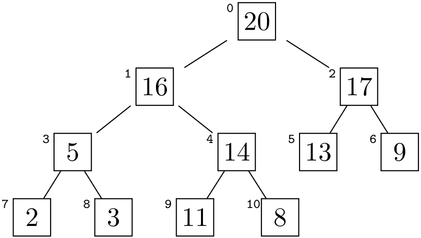
20	16	17	5	14	13	9	2	3	11	8



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

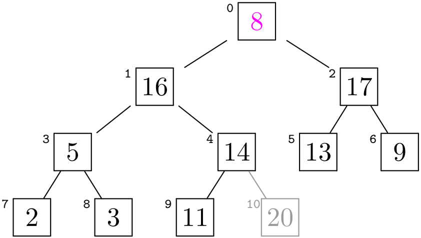
20	16	17	5	14	13	9	2	3	11	8



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

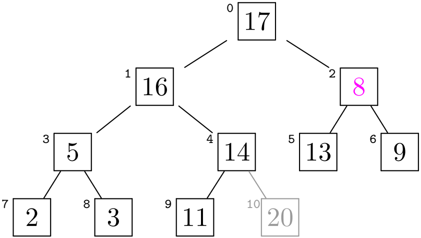
8	16	17	5	14	13	9	2	3	11	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

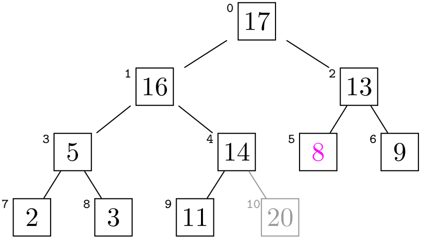
17	16	8	5	14	13	9	2	3	11	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

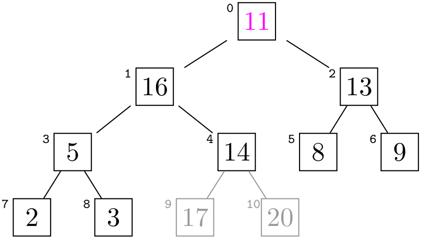
17	16	13	5	14	8	9	2	3	11	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

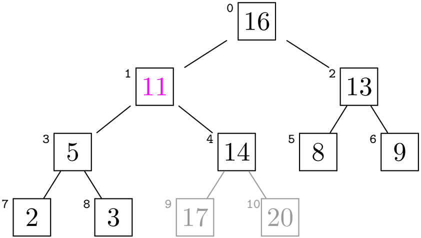
11	16	13	5	14	8	9	2	3	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

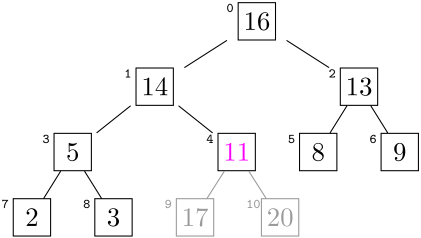
16	11	13	5	14	8	9	2	3	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

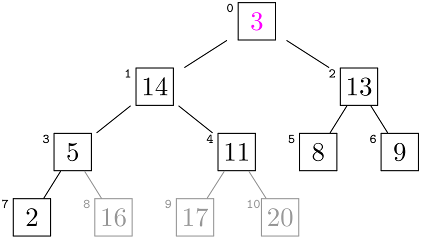
16	14	13	5	11	8	9	2	3	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

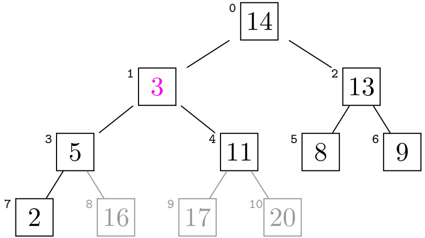
3	14	13	5	11	8	9	2	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

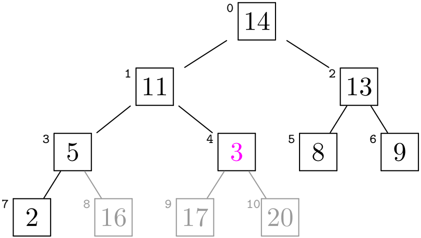
14	3	13	5	11	8	9	2	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

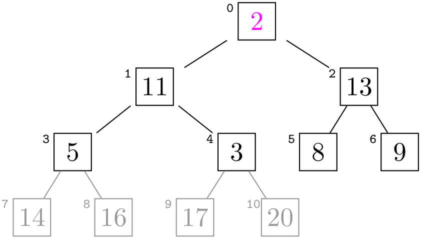
14	11	13	5	3	8	9	2	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

