

3 Sortieren

3 Sortieren in Linearzeit

3.1 Quicksort

3.2 Eigenschaften von Sortieralgorithmen

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortieralgorithmen

3.4 Sortieren in Linearzeit

3.1 Quicksort

initialer Aufruf: $\text{QUICKSORT}(a, 0, n - 1)$

```
QUICKSORT(int[] a, int  $\ell$ , int r)
```

```
1  if ( $\ell < r$ ) {  
2      int  $q = \text{PARTITION}(a, \ell, r)$ ;  
3      QUICKSORT( $a, \ell, q - 1$ );  
4      QUICKSORT( $a, q + 1, r$ );  
5  }
```

3.1 Quicksort

initialer Aufruf: $\text{QUICKSORT}(a, 0, n - 1)$

```
QUICKSORT(int[] a, int  $\ell$ , int r)
1  if ( $\ell < r$ ) {
2      int  $q = \text{PARTITION}(a, \ell, r)$ ;
3      QUICKSORT( $a, \ell, q - 1$ );
4      QUICKSORT( $a, q + 1, r$ );
5  }
```

PARTITION permutiert a , sodass gilt:

1. $a[i] \leq a[q]$ für alle $i \in \{\ell, \dots, q - 1\}$,
2. $a[i] \geq a[q]$ für alle $i \in \{q + 1, \dots, r\}$.

3.1 Quicksort

initialer Aufruf: $\text{QUICKSORT}(a, 0, n - 1)$

```
QUICKSORT(int[] a, int  $\ell$ , int r)
```

```
1  if ( $\ell < r$ ) {  
2      int  $q = \text{PARTITION}(a, \ell, r)$ ;  
3      QUICKSORT}(a, \ell, q - 1);  
4      QUICKSORT}(a, q + 1, r);  
5  }
```

PARTITION permutiert a , sodass gilt:

1. $a[i] \leq a[q]$ für alle $i \in \{\ell, \dots, q - 1\}$,
2. $a[i] \geq a[q]$ für alle $i \in \{q + 1, \dots, r\}$.

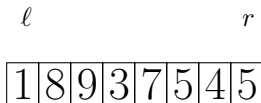
```
PARTITION(int[] a, int  $\ell$ , int r)
```

```
1  int  $x = a[r]$ ;  
2  int  $i = \ell - 1$ ;  
3  for ( $j = \ell; j < r; j++$ ) {  
4      if ( $a[j] \leq x$ ) {  
5           $i++$ ;  
6          vertausche  $a[i]$  und  $a[j]$ ;  
7      }  
8  }  
9  vertausche  $a[i + 1]$  und  $a[r]$ ;  
10 return  $i + 1$ ;
```

3.1 Quicksort

PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```

$x = 5$

ℓ

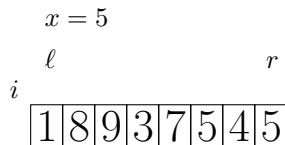
r

1	8	9	3	7	5	4	5
---	---	---	---	---	---	---	---

3.1 Quicksort

PARTITION(int[] a, int ℓ , int r)

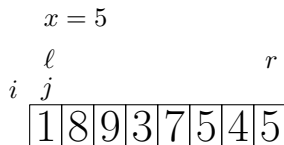
```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

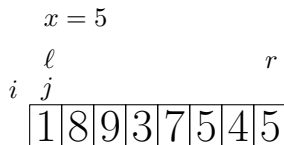
PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

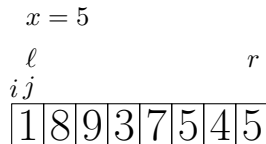
```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1  int  $x = a[r]$ ;
2  int  $i = \ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5          i++;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for (j =  $\ell$ ; j < r; j++) {
4      if (a[j] <= x) {
5          i++;
6          vertausche a[i] und a[j];
7      }
8  }
9  vertausche a[i + 1] und a[r];
10 return i + 1;
```

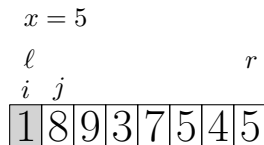
$x = 5$

ℓ r

i	j						
1	8	9	3	7	5	4	5

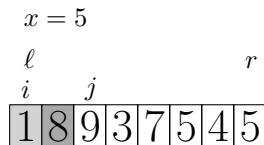
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1  int  $x = a[r]$ ;
2  int  $i = \ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



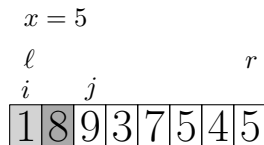
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6           vertausche  $a[i]$  und  $a[j]$ ;
7       }
8   }
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```



3.1 Quicksort

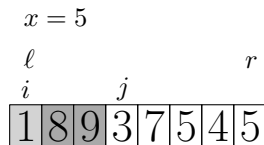
```
PARTITION(int[] a, int  $\ell$ , int  $r$ )  
1   int  $x = a[r]$ ;  
2   int  $i = \ell - 1$ ;  
3   for ( $j = \ell; j < r; j++$ ) {  
4       if ( $a[j] \leq x$ ) {  
5            $i++$ ;  
6           vertausche  $a[i]$  und  $a[j]$ ;  
7       }  
8   }  
9   vertausche  $a[i + 1]$  und  $a[r]$ ;  
10  return  $i + 1$ ;
```



3.1 Quicksort

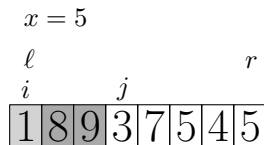
PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5          i++;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



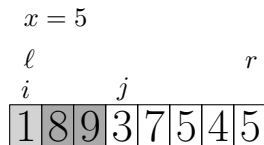
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell; j < r; j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6           vertausche  $a[i]$  und  $a[j]$ ;
7       }
8   }
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```



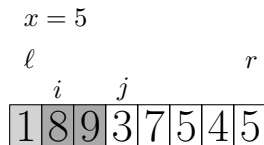
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1  int  $x = a[r]$ ;
2  int  $i = \ell - 1$ ;
3  for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



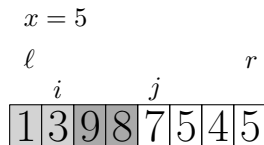
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1  int  $x = a[r]$ ;
2  int  $i = \ell - 1$ ;
3  for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



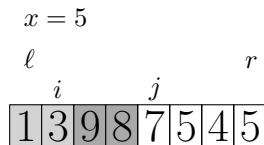
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1  int  $x = a[r]$ ;
2  int  $i = \ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

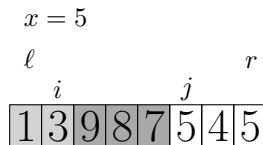
```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell; j < r; j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6           vertausche  $a[i]$  und  $a[j]$ ;
7       }
8   }
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```



3.1 Quicksort

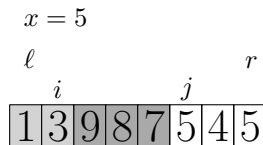
PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5          i++;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



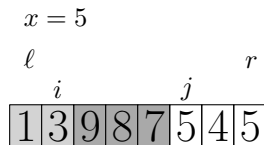
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell; j < r; j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6           vertausche  $a[i]$  und  $a[j]$ ;
7       }
8   }
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```



