Algorithmen und Datenstrukturen

Prof. Dr. Thomas Wiemann - FB AI

Hochschule Fulda
University of Applied Sciences

Inhalt

Gliederung

1. Laufzeit, Komplexität, Verifikation
2. **Sortieren**
3. Abstrakte Datentypen
4. Hashing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

Inhalt

Gliederung

1. Laufzeit und Komplexität
2. **Sortieren**
3. Abstrakte Datentypen
4. Hashing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

2. Sortieren

1. Simple Sortierverfahren
2. Komplexe Sortierverfahren
3. Untere Schranke für vergleichsbasiertes Sortieren
4. Sortieren in linearer Zeit

Warum Sortieren

- Sortieren hilft beim schnellen Auffinden von Informationen
- Beispiele:
  - Lexika
  - Telefonbücher
  - Kataloge
- Formale Formulierung des Sortierproblems

Sei A ein Alphabet oder eine geordnete Menge von Elementen. Eine Folge $v = v_1, \ldots, v_n$ heißt **geordnet**, falls $v_i \leq v_{i+1}$ für $i = 1 \ldots n$. **Sortierproblem**: Gegeben seine eine Folge $v = v_1, \ldots, v_n$ in A. Ordne v so um, dass eine geordnete Folge entsteht. Gesucht wird also eine **Permutation** $\pi : \{1,\ldots,n\} \rightarrow \{1,\ldots,n\}$ so dass die Folge $w = v_{\pi 1} \ldots v_{\pi n}$ geordnet ist

Bubblesort

Simple Sortieralgorithmen: Bubblesort (1)

```
public static void sort(int[] elements) {
    for (int max = elements.length - 1; max > 0; max--) {
      for (int i = 0; i < max; i++) {
        int left = elements[i];
        int right = elements[i + 1];
        if (left > right) {
          elements[i + 1] = left;
          elements[i] = right;
        }
      }
    }
  }
```

- Große Zahlen blubbern nach hinten
- Best, Worst und Average-Case $\mathcal{O}(n^2)$
- :-(
- Verbesserungsvorschläge?

Simple Sortieralgorithmen: Bubblesort (2)

```
public static void sort(int[] elements) {
    for (int max = elements.length - 1; max > 0; max--) {
      boolean swapped = false;
      for (int i = 0; i < max; i++) {
        int left = elements[i];
        int right = elements[i + 1];
        if (left > right) {
          elements[i + 1] = left;
          elements[i] = right;
          swapped = true;
        }
      }
      if (!swapped) break;
    }
  }
```

- Große Zahlen blubbern nach hinten
- Worst und Average-Case $\mathcal{O}(n^2)$
- Best-Case: $\mathcal{O}(n)$

Simple Sortieralgorithmen: Selection Sort (1)

```
public static void sort(int[] elements)
{
    int length = elements.length;

    for (int i = 0; i < length - 1; i++) {
      int minPos = i;
      int min = elements[minPos];
      for (int j = i + 1; j < length; j++) {
        if (elements[j] < min) {
          minPos = j;
          min = elements[minPos];
        }
      }
      if (minPos != i) {
        elements[minPos] = elements[i];
        elements[i] = min;
      }
    }
}
```

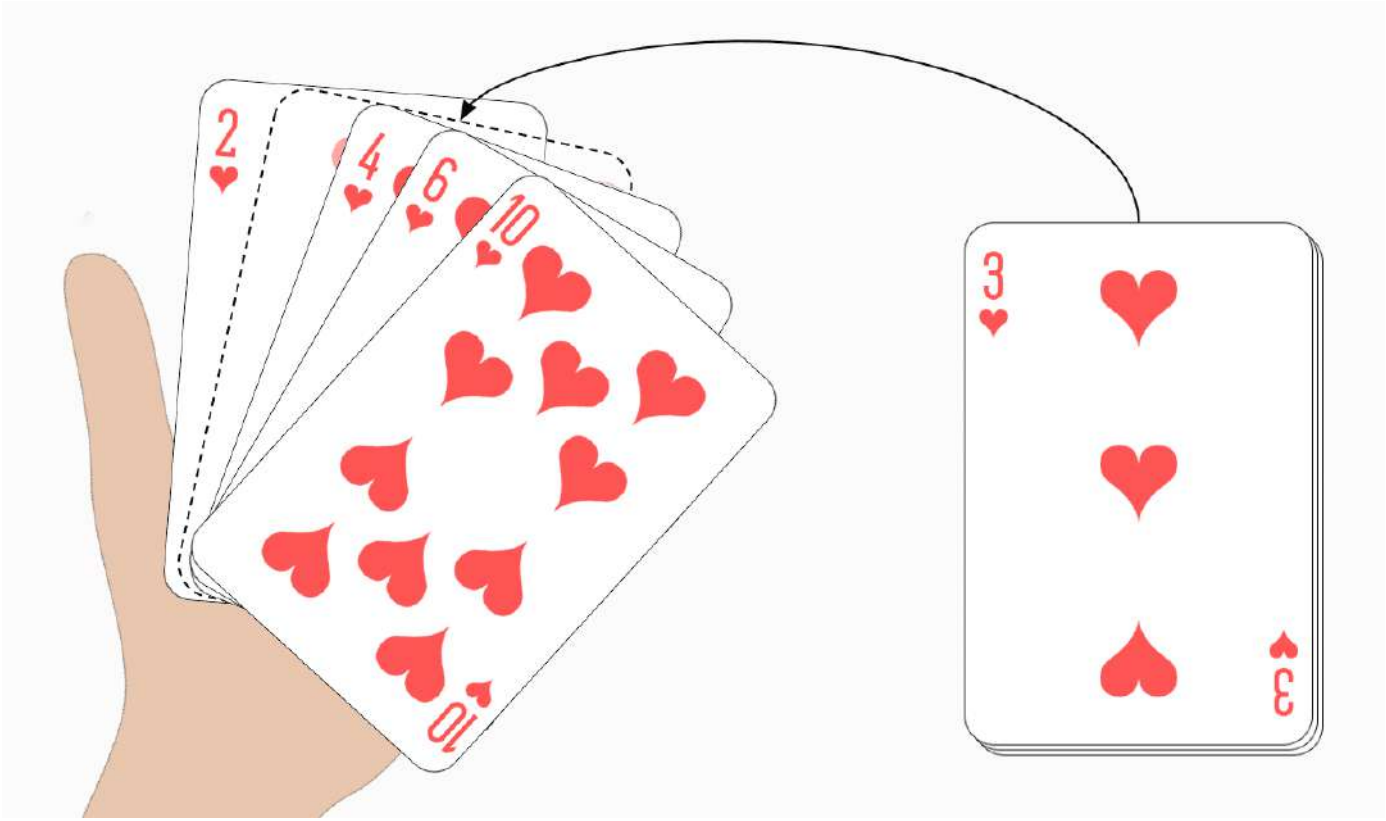
Laufe durch den noch nicht sortierten Bereich

Suche nach dem kleinsten Element

Swappe kleinstes Element an den Anfang des sortierten Bereichs

Simple Sortieralgorithmen: Insertion Sort (1)

- Beispiel an der Tafel



https://www.happycoders.eu/de/algorithmen/insertion-sort/

Simple Sortieralgorithmen - Insertion Sort (2)

- Sortieren wie beim Kartenstapel
- Wir haben die schon sortierten Karten in der Hand
- Ziehen eine neue Karte und stecken sie an der passenden Stelle in den sortierten Stapel

```
public static int[] sort(int arr[])
{
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
            greater than key, to one position ahead
            of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
    return arr
}
```

Laufe durch den noch nicht sortierten Bereich (rechts)

Wähle erstes Element als Vergleichsschlüssel

Schiebe alle Elemente passend um eins nach rechts

Füge Schlüssel an passender Stelle im sortieren Bereich ein

Simple Sortieralgorithmen - Insertion Sort (3)

- Laufzeitanalyse
- Worst Case: Array ist komplett absteigend sortiert. In jedem inneren Schleifendurchlauf müssen alle Elemente verschoben werden. ➠ $\mathcal{O}(n^2)$
- Average Case:
  - Das einzusortierende Element muss im Durchschnitt bis zur Mitte geschoben werden
  - Nochmal geteilt durch 2, da im Mittel bereits die Hälfte der Einträge sortiert ist
  - Ausgeschrieben: $\frac{n(n-1)}{4} = \frac{n^2}{4} - \frac{n}{4}$ ➠ $\mathcal{O}(n^2)$
- Best Case: Wenn alle Elemente sortiert sind, gibt es in der inneren Schleife jeweils einen Vergleich und keine Verschiebungen ➠ $\mathcal{O}(n)$

Best-Case: $\mathcal{O}(n)$
Worst-Case: $\mathcal{O}(n^2)$
Average-Case: $\mathcal{O}(n^2)$

Inhalt

Gliederung

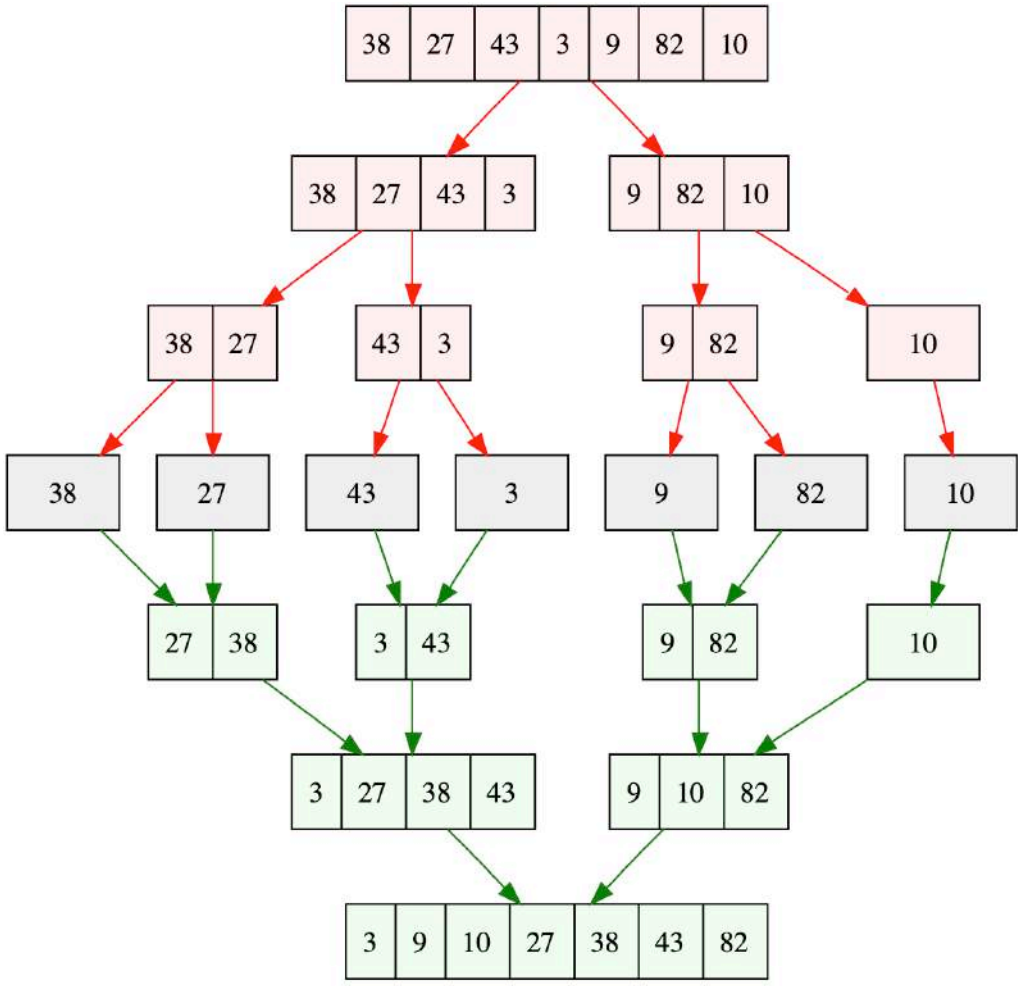
1. Laufzeit und Komplexität
2. **Sortieren**
3. Abstrakte Datentypen
4. Hashing
5. Suchbäume
6. Graphen- und Graphenalgorithmen
7. Ausblick

2. Sortieren

1. Simple Sortierverfahren
2. **Komplexe Sortierverfahren**
3. Untere Schranke für vergleichsbasiertes Sortieren
4. Sortieren in linearer Zeit

Sortieralgorithmen: Mergesort (1)

- Beispiel: Mergesort



Mergesort (2)

```
private int[] mergeSort(int[] elements, int left, int right)
{
    // End of recursion reached?
    if (left == right) return new int[]{elements[left]};

    int middle = left + (right - left) / 2;
    int[] leftArray = mergeSort(elements, left, middle);
    int[] rightArray = mergeSort(elements, middle + 1, right);
    return merge(leftArray, rightArray);
}
```

- Wie implementiere ich den Merge?

