Datenstrukturen und Algorithmen

Vorlesung 9: Quicksort (K7)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

`http://www-i2.rwth-aachen.de/i2/dsal10/`

18. Mai 2010

Übersicht

Übersicht

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.

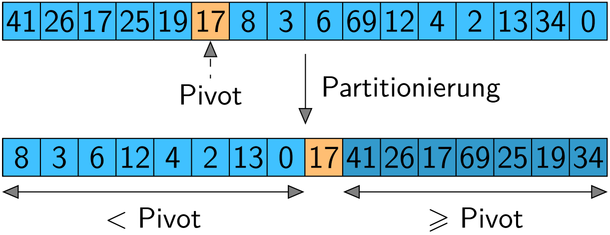
Quicksort – Idee

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

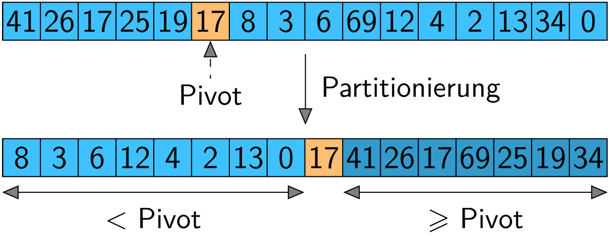
Bei Quicksort werden die Elemente zuerst auf die richtige Seite („Hälfte“) des Arrays gebracht, dann wird jeweils rekursiv sortiert.

Quicksort wurde 1961 von Tony Hoare (Großbritanien) entwickelt.

Quicksort – Strategie

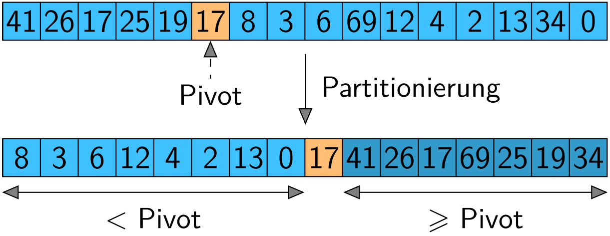


Quicksort – Strategie



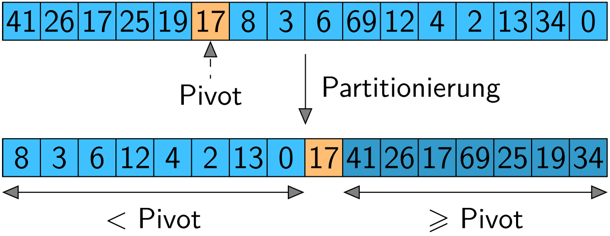
Teile Wähle ein Pivotelement aus dem zu sortierenden Array

Quicksort – Strategie



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

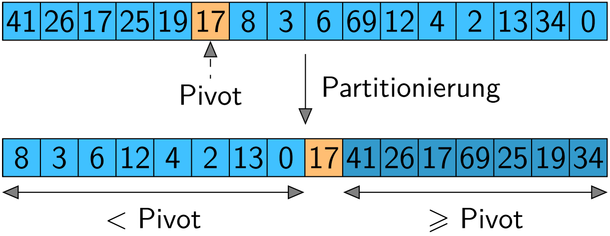
Quicksort – Strategie



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

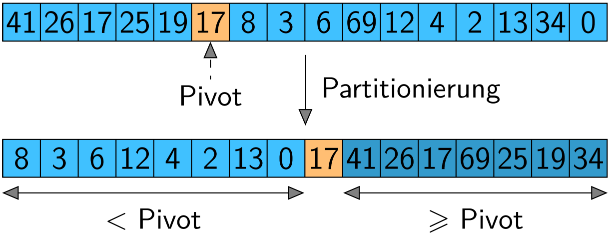


Teile Wähle ein Pivotelement aus dem zu sortierenden Array und partitioniere das zu sortierende Array in zwei Teile auf:

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: „$<$ Pivot“, „$\geqslant$ 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: „$<$ Pivot", „$\geqslant$ Pivot" und „ungeprüft".
- Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $<$ Pivot ist.

Partitionierung (I)

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

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

Partitionierung (I)

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

- Arbeite mit drei Bereichen: „$<$ Pivot“, „$\geqslant$ Pivot“ und „ungeprüft“.
- Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $<$ Pivot ist.
- Schiebe die rechte Grenze nach links, solange das zusäztliche Element $\geqslant$ 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: „$<$ Pivot“, „$\geqslant$ Pivot“ und „ungeprüft“.
- Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $<$ Pivot ist.
- Schiebe die rechte Grenze nach links, solange das zusäztliche Element $\geqslant$ 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: „$<$ Pivot“, „$\geqslant$ Pivot“ und „ungeprüft“.
- Schiebe die linke Grenze nach rechts, solange das zusätzliche Element $<$ Pivot ist.
- Schiebe die rechte Grenze nach links, solange das zusäztliche Element $\geqslant$ 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)