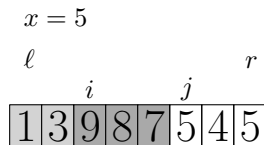
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1  int  $x = a[r]$ ;
2  int  $i = \ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

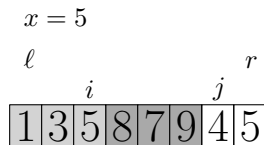
```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell; j < r; j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6       }
7   }
8   vertausche  $a[i]$  und  $a[j]$ ;
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```



3.1 Quicksort

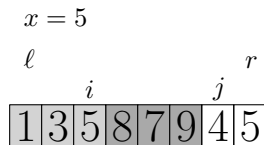
PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5          i++;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

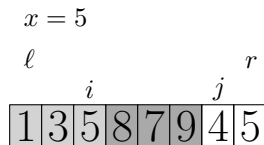
```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1  int  $x = a[r]$ ;
2  int  $i = \ell - 1$ ;
3  for ( $j = \ell; j < r; j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

PARTITION(int[] a, int ℓ , int r)

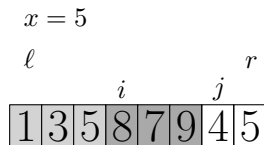
```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4      if ( $a[j] \leq x$ ) {
5          i++;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



3.1 Quicksort

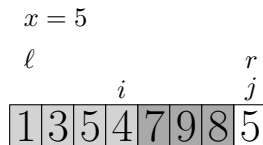
PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```



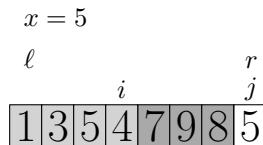
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )  
1   int  $x = a[r]$ ;  
2   int  $i = \ell - 1$ ;  
3   for ( $j = \ell; j < r; j++$ ) {  
4       if ( $a[j] \leq x$ ) {  
5            $i++$ ;  
6           vertausche  $a[i]$  und  $a[j]$ ;  
7       }  
8   }  
9   vertausche  $a[i + 1]$  und  $a[r]$ ;  
10  return  $i + 1$ ;
```



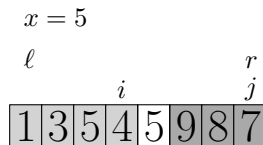
3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )  
1   int  $x = a[r]$ ;  
2   int  $i = \ell - 1$ ;  
3   for ( $j = \ell; j < r; j++$ ) {  
4       if ( $a[j] \leq x$ ) {  
5            $i++$ ;  
6           vertausche  $a[i]$  und  $a[j]$ ;  
7       }  
8   }  
9   vertausche  $a[i + 1]$  und  $a[r]$ ;  
10  return  $i + 1$ ;
```



3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1  int  $x = a[r]$ ;
2  int  $i = \ell - 1$ ;
3  for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```

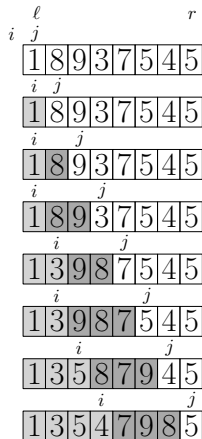
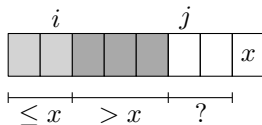


3.1 Quicksort

Lemma 3.1

In Zeile 3 der Methode PARTITION gelten für das aktuelle j und das aktuelle i stets die folgenden Aussagen.

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i+1, \dots, j-1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

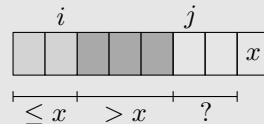


3.1 Quicksort

Lemma 3.1

In Zeile 3 der Methode PARTITION gelten für das aktuelle j und das aktuelle i stets die folgenden Aussagen.

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i+1, \dots, j-1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

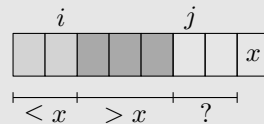


3.1 Quicksort

Lemma 3.1

In Zeile 3 der Methode PARTITION gelten für das aktuelle j und das aktuelle i stets die folgenden Aussagen.

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i+1, \dots, j-1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.



Beweis:

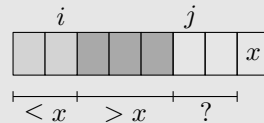
Induktionsanfang: $i = \ell - 1, j = \ell$

3.1 Quicksort

Lemma 3.1

In Zeile 3 der Methode PARTITION gelten für das aktuelle j und das aktuelle i stets die folgenden Aussagen.

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i+1, \dots, j-1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.



Beweis:

Induktionsanfang: $i = \ell - 1, j = \ell$

1. Für alle $k \in \{\ell, \dots, \ell - 1\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{\ell, \dots, \ell - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell; j < r; j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6           vertausche  $a[i]$  und  $a[j]$ ;
7       }
8   }
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```

Induktionsschritt: Zu Beginn von Zeile 3:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i + 1, \dots, j - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell; j < r; j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6           vertausche  $a[i]$  und  $a[j]$ ;
7       }
8   }
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```

Induktionsschritt: Zu Beginn von Zeile 3:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
 2. Für alle $k \in \{i + 1, \dots, j - 1\}$ gilt $a[k] > x$.
 3. Es gilt $a[r] = x$.
1. Fall: $a[j] > x$.

3.1 Quicksort

PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```

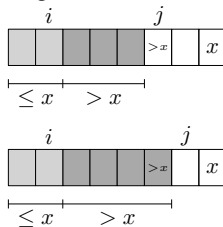
Induktionsschritt: Zu Beginn von Zeile 3:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i + 1, \dots, j - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

1. Fall: $a[j] > x$.

a und i werden nicht verändert.

j wird um eins vergrößert.



3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell; j < r; j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6           vertausche  $a[i]$  und  $a[j]$ ;
7       }
8   }
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```

Induktionsschritt: Zu Beginn von Zeile 3:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i + 1, \dots, j - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

3.1 Quicksort

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
1   int  $x = a[r]$ ;
2   int  $i = \ell - 1$ ;
3   for ( $j = \ell; j < r; j++$ ) {
4       if ( $a[j] \leq x$ ) {
5            $i++$ ;
6           vertausche  $a[i]$  und  $a[j]$ ;
7       }
8   }
9   vertausche  $a[i + 1]$  und  $a[r]$ ;
10  return  $i + 1$ ;
```

Induktionsschritt: Zu Beginn von Zeile 3:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
 2. Für alle $k \in \{i + 1, \dots, j - 1\}$ gilt $a[k] > x$.
 3. Es gilt $a[r] = x$.
2. Fall: $a[j] \leq x$.

3.1 Quicksort

PARTITION(int[] a, int ℓ , int r)

```
1  int x = a[r];
2  int i =  $\ell - 1$ ;
3  for ( $j = \ell$ ;  $j < r$ ;  $j++$ ) {
4      if ( $a[j] \leq x$ ) {
5           $i++$ ;
6          vertausche  $a[i]$  und  $a[j]$ ;
7      }
8  }
9  vertausche  $a[i + 1]$  und  $a[r]$ ;
10 return  $i + 1$ ;
```

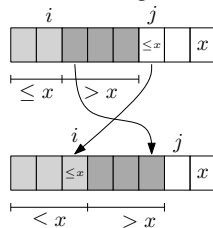
Induktionsschritt: Zu Beginn von Zeile 3:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i + 1, \dots, j - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

2. Fall: $a[j] \leq x$.

$a[i]$ und $a[j]$ werden vertauscht.

i und j werden um eins vergrößert.