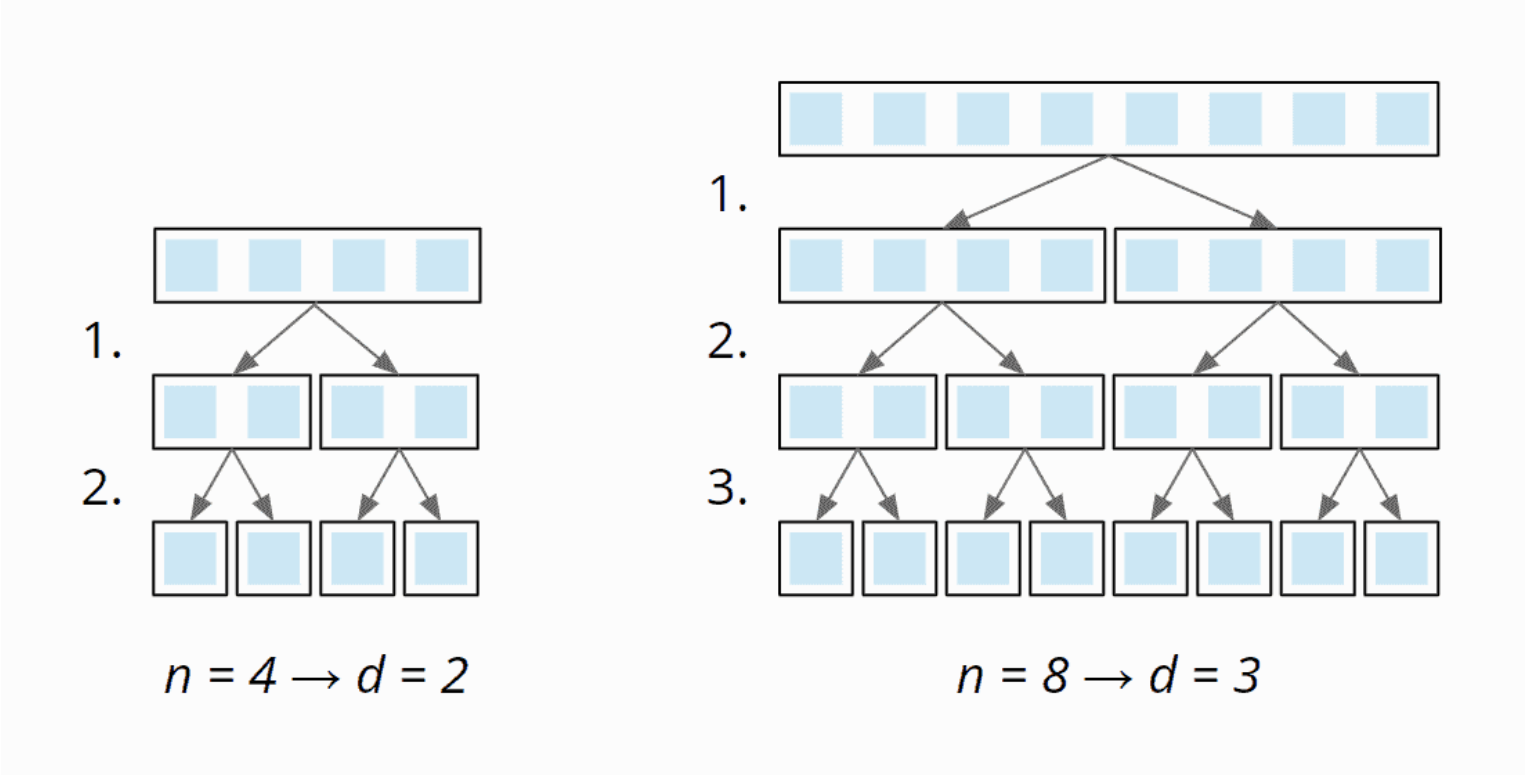
Mergesort (3)

```
public int[] merge(int[] leftArray, int[] rightArray) {
    int leftLen = leftArray.length;
    int rightLen = rightArray.length;

    int[] target = new int[leftLen + rightLen];
    int targetPos = 0;
    int leftPos = 0;
    int rightPos = 0;

    // As long as both arrays contain elements...
    while (leftPos < leftLen && rightPos < rightLen) {
      // Which one is smaller?
      int leftValue = leftArray[leftPos];
      int rightValue = rightArray[rightPos];
      if (leftValue <= rightValue) {
        target[targetPos++] = leftValue;
        leftPos++;
      } else {
        target[targetPos++] = rightValue;
        rightPos++;
      }
    }
    // Copy the rest
    while (leftPos < leftLen) {
      target[targetPos++] = leftArray[leftPos++];
    }
    while (rightPos < rightLen) {
      target[targetPos++] = rightArray[rightPos++];
    }
    return target;
  }
```

Simple Merge-Strategie

Mergesort - Laufzeitanalyse (1)

- Partitionierung: Wie viele rekursive Aufrufe werden benötigt?

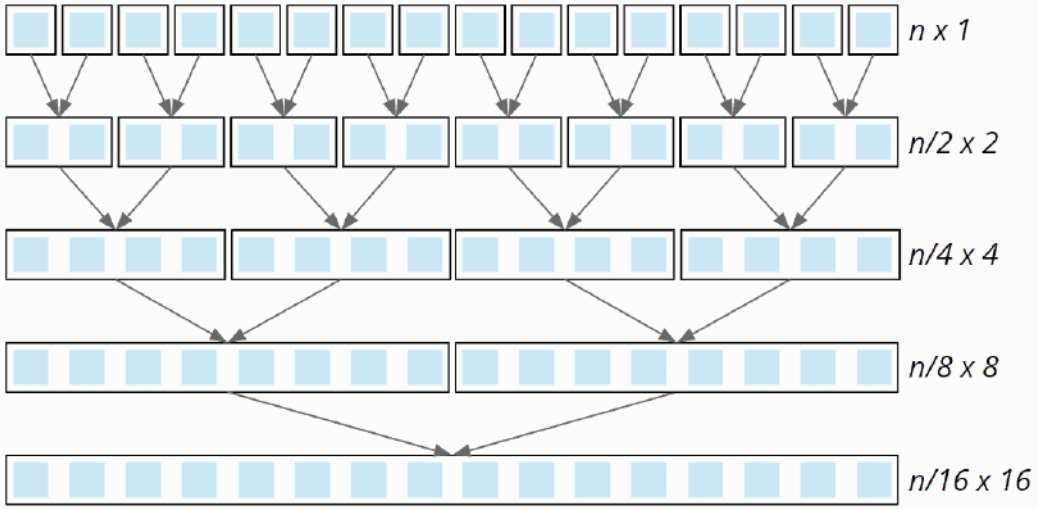


Mergesort - Laufzeitanalyse (2)

- Laufzeit beim Zusammenführen?
  - Auf jeder Ebene müssen n Elemente gemerged werden
  - $n \times 1$ auf der Ersten, $n/2 \times 2$ usw.
  - Das passiert d-Mal, wobei d konstant ist
  - Keine inneren Schleifen
  - Also Laufzeit $\mathcal{O}(n)$

Insgesamt liegt die Laufzeit von Mergesort in $\mathcal{O}(n \cdot \log n)$

Mergesort sortiert nicht in-Place. Es wird Extraspeicher benötigt!



Wie klingen Sortieralgorithmen?

Pivoting

- Verschiedene Algorithmen müssen zunächst ein Element aus einer Menge auswählen, um bestimmte Operationen ausführen zu können
- “Pivoting”
- Anhand dieses Elements werden alle anderen Elemente in Kategorien eingeteilt

Rätsel:

In einer Höhle leben Zwerge mit roten und blauen Mützen. Die Höhle hat einen enge Ausgang, aus dem immer nur ein Zwerg zur Zeit herausgehen kann. Bei einem Angriff auf die Höhle werden alle Zwerge zum Appell gerufen. Dabei sollen sie sich nach Farben geordnet in einer Reihe aufstellen. Wie erreicht man das am schnellsten?

Quicksort - Grundlagen

- Weiteres Beispiel für einen Divide-and-Conquer-Algorithmus
- Zentraler Bestandteil ist hier die Wahl eines Pivot-Elementes
- Quicksort ist für große Mengen in der Praxis meistens der schnellste Algorithmus
- Ansatz:
  - Teile die Menge der zu sortierenden Elemente in “kleine” und “große” auf
  - “Klein” und “groß” werden anhand eines zuvor gewählten Pivot-Elements festgemacht
  - Das Array wird jetzt so sortiert, das die kleinen Einträge im linken Teil und die großen Elemente im rechten Teil landen
  - Die Bereiche werden dann rekursiv weiter unterteilt

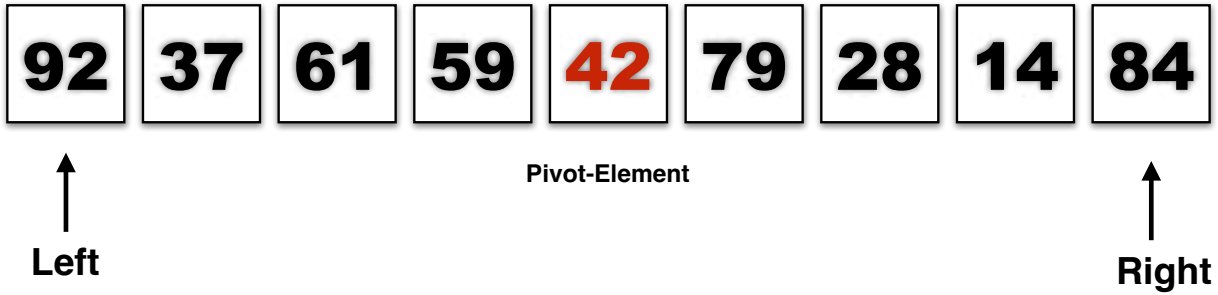
Quicksort - Partitionierung (1)

- Partitionierung, indem von links beginnend nach Elementen gesucht wird, die größer als das Pivot-Element sind
- Von rechts wird nach kleineren Elementen gesucht
- Die gefundenen Elemente werden vertauscht
- Wiederholen, bis die linke oder rechte Suchposition gleich sind oder aneinander vorbei gelaufen sind