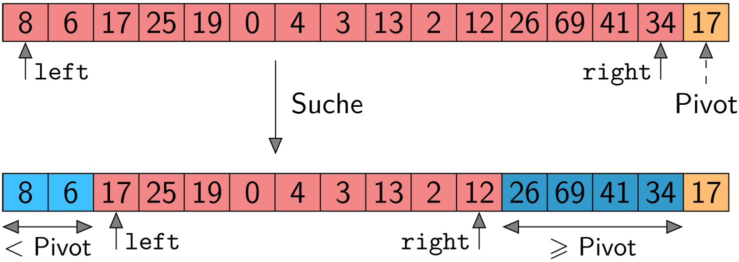
Sobald ein Pivot ausgewählt ist, kann das Array in $O(n)$ partitioniert werden, z. B. folgendermaßen:

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

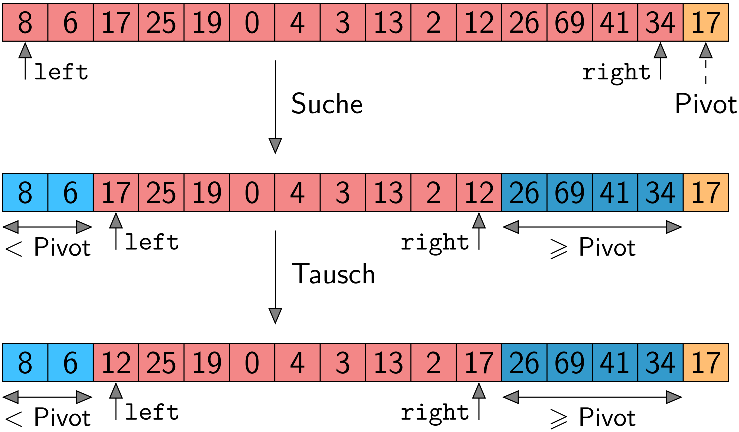
(Es gibt auch andere Verfahren.)

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

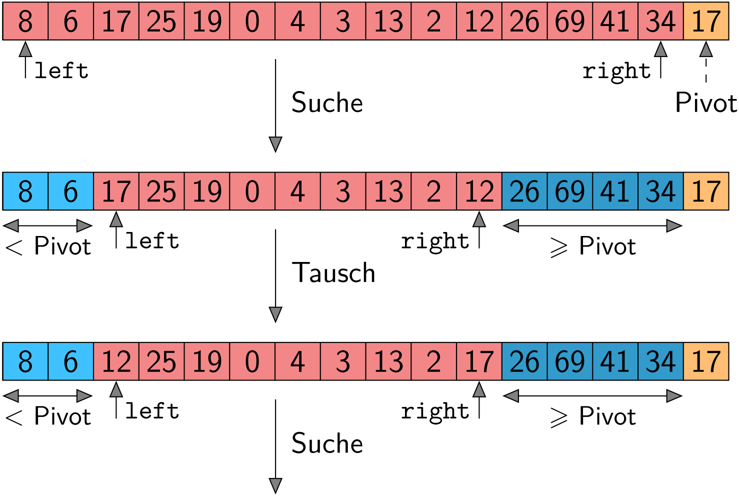
Partitionierung (II)



Partitionierung (II)



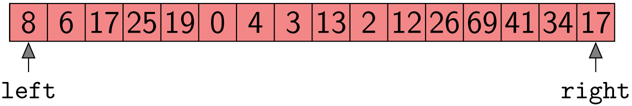
Partitionierung (II)



Partitionierung – Algorithmus

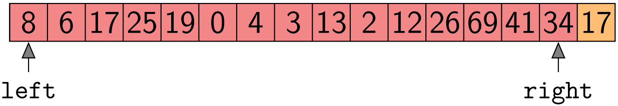
```
int partition(int E[], int left, int right) {
  // Wähle einfaches Pivotelement
  int ppos = right, pivot = E[ppos];
  right--; // Pivot ausgenommen
  while (true) {
    // Bilineare Suche
    while (left < right && E[left] < pivot) left++;
    while (left < right && E[right] >= pivot) right--;
    if (left >= right) {
      break;
    }
    swap(E[left], E[right]);
  }
  if (E[left] < pivot) { // nur bei (left==ppos-1) möglich
    return ppos;
  }
  swap(E[left], E[ppos]);
  return left; // gib neue Pivotposition als Splitpunkt zurück
}
```

