

Datenstrukturen und Algorithmen

Vorlesung 9: Quicksort (K7)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

<https://moves.rwth-aachen.de/teaching/ss-18/dsal/>

18. Mai 2018

WICHTIG: Anmeldung Klausur u. Präsenzübung

Übersicht

1 Quicksort

- Das Divide-and-Conquer Paradigma
- Partitionierung
- Quicksort Algorithmus
- Komplexitätsanalyse

mergesort

worst case

best case

average case

2 Weitere Sortierverfahren

- Bubblesort
- Countingsort

3 Vergleich der Sortieralgorithmen

Übersicht

1 Quicksort

- Das Divide-and-Conquer Paradigma
- Partitionierung
- Quicksort Algorithmus
- Komplexitätsanalyse

2 Weitere Sortierverfahren

- Bubblesort
- Countingsort

3 Vergleich der Sortieralgorithmen

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide and conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide and conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme **rekursiv** und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide and conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme **rekursiv** und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide and conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme **rekursiv** und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

- Teile** das Problem in eine Anzahl von Teilproblemen auf.

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide and conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme **rekursiv** und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

Teile das Problem in eine Anzahl von Teilproblemen auf.

Beherrsche die Teilprobleme durch rekursives Lösen. Hinreichend kleine Teilprobleme werden direkt gelöst.

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide and conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme **rekursiv** und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

- Teile** das Problem in eine Anzahl von Teilproblemen auf.

- Beherrsche** die Teilprobleme durch rekursives Lösen. Hinreichend kleine Teilprobleme werden direkt gelöst.

- Verbinde** die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems.

Divide-and-Conquer

Teile-und-Beherrsche Algorithmen (divide and conquer) teilen das Problem in mehrere Teilprobleme auf, die dem Ausgangsproblem ähneln, jedoch von kleinerer Größe sind.

Sie lösen die Teilprobleme **rekursiv** und kombinieren diese Lösungen dann, um die Lösung des eigentlichen Problems zu erstellen.

Das Paradigma von Teile-und-Beherrsche umfasst 3 Schritte auf jeder Rekursionsebene:

- Teile** das Problem in eine Anzahl von Teilproblemen auf.

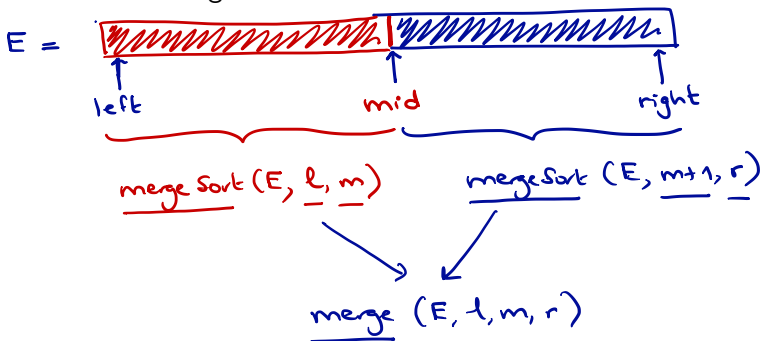
- Beherrsche** die Teilprobleme durch rekursives Lösen. Hinreichend kleine Teilprobleme werden direkt gelöst.

- Verbinde** die Lösungen der Teilprobleme zur Lösung des Ausgangsproblems.

Beispiel: Mergesort (s. Vorlesung 7).

Quicksort – Idee

Mergesort sortiert zunächst rekursiv, danach verteilt er sozusagen die Elemente an die richtigen Stellen.



Quicksort – Idee

Mergesort sortiert zunächst rekursiv, danach verteilt er sozusagen die Elemente an die richtigen Stellen.

Bei **Quicksort** werden die Elemente zuerst auf die richtige Seite („Hälfte“) des Arrays gebracht, dann wird jeweils rekursiv sortiert.



Quicksort – Idee

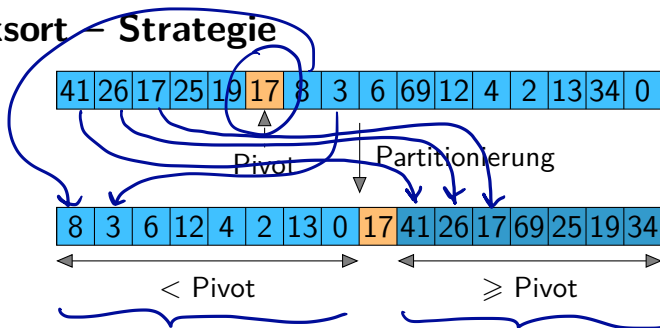
Mergesort sortiert zunächst rekursiv, danach verteilt er sozusagen die Elemente an die richtigen Stellen.

Bei **Quicksort** werden die Elemente zuerst auf die richtige Seite („Hälfte“) des Arrays gebracht, dann wird jeweils rekursiv sortiert.

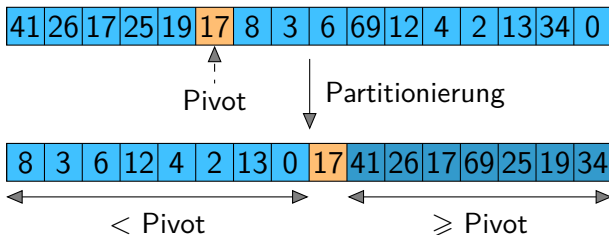
Quicksort wurde 1961 von Tony Hoare (Großbritannien) entwickelt.

Turing Award

Quicksort – Strategie

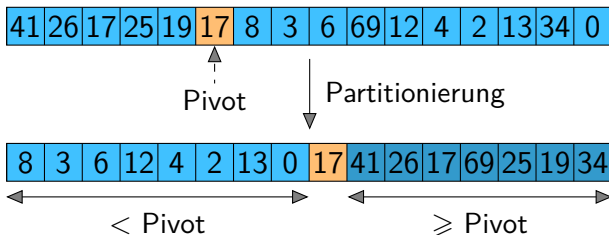


Quicksort – Strategie



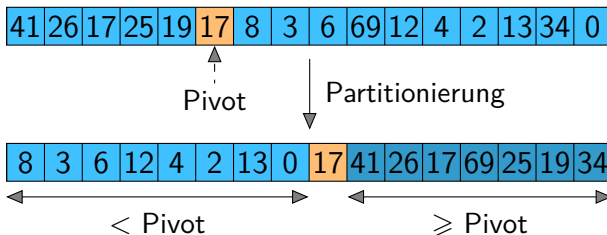
Teile Wähle ein **Pivotelement** aus dem zu sortierenden Array

Quicksort – Strategie



Teile Wähle ein **Pivotelement** aus dem zu sortierenden Array und **partitioniere** das zu sortierende Array in zwei Teile auf:

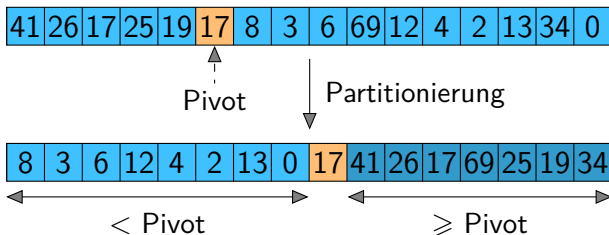
Quicksort – Strategie



Teile Wähle ein **Pivotelement** aus dem zu sortierenden Array und **partitioniere** das zu sortierende Array in zwei Teile auf:

1. **Kleiner** als das Pivotelement, sowie
2. **mindestens so groß** wie das Pivotelement.

Quicksort – Strategie

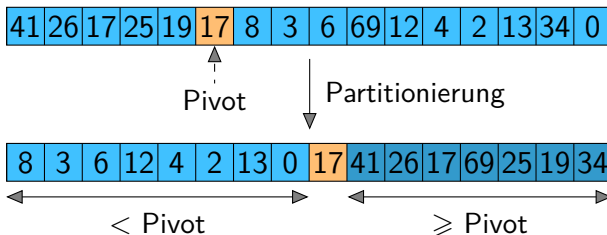


Teile Wähle ein **Pivotelement** aus dem zu sortierenden Array und **partitioniere** das zu sortierende Array in zwei Teile auf:

1. **Kleiner** als das Pivotelement, sowie
2. **mindestens so groß** wie das Pivotelement.

Beherrsche: Sortiere die Teile rekursiv und setze dann das Pivotelement zwischen die sortierten Teile.

Quicksort – Strategie



Teile Wähle ein **Pivotelement** aus dem zu sortierenden Array und **partitioniere** das zu sortierende Array in zwei Teile auf:

1. **Kleiner** als das Pivotelement, sowie
2. **mindestens so groß** wie das Pivotelement.

Beherrsche: Sortiere die Teile rekursiv und setze dann das Pivotelement zwischen die sortierten Teile.

Verbinde: Da die Teilfelder in-place sortiert werden ist keine Arbeit nötig, um sie zu verbinden.

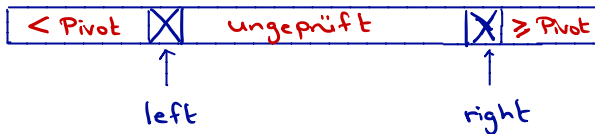
Partitionierung (I)

Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

Partitionierung (I)

Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

- ▶ Arbeite mit drei Bereichen: „ $< \text{Pivot}$ “, „ $\geq \text{Pivot}$ “ und „ungeprüft“.



Partitionierung (I)

Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

- ▶ Arbeite mit drei Bereichen: „ $< \text{Pivot}$ “, „ $\geq \text{Pivot}$ “ und „ungeprüft“.
- ▶ Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $< \text{Pivot}$ ist.

Partitionierung (I)

Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

- ▶ Arbeite mit drei Bereichen: „ $< \text{Pivot}$ “, „ $\geq \text{Pivot}$ “ und „ungeprüft“.
- ▶ Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $< \text{Pivot}$ ist.
- ▶ Schiebe die rechte Grenze nach links, solange das zusätzliche Element $\geq \text{Pivot}$ ist.

Partitionierung (I)

Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

- ▶ Arbeite mit drei Bereichen: „ $< \text{Pivot}$ “, „ $\geq \text{Pivot}$ “ und „ungeprüft“.
- ▶ Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $< \text{Pivot}$ ist.
- ▶ Schiebe die rechte Grenze nach links, solange das zusätzliche Element $\geq \text{Pivot}$ ist.
- ▶ Tausche das links gefundene mit dem rechts gefundenen Element.

Partitionierung (I)

Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

- ▶ Arbeite mit drei Bereichen: „ $< \text{Pivot}$ “, „ $\geq \text{Pivot}$ “ und „ungeprüft“.
- ▶ Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $< \text{Pivot}$ ist.
- ▶ Schiebe die rechte Grenze nach links, solange das zusätzliche Element $\geq \text{Pivot}$ ist.
- ▶ Tausche das links gefundene mit dem rechts gefundenen Element.
- ▶ Fahre fort, bis sich die Grenzen treffen.

Partitionierung (I)

Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

- ▶ Arbeite mit drei Bereichen: „ $< \text{Pivot}$ “, „ $\geq \text{Pivot}$ “ und „ungeprüft“.
- ▶ Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $< \text{Pivot}$ ist.
- ▶ Schiebe die rechte Grenze nach links, solange das zusätzliche Element $\geq \text{Pivot}$ ist.
- ▶ Tausche das links gefundene mit dem rechts gefundenen Element.
- ▶ Fahre fort, bis sich die Grenzen treffen.

(Es gibt auch andere Verfahren.)

Partitionierung (I)

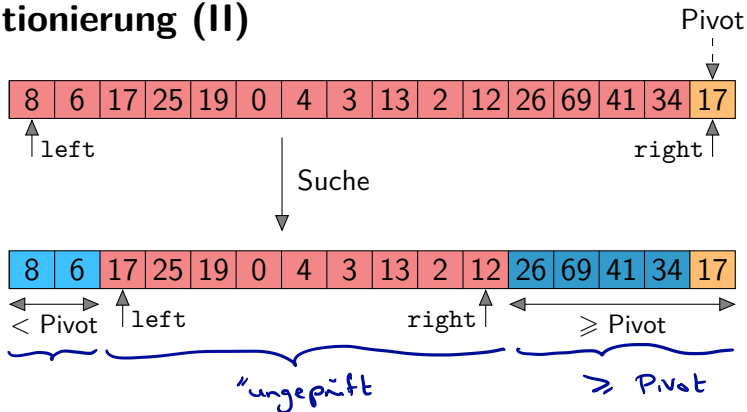
Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

- ▶ Arbeite mit drei Bereichen: „ $< \text{Pivot}$ “, „ $\geq \text{Pivot}$ “ und „ungeprüft“.
- ▶ Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $< \text{Pivot}$ ist.
- ▶ Schiebe die rechte Grenze nach links, solange das zusätzliche Element $\geq \text{Pivot}$ ist.
- ▶ Tausche das links gefundene mit dem rechts gefundenen Element.
- ▶ Fahre fort, bis sich die Grenzen treffen.