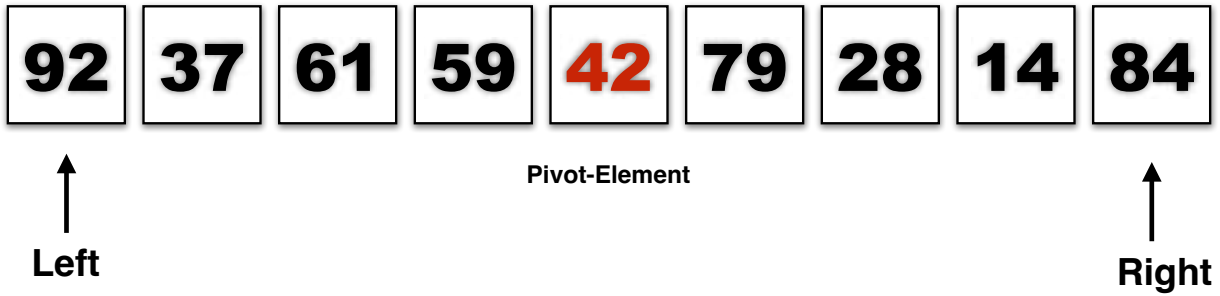
Beispiel

1.) Wähle Pivot-Element

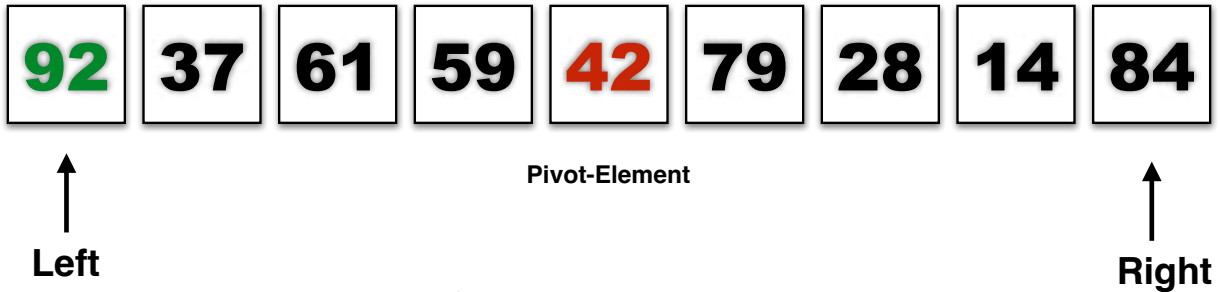
2.) Initialisiere Zeiger links und rechts



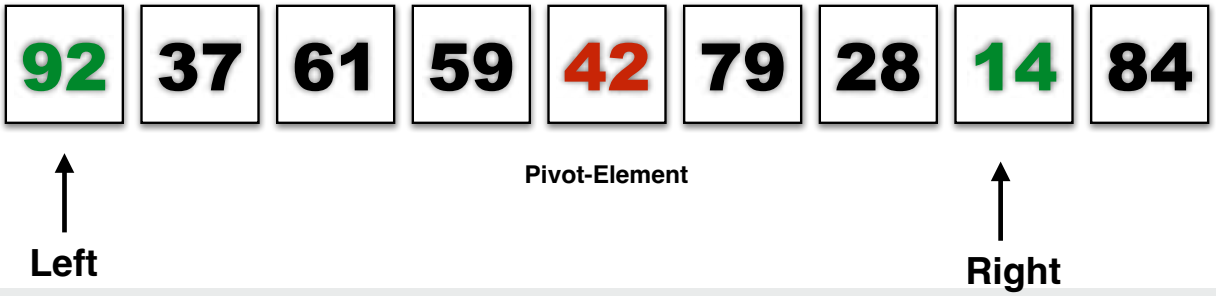
Beispiel



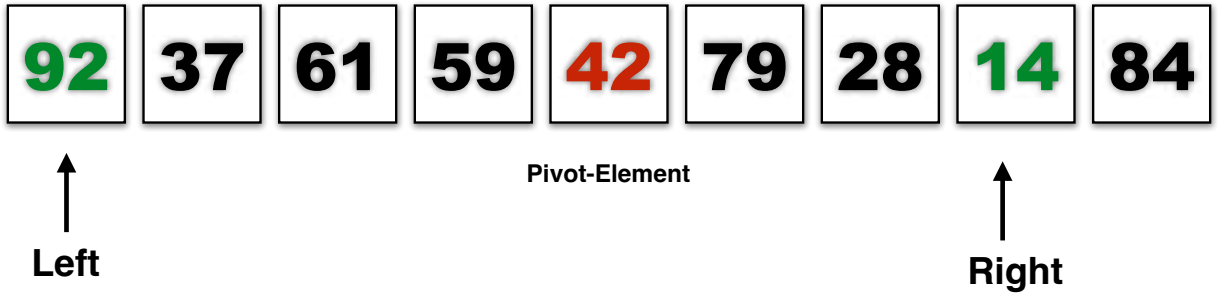
3.) Lass links soweit laufen, bis links $\geq$ pivot



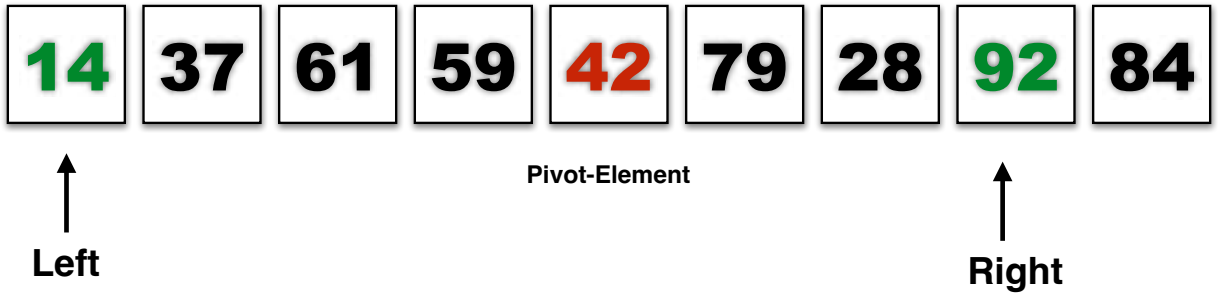
4.) Lass rechts soweit laufen, bis rechts $\leq$ pivot