Quicksort – Algorithmus und Animation



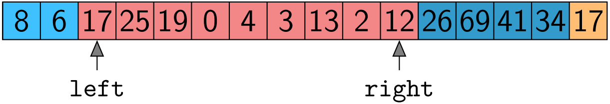

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



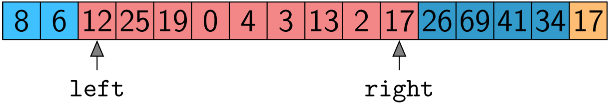

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



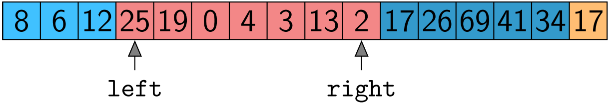

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



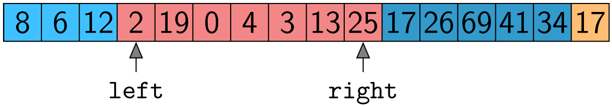

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



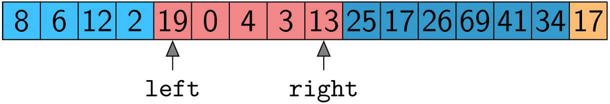

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



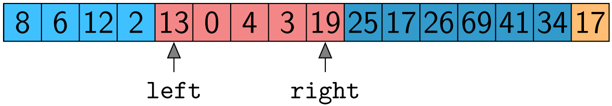

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



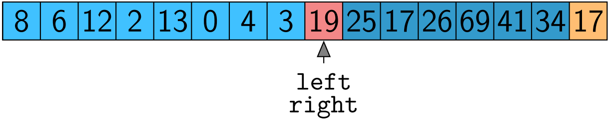

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



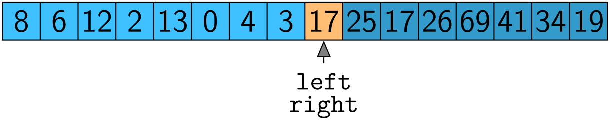

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



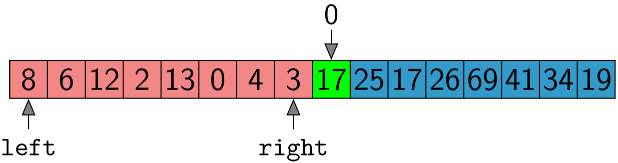

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



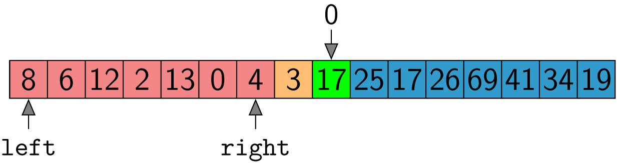

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



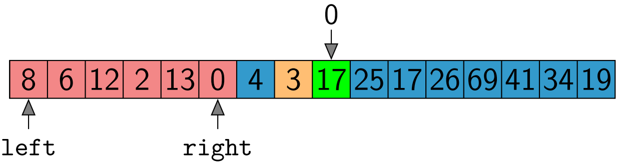

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



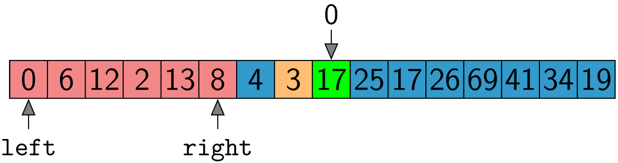

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



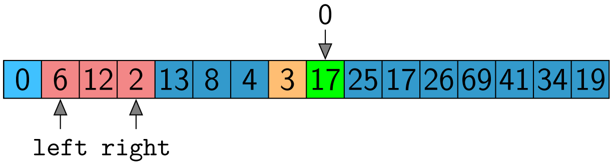

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



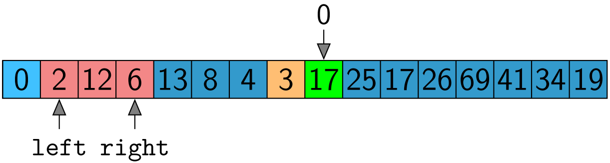

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