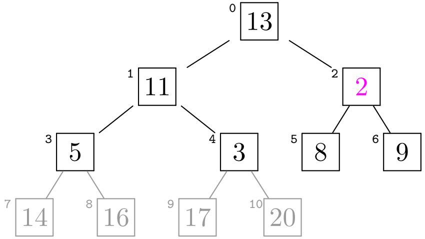
2	11	13	5	3	8	9	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

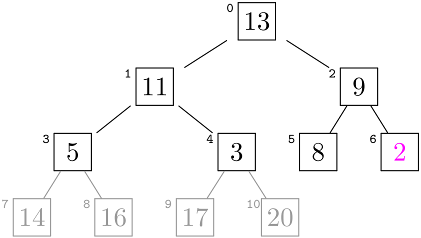
13	11	2	5	3	8	9	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

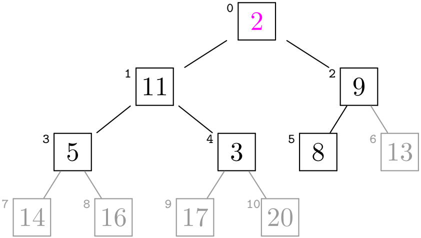
13	11	9	5	3	8	2	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

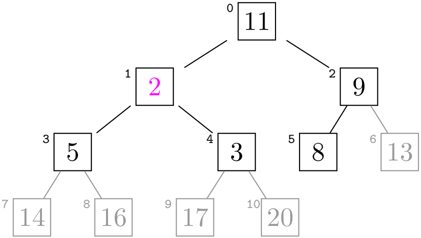
2	11	9	5	3	8	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

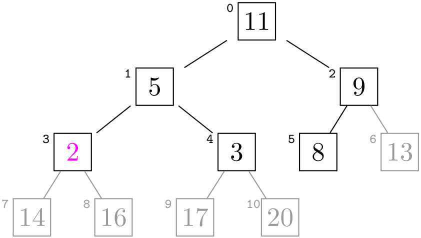
11	2	9	5	3	8	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

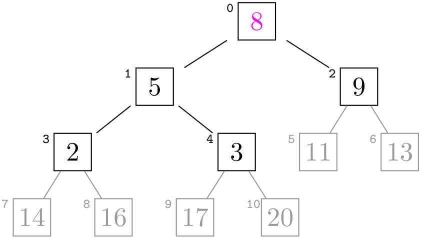
11	5	9	2	3	8	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

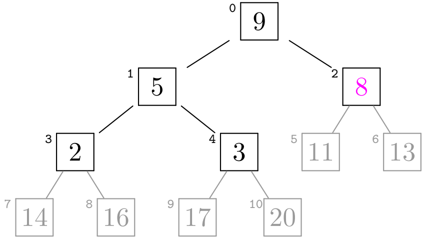
8	5	9	2	3	11	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

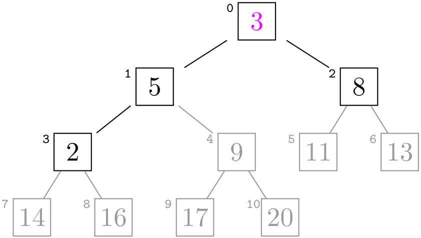
9	5	8	2	3	11	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

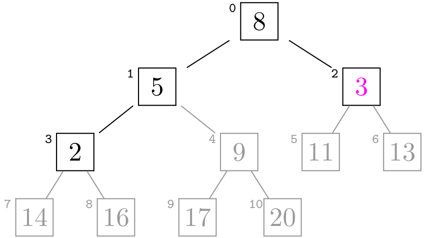
3	5	8	2	9	11	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

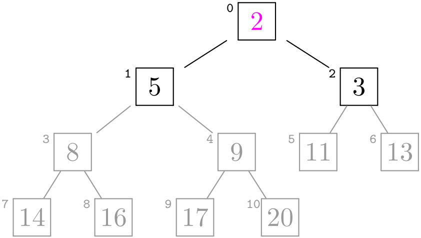
8	5	3	2	9	11	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

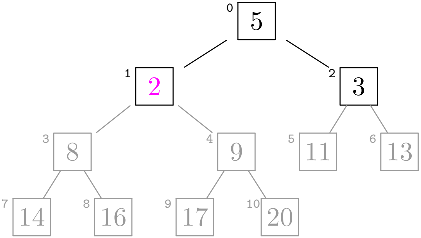
2	5	3	8	9	11	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

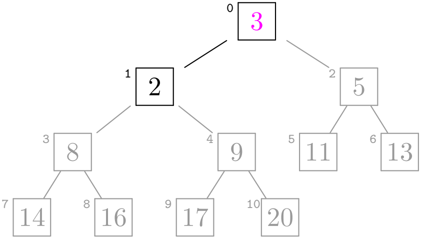
5	2	3	8	9	11	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