Beispiel

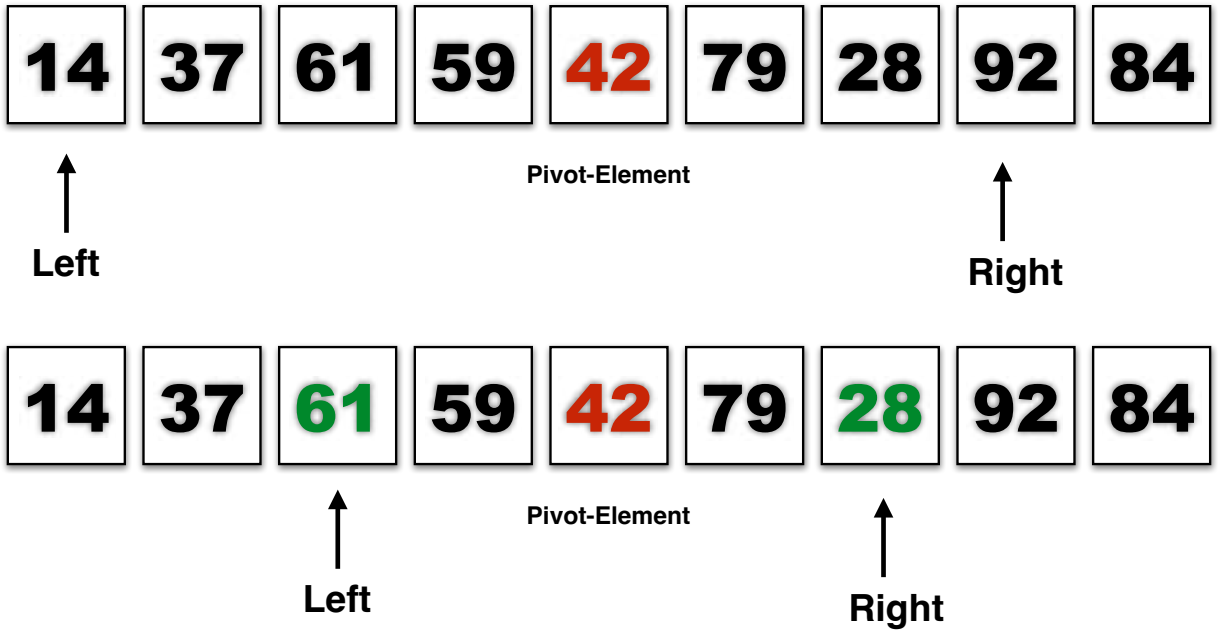


5.) Tausche die Werte von links und rechts

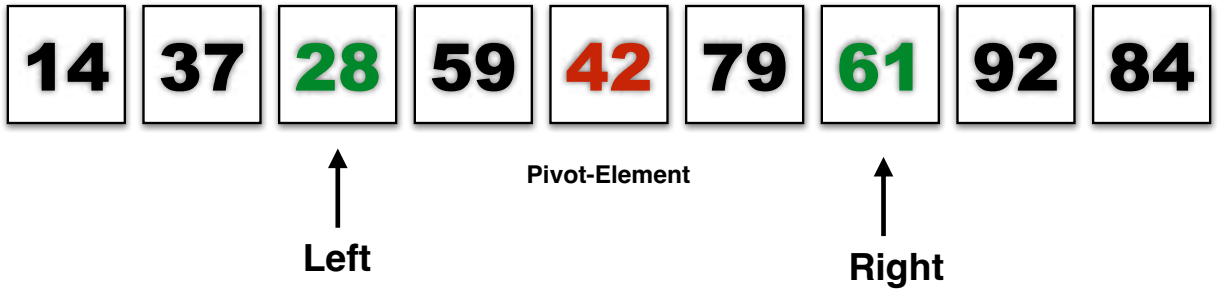


Beispiel

6.) Lass die Zeiger wieder laufen

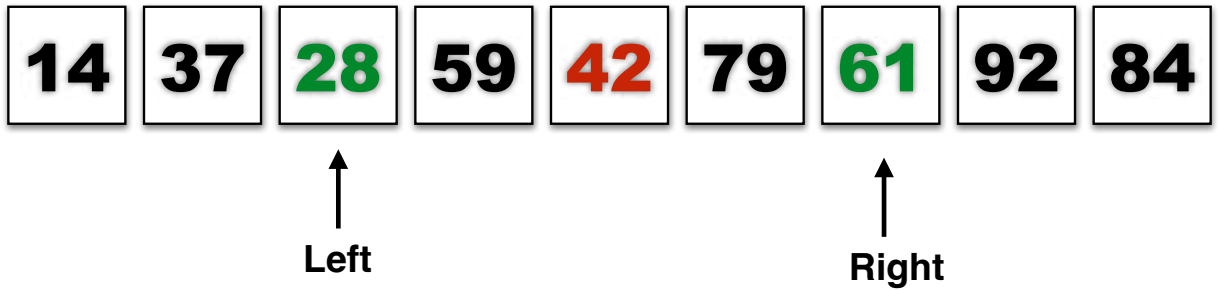


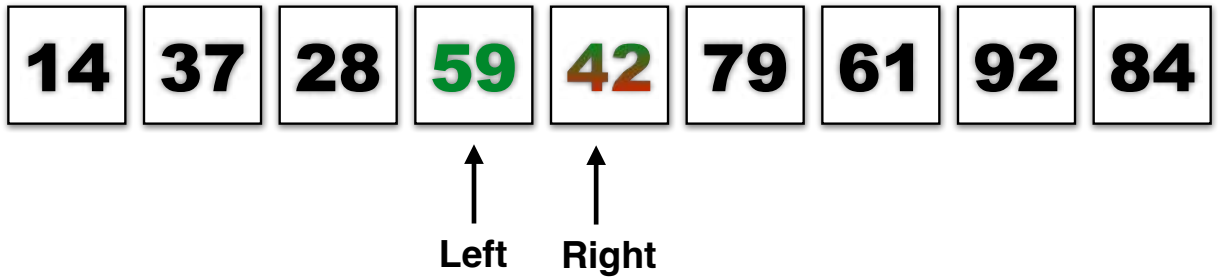
7.) Tausche



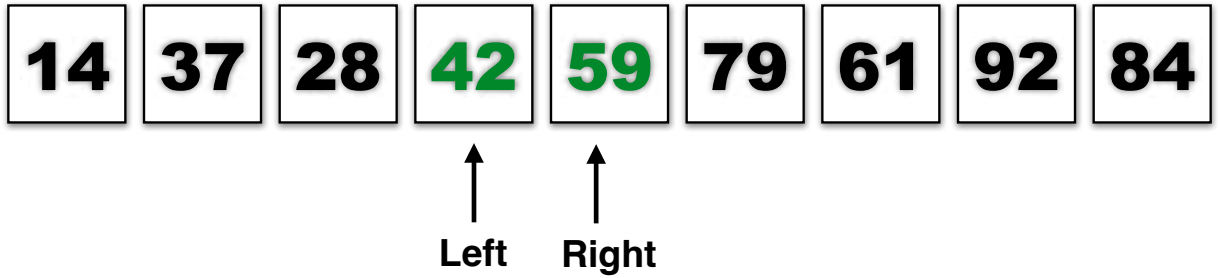
Beispiel

9.) Lass die Zeiger wieder laufen



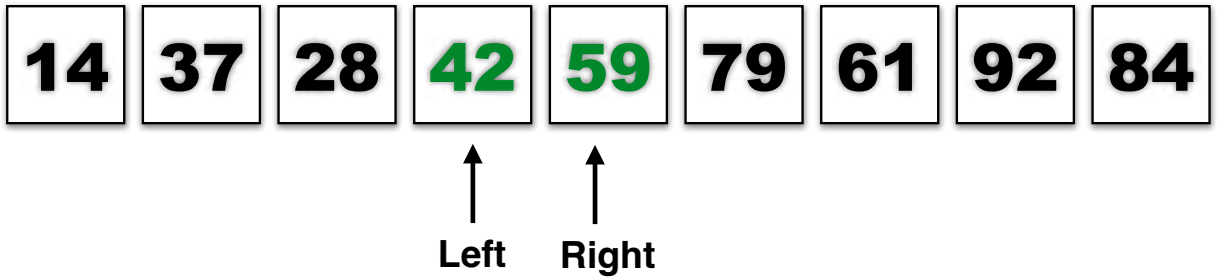


10.) Tauschen

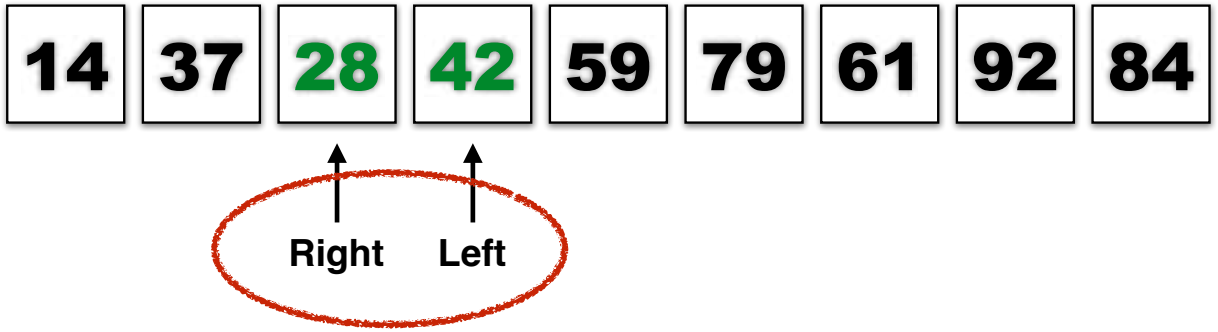


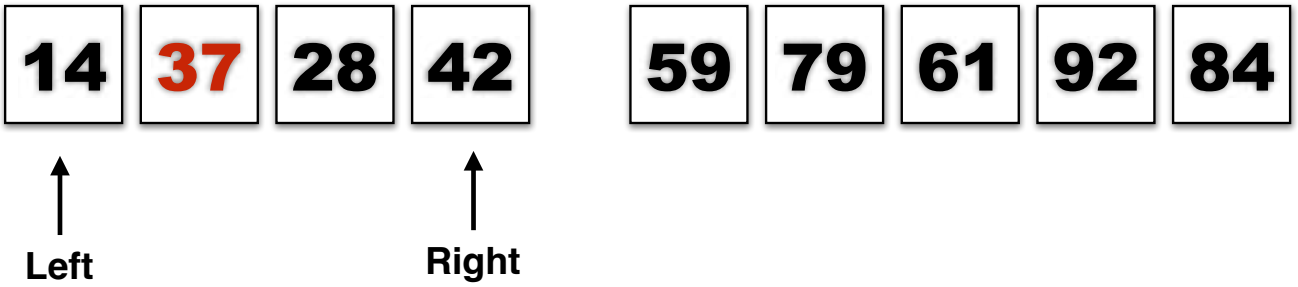