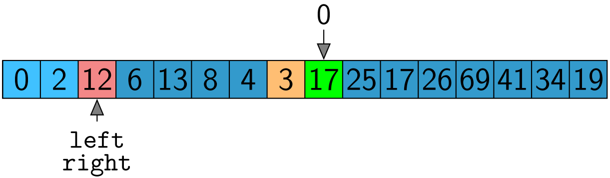

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



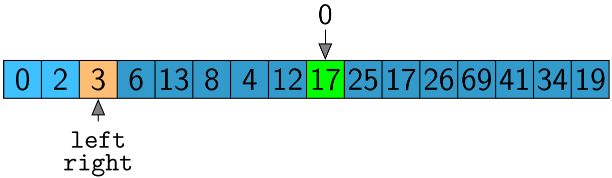

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



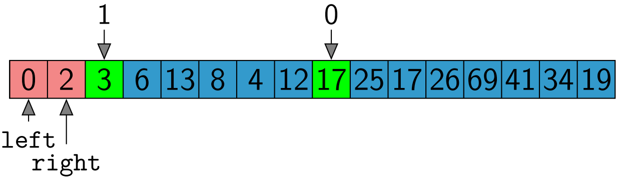

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



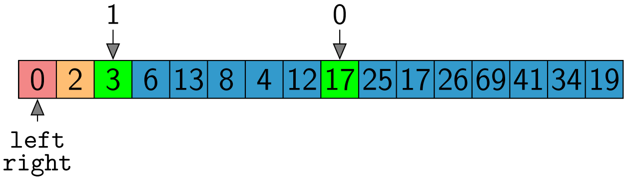

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



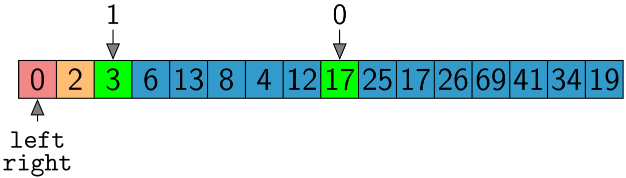

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



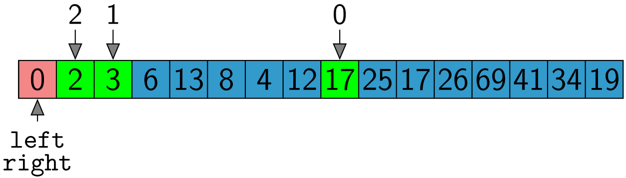

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



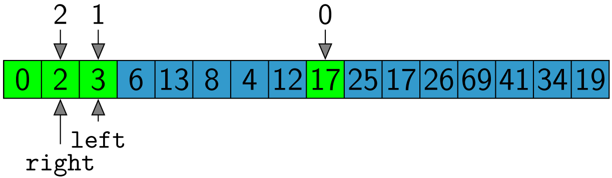

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



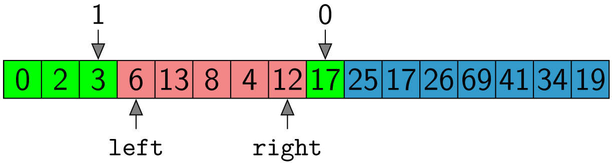

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



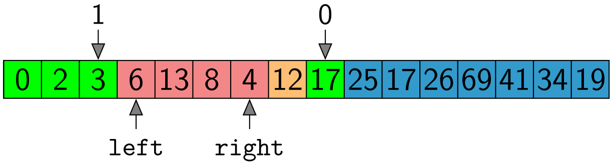

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



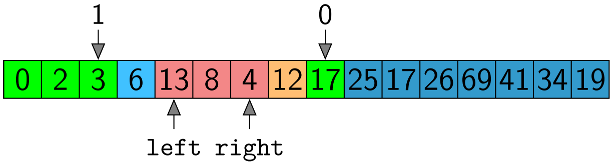

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



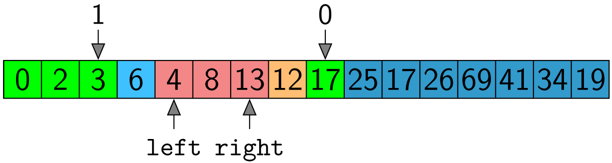

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



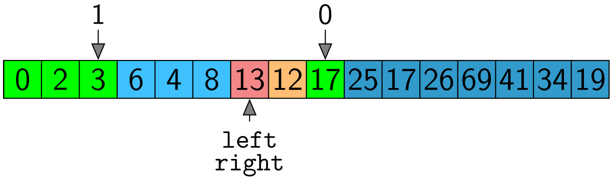

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



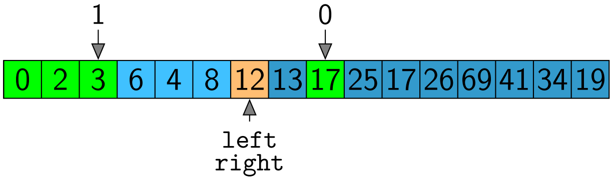

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



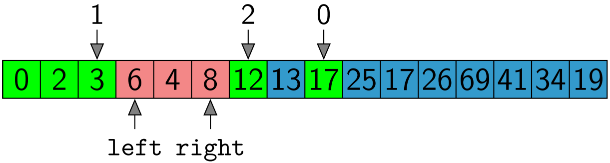

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



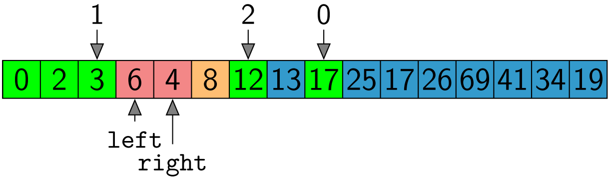

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