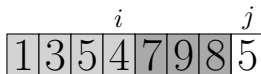


3.1 Quicksort

Am Ende gilt $j = r$.

Dafür besagt Lemma 3.1:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i + 1, \dots, r - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.



1	3	5	4	7	9	8	5
---	---	---	---	---	---	---	---

3.1 Quicksort

Am Ende gilt $j = r$.

Dafür besagt Lemma 3.1:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i + 1, \dots, r - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

				i			j
1	3	5	4	7	9	8	5

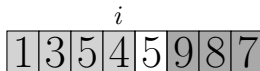
In Zeile 9 werden dann $a[i + 1]$ und $a[r]$ vertauscht.

3.1 Quicksort

Am Ende gilt $j = r$.

Dafür besagt Lemma 3.1:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i + 1, \dots, r - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.



i							
1	3	5	4	5	9	8	7

In Zeile 9 werden dann $a[i + 1]$ und $a[r]$ vertauscht.

3.1 Quicksort

Am Ende gilt $j = r$.

Dafür besagt Lemma 3.1:

1. Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.
2. Für alle $k \in \{i + 1, \dots, r - 1\}$ gilt $a[k] > x$.
3. Es gilt $a[r] = x$.

i							
1	3	5	4	5	9	8	7

In Zeile 9 werden dann $a[i + 1]$ und $a[r]$ vertauscht.

\Rightarrow PARTITION arbeitet korrekt.

3.1 Quicksort

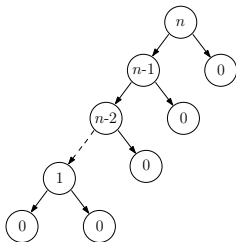
Wie groß ist die Laufzeit von QUICKSORT?

Stets das größte Element wird Pivot.

3.1 Quicksort

Wie groß ist die Laufzeit von QUICKSORT?

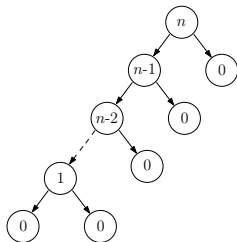
Stets das größte Element wird Pivot.



3.1 Quicksort

Wie groß ist die Laufzeit von QUICKSORT?

Stets das größte Element wird Pivot.

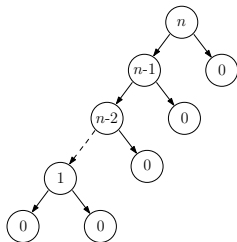


$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 0, \\ T(n-1) + cn & \text{falls } n > 0. \end{cases}$$

3.1 Quicksort

Wie groß ist die Laufzeit von QUICKSORT?

Stets das größte Element wird Pivot.



$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 0, \\ T(n-1) + cn & \text{falls } n > 0. \end{cases}$$

$$\Rightarrow T(n) = \Theta(n^2).$$

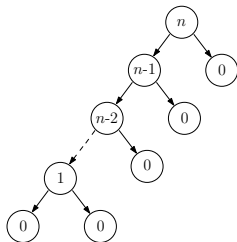
Dies ist der Worst Case.

3.1 Quicksort

Wie groß ist die Laufzeit von QUICKSORT?

Stets das größte Element wird Pivot.

Stets der Median wird Pivot.



$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 0, \\ T(n-1) + cn & \text{falls } n > 0. \end{cases}$$

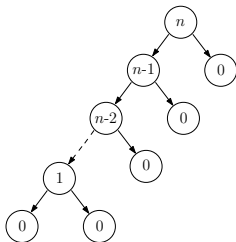
$$\Rightarrow T(n) = \Theta(n^2).$$

Dies ist der Worst Case.

3.1 Quicksort

Wie groß ist die Laufzeit von QUICKSORT?

Stets das größte Element wird Pivot.

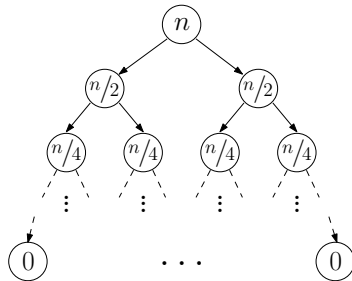


$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 0, \\ T(n-1) + cn & \text{falls } n > 0. \end{cases}$$

$$\Rightarrow T(n) = \Theta(n^2).$$

Dies ist der Worst Case.

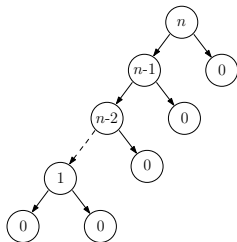
Stets der Median wird Pivot.



3.1 Quicksort

Wie groß ist die Laufzeit von QUICKSORT?

Stets das größte Element wird Pivot.

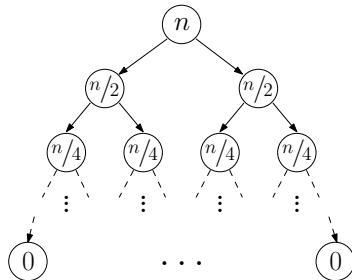


$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 0, \\ T(n-1) + cn & \text{falls } n > 0. \end{cases}$$

$$\Rightarrow T(n) = \Theta(n^2).$$

Dies ist der Worst Case.

Stets der Median wird Pivot.



Mit Variante von Theorem 2.2 folgt

$$T(n) = \Theta(n \log n).$$

Dies ist der Best Case.

3.1 Quicksort

RANDOMQUICKSORT: Wähle stets ein **uniform zufälliges Element** als Pivotelement.

3.1 Quicksort

RANDOMQUICKSORT: Wähle stets ein **uniform zufälliges Element** als Pivotelement.

Kann einfach erreicht werden, indem zu Beginn von PARTITION $a[r]$ mit $a[k]$ für ein uniform zufälliges $k \in \{\ell, \dots, r\}$ vertauscht wird.

3.1 Quicksort

RANDOMQUICKSORT: Wähle stets ein **uniform zufälliges Element** als Pivotelement.

Kann einfach erreicht werden, indem zu Beginn von PARTITION $a[r]$ mit $a[k]$ für ein uniform zufälliges $k \in \{\ell, \dots, r\}$ vertauscht wird.

Für eine feste Eingabe ist die Laufzeit von RANDOMQUICKSORT nicht fest, sondern eine **Zufallsvariable**.

3.1 Quicksort

RANDOMQUICKSORT: Wähle stets ein **uniform zufälliges Element** als Pivotelement.

Kann einfach erreicht werden, indem zu Beginn von PARTITION $a[r]$ mit $a[k]$ für ein uniform zufälliges $k \in \{\ell, \dots, r\}$ vertauscht wird.

Für eine feste Eingabe ist die Laufzeit von RANDOMQUICKSORT nicht fest, sondern eine **Zufallsvariable**.

Uns interessiert die **erwartete Laufzeit**.

