Rekursion

Andreas M. Chwatal

Programmieren und Software-Engineering
Theorie

12. März 2024

Definition

Definition (Rekursion)

Unter einer rekursiven Methode versteht man eine Methode die sich selbst aufruft.

- Die Definition gilt auch für *Funktionen* oder *Programme* (statt Methoden).
- Viele Problemstellungen/Algorithmen lassen sich sehr einfach und elegant mittels Rekursion formulieren.
- Abbruchbedingung: muss vorhanden sein um die rekursiven Aufrufe zu beenden.

Fakultät

In der Mathematik bezeichnet $n!$ die Fakultät, und es gilt

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

für alle $n > 0$ und $0! = 1$.

Rekursive Formulierung der Fakultät:

$$n! = n \cdot (n-1)! \qquad \text{für } n > 0$$

$$0! = 1$$

Direkte Umsetzung in Programmcode:

```
1       long factorial(long n) {
2           if (n == 0) return 1;
3           return n * factorial(n-1);
4       }
```

Fakultät

In der Mathematik bezeichnet $n!$ die Fakultät, und es gilt

$$n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

für alle $n > 0$ und $0! = 1$.

Rekursive Formulierung der Fakultät:

$$n! = n \cdot (n-1)! \qquad \text{für } n > 0$$

$$0! = 1$$

Direkte Umsetzung in Programmcode:

```
1    long factorial(long n) {
2        if (n == 0) return 1;
3        return n * factorial(n-1);
4    }
```

Achtung: der Aufruf `factorial(-1)` führt zu einer endlosen Rekursion! Dieser Fall sollte im Code abgefangen werden.

Negativbeispiel: Fibonacci-Zahlen

Die Folge der Fibonacci-Zahlen ist für $n \geq 2$ definiert als

$$f_n = f_{n-1} + f_{n-2},$$

weiters gilt $f_0 = f_1 = 1$.

Hierdurch erhalten wir die Folge:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \ldots$$

Negativbeispiel: Fibonacci-Zahlen

Die direkte Umsetzung als rekursives Programm ist jedoch sehr unvorteilhaft:

```
1       long fibonacci(long n) {
2           if (n <= 1) return 1;
3           return fibonacci(n-1) + fibonacci(n-2);
4       }
```

Negativbeispiel: Fibonacci-Zahlen

Die direkte Umsetzung als rekursives Programm ist jedoch sehr unvorteilhaft:

```
1       long fibonacci(long n) {
2           if (n <= 1) return 1;
3           return fibonacci(n-1) + fibonacci(n-2);
4       }
```

Frage

Wo genau liegt das Problem bei der rekursiven Implementierung der Fibonacci-Zahlen?

Negativbeispiel: Fibonacci-Zahlen

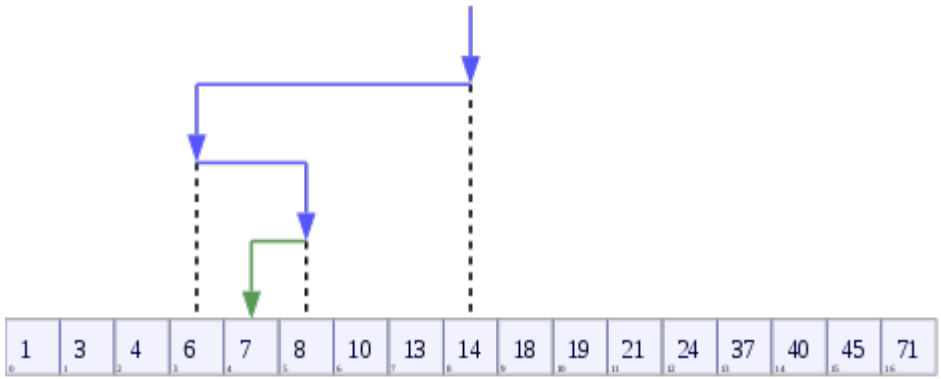
Die *iterative* Implementierung ist hier vorteilhaft, da wesentlich weniger Berechnungen ausgeführte werden:

```
1     long fibonacci(long n) {
2         if (n < 0 || n > MAX) {
3             // Ausnahmebehandlung
4         }
5         long[] f = new long[n+1];
6         f[0] = 1; f[1] = 1;
7         for (long i=2; i<=n; i++) {
8             f[i] = f[i-1] + f[i-2];
9         }
10        return f[n];
11    }
```