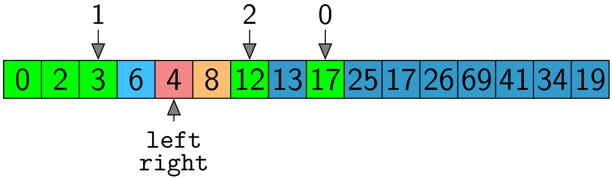

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



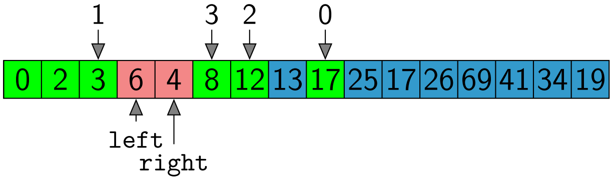

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



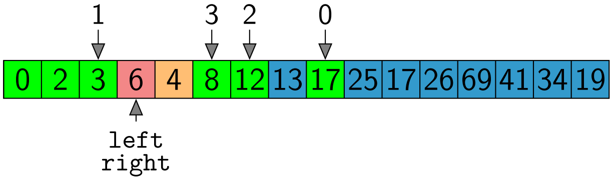

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



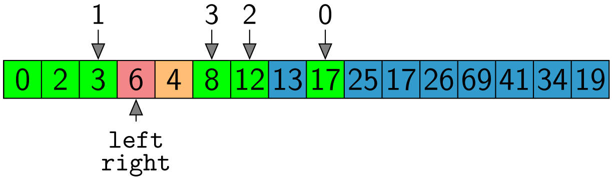

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation




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

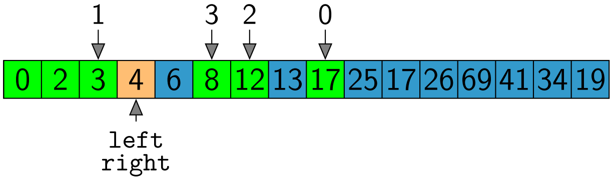
Quicksort – Algorithmus und Animation




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

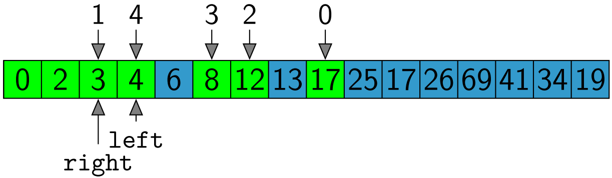
 Quicksort

Quicksort – Algorithmus und Animation



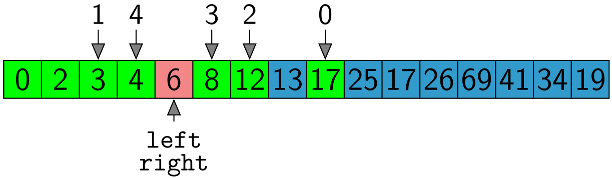

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



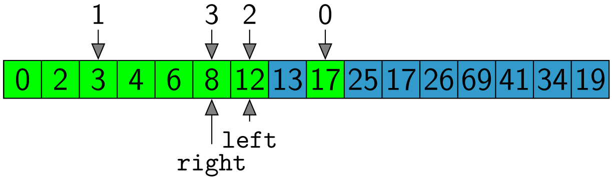

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



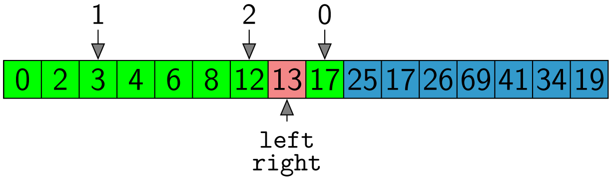

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



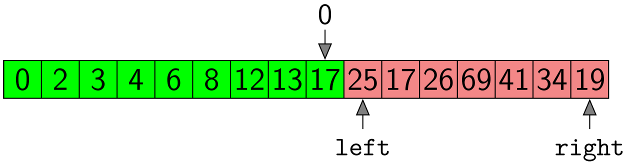

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



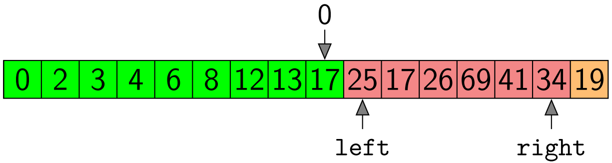

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



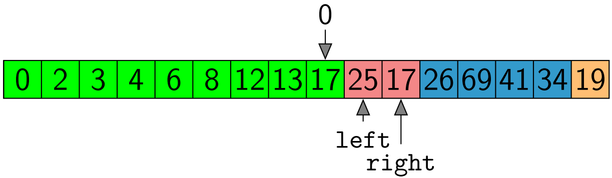

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



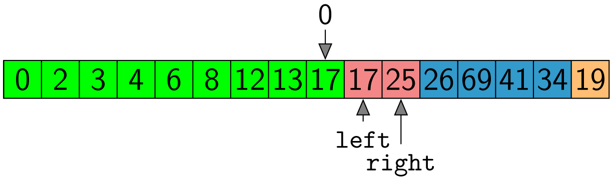

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



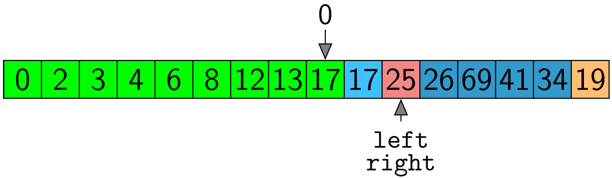