3.1 Quicksort

Exkurs: Wahrscheinlichkeitsrechnung

Für ein Ereignis A bezeichnen wir mit $\Pr[A]$ die **Wahrscheinlichkeit, dass es eintritt**.

3.1 Quicksort

Exkurs: Wahrscheinlichkeitsrechnung

Für ein Ereignis A bezeichnen wir mit $\Pr[A]$ die **Wahrscheinlichkeit, dass es eintritt**.

Interpretation: Wenn das Zufallsexperiment sehr oft wiederholt wird, dann ist $\Pr[A]$ die **relative Häufigkeit** von A .

3.1 Quicksort

Exkurs: Wahrscheinlichkeitsrechnung

Für ein Ereignis A bezeichnen wir mit $\Pr[A]$ die **Wahrscheinlichkeit, dass es eintritt**.

Interpretation: Wenn das Zufallsexperiment sehr oft wiederholt wird, dann ist $\Pr[A]$ die **relative Häufigkeit** von A .

Beispiele:

- Bezeichne X die Augenzahl beim Wurf eines fairen Würfels. Es gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, 2, 3, 4, 5, 6\}$.

3.1 Quicksort

Exkurs: Wahrscheinlichkeitsrechnung

Für ein Ereignis A bezeichnen wir mit $\Pr[A]$ die **Wahrscheinlichkeit, dass es eintritt**.

Interpretation: Wenn das Zufallsexperiment sehr oft wiederholt wird, dann ist $\Pr[A]$ die **relative Häufigkeit** von A .

Beispiele:

- Bezeichne X die Augenzahl beim Wurf eines fairen Würfels. Es gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, 2, 3, 4, 5, 6\}$.
- Wird Index k des Pivotelementes uniform zufällig aus der Menge $\{\ell, \dots, r\}$ gewählt, so gilt $\Pr[k = i] = 1/(r - \ell + 1)$ für alle $i \in \{\ell, \dots, r\}$.

3.1 Quicksort

Exkurs: Wahrscheinlichkeitsrechnung

Der **Erwartungswert** $E[X]$ einer Zufallsvariable X ist der Wert, den die Zufallsvariable **im Durchschnitt** annimmt, wenn man das Zufallsexperiment unendlich oft wiederholt.

3.1 Quicksort

Exkurs: Wahrscheinlichkeitsrechnung

Der **Erwartungswert** $\mathbf{E}[X]$ einer Zufallsvariable X ist der Wert, den die Zufallsvariable **im Durchschnitt** annimmt, wenn man das Zufallsexperiment unendlich oft wiederholt.

Formal: $\mathbf{E}[X]$ ist Summe über alle möglichen Werte von X , wobei jeder Wert mit der Wahrscheinlichkeit gewichtet ist, mit der er auftritt.

3.1 Quicksort

Exkurs: Wahrscheinlichkeitsrechnung

Der **Erwartungswert** $\mathbf{E}[X]$ einer Zufallsvariable X ist der Wert, den die Zufallsvariable **im Durchschnitt** annimmt, wenn man das Zufallsexperiment unendlich oft wiederholt.

Formal: $\mathbf{E}[X]$ ist Summe über alle möglichen Werte von X , wobei jeder Wert mit der Wahrscheinlichkeit gewichtet ist, mit der er auftritt.

Für eine Zufallsvariable X , die nur Werte aus \mathbb{Z} annimmt, gilt dementsprechend

$$\mathbf{E}[X] = \sum_{i=-\infty}^{\infty} i \cdot \mathbf{Pr}[X = i].$$

3.1 Quicksort

Exkurs: Wahrscheinlichkeitsrechnung

Der **Erwartungswert** $\mathbf{E}[X]$ einer Zufallsvariable X ist der Wert, den die Zufallsvariable **im Durchschnitt** annimmt, wenn man das Zufallsexperiment unendlich oft wiederholt.

Formal: $\mathbf{E}[X]$ ist Summe über alle möglichen Werte von X , wobei jeder Wert mit der Wahrscheinlichkeit gewichtet ist, mit der er auftritt.

Für eine Zufallsvariable X , die nur Werte aus \mathbb{Z} annimmt, gilt dementsprechend

$$\mathbf{E}[X] = \sum_{i=-\infty}^{\infty} i \cdot \mathbf{Pr}[X = i].$$

Nimmt X nur Werte aus \mathbb{N}_0 an, so gilt außerdem

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i].$$

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt.

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt. Dann gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, \dots, 6\}$ und $\Pr[X = i] = 0$ für jedes $i \notin \{1, \dots, 6\}$.

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt. Dann gilt $\mathbf{Pr}[X = i] = 1/6$ für jedes $i \in \{1, \dots, 6\}$ und $\mathbf{Pr}[X = i] = 0$ für jedes $i \notin \{1, \dots, 6\}$. Damit gilt für den Erwartungswert von X

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \mathbf{Pr}[X = i] = \sum_{i=1}^6 i \cdot \mathbf{Pr}[X = i] = \sum_{i=1}^6 \frac{i}{6} = 3,5.$$

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt. Dann gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, \dots, 6\}$ und $\Pr[X = i] = 0$ für jedes $i \notin \{1, \dots, 6\}$. Damit gilt für den Erwartungswert von X

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \Pr[X = i] = \sum_{i=1}^6 i \cdot \Pr[X = i] = \sum_{i=1}^6 \frac{i}{6} = 3,5.$$

- Bezeichne nun X die Zufallsvariable, die angibt, **wie oft man einen fairen Würfel werfen muss, bis er das erste Mal eine Sechs zeigt**.

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt. Dann gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, \dots, 6\}$ und $\Pr[X = i] = 0$ für jedes $i \notin \{1, \dots, 6\}$. Damit gilt für den Erwartungswert von X

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \Pr[X = i] = \sum_{i=1}^6 i \cdot \Pr[X = i] = \sum_{i=1}^6 \frac{i}{6} = 3,5.$$

- Bezeichne nun X die Zufallsvariable, die angibt, **wie oft man einen fairen Würfel werfen muss, bis er das erste Mal eine Sechs zeigt**.

Es gilt $\Pr[X \geq 1] = 1$,

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt. Dann gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, \dots, 6\}$ und $\Pr[X = i] = 0$ für jedes $i \notin \{1, \dots, 6\}$. Damit gilt für den Erwartungswert von X

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \Pr[X = i] = \sum_{i=1}^6 i \cdot \Pr[X = i] = \sum_{i=1}^6 \frac{i}{6} = 3,5.$$

- Bezeichne nun X die Zufallsvariable, die angibt, **wie oft man einen fairen Würfel werfen muss, bis er das erste Mal eine Sechs zeigt**.

Es gilt $\Pr[X \geq 1] = 1$, $\Pr[X \geq 2] = 5/6$,

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt. Dann gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, \dots, 6\}$ und $\Pr[X = i] = 0$ für jedes $i \notin \{1, \dots, 6\}$. Damit gilt für den Erwartungswert von X

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \Pr[X = i] = \sum_{i=1}^6 i \cdot \Pr[X = i] = \sum_{i=1}^6 \frac{i}{6} = 3,5.$$

- Bezeichne nun X die Zufallsvariable, die angibt, **wie oft man einen fairen Würfel werfen muss, bis er das erste Mal eine Sechs zeigt**.