Beispiel

10.) Lass die Zeiger wieder laufen

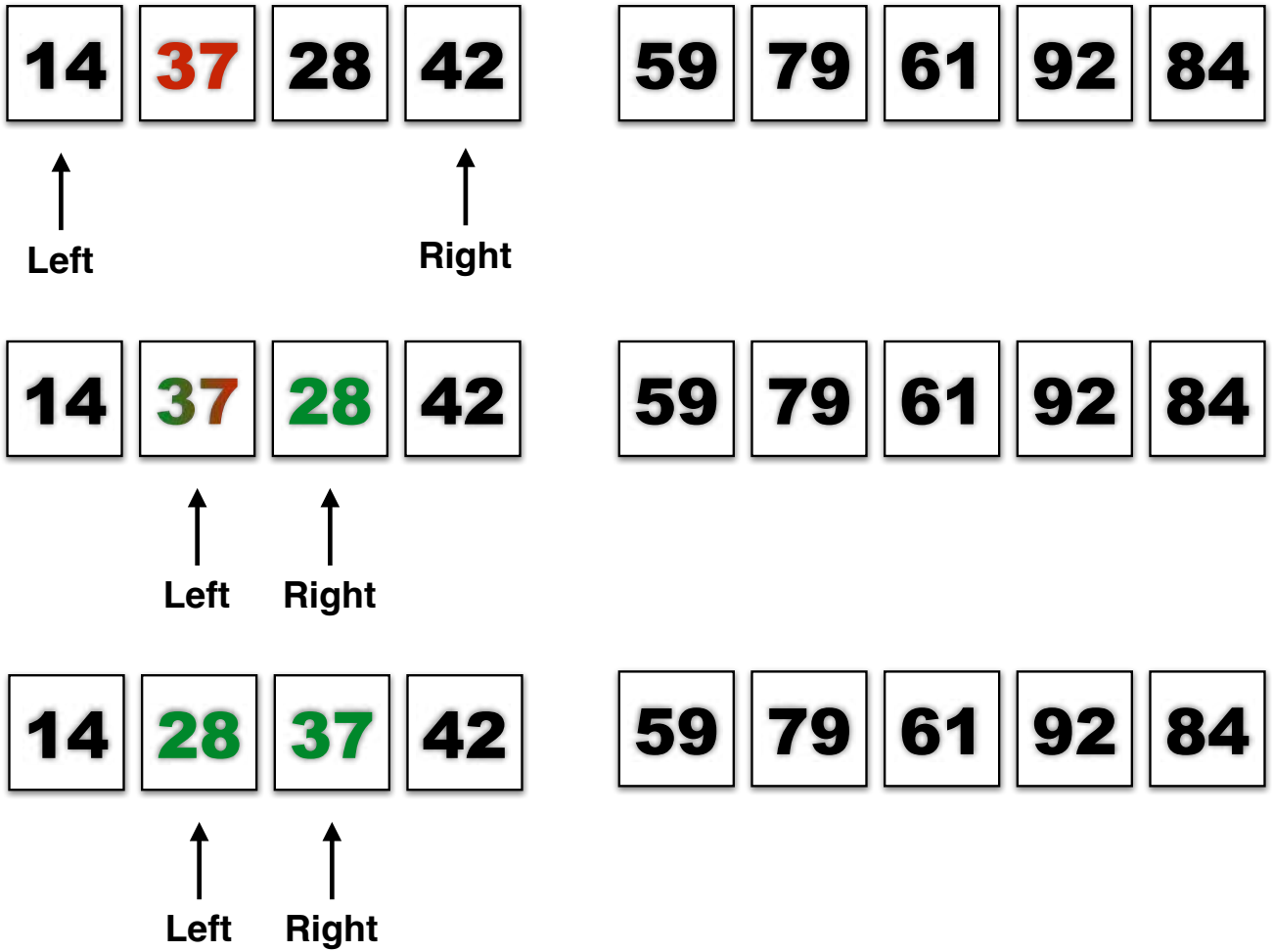


11.) Teile die Mengen rekursiv auf

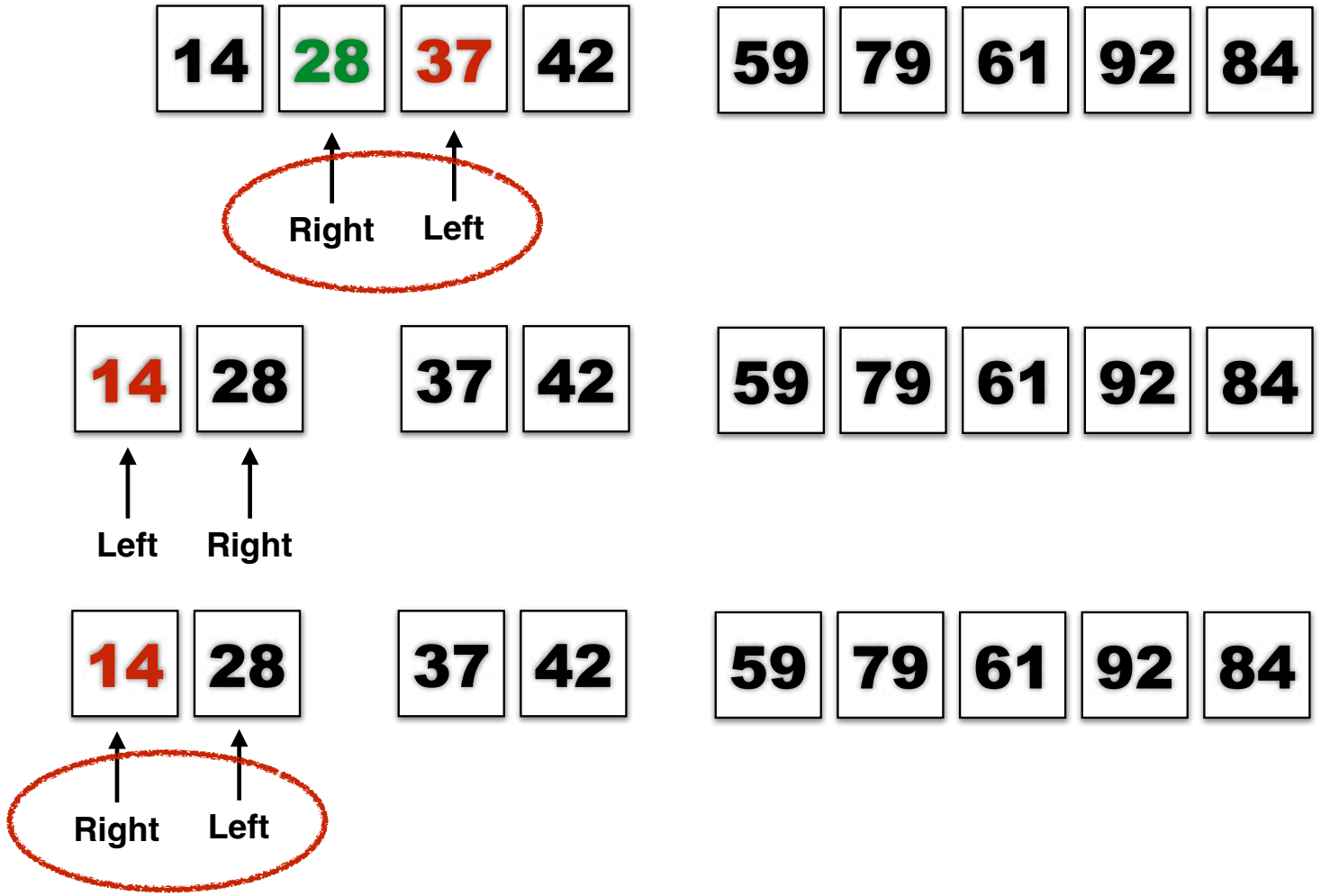




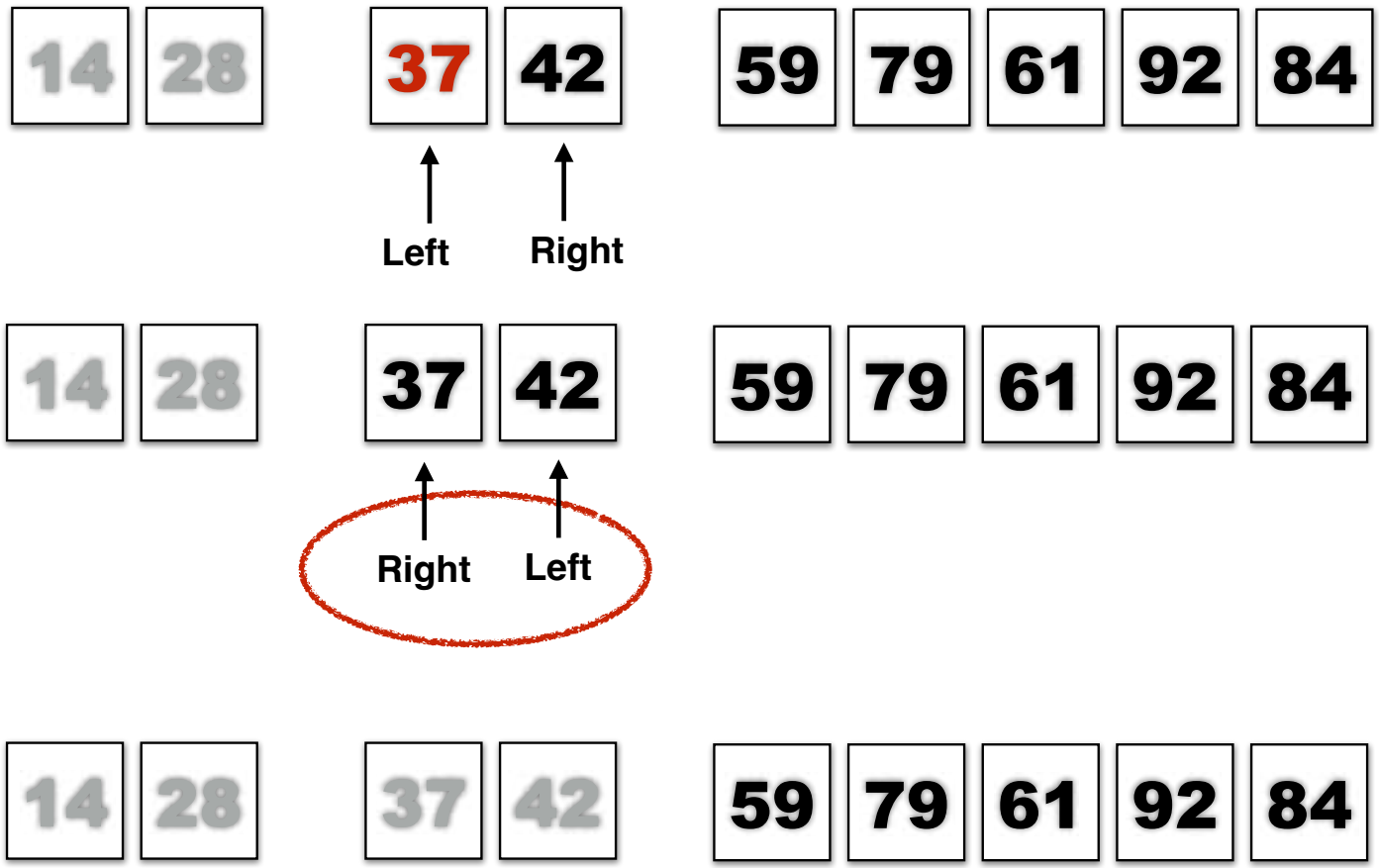
Beispiel


14 37 28 42 59 79 61 92 84
Left
Right
14 37 28 42 59 79 61 92 84
Left
Right
14 28 37 42 59 79 61 92 84
Left
Right

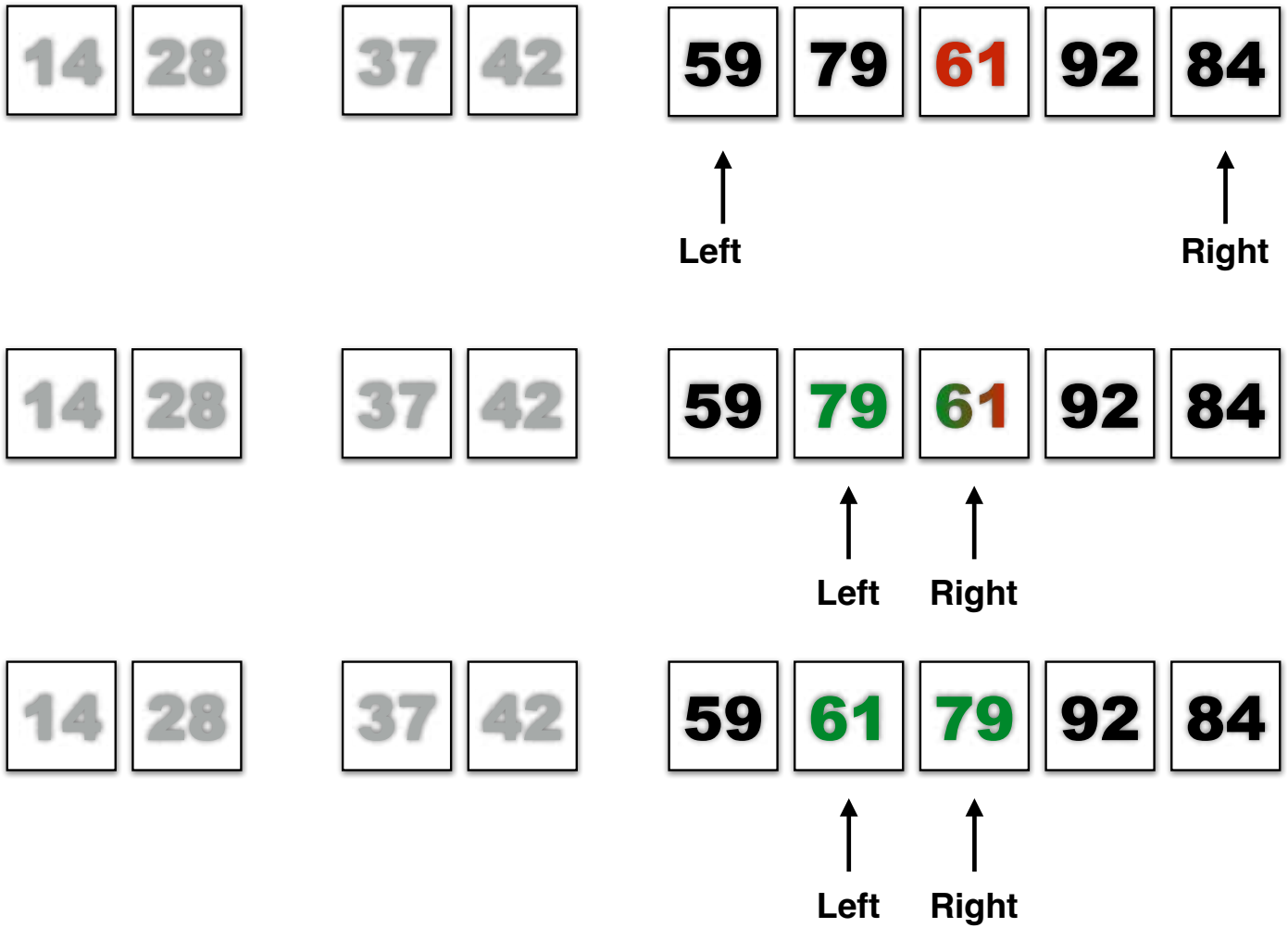
Beispiel


14 28 37 42 59 79 61 92 84
Right Left
14 28 37 42 59 79 61 92 84
Left Right
14 28 37 42 59 79 61 92 84
Right Left