Anmerkung: Die Berechnung des n-ten Folgegliedes ist auch ohne Array möglich!

Suche in sortierten Listen

- Aufgabenstellung: Suchen eines Elementes in einer sortierten Liste (Array)
- Naiver Zugang: $O(n)$ Schritte (Erwartungswert $n/2$)
- Mittels binärer Suche: nur $O(\log n)$ Schritte notwendig
- Vorgehensweise: analog zu Suche in Telefonbuch
  - Beliebige Seite aufschlagen
  - Steht Name davor oder danach?
  - Weitere Suche erfolgt nur mehr im verbleibenden Teil



Binäre Suche

Algorithm 1: (Iterative) Binäre Suche

```
1  Function BinarySearch(sorted array a[], element e)
       Result: found element, or NULL if element not
               exists
2      l = 0;
3      r = a.length;
4      while l ≤ r do
5          m = l + (r−l)/2;
6          if (a[m] == e) then
7              return e;
8          else
9              if a[m] > e then
10                 r = m − 1;
11             else
12                 l = m + 1;
13     return NULL;
```

Rekursive Binäre Suche

Algorithm 2: Rekursive Binäre Suche

```
1  Function RecBinarySearch(sorted array a[], element e, l, r)
       Result: found element, or NULL if element not exists
2      if l ≤ r then
3          m = l + (r−l)/2;
4          if (a[m] == e) then
5              return e;
6          else
7              if a[m] > e then
8                  return RecBinarySearch(a, e, m+1, r);
9              else
10                 return RecBinarySearch(a, e, l, m-1);
11     else
12         return NULL;
```

Aufruf mit array a und gesuchtem Element e und `RecBinarySearch(a, e, 0, a.length-1)`.

Rekursive Binäre Suche

Eine rekursive Variante in Python:

```
1   def binaersuche_rekursiv(werte, gesucht, start, ende):
2       if ende < start:
3           return 'nicht␣gefunden'
4           # alternativ: return -start # bei (-Returnwert) waere
5           # die richtige Einfuege-Position
6
7       mitte = (start + ende) // 2
8       if werte[mitte] == gesucht:
9           return mitte
10      elif werte[mitte] < gesucht:
11          return binaersuche_rekursiv(werte, gesucht, mitte+1, ende)
12      else:
13          return binaersuche_rekursiv(werte, gesucht, start, mitte-1)
14
15  def binaersuche(werte, gesucht):
16      return binaersuche_rekursiv(werte, gesucht, 0, len(werte)-1)
```

Quelle: Wikipedia

Tiefensuche

Bei der Tiefensuche ist eine rekursive Implementierung naheliegend. Diese Variante durchläuft alle vom ersten Aufruf mit Knoten $v \in V(G)$ (`RecDFS(v)`) aus erreichbaren Knoten des Graphen G .

Algorithm 3: Rekursive Tiefensuche

```
1 Function RecDFS(Knoten v)
2     markiere v als besucht;
3     for all [v, u] ∈ E(G) do
4         if u noch nicht besucht then
5             RecDFS(u);
```

In der rekursiven Version der Tiefensuche ist kein *Stack* notwendig, die Knoten werden durch die rekursiven Aufrufe automatisch in der richtigen Reihenfolge besucht.