Es gilt $\Pr[X \geq 1] = 1$, $\Pr[X \geq 2] = 5/6$, $\Pr[X \geq 3] = (5/6)^2$

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt. Dann gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, \dots, 6\}$ und $\Pr[X = i] = 0$ für jedes $i \notin \{1, \dots, 6\}$. Damit gilt für den Erwartungswert von X

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \Pr[X = i] = \sum_{i=1}^6 i \cdot \Pr[X = i] = \sum_{i=1}^6 \frac{i}{6} = 3,5.$$

- Bezeichne nun X die Zufallsvariable, die angibt, **wie oft man einen fairen Würfel werfen muss, bis er das erste Mal eine Sechs zeigt**.

Es gilt $\Pr[X \geq 1] = 1$, $\Pr[X \geq 2] = 5/6$, $\Pr[X \geq 3] = (5/6)^2$ und $\Pr[X \geq i] = (5/6)^{i-1}$ für jedes $i \in \mathbb{N}$.

3.1 Quicksort

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den **Ausgang eines fairen Würfelwurfs** beschreibt. Dann gilt $\Pr[X = i] = 1/6$ für jedes $i \in \{1, \dots, 6\}$ und $\Pr[X = i] = 0$ für jedes $i \notin \{1, \dots, 6\}$. Damit gilt für den Erwartungswert von X

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \Pr[X = i] = \sum_{i=1}^6 i \cdot \Pr[X = i] = \sum_{i=1}^6 \frac{i}{6} = 3,5.$$

- Bezeichne nun X die Zufallsvariable, die angibt, **wie oft man einen fairen Würfel werfen muss, bis er das erste Mal eine Sechs zeigt**.

Es gilt $\Pr[X \geq 1] = 1$, $\Pr[X \geq 2] = 5/6$, $\Pr[X \geq 3] = (5/6)^2$ und $\Pr[X \geq i] = (5/6)^{i-1}$ für jedes $i \in \mathbb{N}$.

Damit ergibt sich

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] = \sum_{i=1}^{\infty} (5/6)^{i-1} = \frac{1}{1 - 5/6} = 6.$$

3.1 Quicksort

Beispiele für Erwartungswerte

- Eine Zufallsvariable X , die **nur die Werte 0 und 1 mit positiver Wahrscheinlichkeit annimmt**, nennen wir *0-1-Zufallsvariable*. Für ihren Erwartungswert gilt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \mathbf{Pr}[X = i] = 1 \cdot \mathbf{Pr}[X = 1] = \mathbf{Pr}[X = 1].$$

3.1 Quicksort

Beispiele für Erwartungswerte

- Eine Zufallsvariable X , die **nur die Werte 0 und 1 mit positiver Wahrscheinlichkeit annimmt**, nennen wir *0-1-Zufallsvariable*. Für ihren Erwartungswert gilt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} i \cdot \mathbf{Pr}[X = i] = 1 \cdot \mathbf{Pr}[X = 1] = \mathbf{Pr}[X = 1].$$

Linearität des Erwartungswertes: Für zwei beliebige Zufallsvariablen X und Y gilt stets $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$. Dies ist sogar dann der Fall, wenn die Zufallsvariablen voneinander abhängen.

3.1 Quicksort

Theorem 3.2

Die erwartete Laufzeit von RANDOMQUICKSORT beträgt auf jeder Eingabe mit n Zahlen $O(n \log n)$.

3.1 Quicksort

Theorem 3.2

Die erwartete Laufzeit von RANDOMQUICKSORT beträgt auf jeder Eingabe mit n Zahlen $O(n \log n)$.

Beweis:

QUICKSORT(int[] a , int ℓ , int r)

```
1  if ( $\ell < r$ ) {  
2      int  $q$  = PARTITION( $a, \ell, r$ );  
3      QUICKSORT( $a, \ell, q - 1$ );  
4      QUICKSORT( $a, q + 1, r$ );  
5  }
```

PARTITION(int[] a , int ℓ , int r)

```
1  int  $x = a[r]$ ;  
2  int  $i = \ell - 1$ ;  
3  for ( $j = \ell; j < r; j++$ ) {  
4      if ( $a[j] \leq x$ ) {  
5           $i++$ ;  
6          vertausche  $a[i]$  und  $a[j]$ ;  
7      }  
8  }  
9  vertausche  $a[i + 1]$  und  $a[r]$ ;  
10 return  $i + 1$ ;
```

3.1 Quicksort

Theorem 3.2

Die erwartete Laufzeit von RANDOMQUICKSORT beträgt auf jeder Eingabe mit n Zahlen $O(n \log n)$.

Beweis:

```
QUICKSORT(int[] a, int  $\ell$ , int  $r$ )
```

```
1  if ( $\ell < r$ ) {  
2      int  $q$  = PARTITION( $a, \ell, r$ );  
3      QUICKSORT( $a, \ell, q - 1$ );  
4      QUICKSORT( $a, q + 1, r$ );  
5  }
```

Es genügt, die **Zahl der wesentlichen Vergleiche** (Zeile 4 von PARTITION) zu beschränken.

```
PARTITION(int[] a, int  $\ell$ , int  $r$ )
```

```
1  int  $x$  =  $a[r]$ ;  
2  int  $i$  =  $\ell - 1$ ;  
3  for ( $j = \ell; j < r; j++$ ) {  
4      if ( $a[j] \leq x$ ) {  
5           $i++$ ;  
6          vertausche  $a[i]$  und  $a[j]$ ;  
7      }  
8  }  
9  vertausche  $a[i + 1]$  und  $a[r]$ ;  
10 return  $i + 1$ ;
```

3.1 Quicksort

- **Eingabe:** $a[0 \dots n - 1]$

3.1 Quicksort

- **Eingabe:** $a[0 \dots n - 1]$
- **Ausgabe:** (y_1, \dots, y_n) , d. h. y_i ist die i -t größte Zahl der Eingabe a .

3.1 Quicksort

- **Eingabe:** $a[0 \dots n - 1]$
- **Ausgabe:** (y_1, \dots, y_n) , d. h. y_i ist die i -t größte Zahl der Eingabe a .
- Für jedes Paar $i, j \in \{1, \dots, n\}$ mit $i < j$ definieren wir eine Zufallsvariable

$$X_{ij} = \begin{cases} 1 & \text{falls } y_i \text{ und } y_j \text{ verglichen werden,} \\ 0 & \text{sonst.} \end{cases}$$

3.1 Quicksort

- **Eingabe:** $a[0 \dots n-1]$
- **Ausgabe:** (y_1, \dots, y_n) , d. h. y_i ist die i -t größte Zahl der Eingabe a .
- Für jedes Paar $i, j \in \{1, \dots, n\}$ mit $i < j$ definieren wir eine Zufallsvariable

$$X_{ij} = \begin{cases} 1 & \text{falls } y_i \text{ und } y_j \text{ verglichen werden,} \\ 0 & \text{sonst.} \end{cases}$$

- Sei X die **Anzahl wesentlicher Vergleiche**. Dann gilt

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij},$$

denn derselbe Vergleich kann nicht mehrfach auftreten.