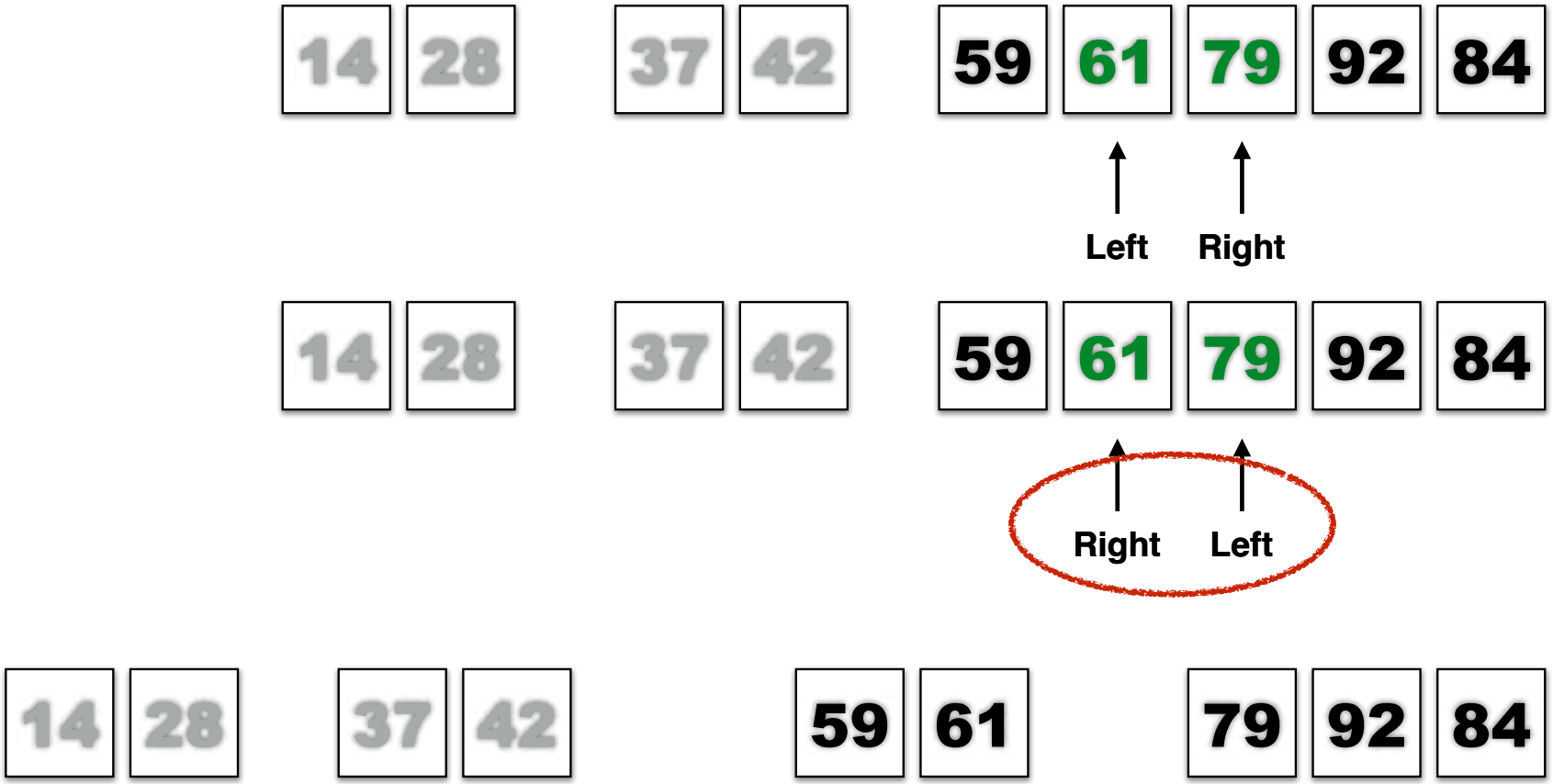
Beispiel


14
28
37
42
59
79
61
92
84
Left
Right
14
28
37
42
59
79
61
92
84
Right
Left
14
28
37
42
59
79
61
92
84

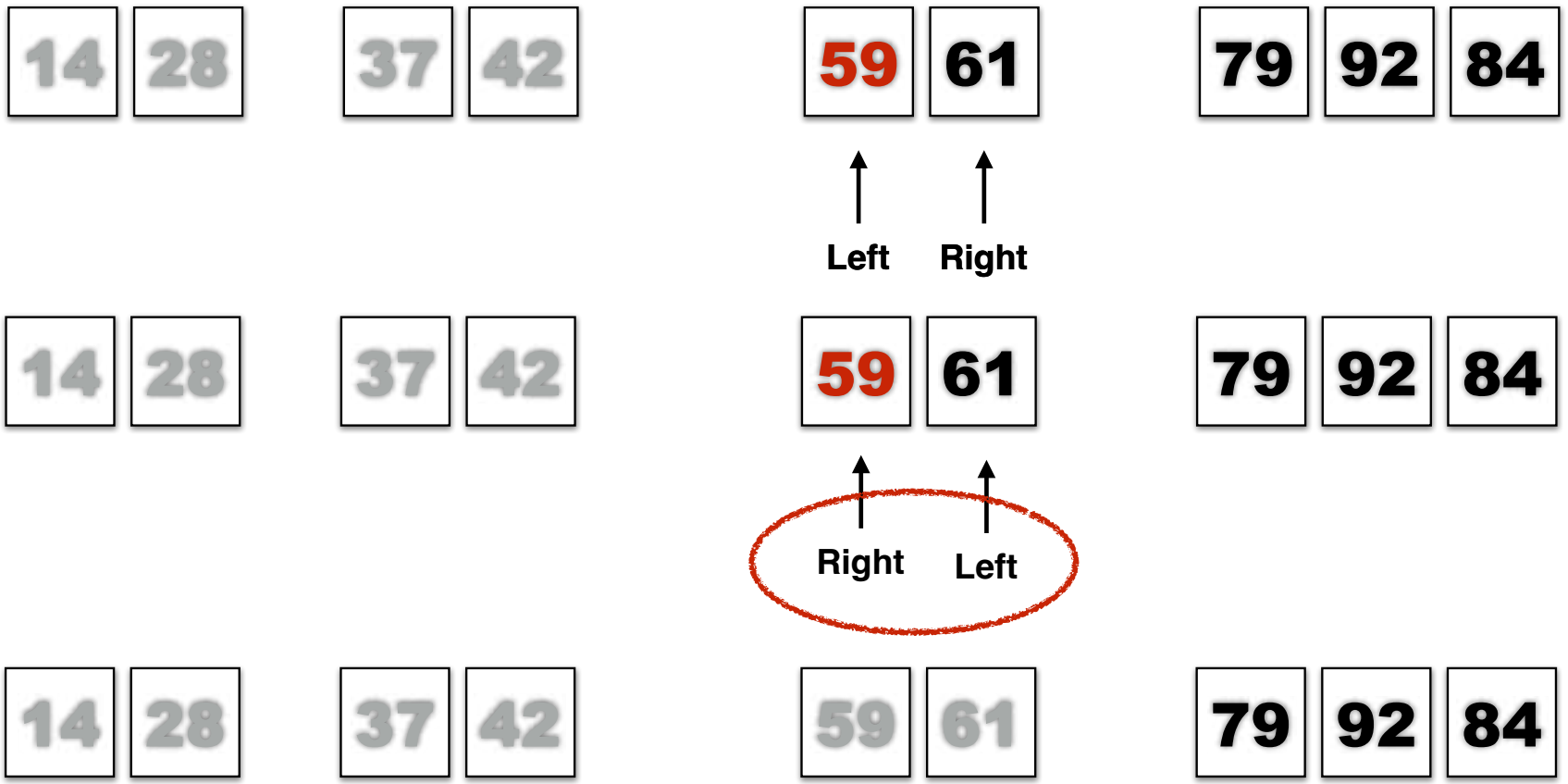
Beispiel


14 28 37 42 59 79 61 92 84
Left
Right
14 28 37 42 59 79 61 92 84
Left
Right
14 28 37 42 59 61 79 92 84
Left
Right

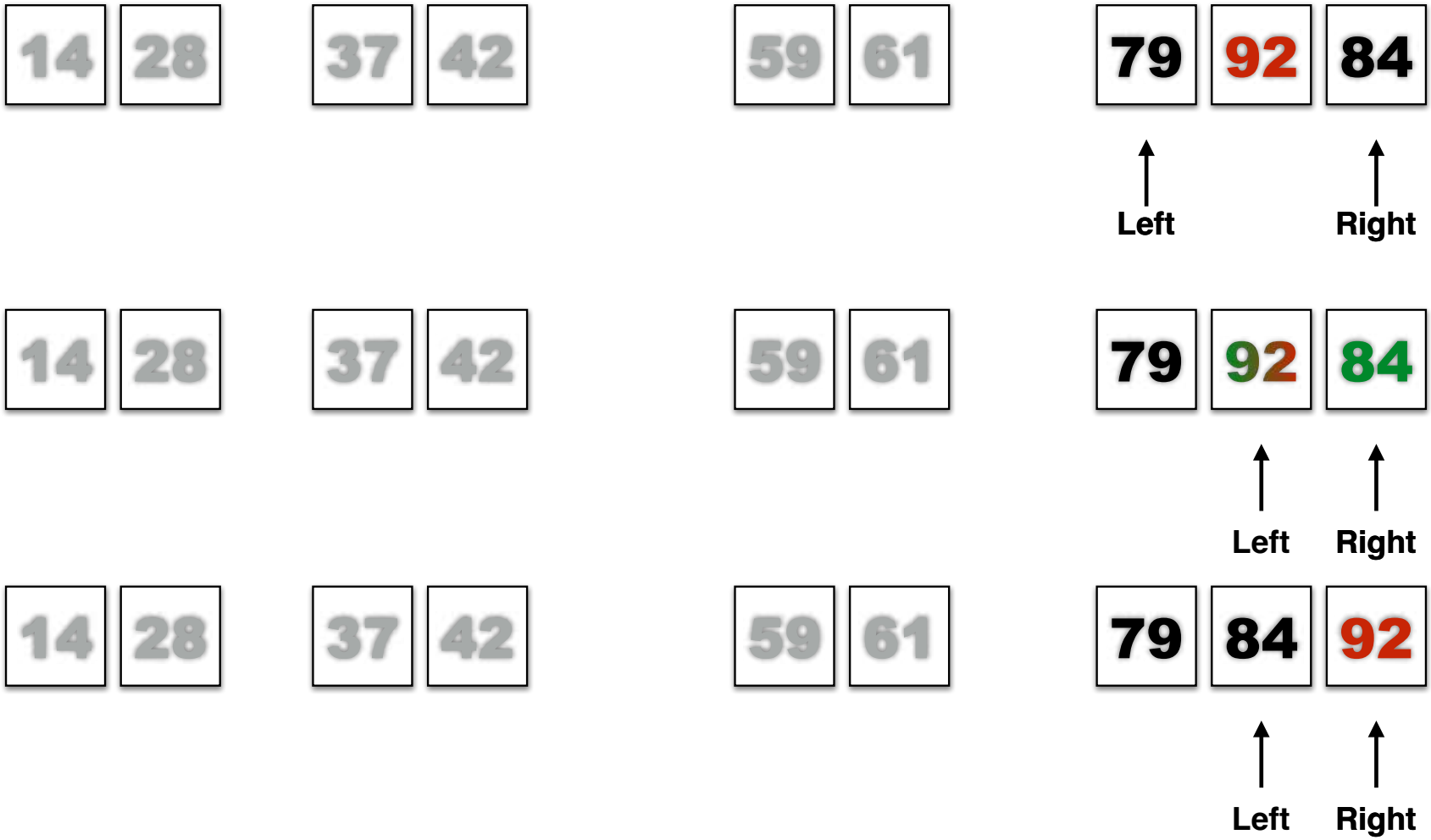
Beispiel


14 28 37 42 59 61 79 92 84
Left
Right
14 28 37 42 59 61 79 92 84
Right
Left
14 28 37 42 59 61 79 92 84