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



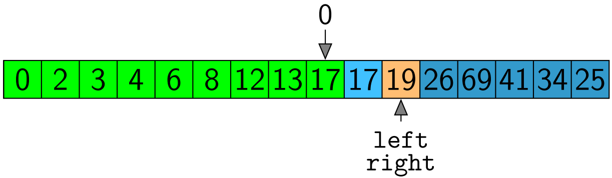

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



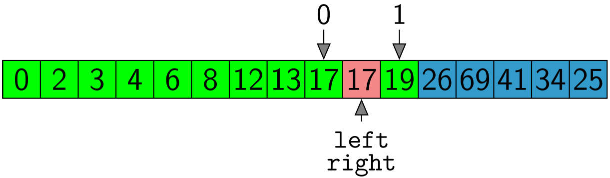

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



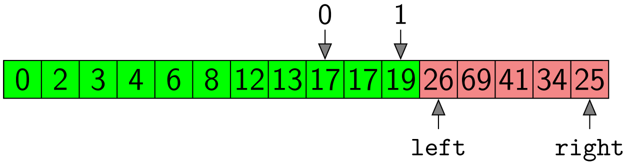

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



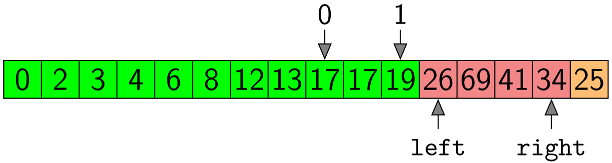

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



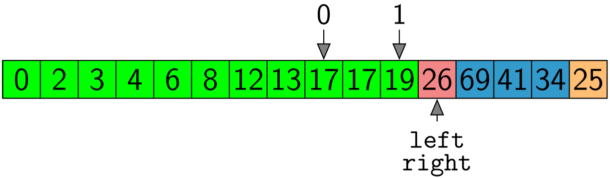

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



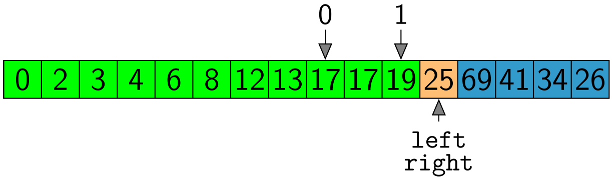

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



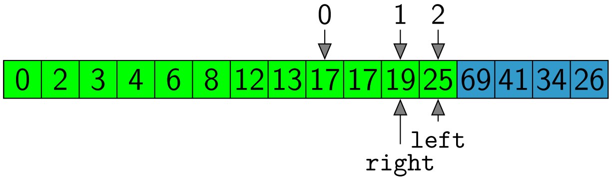

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



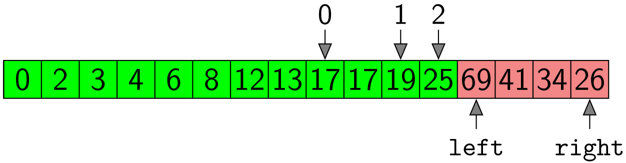

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



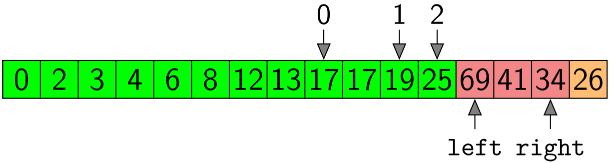

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



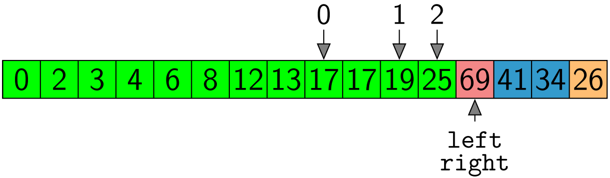

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



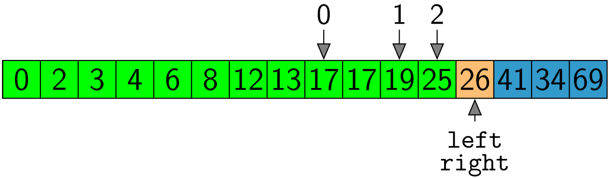

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



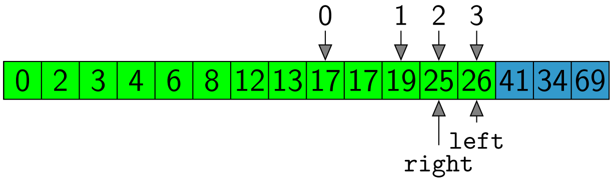

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



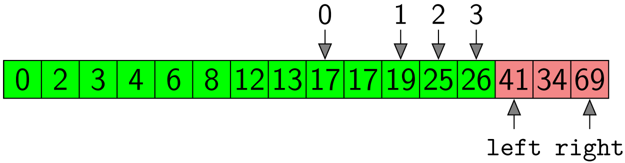

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



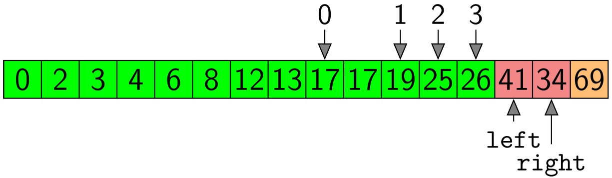

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