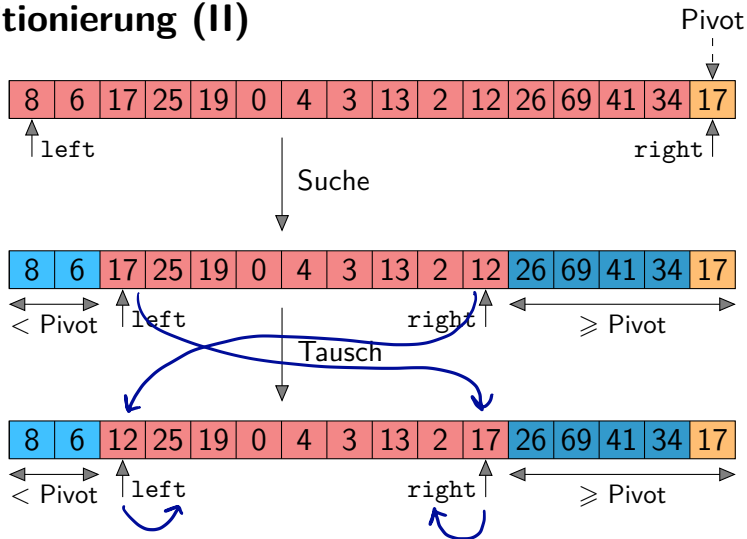
(Es gibt auch andere Verfahren.)

Das obige Schema ist ähnlich zu Dijkstra's **Dutch** National **Flag** Problem.

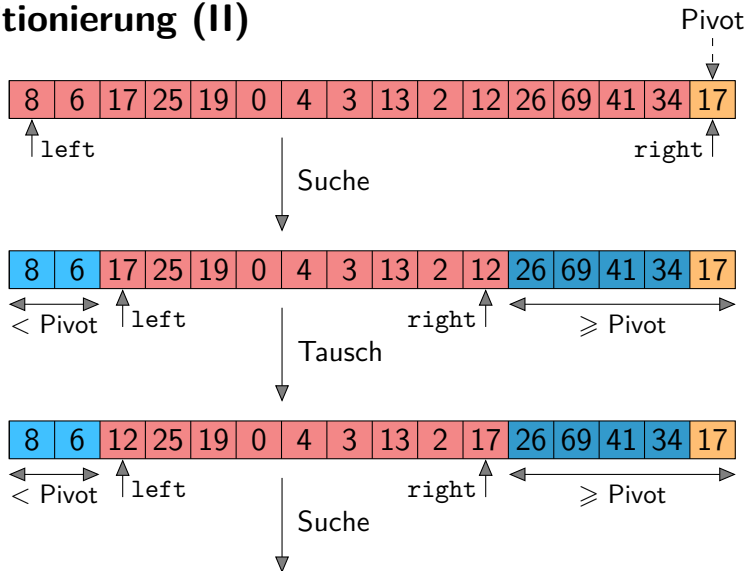
Partitionierung (II)



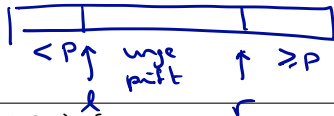
Partitionierung (II)



Partitionierung (II)



Partitionierung – Algorithmus

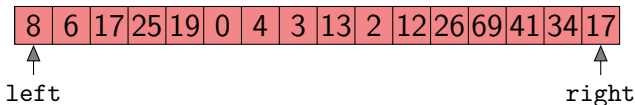


```

1 int partition(int E[], int left, int right) {
2     // Wähle einfaches Pivotelement
3     int ppos = right, pivot = E[ppos];
4     while (true) {
5         // Bilineare Suche
6         while (left < right && E[left] < pivot) left++;
7         while (left < right && E[right] >= pivot) right--;
8         if (left >= right) {
9             break;
10        }
11        swap(E[left], E[right]);
12    }
13    swap(E[left], E[ppos]);
14    return left; // gib neue Pivotposition als Splitpunkt zurück
15 }

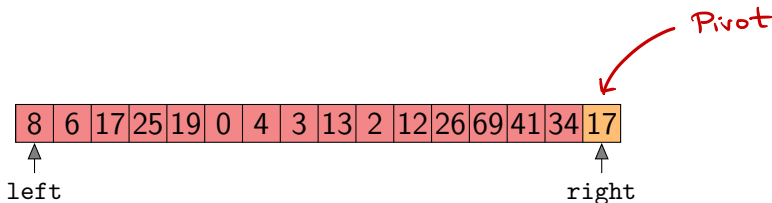
```

Quicksort – Algorithmus und Animation



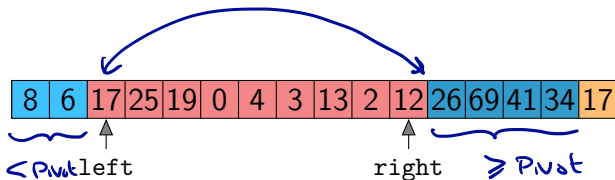
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



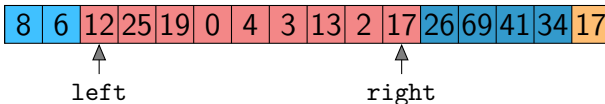
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     * int i = partition(E, left, right);  
4       // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



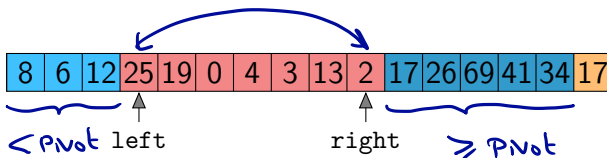
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation

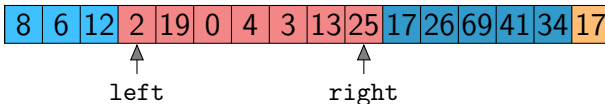


```

1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     * int i = partition(E, left, right);
4       // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }

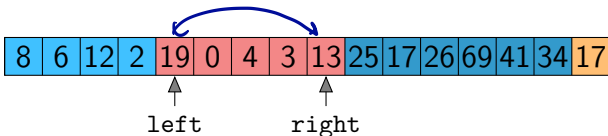
```

Quicksort – Algorithmus und Animation



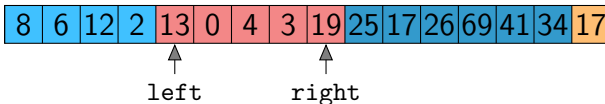
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



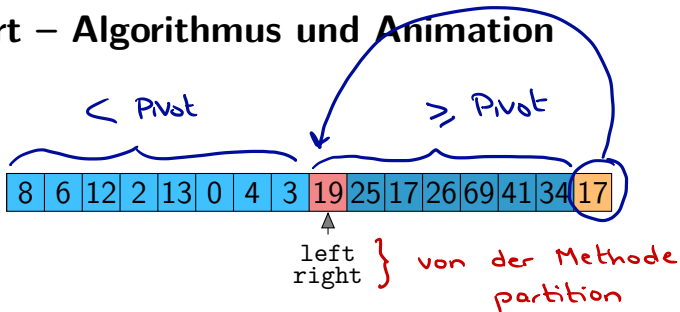
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation

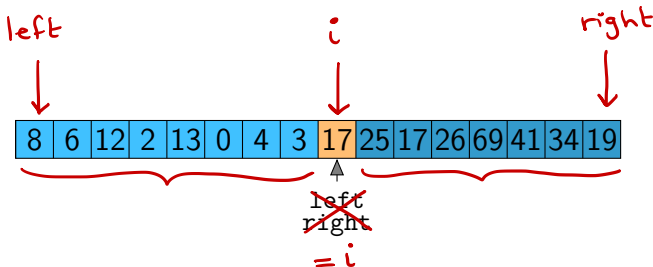


```

1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     * int i = partition(E, left, right);
4       // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }

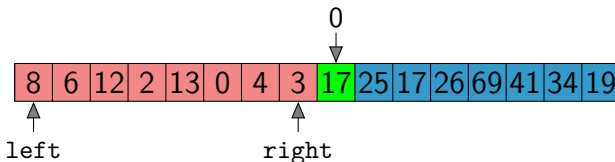
```

Quicksort – Algorithmus und Animation



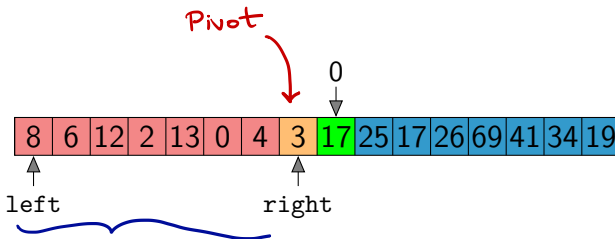
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



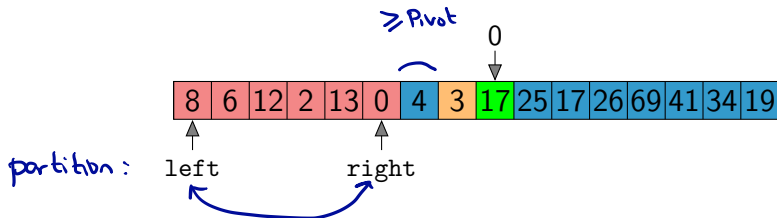
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     * int i = partition(E, left, right);  
4       // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation

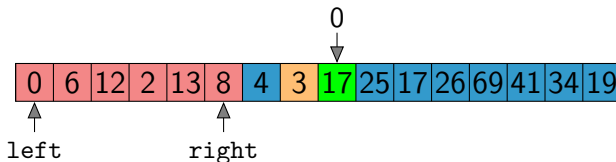


```

1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     int i = partition(E, left, right);
4     // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }

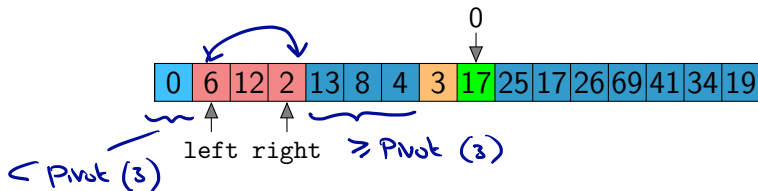
```

Quicksort – Algorithmus und Animation



```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation

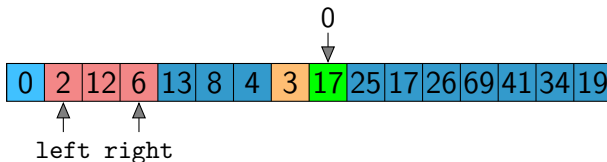


```

1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     int i = partition(E, left, right);
4     // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }

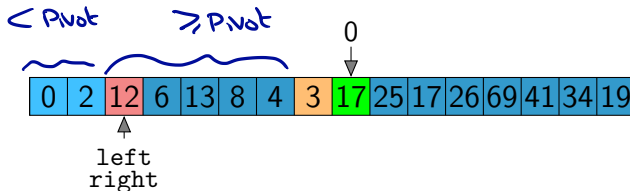
```

Quicksort – Algorithmus und Animation



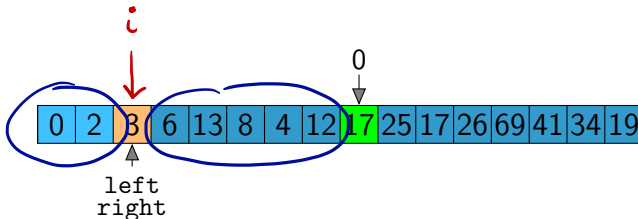
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



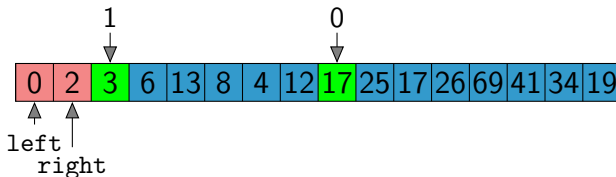
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```


Quicksort – Algorithmus und Animation



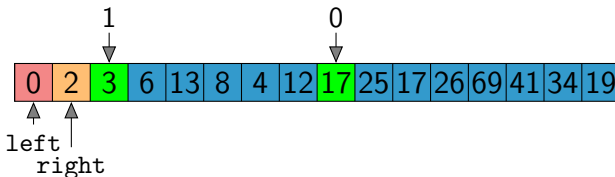
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



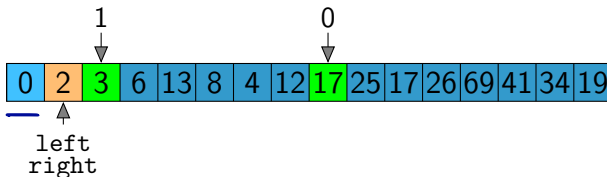
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



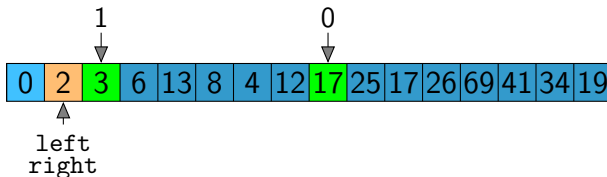
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



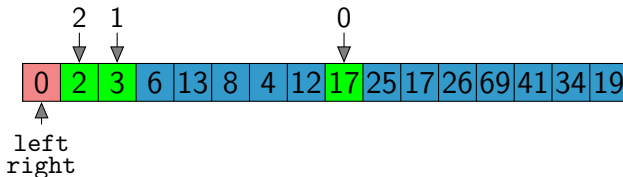
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



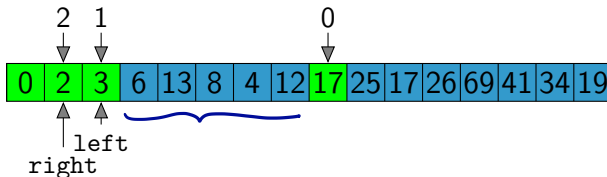
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



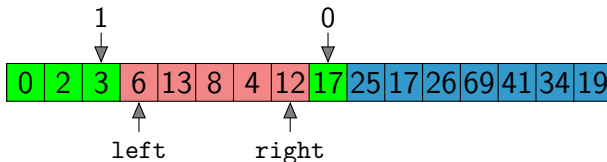
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



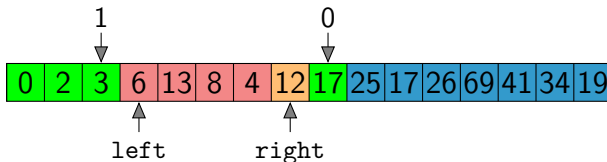
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



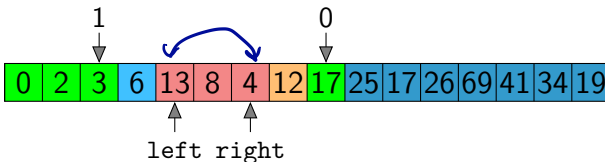
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



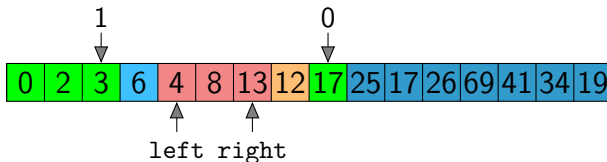
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



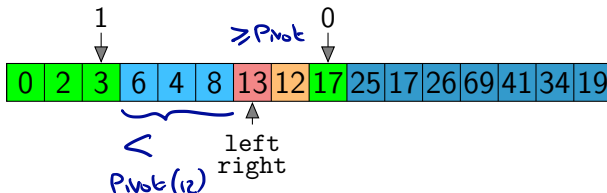
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation

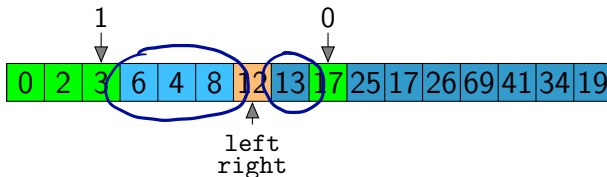


```

1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     int i = partition(E, left, right);
4     // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }

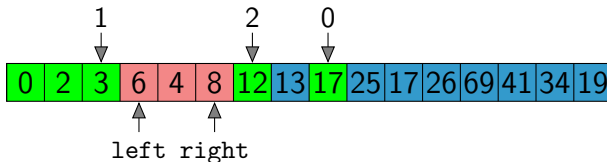
```

Quicksort – Algorithmus und Animation



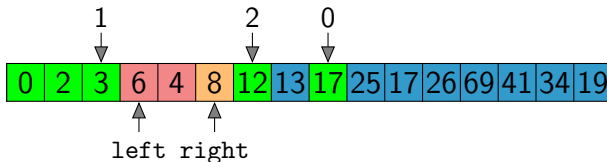
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



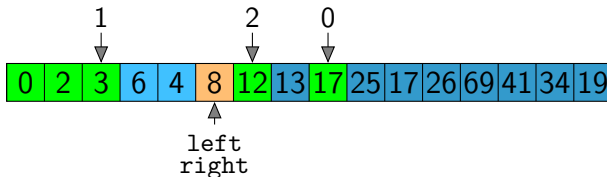
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



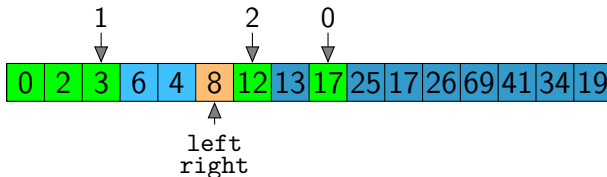
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



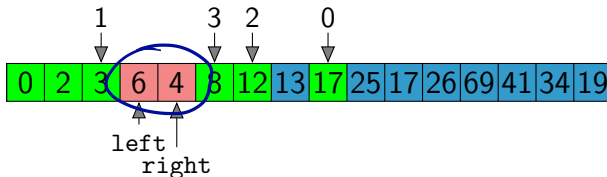
```
1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     int i = partition(E, left, right);
4     // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }
```

Quicksort – Algorithmus und Animation



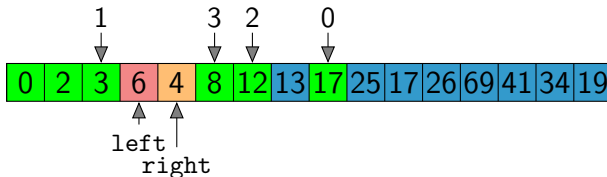
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



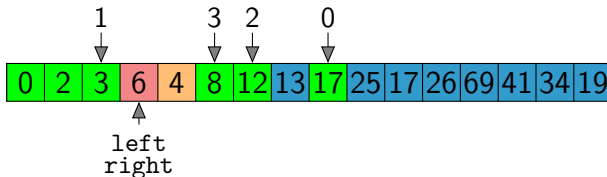
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



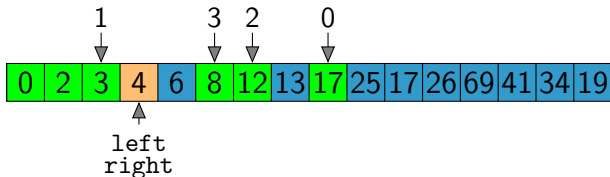
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



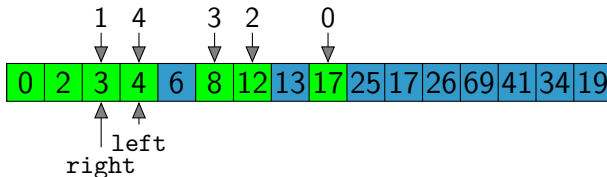
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



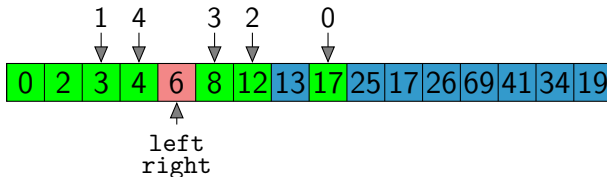
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



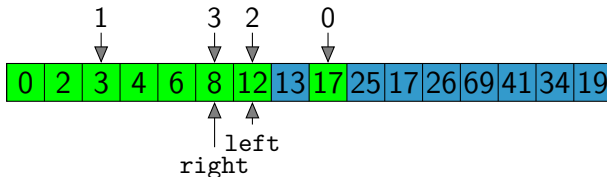
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



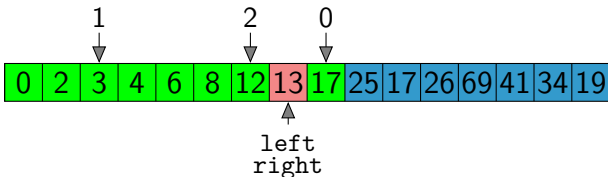
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



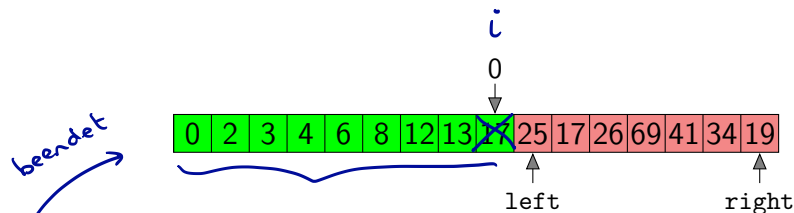
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



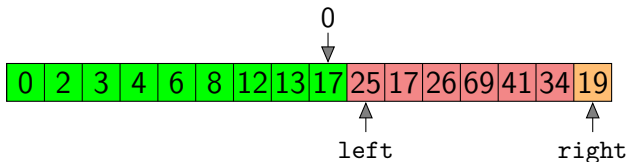
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



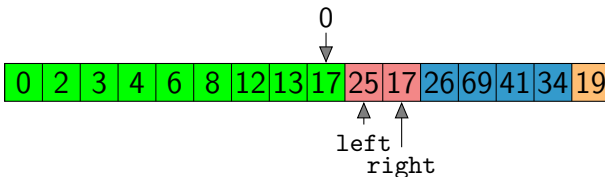
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



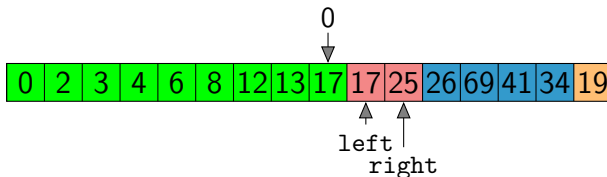
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



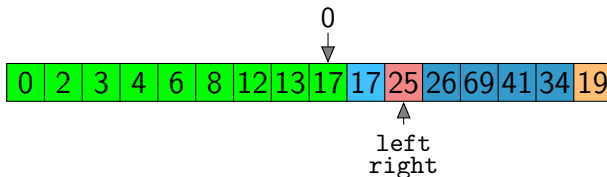
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



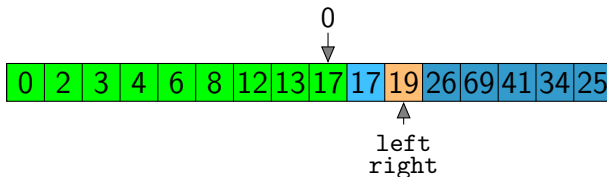
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



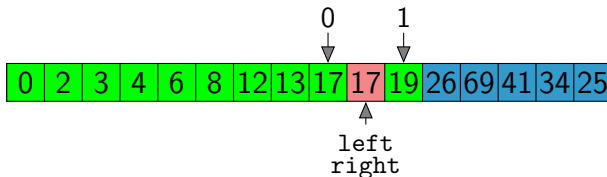
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



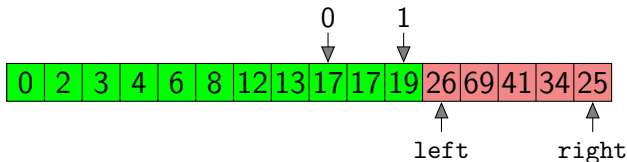
```
1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     int i = partition(E, left, right);
4     // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }
```

Quicksort – Algorithmus und Animation



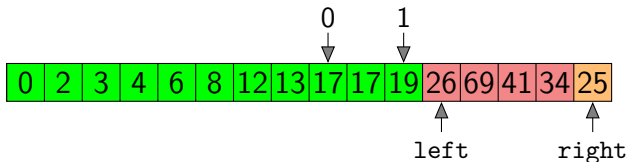
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



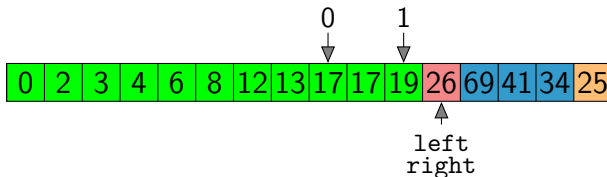
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



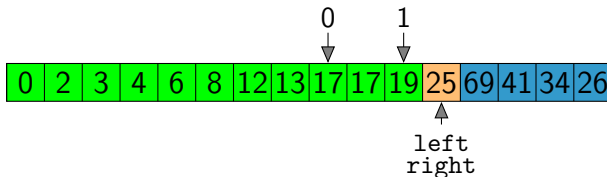
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



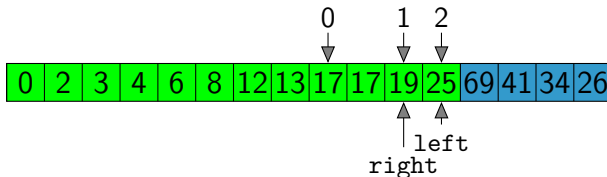
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



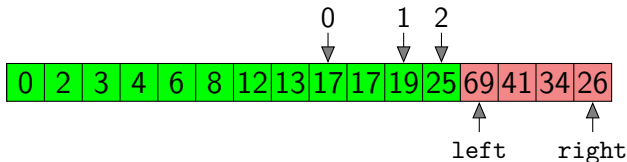
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



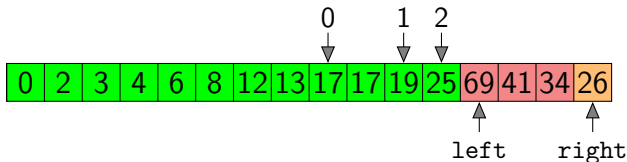
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



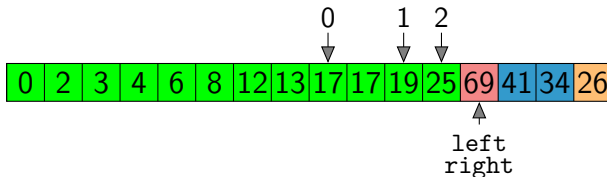
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



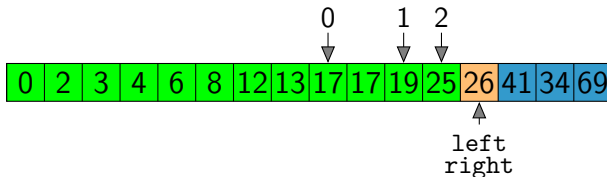
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



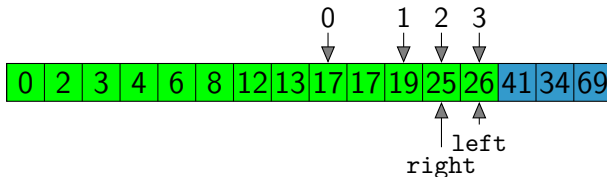
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



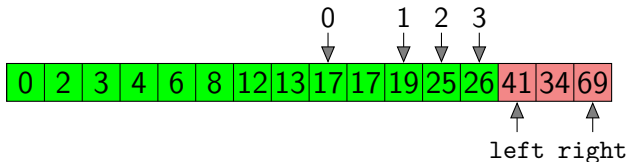
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



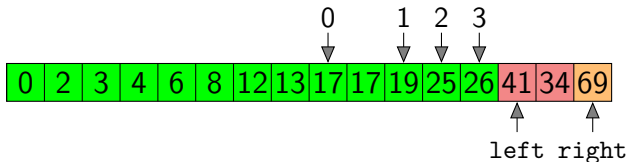
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



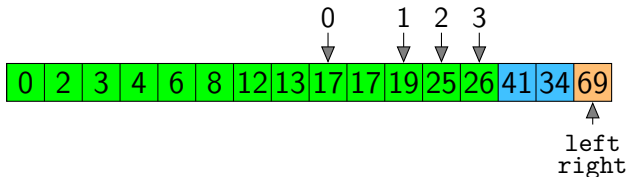
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



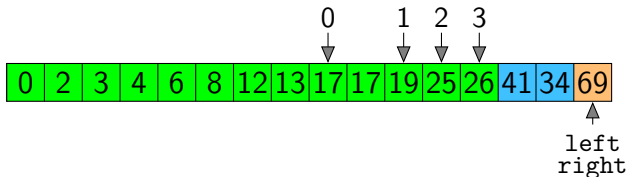
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



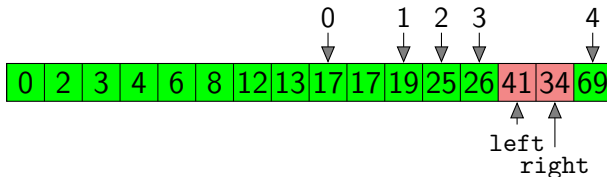
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



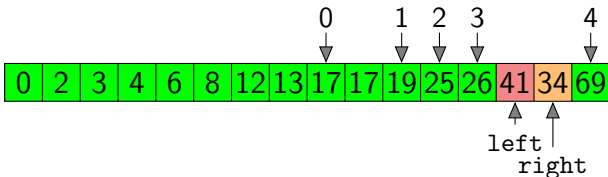
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



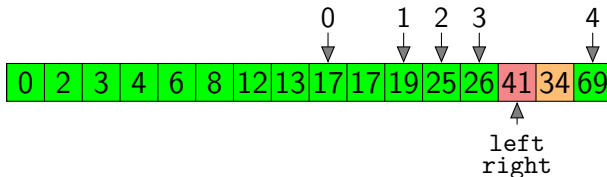
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



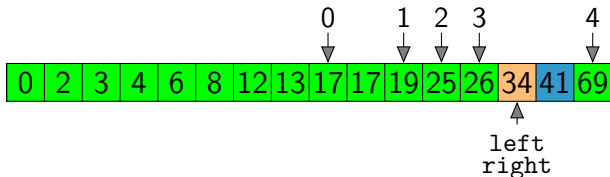
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```


Quicksort – Algorithmus und Animation



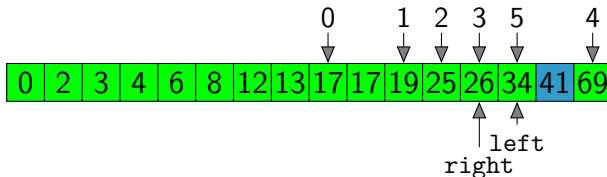
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



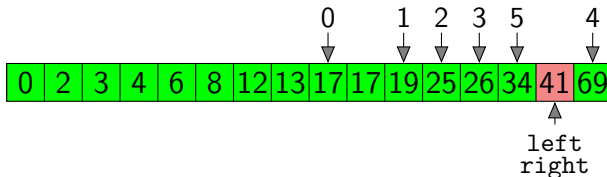
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



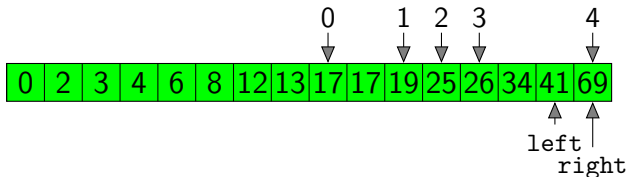
```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation



```
1 void quickSort(int E[], int left, int right) {  
2   if (left < right) {  
3     int i = partition(E, left, right);  
4     // i ist Position des Split-punktes (Pivot)  
5     quickSort(E, left, i - 1); // sortiere den linken Teil  
6     quickSort(E, i + 1, right); // sortiere den rechten Teil  
7   }  
8 }
```