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

- Betrachte **Sequenz von Pivotelementen** bei Ausführung von RANDOMQUICKSORT.
Sei x **erstes Pivotelement aus der Menge** $\{y_i, y_{i+1}, \dots, y_j\}$.

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

- Betrachte **Sequenz von Pivotelementen** bei Ausführung von RANDOMQUICKSORT.
Sei x **erstes Pivotelement aus der Menge** $\{y_i, y_{i+1}, \dots, y_j\}$.

$$X_{ij} = 1 \iff x = y_i \text{ oder } x = y_j$$

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E} \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

- Betrachte **Sequenz von Pivotelementen** bei Ausführung von RANDOMQUICKSORT.
Sei x **erstes Pivotelement aus der Menge** $\{y_i, y_{i+1}, \dots, y_j\}$.

$$X_{ij} = 1 \iff x = y_i \text{ oder } x = y_j$$

- Sei $x \neq y_i$ und $x \neq y_j$.

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E} \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

- Betrachte **Sequenz von Pivotelementen** bei Ausführung von RANDOMQUICKSORT.
Sei x **erstes Pivotelement aus der Menge** $\{y_i, y_{i+1}, \dots, y_j\}$.

$$X_{ij} = 1 \iff x = y_i \text{ oder } x = y_j$$

- Sei $x \neq y_i$ und $x \neq y_j$.
Dann gilt $y_i < x < y_j$.

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

- Betrachte **Sequenz von Pivotelementen** bei Ausführung von RANDOMQUICKSORT.
Sei x **erstes Pivotelement aus der Menge** $\{y_i, y_{i+1}, \dots, y_j\}$.

$$X_{ij} = 1 \iff x = y_i \text{ oder } x = y_j$$

- Sei $x \neq y_i$ und $x \neq y_j$.

Dann gilt $y_i < x < y_j$.

$\Rightarrow y_i$ landet im linken Teilproblem. y_j landet im rechten Teilproblem.

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

- Betrachte **Sequenz von Pivotelementen** bei Ausführung von RANDOMQUICKSORT.
Sei x **erstes Pivotelement aus der Menge** $\{y_i, y_{i+1}, \dots, y_j\}$.

$$X_{ij} = 1 \iff x = y_i \text{ oder } x = y_j$$

- Sei $x \neq y_i$ und $x \neq y_j$.
Dann gilt $y_i < x < y_j$.
 $\Rightarrow y_i$ landet im linken Teilproblem. y_j landet im rechten Teilproblem.
- Sei $x = y_i$ oder $x = y_j$.

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E} \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

- Betrachte **Sequenz von Pivotelementen** bei Ausführung von RANDOMQUICKSORT.
Sei x **erstes Pivotelement aus der Menge** $\{y_i, y_{i+1}, \dots, y_j\}$.

$$X_{ij} = 1 \iff x = y_i \text{ oder } x = y_j$$

- Sei $x \neq y_i$ und $x \neq y_j$.
Dann gilt $y_i < x < y_j$.
 $\Rightarrow y_i$ landet im linken Teilproblem. y_j landet im rechten Teilproblem.
- Sei $x = y_i$ oder $x = y_j$.
 y_i und y_j sind beide im aktuellen Teilproblem enthalten.

3.1 Quicksort

- Es gilt

$$\mathbf{E}[X] = \mathbf{E} \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1].$$

- Betrachte **Sequenz von Pivotelementen** bei Ausführung von RANDOMQUICKSORT.
Sei x **erstes Pivotelement aus der Menge** $\{y_i, y_{i+1}, \dots, y_j\}$.

$$X_{ij} = 1 \iff x = y_i \text{ oder } x = y_j$$

- Sei $x \neq y_i$ und $x \neq y_j$.
Dann gilt $y_i < x < y_j$.
 $\Rightarrow y_i$ landet im linken Teilproblem. y_j landet im rechten Teilproblem.
- Sei $x = y_i$ oder $x = y_j$.
 y_i und y_j sind beide im aktuellen Teilproblem enthalten.
 $\Rightarrow y_i$ und y_j werden verglichen.

3.1 Quicksort

$$\Rightarrow \mathbf{Pr}[X_{ij} = 1] = \mathbf{Pr}[x = y_i \text{ oder } x = y_j] = \frac{2}{j - i + 1}$$

3.1 Quicksort

$$\Rightarrow \mathbf{Pr}[X_{ij} = 1] = \mathbf{Pr}[x = y_i \text{ oder } x = y_j] = \frac{2}{j - i + 1}$$

Es gilt:

$$\mathbf{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbf{Pr}[X_{ij} = 1]$$

3.1 Quicksort

$$\Rightarrow \Pr[X_{ij} = 1] = \Pr[x = y_i \text{ oder } x = y_j] = \frac{2}{j - i + 1}$$

Es gilt:

$$\mathbf{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1}$$

3.1 Quicksort

$$\Rightarrow \Pr[X_{ij} = 1] = \Pr[x = y_i \text{ oder } x = y_j] = \frac{2}{j - i + 1}$$

Es gilt:

$$\begin{aligned} \mathbf{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \end{aligned}$$

3.1 Quicksort

$$\Rightarrow \Pr[X_{ij} = 1] = \Pr[x = y_i \text{ oder } x = y_j] = \frac{2}{j - i + 1}$$

Es gilt:

$$\begin{aligned} \mathbf{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^n \sum_{i=1}^{n+1-k} \frac{2}{k} \end{aligned}$$

3.1 Quicksort

$$\Rightarrow \Pr[X_{ij} = 1] = \Pr[x = y_i \text{ oder } x = y_j] = \frac{2}{j - i + 1}$$

Es gilt:

$$\begin{aligned} \mathbf{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^n \sum_{i=1}^{n+1-k} \frac{2}{k} = \sum_{k=2}^n \frac{2(n+1-k)}{k} \end{aligned}$$

3.1 Quicksort

$$\Rightarrow \Pr[X_{ij} = 1] = \Pr[x = y_i \text{ oder } x = y_j] = \frac{2}{j - i + 1}$$

Es gilt:

$$\begin{aligned} \mathbf{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^n \sum_{i=1}^{n+1-k} \frac{2}{k} = \sum_{k=2}^n \frac{2(n+1-k)}{k} \\ &= (n+1) \left(\sum_{k=2}^n \frac{2}{k} \right) - 2(n-1) \end{aligned}$$

3.1 Quicksort

$$\Rightarrow \Pr[X_{ij} = 1] = \Pr[x = y_i \text{ oder } x = y_j] = \frac{2}{j - i + 1}$$

Es gilt:

$$\begin{aligned} \mathbf{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} \\ &= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^n \sum_{i=1}^{n+1-k} \frac{2}{k} = \sum_{k=2}^n \frac{2(n+1-k)}{k} \\ &= (n+1) \left(\sum_{k=2}^n \frac{2}{k} \right) - 2(n-1) \\ &= (2n+2)(H_n - 1) - 2(n-1), \end{aligned}$$

wobei $H_n = \sum_{i=1}^n \frac{1}{i}$ die **n -te harmonische Zahl** bezeichnet.