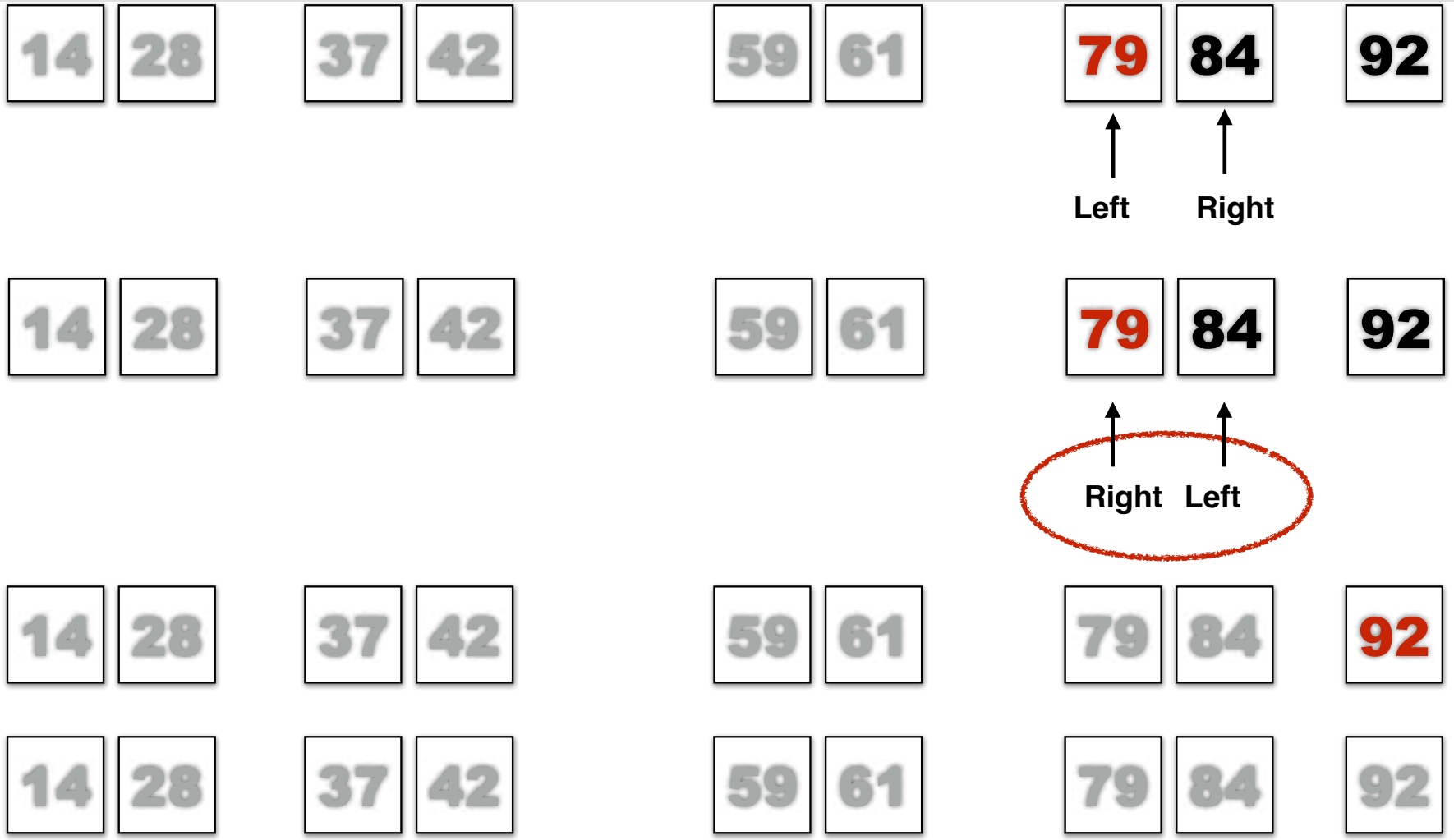
Beispiel


14
28
37
42
59
61
79
92
84
Left
Right
14
28
37
42
59
61
79
92
84
Right
Left
14
28
37
42
59
61
79
92
84

Beispiel


14
28
37
42
59
61
79
92
84
Left
Right
14
28
37
42
59
61
79
92
84
Left
Right
14
28
37
42
59
61
79
84
92
Left
Right

Beispiel

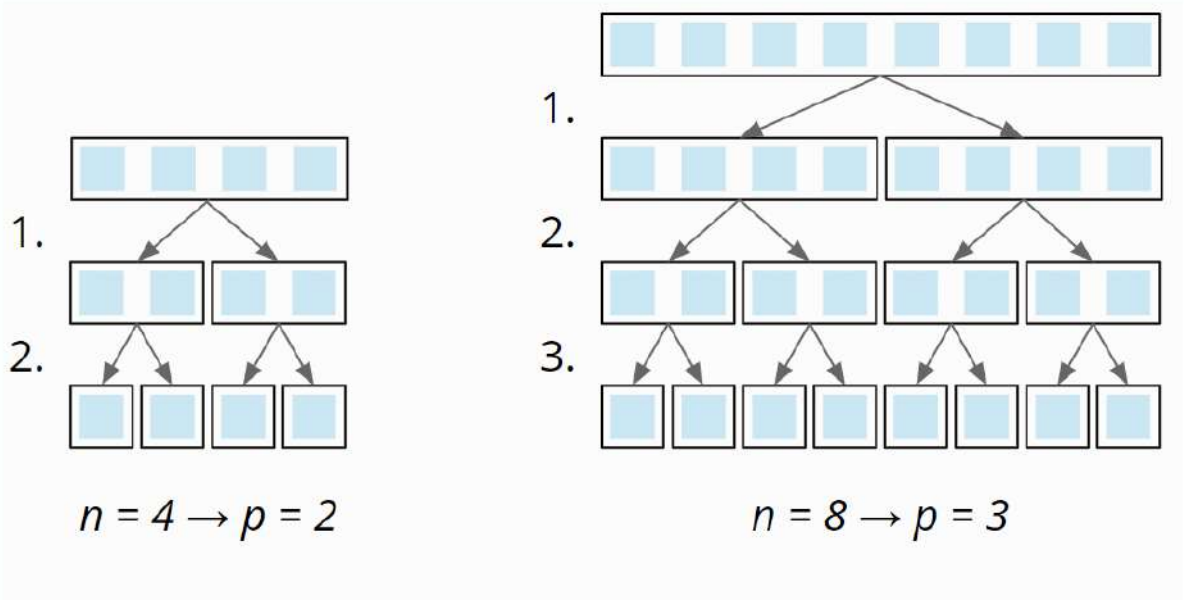

14 28 37 42 59 61 79 84 92
Left
Right
14 28 37 42 59 61 79 84 92
Right
Left
14 28 37 42 59 61 79 84 92
14 28 37 42 59 61 79 84 92

Quicksort - Java

```java
public class QuickSort {

    public static void sort(int[] a){
        quicksort(a, 0, a.length - 1);
    }

    private static void quicksort(int[]a, int left, int right) {
        int i = left; int j = right; int middle = (left + right) / 2; int tmp;
        int pivot = a[middle];
        do
        {
            while(a[i] < pivot) i++;
            while(a[j] > pivot) j--;
            if(i <= j)
            {
                tmp = a[i]; a[i] = a[j]; a[j] = tmp;
                i++; j--;
            }
        } while(i <= j);
        if(left < j) quicksort(a, left, j);
        if(i < right) quicksort(a, i, right);
    }

}
```

Laufzeitbetrachtung - Best Case (1)

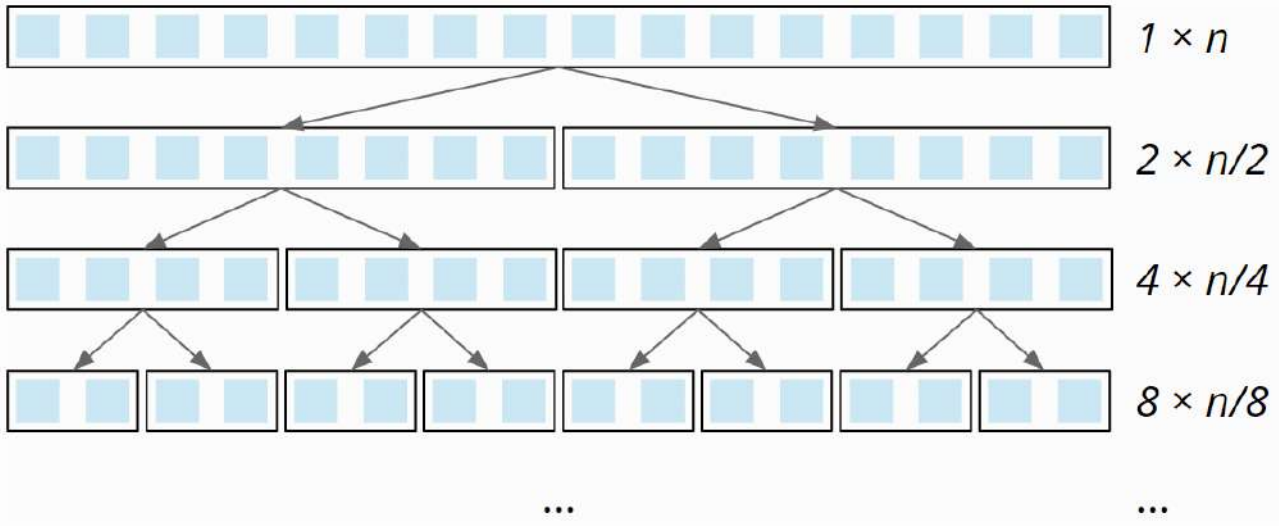
- Quicksort erreicht optimale Performance, wenn `partition()` die Elemente immer gleichmäßig verteilt
- Wenn ist das der Fall?
- Immer dann, wenn das Pivot-Element der Median der Menge ist



Anzahl der Partitionierungsstufen ist dann in $\mathcal{O}(\log_2 n)$

Laufzeitbetrachtung - Best Case (2)

- Auf jeder Stufe müssen n Elemente auf die linke und rechte Partition aufgeteilt werden



Anzahl der Vertauschungen ist dann in $\mathcal{O}(n)$

Insgesamt also $\mathcal{O}(n \cdot \log n)$

Laufzeitbetrachtung - Worst Case

- Wenn durch Zufall immer das Minimum in der Mitte steht
- Partionierungsaufwand sinkt linear von n bis 0
- Im Mittel Beträgt der Gesamtaufwand aber $n \cdot (n/2)$

Laufzeitbetrachtung Average Case

- Diskussion + Tafel

Quicksort - Varianten

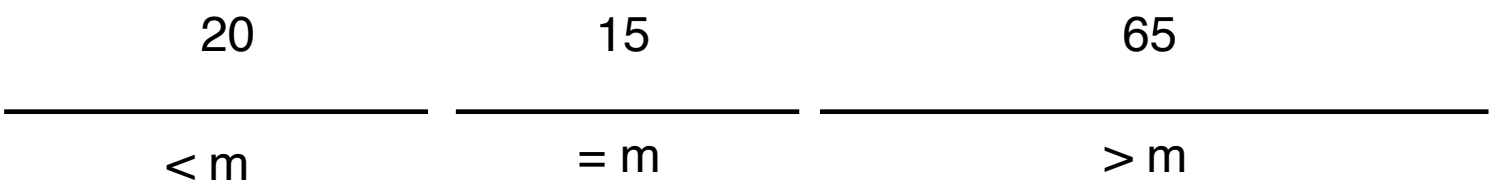
- Idealerweise würde man den Median wählen
- Warum ist das nicht möglich?
- Zur Bestimmung des Medians muss die Menge sortiert sein!
- Gängige Heuristiken:
  - Random: Ziehe ein zufälliges Element
  - Left: Nimm das linke Element
  - Middle: Nimm das Element in der Mitte
  - MedianN: Nimm den Median aus N Elementen (üblich 3 oder 5)
- Grundsätzlich bleibt man aber auch hier natürlich in $\mathcal{O}(n \cdot \log n)$