Quicksort – Algorithmus und Animation

0	2	3	4	6	8	12	13	17	17	19	25	26	34	41	69
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

```
1 void quickSort(int E[], int left, int right) {
2   if (left < right) {
3     int i = partition(E, left, right);
4     // i ist Position des Split-punktes (Pivot)
5     quickSort(E, left, i - 1); // sortiere den linken Teil
6     quickSort(E, i + 1, right); // sortiere den rechten Teil
7   }
8 }
```

Quicksort – Platzbedarf

Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus.

Quicksort – Platzbedarf

Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus. – **Nicht ganz:**

Quicksort – Platzbedarf

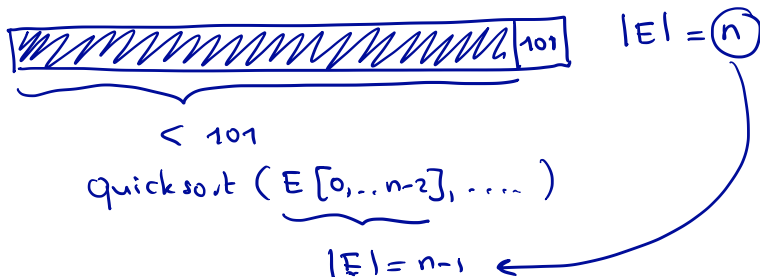
Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus. – **Nicht ganz:**

- ▶ Die rekursiven Aufrufe benötigen Speicherplatz für alle left und right Parameter.

Quicksort – Platzbedarf

Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus. – **Nicht ganz:**

- ▶ Die rekursiven Aufrufe benötigen Speicherplatz für alle `left` und `right` Parameter.
- ▶ Im Worst-Case wird durch die Partitionierung nur ein Element abgespalten.



Quicksort – Platzbedarf

Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus. – **Nicht ganz:**

- ▶ Die rekursiven Aufrufe benötigen Speicherplatz für alle `left` und `right` Parameter.
- ▶ Im Worst-Case wird durch die Partitionierung nur ein Element abgespalten.
- ▶ Dann wird für diese n Elemente $\Theta(n)$ Platz auf dem Stack benötigt.

Quicksort – Platzbedarf

Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus. – **Nicht ganz:**

- ▶ Die rekursiven Aufrufe benötigen Speicherplatz für alle `left` und `right` Parameter.
- ▶ Im Worst-Case wird durch die Partitionierung nur ein Element abgespalten.
- ▶ Dann wird für diese n Elemente $\Theta(n)$ Platz auf dem Stack benötigt.
- ▶ Man kann den Platzbedarf aber auf $\Theta(\log(n))$ reduzieren (siehe Aufgabe 7-4 im Buch).

Quicksort – Platzbedarf

Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus. – **Nicht ganz:**

- ▶ Die rekursiven Aufrufe benötigen Speicherplatz für alle `left` und `right` Parameter.
- ▶ Im Worst-Case wird durch die Partitionierung nur ein Element abgespalten.
- ▶ Dann wird für diese n Elemente $\Theta(n)$ Platz auf dem Stack benötigt.
- ▶ Man kann den Platzbedarf aber auf $\Theta(\log(n))$ reduzieren (siehe Aufgabe 7-4 im Buch).
- ▶ Hauptidee: sortiere nur das größte Teilarray rekursiv, die kleineren iterativ.

Quicksort – Platzbedarf

Auf den ersten Blick sieht Quicksort nach einem **in-place** Sortieralgorithmus aus. – **Nicht ganz:**

- ▶ Die rekursiven Aufrufe benötigen Speicherplatz für alle `left` und `right` Parameter.
- ▶ Im Worst-Case wird durch die Partitionierung nur ein Element abgespalten.
- ▶ Dann wird für diese n Elemente $\Theta(n)$ Platz auf dem Stack benötigt.
- ▶ Man kann den Platzbedarf aber auf $\Theta(\log(n))$ reduzieren (siehe Aufgabe 7-4 im Buch).
- ▶ Hauptidee: sortiere nur das größte Teilarray rekursiv, die kleineren iterativ.

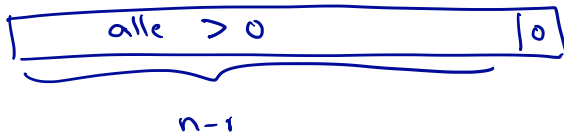
Theorem

Die Platzkomplexität von Quicksort ist in $\Theta(\log(n))$.

Quicksort – Worst-Case Analyse

Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.



Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
- ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:

Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
- ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:
- ▶ Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.

Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
- ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:
- ▶ Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.
- ▶ Das passiert etwa, wenn das Array bereits auf- oder absteigend sortiert ist.

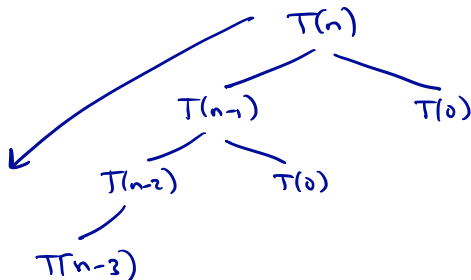
$$T(n) = \underbrace{T(n-1)} + \underbrace{T(0)} + \underbrace{c \cdot n}$$

→ Kosten der Partitionierung



Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
 - ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:
 - ▶ Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.
 - ▶ Das passiert etwa, wenn das Array bereits auf- oder absteigend sortiert ist.
- ⇒ Der Rekursionsbaum für Quicksort enthält **$n-1$** Ebenen.



Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
- ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:
- ▶ Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.
- ▶ Das passiert etwa, wenn das Array bereits auf- oder absteigend sortiert ist.

⇒ Der Rekursionsbaum für Quicksort enthält **$n-1$** Ebenen.

- ▶ Man erhält:
$$W(n) = \sum_{i=1}^n (i-1) = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$$

Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
- ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:
- ▶ Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.
- ▶ Das passiert etwa, wenn das Array bereits auf- oder absteigend sortiert ist.

⇒ Der Rekursionsbaum für Quicksort enthält **$n-1$** Ebenen.

- ▶ Man erhält:
$$W(n) = \sum_{i=1}^n (i-1) = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$$
- ▶ Das ist aber genauso schlecht wie Insertionsort, Bubblesort, usw.

Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
- ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:
- ▶ Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.
- ▶ Das passiert etwa, wenn das Array bereits auf- oder absteigend sortiert ist.

⇒ Der Rekursionsbaum für Quicksort enthält **$n-1$** Ebenen.

- ▶ Man erhält:
$$W(n) = \sum_{i=1}^n (i-1) = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$$
- ▶ Das ist aber genauso schlecht wie Insertionsort, Bubblesort, usw.
- ▶ Wenn das Array bereits aufsteigend sortiert ist, braucht Insertionsort nur $O(n)$.

Quicksort – Worst-Case Analyse

- ▶ Im Worst-Case wird als Pivot **immer** das kleinste oder größte Element im Array genommen.
- ▶ Dadurch ist das Partitionieren **maximal unbalanciert**:
- ▶ Ein Teil ist leer, der andere enthält alle verbleibenden Elemente.
- ▶ Das passiert etwa, wenn das Array bereits auf- oder absteigend sortiert ist.

⇒ Der Rekursionsbaum für Quicksort enthält **$n-1$** Ebenen.

- ▶ Man erhält:
$$W(n) = \sum_{i=1}^n (i-1) = \frac{n \cdot (n-1)}{2} \in \Theta(n^2)$$
- ▶ Das ist aber genauso schlecht wie Insertionsort, Bubblesort, usw.
- ▶ Wenn das Array bereits aufsteigend sortiert ist, braucht Insertionsort nur $O(n)$.

Theorem

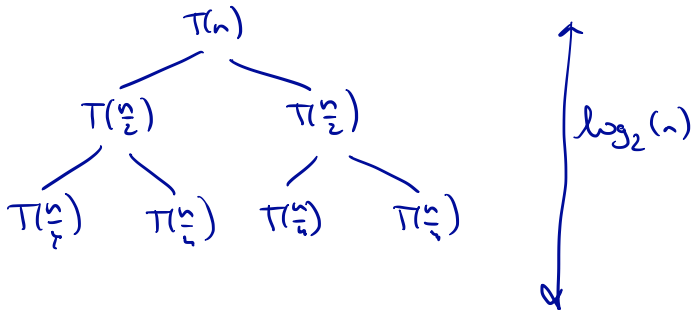
Die Worst-Case Laufzeit von Quicksort ist in $\Theta(n^2)$.

Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.

Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.
- ▶ Wenn wir also das Array mit n Elementen in zwei der Größe $n/2$ teilen, so erhalten wir $\log_2(n)$ Ebenen im Rekursionsbaum.



Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.
- ▶ Wenn wir also das Array mit n Elementen in zwei der Größe $n/2$ teilen, so erhalten wir $\log_2(n)$ Ebenen im Rekursionsbaum.
- ▶ Die Partitionierung hat eine lineare Zeitkomplexität, d. h. jede Ebene braucht $O(n)$ Zeit.

Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.
- ▶ Wenn wir also das Array mit n Elementen in zwei der Größe $n/2$ teilen, so erhalten wir $\log_2(n)$ Ebenen im Rekursionsbaum.
- ▶ Die Partitionierung hat eine lineare Zeitkomplexität, d. h. jede Ebene braucht $O(n)$ Zeit.
- ▶ Man erhält: $T(n) = 2 \cdot T(n/2) + k \cdot n$ für $n > 1$ mit $T(1) = 1$.

Master theorem :

$$b = 2 \quad E = \frac{\log b}{\log c} = 1$$
$$c = 2$$

$$f(n) = k \cdot n \in \Theta(n^E)$$

→ 2. Fall anwendbar → $T(n) \in \Theta(n^E \cdot \log n)$

Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.
- ▶ Wenn wir also das Array mit n Elementen in zwei der Größe $n/2$ teilen, so erhalten wir **$\log_2(n)$** Ebenen im Rekursionsbaum.
- ▶ Die Partitionierung hat eine lineare Zeitkomplexität, d. h. jede Ebene braucht $O(n)$ Zeit.
- ▶ Man erhält: **$T(n) = 2 \cdot T(n/2) + c \cdot n$ für $n > 1$ mit $T(1) = 1$.**
- ▶ Anwendung des Mastertheorems liefert: **$T(n) \in \Theta(n \cdot \log(n))$.**

Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.
- ▶ Wenn wir also das Array mit n Elementen in zwei der Größe $n/2$ teilen, so erhalten wir $\log_2(n)$ Ebenen im Rekursionsbaum.
- ▶ Die Partitionierung hat eine lineare Zeitkomplexität, d. h. jede Ebene braucht $O(n)$ Zeit.
- ▶ Man erhält: $T(n) = 2 \cdot T(n/2) + c \cdot n$ für $n > 1$ mit $T(1) = 1$.
- ▶ Anwendung des Mastertheorems liefert: $T(n) \in \Theta(n \cdot \log(n))$.

Die Ausbalanciertheit der beiden Hälften der Zerlegung in jeder Rekursionsstufe erzeugt also einen **asymptotisch schnelleren** Algorithmus.

Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.
- ▶ Wenn wir also das Array mit n Elementen in zwei der Größe $n/2$ teilen, so erhalten wir $\log_2(n)$ Ebenen im Rekursionsbaum.
- ▶ Die Partitionierung hat eine lineare Zeitkomplexität, d. h. jede Ebene braucht $O(n)$ Zeit.
- ▶ Man erhält: $T(n) = 2 \cdot T(n/2) + c \cdot n$ für $n > 1$ mit $T(1) = 1$.
- ▶ Anwendung des Mastertheorems liefert: $T(n) \in \Theta(n \cdot \log(n))$.

Die Ausbalanciertheit der beiden Hälften der Zerlegung in jeder Rekursionsstufe erzeugt also einen **asymptotisch schnelleren** Algorithmus.

Fazit: Wenn man eine Aufgabe zerlegt, ist es am besten, sie in gleich große Teile zu teilen.

Quicksort – Best-Case Analyse

- ▶ Divide-and-conquer funktioniert besonders gut, wenn die Aufteilung so **gleichmäßig wie möglich** geschieht.
- ▶ Wenn wir also das Array mit n Elementen in zwei der Größe $n/2$ teilen, so erhalten wir $\log_2(n)$ Ebenen im Rekursionsbaum.
- ▶ Die Partitionierung hat eine lineare Zeitkomplexität, d. h. jede Ebene braucht $O(n)$ Zeit.
- ▶ Man erhält: $T(n) = 2 \cdot T(n/2) + c \cdot n$ für $n > 1$ mit $T(1) = 1$.
- ▶ Anwendung des Mastertheorems liefert: $T(n) \in \Theta(n \cdot \log(n))$.

Die Ausbalanciertheit der beiden Hälften der Zerlegung in jeder Rekursionsstufe erzeugt also einen **asymptotisch schnelleren** Algorithmus.

Fazit: Wenn man eine Aufgabe zerlegt, ist es am besten, sie in gleich große Teile zu teilen.

Theorem

Die Best-Case Laufzeit von Quicksort ist in $\Theta(n \cdot \log(n))$.

Quicksort – Balancierte Zerlegung

- ▶ Die mittlere Laufzeit von Quicksort ist viel näher an der des besten Falls als an der des schlechtesten Falls.

1 1
B A

1
W

Quicksort – Balancierte Zerlegung

- ▶ Die mittlere Laufzeit von Quicksort ist viel näher an der des besten Falls als an der des schlechtesten Falls.
- ▶ Schlüssel zum Verständnis: wie schlägt sich die Balanciertheit der Zerlegung in der Rekursionsgleichung nieder?

Quicksort – Balancierte Zerlegung

- ▶ Die mittlere Laufzeit von Quicksort ist viel näher an der des besten Falls als an der des schlechtesten Falls.
- ▶ Schlüssel zum Verständnis: wie schlägt sich die Balanciertheit der Zerlegung in der Rekursionsgleichung nieder?
- ▶ Betrachte z. B. eine Zerlegung im Verhältnis 9:1. Dann erhält man für $n > 1$:

$$\underline{T(n)} \leq \underline{T(9n/10)} + \underline{T(n/10)} + \underbrace{c \cdot n}_{\text{Partitionierung}}$$

Diagram illustrating the partitioning process. An array is shown divided into three sections. The first section is labeled '9' and is bracketed below with the text '< Pivot'. The second section is labeled '1' and is bracketed below with the text '≥ Pivot'. The third section is a single element, indicated by an arrow labeled 'Pivot' pointing to it. The entire diagram is drawn in blue ink.

Quicksort – Balancierte Zerlegung

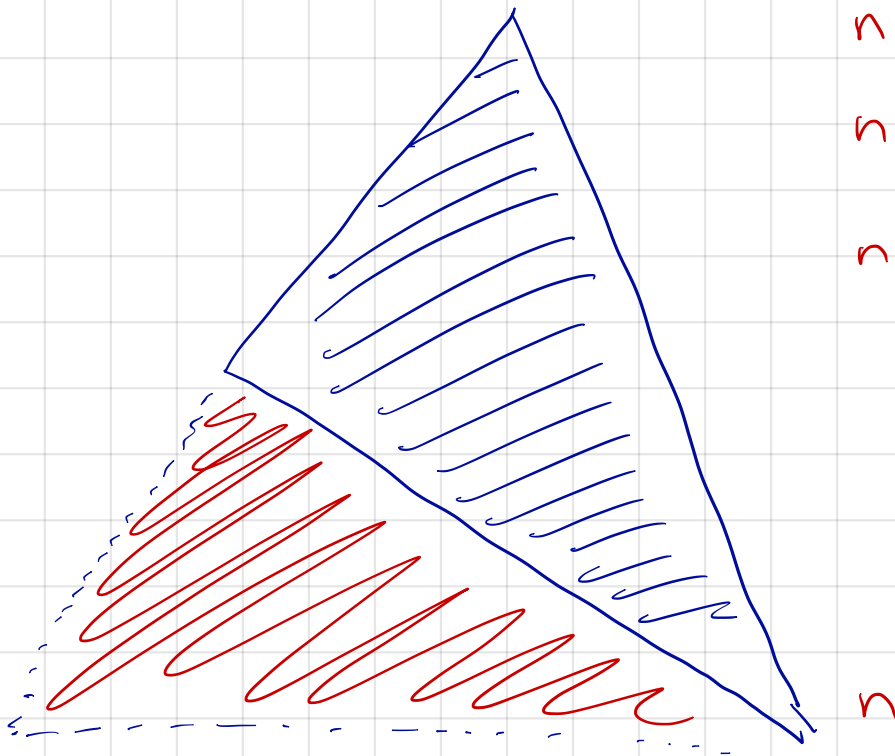
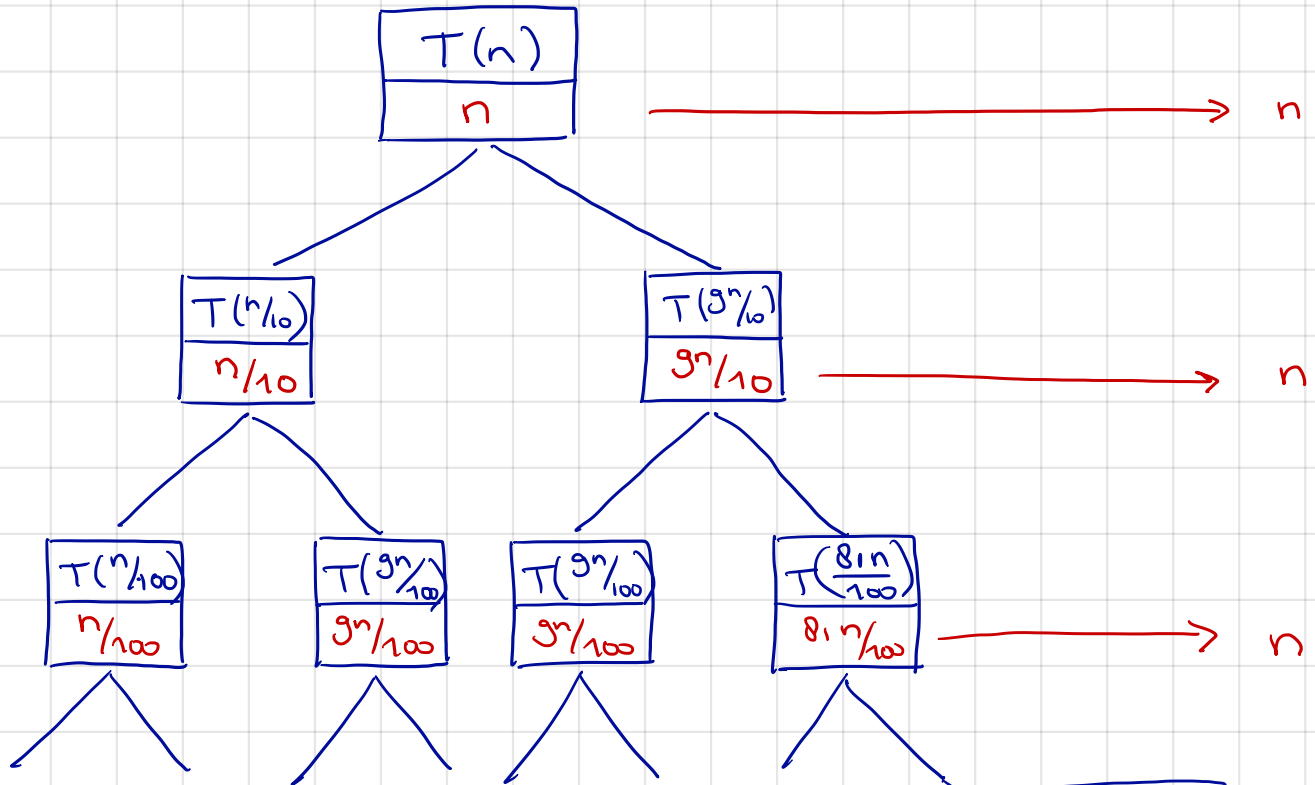
- ▶ Die mittlere Laufzeit von Quicksort ist viel näher an der des besten Falls als an der des schlechtesten Falls.
- ▶ Schlüssel zum Verständnis: wie schlägt sich die Balanciertheit der Zerlegung in der Rekursionsgleichung nieder?
- ▶ Betrachte z. B. eine Zerlegung im Verhältnis 9:1. Dann erhält man für $n > 1$:

$$T(n) \leq T(9n/10) + T(n/10) + c \cdot n$$

- ▶ Rekursionsbaumanalyse liefert: $T(n) \in \mathcal{O}(n \cdot \log(n))$.

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n$$

$\nwarrow k=1$



n
 n
 n

$\log_{10/9} n$

Quicksort – Balancierte Zerlegung

- ▶ Die mittlere Laufzeit von Quicksort ist viel näher an der des besten Falls als an der des schlechtesten Falls.
- ▶ Schlüssel zum Verständnis: wie schlägt sich die Balanciertheit der Zerlegung in der Rekursionsgleichung nieder?
- ▶ Betrachte z. B. eine Zerlegung im Verhältnis 9:1. Dann erhält man für $n > 1$:

$$T(n) \leq T(9n/10) + T(n/10) + c \cdot n$$

- ▶ Rekursionsbaumanalyse liefert: $T(n) \in \mathcal{O}(n \cdot \log(n))$.
- ▶ Diese 9:1 “unbalancierte” Zerlegung liefert asymptotisch die gleiche Zeit wie bei einer Aufteilung zu gleichen Teilen!

Quicksort – Balancierte Zerlegung

- ▶ Die mittlere Laufzeit von Quicksort ist viel näher an der des besten Falls als an der des schlechtesten Falls.
- ▶ Schlüssel zum Verständnis: wie schlägt sich die Balanciertheit der Zerlegung in der Rekursionsgleichung nieder?
- ▶ Betrachte z. B. eine Zerlegung im Verhältnis 9:1. Dann erhält man für $n > 1$:

$$T(n) \leq T(9n/10) + T(n/10) + c \cdot n$$

- ▶ Rekursionsbaumanalyse liefert: $T(n) \in \mathcal{O}(n \cdot \log(n))$.
- ▶ Diese 9:1 “unbalancierte” Zerlegung liefert asymptotisch die gleiche Zeit wie bei einer Aufteilung zu gleichen Teilen!
- ▶ Eine Aufteilung im Verhältnis 99:1 liefert auch: $T(n) \in \mathcal{O}(n \cdot \log(n))$.

Quicksort – Average-Case-Analyse (I)

- ▶ Annahmen:

Quicksort – Average-Case-Analyse (I)

► Annahmen:

1. das Pivotelement kann in $O(1)$ Zeit gewählt werden

Quicksort – Average-Case-Analyse (I)

► Annahmen:

1. das Pivotelement kann in $O(1)$ Zeit gewählt werden
2. alle Elemente im zu sortierenden Array E sind unterschiedlich

Quicksort – Average-Case-Analyse (I)

► Annahmen:

1. das Pivotelement kann in $O(1)$ Zeit gewählt werden
2. alle Elemente im zu sortierenden Array E sind unterschiedlich
3. alle möglichen Permutationen haben die gleiche Wahrscheinlichkeit

Quicksort – Average-Case-Analyse (I)

- ▶ Annahmen:
 1. das Pivotelement kann in $O(1)$ Zeit gewählt werden
 2. alle Elemente im zu sortierenden Array E sind unterschiedlich
 3. alle möglichen Permutationen haben die gleiche Wahrscheinlichkeit
- ▶ Elemente in den Teilarrays sind noch nicht verglichen worden

Quicksort – Average-Case-Analyse (I)

- ▶ Annahmen:
 1. das Pivotelement kann in $O(1)$ Zeit gewählt werden
 2. alle Elemente im zu sortierenden Array E sind unterschiedlich
 3. alle möglichen Permutationen haben die gleiche Wahrscheinlichkeit
- ▶ Elemente in den Teilarrays sind noch nicht verglichen worden
 - ⇒ Permutationen in Teilarrays haben die gleiche Wahrscheinlichkeit

Quicksort – Average-Case-Analyse (I)

► Annahmen:

1. das Pivotelement kann in $O(1)$ Zeit gewählt werden
2. alle Elemente im zu sortierenden Array E sind unterschiedlich
3. alle möglichen Permutationen haben die gleiche Wahrscheinlichkeit

► Elemente in den Teilarrays sind noch nicht verglichen worden

⇒ Permutationen in Teilarrays haben die gleiche Wahrscheinlichkeit

► Partitionierung eines Arrays mit $n-1$ Elemente fordert $n-1$ Vergleiche

Quicksort – Average-Case-Analyse (I)

► Annahmen:

1. das Pivotelement kann in $O(1)$ Zeit gewählt werden
2. alle Elemente im zu sortierenden Array E sind unterschiedlich
3. alle möglichen Permutationen haben die gleiche Wahrscheinlichkeit

► Elemente in den Teilarrays sind noch nicht verglichen worden

⇒ Permutationen in Teilarrays haben die gleiche Wahrscheinlichkeit

► Partitionierung eines Arrays mit $n-1$ Elemente fordert $n-1$ Vergleiche

► Wir erhalten damit für $n > 1$ folgende Rekursionsgleichung:

$$A(n) = \underbrace{n-1} + \sum_{i=0}^{n-1} \Pr\{\text{Pivot endet an Stelle } i\} \cdot \underbrace{(A(i) + A(n-i-1))}_{\text{kosten } qs(i+1, n-1)}$$

wobei $A(0) = A(1) = 0$.

Partitionieren

kosten
quicksort($l, i-1$)
= 0

Quicksort – Average-Case-Analyse (II)

$$A(n) = n - 1 + \sum_{i=0}^{n-1} \frac{1}{n} \cdot (A(i) + A(n - i - 1))$$

$$\quad \mid \quad \sum_{i=0}^{n-1} A(n - i - 1) = A(n - 1) + A(n - 2) + \dots + A(0)$$

$$= n - 1 + \sum_{i=1}^{n-1} \frac{2}{n} \cdot A(i).$$

$$A(n) = n-1 + \sum_{i=0}^{n-1} \underbrace{\Pr \{ \text{Pivot an Position } i \}}_{n \text{ Position}} \cdot (A(i) + A(n-i-1))$$

$$= n-1 + \sum_{i=0}^{n-1} \frac{1}{n} (A(i) + A(n-i-1))$$

$$(+ \sum A(n-i-1) = \underbrace{A(n-1)}_{i=0} + A(n-2) + \dots + \underbrace{A(0)}_{i=n-1} \quad *)$$

$$= \boxed{n-1 + \sum_{i=0}^{n-1} \frac{2}{n} A(i)} \quad (*)$$

Zum loswerden von $\sum A(i)$ mache folgendes:

$$n \cdot A(n) - \underline{(n-1) A(n-1)}$$

$$= n \cdot (n-1) + \sum_{i=0}^{n-1} 2 A(i) - (n-1)(n-2) - \sum_{i=0}^{n-2} 2 A(i)$$

$$= n \cdot (n-1) - (n-1)(n-2) + 2 A(n-1)$$

$$= \underline{2 A(n-1) + 2(n-1)}$$

Also: $n \cdot A(n) = (n+1) A(n-1) + 2(n-1)$

Teile durch $n \cdot (n+1)$:

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

Sei $\underline{A'(n)} = \frac{A(n)}{n+1}$ $A'(0) = 1$

$$A'(n) = A'(n-1) + \frac{2(n-1)}{n(n+1)}$$

Ergo $A'(n) = \sum_{i=1}^n \frac{2(i-1)}{i(i+1)}$

$$\begin{aligned} \Leftrightarrow A'(n) &= \sum_{i=1}^n \frac{2}{i+1} - \sum_{i=1}^n \frac{2}{i(i+1)} \\ &= 2 \cdot \sum_{i=1}^{n+1} \frac{1}{i} - 2 - \sum_{i=1}^n \frac{2}{i(i+1)} \\ &= 2 \cdot \sum_{i=1}^n \frac{1}{i} + \frac{2}{n+1} - 2 - \underbrace{\sum_{i=1}^n \frac{2}{i(i+1)}}_{\substack{\text{kleinste} \\ \text{Term} \\ \frac{2}{n(n+1)}}} \\ &\approx 2 \cdot \ln(n) \leq 1 \\ &\leq 2 \cdot \ln \cdot n - \frac{2 \cdot \cancel{n}}{\cancel{n}(n+1)} \end{aligned}$$

$$A'(n) \leq 2 \cdot \ln(n) - \frac{2}{n+1}$$

$A'(n) = \frac{A(n)}{n+1}$ d.h. $A(n) = (n+1) \cdot A'(n)$

$$A(n) \leq 2 \cdot (n+1) \cdot \ln(n) - 2$$

Quicksort – Average-Case-Analyse (II)

$$\begin{aligned}
 A(n) &= n - 1 + \sum_{i=0}^{n-1} \frac{1}{n} \cdot (A(i) + A(n - i - 1)) \\
 &\quad \left| \sum_{i=0}^{n-1} A(n - i - 1) = A(n - 1) + A(n - 2) + \dots + A(0) \right. \\
 &= n - 1 + \sum_{i=1}^{n-1} \frac{2}{n} \cdot A(i).
 \end{aligned}$$

Intermezzo: wir wollen $\sum_i A(i)$ loswerden; folgender Trick hilft:

$$\begin{aligned}
 n \cdot A(n) - (n - 1) \cdot A(n - 1) &= 2 \cdot A(n - 1) + 2 \cdot (n - 1) \\
 &\quad \left| \text{teile durch } n \cdot (n + 1) \text{ und setze } A'(n) = A(n)/(n + 1) \right.
 \end{aligned}$$

$$\begin{aligned}
 A'(n) &= A'(n - 1) + \frac{2 \cdot (n - 1)}{n \cdot (n + 1)} \quad \text{mit } A'(0) = 1 \quad \left| \text{Umformen} \right. \\
 &= \sum_{i=1}^n \frac{2 \cdot (i - 1)}{i \cdot (i + 1)}
 \end{aligned}$$

Quicksort – Average-Case-Analyse (III)

$$A'(n) = \sum_{i=1}^n \frac{2 \cdot (i-1)}{i \cdot (i+1)} \quad | \text{ calculus}$$

$$= 2 \cdot \sum_{i=1}^n \frac{1}{i+1} - 2 \cdot \sum_{i=1}^n \frac{1}{i \cdot (i+1)} \quad | \text{ calculus}$$

$$= 2 \cdot \sum_{i=1}^n \frac{1}{i} - 2 + \frac{2}{n+1} - 2 \cdot \sum_{i=1}^n \frac{1}{i \cdot (i+1)} \quad | \text{ harmonische Reihe}$$

$$\leq 2 \cdot \ln(n) - \frac{2n}{n+1} \quad | \quad A'(n) = A(n)/(n+1)$$

$$A(n) \in \mathcal{O}(n \cdot \log(n))$$

Quicksort – Average-Case-Analyse (III)

$$A'(n) = \sum_{i=1}^n \frac{2 \cdot (i-1)}{i \cdot (i+1)} \quad | \text{ calculus}$$

$$= 2 \cdot \sum_{i=1}^n \frac{1}{i+1} - 2 \cdot \sum_{i=1}^n \frac{1}{i \cdot (i+1)} \quad | \text{ calculus}$$

$$= 2 \cdot \sum_{i=1}^n \frac{1}{i} - 2 + \frac{2}{n+1} - 2 \cdot \sum_{i=1}^n \frac{1}{i \cdot (i+1)} \quad | \text{ harmonische Reihe}$$

$$\leq 2 \cdot \ln(n) - \frac{2n}{n+1} \quad | \quad A'(n) = A(n)/(n+1)$$

$$A(n) \in \mathcal{O}(n \cdot \log(n))$$

Da die Best-Case Laufzeit in $\Omega(n \cdot \log(n))$ liegt, folgt folgender Satz:

Theorem

Die mittlere Laufzeit von Quicksort ist in $\Theta(n \cdot \log(n))$.