3.1 Quicksort

Es gilt $H_n \leq \ln n + 1$

3.1 Quicksort

Es gilt $H_n \leq \ln n + 1$ und damit

$$\mathbf{E}[X] = (2n + 2)(H_n - 1) - 2(n - 1)$$

3.1 Quicksort

Es gilt $H_n \leq \ln n + 1$ und damit

$$\begin{aligned}\mathbf{E}[X] &= (2n + 2)(H_n - 1) - 2(n - 1) \\ &\leq 2n \ln n + 2 \ln n - 2n + 2\end{aligned}$$

3.1 Quicksort

Es gilt $H_n \leq \ln n + 1$ und damit

$$\begin{aligned}\mathbf{E}[X] &= (2n + 2)(H_n - 1) - 2(n - 1) \\ &\leq 2n \ln n + 2 \ln n - 2n + 2 \\ &= 2n \ln n - \Theta(n)\end{aligned}$$

3.1 Quicksort

Es gilt $H_n \leq \ln n + 1$ und damit

$$\begin{aligned}\mathbf{E}[X] &= (2n+2)(H_n-1) - 2(n-1) \\ &\leq 2n \ln n + 2 \ln n - 2n + 2 \\ &= 2n \ln n - \Theta(n) \\ &= O(n \log n).\end{aligned}$$



3.2 Eigenschaften von Sortieralgorithmen

Definition

Ein Sortieralgorithmus heißt **stabil**, wenn die relative Ordnung gleicher Elemente beim Sortieren erhalten bleibt.

3.2 Eigenschaften von Sortieralgorithmen

Definition

Ein Sortieralgorithmus heißt **stabil**, wenn die relative Ordnung gleicher Elemente beim Sortieren erhalten bleibt.

Unsere Implementierungen von INSERTIONSORT und MERGESORT sind stabil. Unsere Implementierung von QUICKSORT hingegen ist nicht.

3.2 Eigenschaften von Sortialgorithmen

Definition

Ein Sortialgorithmus heißt **stabil**, wenn die relative Ordnung gleicher Elemente beim Sortieren erhalten bleibt.

Unsere Implementierungen von INSERTIONSORT und MERGESORT sind stabil. Unsere Implementierung von QUICKSORT hingegen ist nicht.

Definition

Ein Sortialgorithmus arbeitet **in situ (in-place)**, wenn er zusätzlich zur Eingabe a nur konstant viel Speicherplatz benötigt.

3.2 Eigenschaften von Sortialgorithmen

Definition

Ein Sortialgorithmus heißt **stabil**, wenn die relative Ordnung gleicher Elemente beim Sortieren erhalten bleibt.

Unsere Implementierungen von INSERTIONSORT und MERGESORT sind stabil. Unsere Implementierung von QUICKSORT hingegen ist nicht.

Definition

Ein Sortialgorithmus arbeitet **in situ (in-place)**, wenn er zusätzlich zur Eingabe a nur konstant viel Speicherplatz benötigt.

INSERTIONSORT arbeitet in situ. MERGESORT hingegen benötigt zusätzlichen Speicher der Größe $\Theta(n)$ für die Felder *left* und *right* in der Methode MERGE.

3.2 Eigenschaften von Sortialgorithmen

Definition

Ein Sortialgorithmus heißt **stabil**, wenn die relative Ordnung gleicher Elemente beim Sortieren erhalten bleibt.

Unsere Implementierungen von INSERTIONSORT und MERGESORT sind stabil. Unsere Implementierung von QUICKSORT hingegen ist nicht.

Definition

Ein Sortialgorithmus arbeitet **in situ (in-place)**, wenn er zusätzlich zur Eingabe a nur konstant viel Speicherplatz benötigt.

INSERTIONSORT arbeitet in situ. MERGESORT hingegen benötigt zusätzlichen Speicher der Größe $\Theta(n)$ für die Felder *left* und *right* in der Methode MERGE.

QUICKSORT benötigt Speicherplatz für den Rekursionsstack.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortialgorithmen

Definition

Ein Sortialgorithmus heißt **vergleichsbasiert**, wenn er nur durch die Vergleiche zweier Objekte aus der Eingabe Informationen über die Eingabe gewinnt.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortieralgorithmen

Definition

Ein Sortieralgorithmus heißt **vergleichsbasiert**, wenn er nur durch die Vergleiche zweier Objekte aus der Eingabe Informationen über die Eingabe gewinnt.

INSERTIONSORT, MERGESORT und QUICKSORT sind vergleichsbasierte Sortieralgorithmen.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortieralgorithmen

Definition

Ein Sortieralgorithmus heißt **vergleichsbasiert**, wenn er nur durch die Vergleiche zweier Objekte aus der Eingabe Informationen über die Eingabe gewinnt.

INSERTIONSORT, MERGESORT und QUICKSORT sind vergleichsbasierte Sortieralgorithmen.

Theorem 3.3

Jeder vergleichsbasierte Sortieralgorithmus benötigt zum Sortieren von Feldern der Länge n **im Worst Case** $\Omega(n \log n)$ **Vergleiche**. Damit beträgt insbesondere seine Worst-Case-Laufzeit $\Omega(n \log n)$.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortieralgorithmen

Beweis:

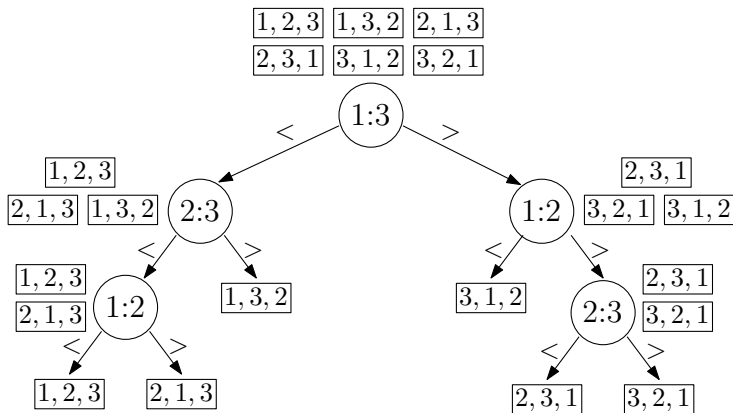
Definition

Sei (ℓ_1, \dots, ℓ_n) die eindeutige Permutation der Zahlen $1, \dots, n$, für die $a[\ell_1] < \dots < a[\ell_n]$ gilt. Wir nennen (ℓ_1, \dots, ℓ_n) den **Ordnungstyp** von a .

Ein Sortieralgorithmus darf sich auf zwei Eingaben, die verschiedene Ordnungstypen besitzen, nicht identisch verhalten.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortialgorithmen

Jeder vergleichsbasierte Algorithmus induziert für ein festes n einen **Entscheidungsbaum**.



3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortialgorithmen

Sei ein beliebiger vergleichsbasierter Sortialgorithmus gegeben.

Für festes n entspricht dieser einem **Entscheidungsbaum mit $n!$ Blättern**.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortialgorithmen

Sei ein beliebiger vergleichsbasierter Sortialgorithmus gegeben.

Für festes n entspricht dieser einem **Entscheidungsbaum mit $n!$ Blättern**.

Es bezeichne h die Höhe des Baumes.

Diese entspricht der **Anzahl der Vergleiche im Worst Case**.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortialgorithmen

Sei ein beliebiger vergleichsbasierter Sortialgorithmus gegeben.