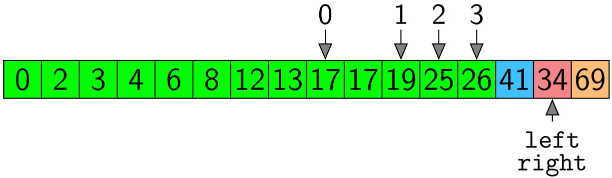

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



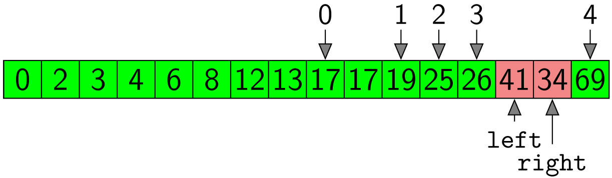

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



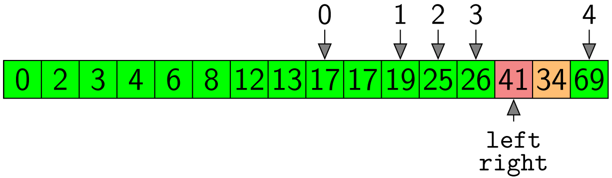

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



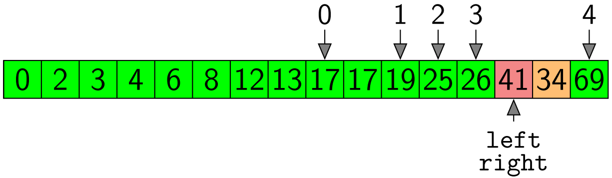

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



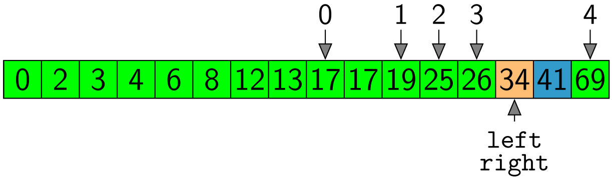

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



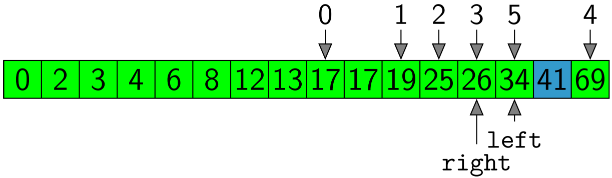

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



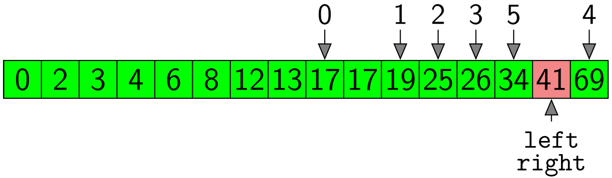

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation



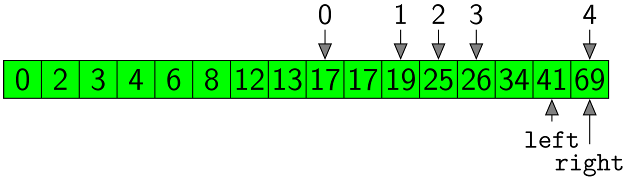

```
void quickSort(int E[], int left, int right) {
  if (left < right) {
    int i = partition(E, left, right);
      // i ist Position des Split-punktes (Pivot)
    quickSort(E, left, i - 1); // sortiere den linken Teil
    quickSort(E, i + 1, right); // sortiere den rechten Teil
  }
}
```

Quicksort – Algorithmus und Animation




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

Quicksort – Algorithmus und Animation




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

Quicksort – Algorithmus und Animation

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

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

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.