Bessere Wahl des Pivot-Elements

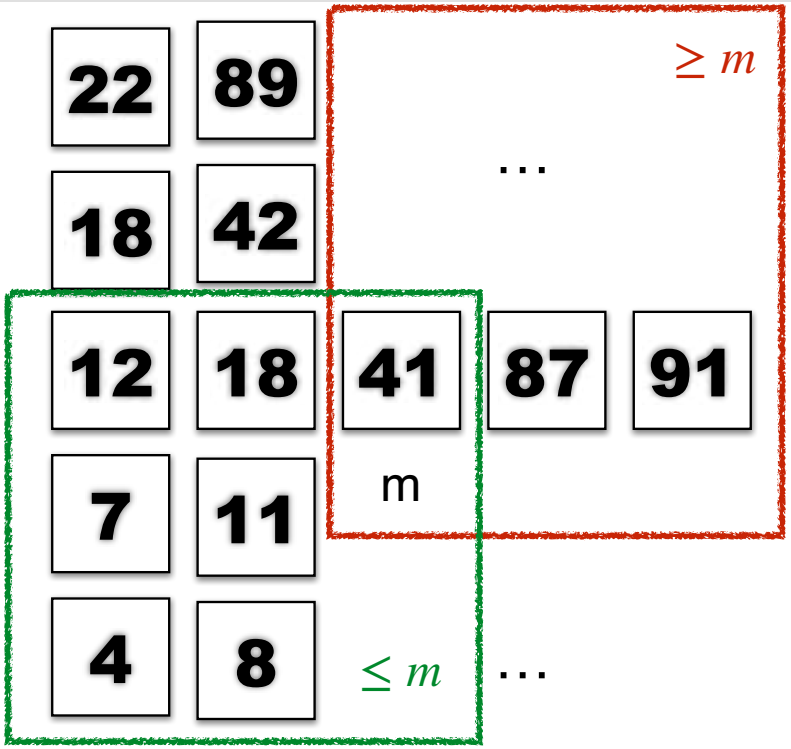
- Idealerweise würde man den Median wählen
- Warum ist das schwierig?
- Zur Bestimmung des Medians muss die Menge sortiert sein!
- Bestimmung des Medians in linearer Zeit? Geht ;-)

Medianberechnung - Pseudocode

```
// calk the k-th smallest element of s
public static int select(int []S, int k) {
    int n = |S|
    if(n < 50) return k-th smallest element by hand; else {
        Build groups of 5: S_1, S_2, S_3, ...S_{n/5}
        Compute in each group S_i the median m_i
        Compute the median of medians m := select( ∪_i m_i, n/10)
        A := {x ∈ S | x < m}
        B := {x ∈ S | x = m}
        C := {x ∈ S | x > m}
        if(|A| >= k) return select(A,k) else
        if(|A| + |B| >= k) return m; else
        if(|A| + |B| + |C| >= k return select(C, k - |A| - |B|)
}
```



Medianberechnung - Komplexität (1)



mindestens $\frac{1}{4}$ der Daten ist $\geq m$ 👉 höchstens $\frac{3}{4} < m$

mindestens $\frac{1}{4}$ der Daten ist $\leq m$ 👉 höchstens $\frac{3}{4} > m$

Medianberechnung - Komplexität (2)

$$f(n) \leq \begin{cases} c & \text{, für } n < 50 \\ \underbrace{c \cdot n}_{\text{Mediane der 5er Gruppen}} + \underbrace{f(\frac{n}{5})}_{\text{Median der Mediane}} + \underbrace{f(\frac{3}{4}n)}_{\text{Aufruf mit A oder C}} & \text{, sonst} \end{cases}$$

Behauptung: $f(n) \leq 20 \cdot c \cdot n$

Beweis durch vollständige Induktion

Verankerung (n < 50): $f(n) \leq c \leq 20 \cdot c \cdot n$

Sei Behauptung bis n-1 bewiesen.

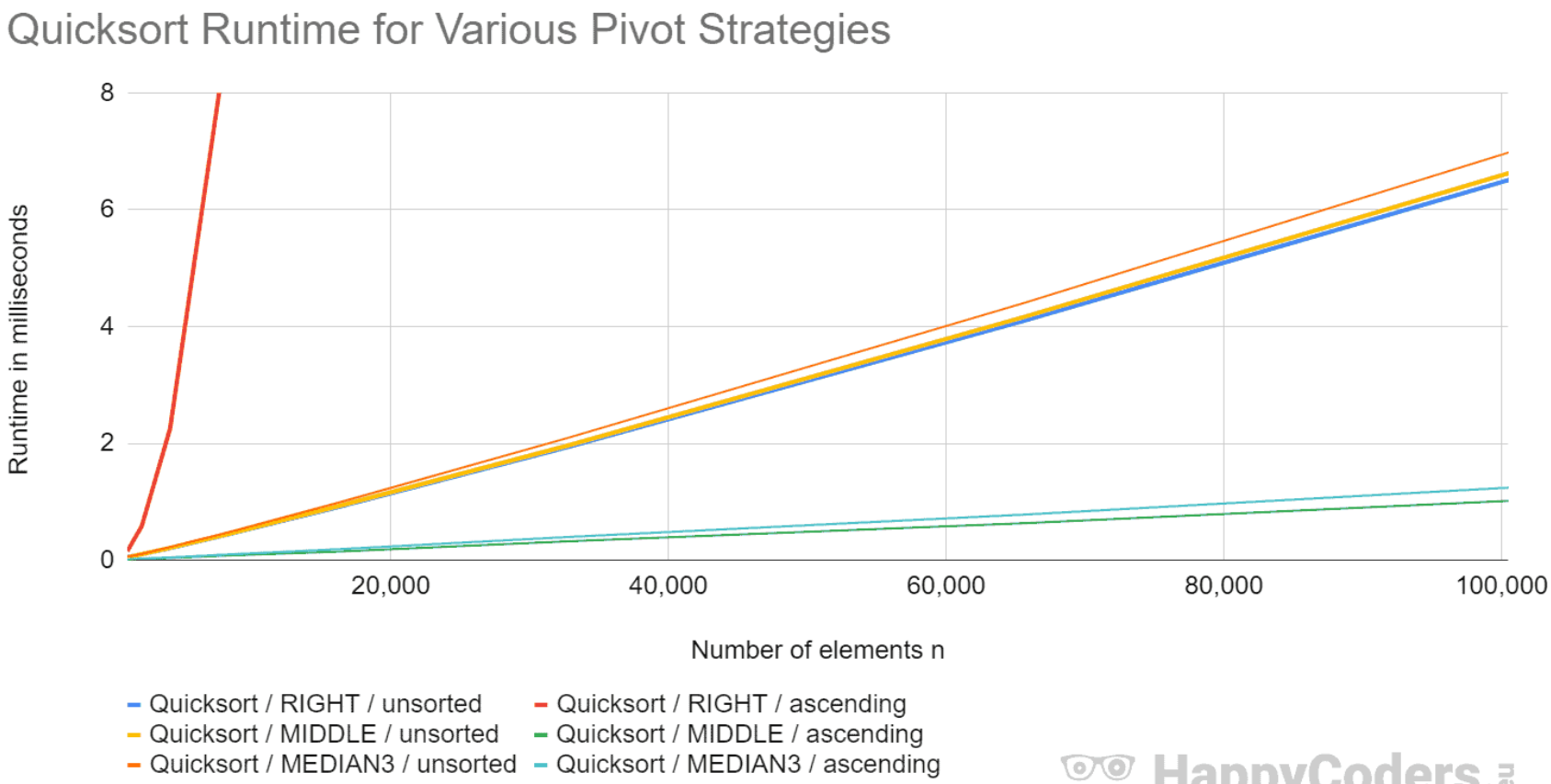
$$f(n) \leq c \cdot n + f(\frac{n}{5}) + f(\frac{3}{4}n)$$

$$f(n) \leq 1 \cdot c \cdot n + 20 \cdot c \cdot \frac{n}{5} + 20 \cdot c \cdot \frac{3}{4}$$

$$f(n) \leq 1 \cdot c \cdot n + 4 \cdot c \cdot n + 15 \cdot c \cdot n = 20 \cdot c \cdot n$$

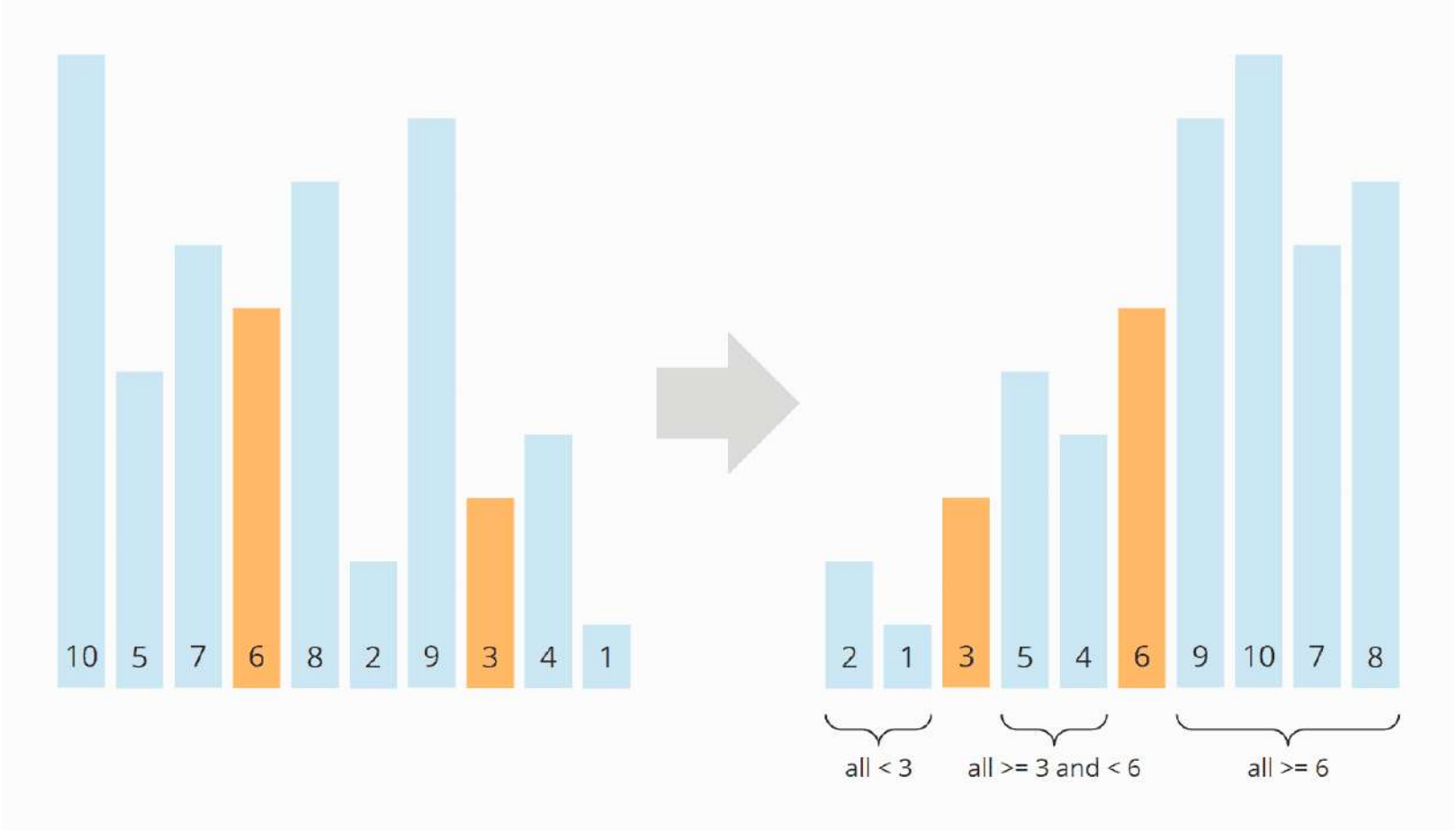
$f(n) \in \mathcal{O}(n)$

Laufzeitentwicklung qualitativ



Quelle: https://www.happycoders.eu/de/algorithmen/quicksort/

Dual-Pivot-Quicksort



Kombination mit Insertion Sort

- Insertion Sort ist für kleine Arrays oft schneller als Quick Sort
- Rekursionsoverhead etc.
- Daher werden die Verfahren oft kombiniert
- Die Sortierfunktion in Java macht Dual-Pivot mit Schwellwert 44

https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/DualPivotQuicksort.java