Übersicht

- 1 Quicksort
 - Das Divide-and-Conquer Paradigma
 - Partitionierung
 - Quicksort Algorithmus
 - Komplexitätsanalyse
- 2 Weitere Sortierverfahren
 - Bubblesort
 - Countingsort
- 3 Vergleich der Sortialgorithmen

Bubblesort - Idee

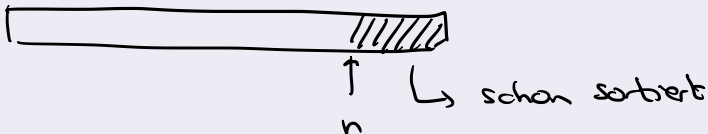
Idee

- ▶ Gehe durch die Liste und vertausche alle nebeneinander stehenden Paare, die in falscher Ordnung stehen.

Bubblesort - Idee

Idee

- ▶ Gehe durch die Liste und vertausche alle nebeneinander stehenden Paare, die in falscher Ordnung stehen.
- ▶ Die Liste rechts von der letzten Tauschposition ist bereits sortiert.



Bubblesort - Idee

Idee

- ▶ Gehe durch die Liste und vertausche alle nebeneinander stehenden Paare, die in falscher Ordnung stehen.
- ▶ Die Liste rechts von der letzten Tauschposition ist bereits sortiert.
- ▶ Wenn keine Änderung, dann fertig.

Bubblesort - Idee

Idee

- ▶ Gehe durch die Liste und vertausche alle nebeneinander stehenden Paare, die in falscher Ordnung stehen.
- ▶ Die Liste rechts von der letzten Tauschposition ist bereits sortiert.
- ▶ Wenn keine Änderung, dann fertig.
- ▶ Sonst gehe wieder durch die Liste. . .

Bubblesort - Idee

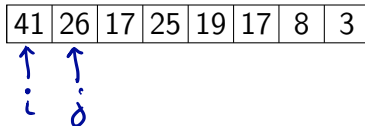
Idee

- ▶ Gehe durch die Liste und vertausche alle nebeneinander stehenden Paare, die in falscher Ordnung stehen.
- ▶ Die Liste rechts von der letzten Tauschposition ist bereits sortiert.
- ▶ Wenn keine Änderung, dann fertig.
- ▶ Sonst gehe wieder durch die Liste. . .

Beim Sortieren steigen/sinken die Elemente zu ihrem Platz in der Ordnung wie Blasen im Wasser auf (daher der Name).

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

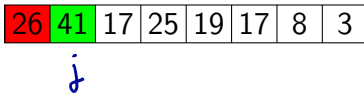
```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```

41	26	17	25	19	17	8	3
----	----	----	----	----	----	---	---

i j

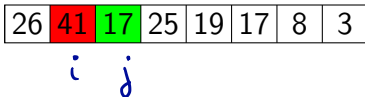
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



26	17	41	25	19	17	8	3
----	----	----	----	----	----	---	---

j

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

26	17	41	25	19	17	8	3
----	----	----	----	----	----	---	---

i
 j

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



26	17	25	41	19	17	8	3
----	----	----	----	----	----	---	---

i

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```

26	17	25	41	19	17	8	3
----	----	----	----	----	----	---	---

j

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

26	17	25	19	41	17	8	3
----	----	----	----	----	----	---	---

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

26	17	25	19	41	17	8	3
----	----	----	----	----	----	---	---

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

26	17	25	19	17	41	8	3
----	----	----	----	----	----	---	---

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

26	17	25	19	17	41	8	3
----	----	----	----	----	----	---	---

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

26	17	25	19	17	8	41	3
----	----	----	----	----	---	----	---

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```

26	17	25	19	17	8	41	3
----	----	----	----	----	---	----	---

i

n

Bubblesort

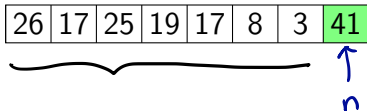
```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    * n = j;  
11  }  
12 }
```

26	17	25	19	17	8	3	41
----	----	----	----	----	---	---	----

↑ ↑
j n

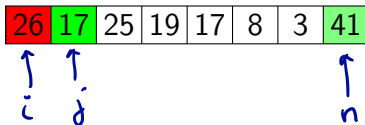
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



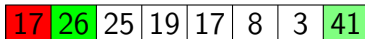
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     * int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       { if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	26	25	19	17	8	3	41
----	----	----	----	----	---	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	25	26	19	17	8	3	41
----	----	----	----	----	---	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	25	26	19	17	8	3	41
----	----	----	----	----	---	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	25	19	26	17	8	3	41
----	----	----	----	----	---	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	25	19	26	17	8	3	41
----	----	----	----	----	---	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	25	19	17	26	8	3	41
----	----	----	----	----	---	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	25	19	17	26	8	3	41
----	----	----	----	----	---	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	25	19	17	8	26	3	41
----	----	----	----	---	----	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	25	19	17	8	26	3	41
----	----	----	----	---	----	---	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```

17	25	19	17	8	3	26	41
----	----	----	----	---	---	----	----

↑ ↑
j n

Bubblesort

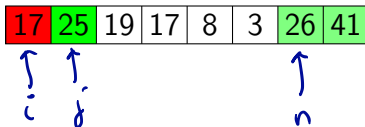
```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```

17	25	19	17	8	3	26	41
----	----	----	----	---	---	----	----



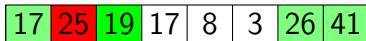
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



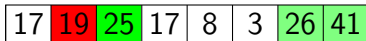
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	19	25	17	8	3	26	41
----	----	----	----	---	---	----	----



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	19	17	25	8	3	26	41
----	----	----	----	---	---	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	19	17	25	8	3	26	41
----	----	----	----	---	---	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	19	17	8	25	3	26	41
----	----	----	---	----	---	----	----

Bubblesort

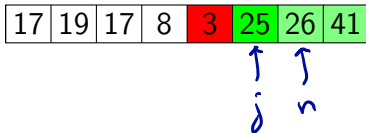
```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```

17	19	17	8	25	3	26	41
----	----	----	---	----	---	----	----

↑
i, j

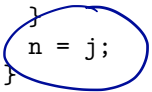
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2   while (n > 1) {  
3     int j = 1;  
4     for (int i = 0; i < n - 1; i++) {  
5       if (E[i] > E[i+1]){  
6         swap(E[i], E[i+1]);  
7         j = i + 1;  
8       }  
9     }  
10    n = j;  
11  }  
12 }
```

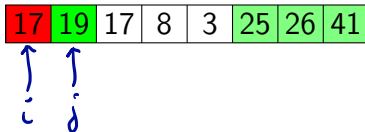


17	19	17	8	3	25	26	41
----	----	----	---	---	----	----	----

↑
 n, j

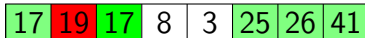
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	17	19	8	3	25	26	41
----	----	----	---	---	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	17	19	8	3	25	26	41
----	----	----	---	---	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	17	8	19	3	25	26	41
----	----	---	----	---	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	17	8	19	3	25	26	41
----	----	---	----	---	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	17	8	3	19	25	26	41
----	----	---	---	----	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	17	8	3	19	25	26	41
----	----	---	---	----	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	8	17	3	19	25	26	41
----	---	----	---	----	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	8	17	3	19	25	26	41
----	---	----	---	----	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	8	3	17	19	25	26	41
----	---	---	----	----	----	----	----

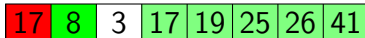
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

17	8	3	17	19	25	26	41
----	---	---	----	----	----	----	----

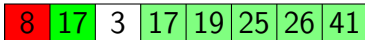
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

8	17	3	17	19	25	26	41
---	----	---	----	----	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

8	3	17	17	19	25	26	41
---	---	----	----	----	----	----	----

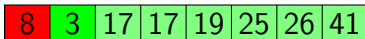
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

8	3	17	17	19	25	26	41
---	---	----	----	----	----	----	----

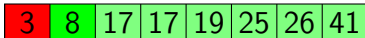
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



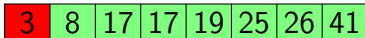
Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

3	8	17	17	19	25	26	41
---	---	----	----	----	----	----	----

Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```



Bubblesort

```
1 void bubbleSort(int E[], int n) {  
2     while (n > 1) {  
3         int j = 1;  
4         for (int i = 0; i < n - 1; i++) {  
5             if (E[i] > E[i+1]){  
6                 swap(E[i], E[i+1]);  
7                 j = i + 1;  
8             }  
9         }  
10        n = j;  
11    }  
12 }
```

3	8	17	17	19	25	26	41
---	---	----	----	----	----	----	----

Countingsort

Countingsort: Idee

- ▶ **Annahme:** Die Schlüssel aller Elemente des Arrays E sind aus $\{0, \dots, k\}$ für ein bekanntes k .
- ▶ **Idee:** Zähle wie oft die einzelnen Schlüssel auftreten und berechne daraus die Positionen der Arrayelemente nach dem Sortieren.
- ▶ Benötigt über Vergleich/Kopieren hinaus weitere **Berechnungen**.
- ▶ Benötigt zusätzlichen **Speicher**.