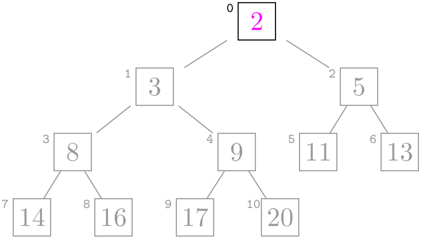
3	2	5	8	9	11	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

2	3	5	8	9	11	13	14	16	17	20



2. Phase: Wiederholtes Abspalten

- Abspalten des jeweils größten Elements nach Tausch von Wurzel ans Feldende
- Wiederherstellen der Heap-Eigenschaft
- Wiederholung bis gesamtes Feld sortiert

2	3	5	8	9	11	13	14	16	17	20

Heapsort in C

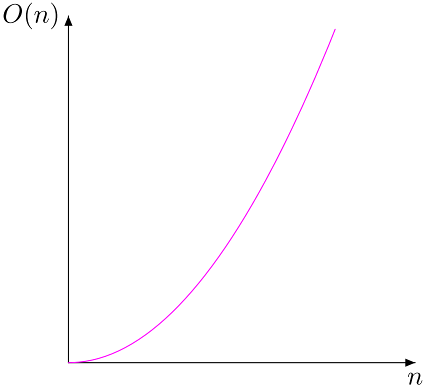
```
void sink(int i,int r)
{ int x,j;
  x=a[i];
  while (1)
    { j=2*i+1;
      if (j>r) break;
      if ((j<r) && (a[j]<a[j+1])) j++;
      if (x>a[j]) break;
      a[i]=a[j]; i=j;
    }
  a[i]=x;
}

int li,re,w;
for (li=n/2-1; li>=0; li--) sink(li,n-1);
for (re=n-1; re>0; re--) { w=a[0]; a[0]=a[re]; a[re]=w;
                           sink(0,re-1);}
```

Heapsort — Komplexität

Worst-case $=$ Average-case: $n \cdot \log(n)$

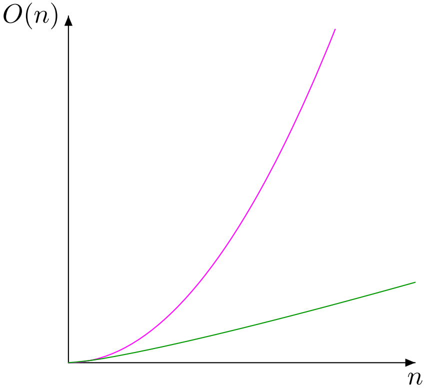
Heapsort — Komplexität

Worst-case = Average-case: $n \cdot \log(n)$



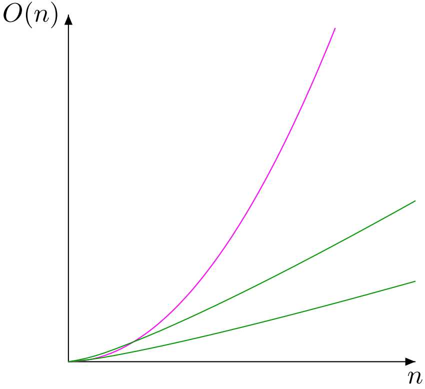
Heapsort — Komplexität

Worst-case = Average-case: $n \cdot \log(n)$



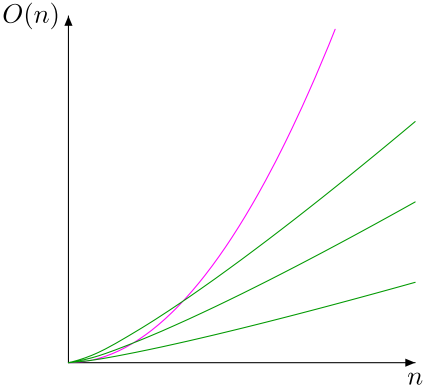
Heapsort — Komplexität

Worst-case = Average-case: $n \cdot \log(n)$



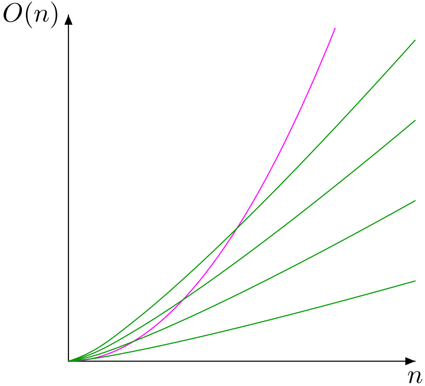
Heapsort — Komplexität

Worst-case $=$ Average-case: $n \cdot \log(n)$



Heapsort — Komplexität

Worst-case = Average-case: $n \cdot \log(n)$



Heapsort — Komplexität

Worst-case = Average-case: $n \cdot \log(n)$