Für festes n entspricht dieser einem **Entscheidungsbaum mit $n!$ Blättern**.

Es bezeichne h die Höhe des Baumes.

Diese entspricht der **Anzahl der Vergleiche im Worst Case**.

Da jeder innere Knoten Grad 2 besitzt, besitzt der Baum höchstens 2^h Blätter.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortialgorithmen

Sei ein beliebiger vergleichsbasierter Sortialgorithmus gegeben.

Für festes n entspricht dieser einem **Entscheidungsbaum mit $n!$ Blättern**.

Es bezeichne h die Höhe des Baumes.

Diese entspricht der **Anzahl der Vergleiche im Worst Case**.

Da jeder innere Knoten Grad 2 besitzt, besitzt der Baum höchstens 2^h Blätter.

$$\Rightarrow 2^h \geq n!$$

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortieralgorithmen

Sei ein beliebiger vergleichsbasierter Sortieralgorithmus gegeben.

Für festes n entspricht dieser einem **Entscheidungsbaum mit $n!$ Blättern**.

Es bezeichne h die Höhe des Baumes.

Diese entspricht der **Anzahl der Vergleiche im Worst Case**.

Da jeder innere Knoten Grad 2 besitzt, besitzt der Baum höchstens 2^h Blätter.

$$\Rightarrow 2^h \geq n!$$

$$\Rightarrow h \geq \log_2(n!)$$

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortialgorithmen

Sei ein beliebiger vergleichsbasierter Sortialgorithmus gegeben.

Für festes n entspricht dieser einem **Entscheidungsbaum mit $n!$ Blättern**.

Es bezeichne h die Höhe des Baumes.

Diese entspricht der **Anzahl der Vergleiche im Worst Case**.

Da jeder innere Knoten Grad 2 besitzt, besitzt der Baum höchstens 2^h Blätter.

$$\Rightarrow 2^h \geq n!$$

$$\Rightarrow h \geq \log_2(n!)$$

Das Theorem folgt, da $\log_2(n!) = \Theta(n \log n)$ gilt.



3.4 Sortieren in Linearzeit

Der folgende Algorithmus MSORT sortiert Felder $a[0 \dots n - 1]$, die **Elemente aus** $\{0, \dots, M - 1\}$ für ein $M \in \mathbb{N}$ enthalten.

3.4 Sortieren in Linearzeit

Der folgende Algorithmus MSORT sortiert Felder $a[0 \dots n - 1]$, die **Elemente aus $\{0, \dots, M - 1\}$** für ein $M \in \mathbb{N}$ enthalten.

MSORT(int[] a)

- 1 Initialisiere ein Feld der Länge M , das für jedes $k \in \{0, \dots, M - 1\}$ eine leere verkettete Liste $L[k]$ enthält.
- 2 **for** (**int** $i = 0$; $i < n$; $i++$)
- 3 Füge $a[i]$ an das Ende der Liste $L[a[i]]$ an.
- 4 Erzeuge ein Feld $b[0 \dots n - 1]$ durch das Aneinanderhängen der Listen $L[0], L[1], \dots, L[M - 1]$.
- 5 **return** b ;

3.4 Sortieren in Linearzeit

Der folgende Algorithmus MSORT sortiert Felder $a[0 \dots n - 1]$, die **Elemente aus $\{0, \dots, M - 1\}$** für ein $M \in \mathbb{N}$ enthalten.

MSORT(int[] a)

- 1 Initialisiere ein Feld der Länge M , das für jedes $k \in \{0, \dots, M - 1\}$ eine leere verkettete Liste $L[k]$ enthält.
- 2 **for** (int $i = 0$; $i < n$; $i++$)
- 3 Füge $a[i]$ an das Ende der Liste $L[a[i]]$ an.
- 4 Erzeuge ein Feld $b[0 \dots n - 1]$ durch das Aneinanderhängen der Listen $L[0], L[1], \dots, L[M - 1]$.
- 5 **return** b ;

Lemma 3.4

Der Algorithmus MSORT sortiert n Zahlen aus der Menge $\{0, \dots, M - 1\}$ in Zeit $O(n + M)$. Ferner ist MSORT ein stabiler Sortieralgorithmus.

3.4 Sortieren in Linearzeit

Der folgende Algorithmus RADIXSORT sortiert Felder $a[0 \dots n - 1]$, deren Einträge **Zahlen mit ℓ Ziffern in M -adischer Darstellung** sind.

3.4 Sortieren in Linearzeit

Der folgende Algorithmus RADIXSORT sortiert Felder $a[0 \dots n - 1]$, deren Einträge **Zahlen mit ℓ Ziffern in M -adischer Darstellung** sind.

```
RADIXSORT(int[] a)
```

```
1   for (int  $i = \ell; i \geq 1; i--$ )
```

```
2       Sortiere das Feld  $a$  mit MSORT bezüglich der  $i$ -ten Ziffer.
```

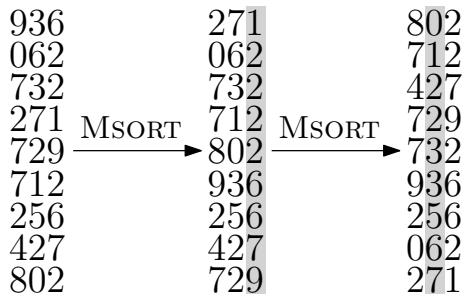
3.4 Sortieren in Linearzeit

936
062
732
271
729
712
256
427
802

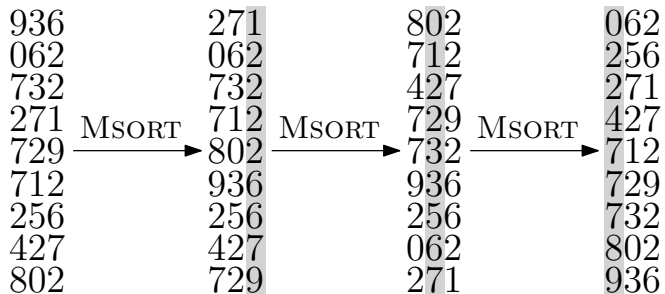
3.4 Sortieren in Linearzeit

936		271
062		062
732		732
271	MSORT	712
729	→	802
712		936
256		256
427		427
802		729

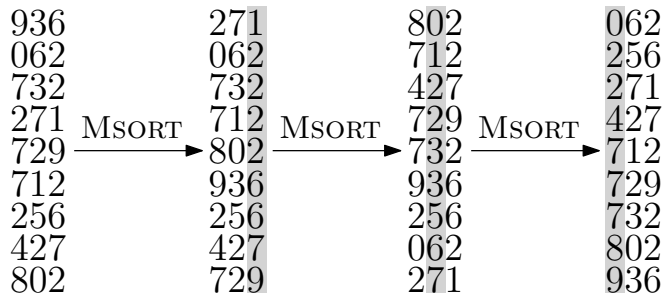
3.4 Sortieren in Linearzeit



3.4 Sortieren in Linearzeit



3.4 Sortieren in Linearzeit



Theorem 3.5

Der Algorithmus RADIXSORT sortiert n Zahlen mit jeweils ℓ Ziffern in M -adischer Darstellung in Zeit $O(\ell(n + M))$.