Countingsort

= E.length

```

1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   [for (int i=0; i<nrOfKeys; i++)      initialisierung
4     index[i] = 0;
5   [for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
	0	1	2	3	4						
index	0	0	0	0	0						
sorted	0	0	0	0	0	0	0	0	0	0	0

k = 4

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	0	0	0	0
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---

Countingsort

```

1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
	0	1	2	3	4						
index	0	0	0	0	0						
sorted	0	0	0	0	0	0	0	0	0	0	0

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
				3							
index	0	1	0	0	0						
sorted	0	0	0	0	0	0	0	0	0	0	0

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	1	0	1	0
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	1	1	1	0
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	1	1	1	0
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

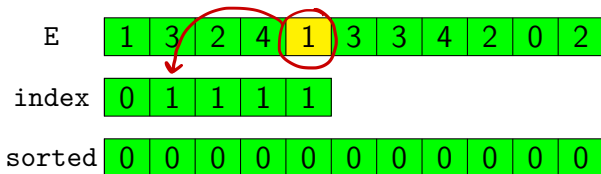
0	1	1	1	1
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
index	0	2	1	1	1						
sorted	0	0	0	0	0	0	0	0	0	0	0

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	2	1	2	1
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	2	1	3	1
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	2	1	3	1
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
index	0	2	1	3	2						
sorted	0	0	0	0	0	0	0	0	0	0	0

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
index	0	2	1	3	2						
sorted	0	0	0	0	0	0	0	0	0	0	0

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index

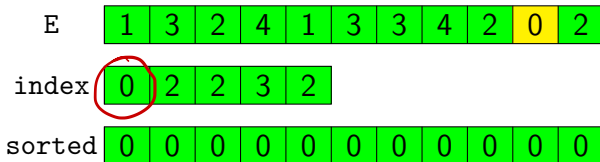


sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

1	2	2	3	2
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
index	1	2	2	3	2						
sorted	0	0	0	0	0	0	0	0	0	0	0

Countingsort

```

1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }

```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---



index

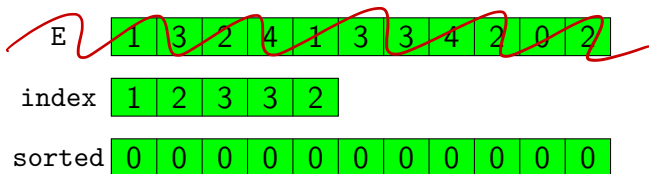
1	2	3	3	2
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

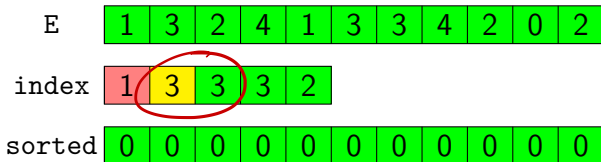
1	2	3	3	2
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index

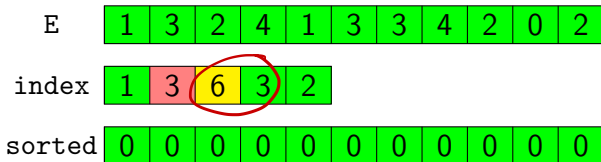


sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```



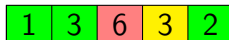
Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
index	1	3	6	9	2						
sorted	0	0	0	0	0	0	0	0	0	0	0

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

1	3	6	9	2
---	---	---	---	---

sorted

0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

1	3	6	9	11
---	---	---	---	----

sorted

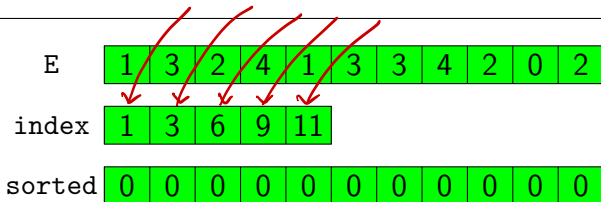
0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```

1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }

```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10     sorted[index[E[j]]-1] = E[j];
11     index[E[j]]--;
12   }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

1	3	6	9	11
---	---	---	---	----

sorted

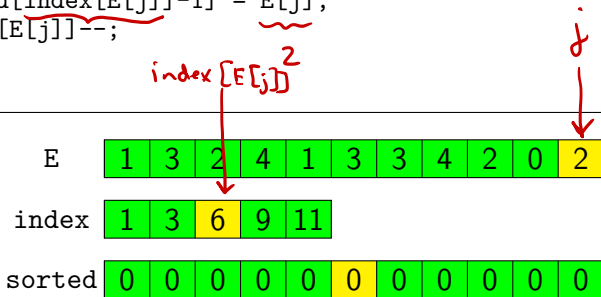
0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```

1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }

```

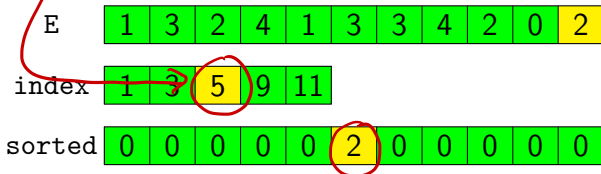


Countingsort

```

1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }

```

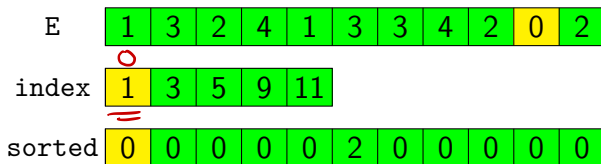


Countingsort

```

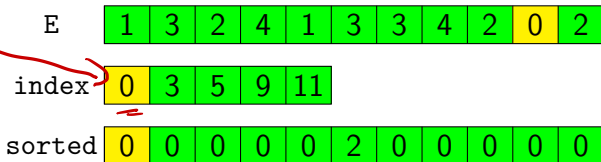
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }

```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

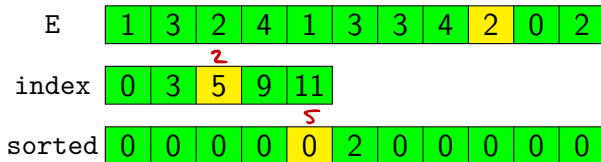


Countingsort

```

1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }

```



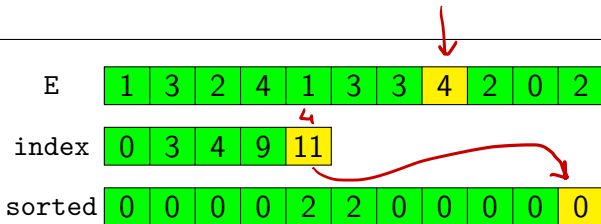
Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E	1	3	2	4	1	3	3	4	2	0	2
index	0	3	4	9	11						
sorted	0	0	0	0	2	2	0	0	0	0	0

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

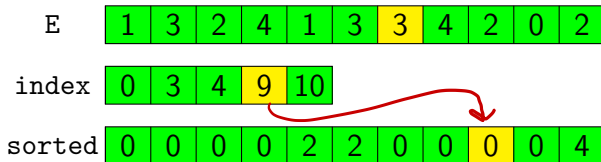
0	3	4	9	10
---	---	---	---	----

sorted

0	0	0	0	2	2	0	0	0	0	4
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

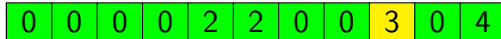
E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

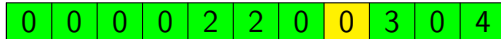
E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index



sorted

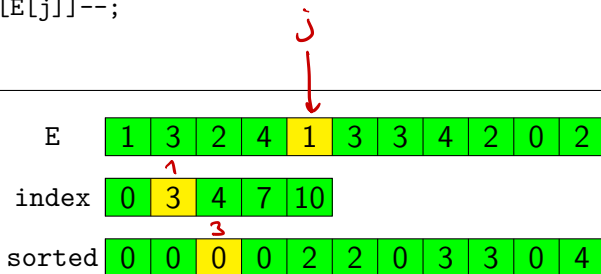


Countingsort

```

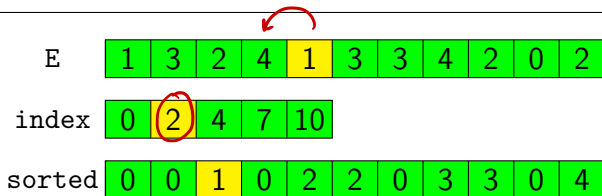
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }

```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	2	4	7	10
---	---	---	---	----

sorted

0	0	1	0	2	2	0	3	3	0	4
---	---	---	---	---	---	---	---	---	---	---

Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

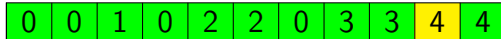
E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

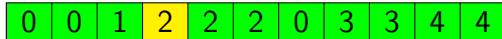
E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

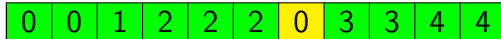
E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

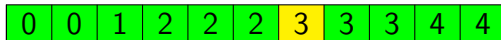
E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

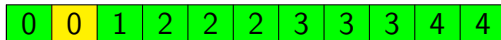
E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

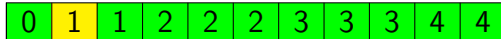
E



index



sorted



Countingsort

```
1 void countingSort(int[] E, int E.length, int nrOfKeys){
2   int sorted[lengthOfE], index[nrOfKeys];
3   for (int i=0; i<nrOfKeys; i++)
4     index[i] = 0;
5   for (int j=0; j<E.length; j++) //index[i] = Anzahl Key i in E
6     index[E[j]]++;
7   for (int i=1; i<nrOfKeys; i++) //index[i] = Anzahl Key <= i in E
8     index[i] += index[i-1];
9   for (j=E.length-1; j>=0; j--){
10    sorted[index[E[j]]-1] = E[j];
11    index[E[j]]--;
12  }
13 }
```

E

1	3	2	4	1	3	3	4	2	0	2
---	---	---	---	---	---	---	---	---	---	---

index

0	1	3	6	9
---	---	---	---	---

sorted

0	1	1	2	2	2	3	3	3	4	4
---	---	---	---	---	---	---	---	---	---	---

Countingsort - Komplexitätsanalyse

Komplexität von Countingsort

- ▶ Laufzeit in $\mathcal{O}(n + \text{number of keys})$
- ▶ Speicherbedarf in $\mathcal{O}(n + \text{number of keys})$
- ▶ Stabil

Übersicht

- 1 Quicksort
 - Das Divide-and-Conquer Paradigma
 - Partitionierung
 - Quicksort Algorithmus
 - Komplexitätsanalyse
- 2 Weitere Sortierverfahren
 - Bubblesort
 - Countingsort
- 3 Vergleich der Sortieralgorithmen

Komplexität von Sortieralgorithmen

	<i>Worst-Case</i>	<i>Average-Case</i>	<i>Platzbedarf</i>	<i>Stabil</i>
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	in-place	J
Bubblesort	$\Theta(n^2)$	$\Theta(n^2)$	in-place	J
Quicksort	$\Theta(n^2)$	$\Theta(n \cdot \log(n))$	$\Theta(\log(n))$	N*
Mergesort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	$\Theta(n)$	J
Heapsort	$\Theta(n \cdot \log(n))$	$\Theta(n \cdot \log(n))$	in-place	N
Countingsort [†]	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	J

* es gibt Varianten die stabil sind.

† benötigt endlichen Wertebereich (k ist Anzahl versch. Werte)

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für kleinere Arrays und fast sortierte Eingaben.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierv Verfahren und wird u. a. benutzt in Perl, Python und Java.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.
- ▶ Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log(n))$ braucht für fast sortierte Eingaben.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.
- ▶ Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log(n))$ braucht für fast sortierte Eingaben.
- ▶ Quicksort ist typischerweise ein effizientes Verfahren.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.
- ▶ Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log(n))$ braucht für fast sortierte Eingaben.
- ▶ Quicksort ist typischerweise ein effizientes Verfahren. Die Wahl des Pivots ist wichtig.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.
- ▶ Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log(n))$ braucht für fast sortierte Eingaben.
- ▶ Quicksort ist typischerweise ein effizientes Verfahren. Die Wahl des Pivots ist wichtig. Nicht stabil, und nicht so effizient auf fast sortierten Eingaben.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.
- ▶ Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log(n))$ braucht für fast sortierte Eingaben.
- ▶ Quicksort ist typischerweise ein effizientes Verfahren. Die Wahl des Pivots ist wichtig. Nicht stabil, und nicht so effizient auf fast sortierten Eingaben.
- ▶ Einige Variationen:

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.
- ▶ Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log(n))$ braucht für fast sortierte Eingaben.
- ▶ Quicksort ist typischerweise ein effizientes Verfahren. Die Wahl des Pivots ist wichtig. Nicht stabil, und nicht so effizient auf fast sortierten Eingaben.
- ▶ Einige Variationen:
 - ▶ **Introsort**: setzt Quicksort ein bis zu einer gewissen Rekursionstiefe und benutzt dann Heapsort.

Einige Bemerkungen

- ▶ Insertion Sort ist einfach und ziemlich effizient für **kleinere** Arrays und **fast sortierte** Eingaben. Wird häufig benutzt als Unterprogramm für andere Sortieralgorithmen für kleinere Instanzen.
- ▶ Mergesort ist ein sehr effizientes Sortierverfahren und wird u. a. benutzt in Perl, Python und Java. Ist leicht anpassbar für Listen und ist stabil.
- ▶ Heapsort ist ein sehr effizientes Sortierverfahren, was jedoch schwieriger auf Listen anzupassen ist und $\mathcal{O}(n \cdot \log(n))$ braucht für fast sortierte Eingaben.
- ▶ Quicksort ist typischerweise ein effizientes Verfahren. Die Wahl des Pivots ist wichtig. Nicht stabil, und nicht so effizient auf fast sortierten Eingaben.
- ▶ Einige Variationen:
 - ▶ **Introsort**: setzt Quicksort ein bis zu einer gewissen Rekursionstiefe und benutzt dann Heapsort.
 - ▶ **Smoothsort**: (komplizierte) Variation von Heapsort die fast $\mathcal{O}(n)$ braucht für fast sortierten Eingaben.

Nächste Vorlesung

Nächste Vorlesung

Montag 28. Mai, 08:30 (Hörsaal H01). Bis dann!

Schöne Pfingsttage!