Tiefensuche

Diese Variante sucht einen Knoten s und returniert ihn sobald gefunden:

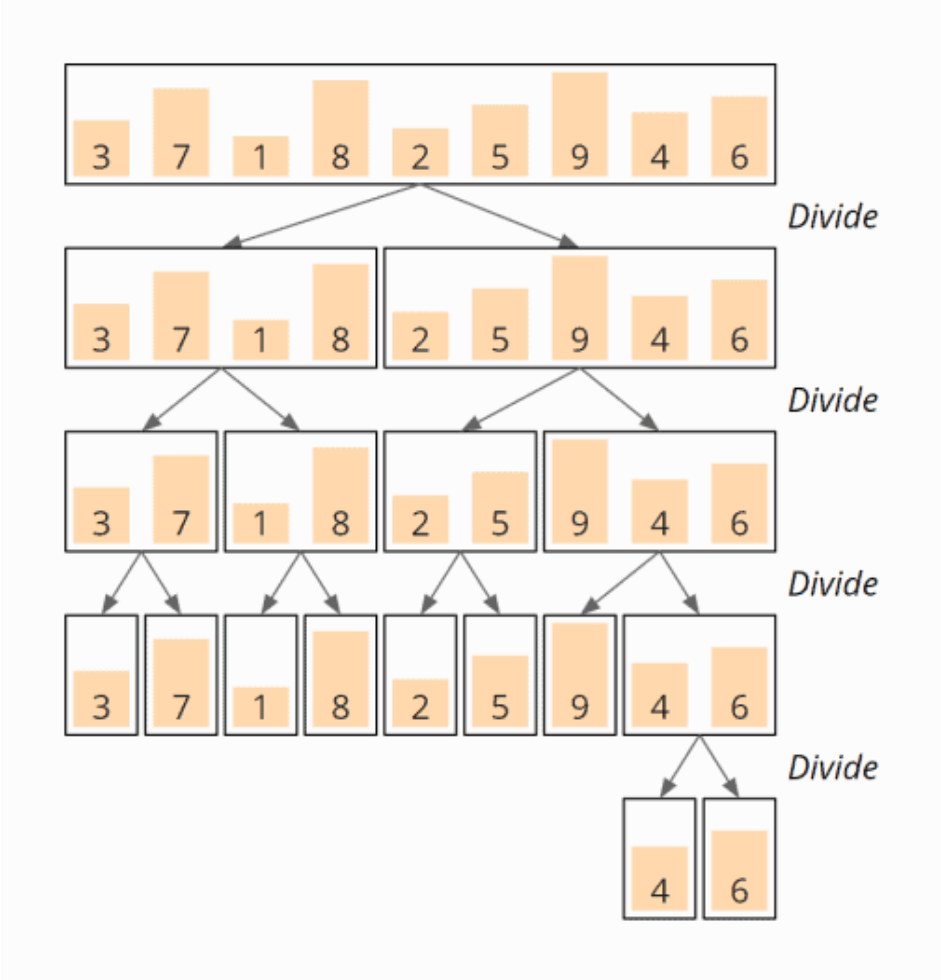
Algorithm 4: Rekursive Tiefensuche (2)

```
1 Function RecDFS(Knoten v, Gesuchter Knoten s)
2     if v gesuchter Knoten s then
3         return v;
4     markiere v als besucht;
5     for all [v, u] ∈ E(G) do
6         if u noch nicht besucht then
7             if RecDFS(u, s) == s then
8                 return s;
9     return null;
```