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.

$\Rightarrow$ 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.

$\Rightarrow$ 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.

$\Rightarrow$ 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, Mergesort, 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.

$\Rightarrow$ 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, Mergesort, 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.

$\Rightarrow$ 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, Mergesort, 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 $\log n$ 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 $\log n$ 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 $\log n$ 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 $T(1) = 1$.

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 $\log n$ 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 $\log n$ 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 $\log n$ 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, 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 $\log n$ 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, 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 schlechtesten Falls.

Quicksort – Balancierte Zerlegung

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

Quicksort – Balancierte Zerlegung

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

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

Quicksort – Balancierte Zerlegung

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

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

- Rekursionsbaumanalyse liefert: $T(n) \in O(n \cdot \log n)$.

Quicksort – Balancierte Zerlegung

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

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

- Rekursionsbaumanalyse liefert: $T(n) \in O(n \cdot \log n)$.
- 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 schlechtesten Falls.
- Schlüssel zum Verständnis: wie schlägt sich die Balanciertheit der Zerlegung sich in der Rekursionsgleichung nieder?
- Betrachte z. B. eine Zerlegung im Verhältnis 9:1. Dann erhält man für $n > 1$:

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

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

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 in der 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 in der zu sortierenden Array `E` sind unterschiedlich
  3. alle mögliche Permutationen haben die gleiche Wahrscheinlichkeit

Quicksort – Average-Case-Analyse (I)

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

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

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

Quicksort – Average-Case-Analyse (II)

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

$$\Big| \; \sum_{i=0}^{n-1} A(n-i-1) = A(n-1) + A(n-2) + \ldots + A(0)$$

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

Quicksort – Average-Case-Analyse (II)

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

$$\Big|\ \textstyle\sum_{i=0}^{n-1} A(n-i-1) = A(n-1) + A(n-2) + \ldots + A(0)$$

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

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

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

$$\Big|\ \text{teile durch } n \cdot (n+1) \text{ und setze } A'(n) = A(n)/(n+1)$$

$$A'(n) = A'(n-1) + \frac{2 \cdot (n-1)}{n \cdot (n+1)} \quad \text{mit } A'(0) = 1 \qquad \Big|\ \text{Umformen}$$

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

Quicksort – Average-Case-Analyse (III)

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

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

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

$$\leqslant 2 \cdot \ln n - \frac{2n}{n+1} \qquad \mid A'(n) = A(n)/(n+1)$$

$A(n) \in O(n \cdot \log n)$

Quicksort – Average-Case-Analyse (III)

$$\begin{aligned}
A'(n) &= \sum_{i=1}^{n} \frac{2\cdot(i-1)}{i\cdot(i+1)} && \mid \text{ calculus} \\
&= 2\cdot\sum_{i=1}^{n}\frac{1}{i+1} - 2\cdot\sum_{i=1}^{n}\frac{1}{i\cdot(i+1)} && \mid \text{ calculus} \\
&= 2\cdot\sum_{i=1}^{n}\frac{1}{n} - 2 + \frac{2}{n+1} - 2\cdot\sum_{i=1}^{n}\frac{1}{i\cdot(i+1)} && \mid \text{ harmonische Reihe} \\
&\leqslant 2\cdot\ln n - \frac{2n}{n+1} && \mid A'(n) = A(n)/(n+1)
\end{aligned}$$

$A(n) \in O(n\cdot\log n)$

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

Komplexität von Sortieralgorithmen

	Worst-Case	*Average-Case*	*Platzbedarf*	*Stabil*
Insertionsort	$\Theta(n^2)$	$\Theta(n^2)$	in-place	J
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	in-place	N*
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

* es gibt Varianten die stabil sind.

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 Sortierverfahren 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 $O(n \cdot \log n)$ 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 $O(n \cdot \log n)$ 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 $O(n \cdot \log n)$ 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 $O(n \cdot \log n)$ 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 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 $O(n \cdot \log n)$ 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 sortierte 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 $O(n \cdot \log n)$ 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 sortierte 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 $O(n \cdot \log n)$ 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 sortierte Eingaben.
- Einige Variationen:
  - Introsort: setzt Quicksort ein bis zu einer gewissen Rekursionstiefe und benutzt dann Heapsort.
  - Smoothsort: (komplizierte) Variation von Heapsort die fast $O(n)$ braucht für fast sortierten Eingaben.