Mergesort

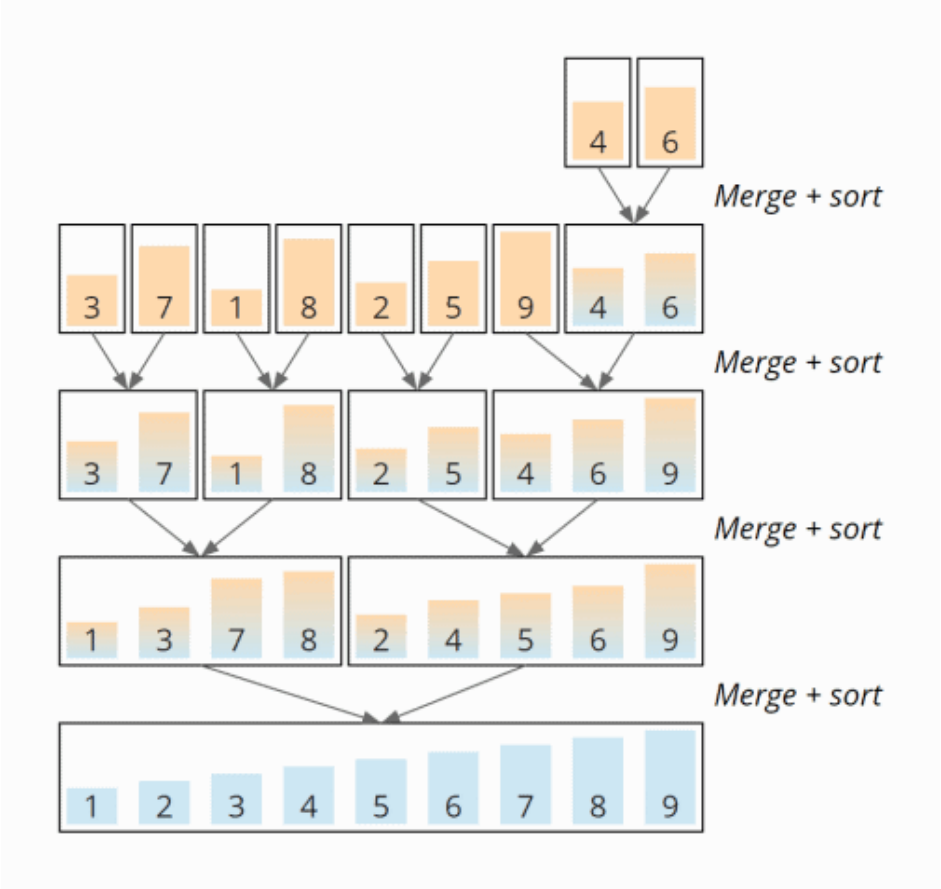
- Der Algorithmus *Mergesort* sortiert die Daten nach dem Prinzip **Divide & Conquer** (dt.: Teile und Herrsche)
- Vorgehensweise bei Divide & Conquer:
  - *Teile* das Problem in kleinere Teilprobleme
  - *Löse* diese Teilprobleme
  - Füge Teillösungen zusammen
- **Mergesort**
  - Zahlenfolge (Array) wird durch rekursive Aufrufe unterteilt.
  - Die Sortierung wird beim anschließenden Zusammenfügen der Arrays erreicht
  - Mergesort weist bessere Laufzeit-Eigenschaften als die bisher besprochenen Sortieralgorithmen auf!

Mergesort



Quelle: `https://www.happycoders.eu/de/algorithmen/mergesort/`

Mergesort



Quelle: `https://www.happycoders.eu/de/algorithmen/mergesort/`

Mergesort

Mergesort-Code in Java. Zunächst der erste Aufruf:

```
1       int[] elements = { 3, 1, 6, 7, 4, 12 }; // zu sortierendes Array
2       int[] sorted = mergeSort(elements, 0, elements.length - 1);
```

Die rekursive Methode sieht wie folgt aus:

```
1       int[] mergeSort(int[] elements, int left, int right) {
2
3           if (left == right) return new int[]{ elements[left] };
4
5           int middle = left + (right - left) / 2;
6           int[] leftArray = mergeSort(elements, left, middle);
7           int[] rightArray = mergeSort(elements, middle + 1, right);
8           return merge(leftArray, rightArray);
9       }
```

```java
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;
  }
```

Quicksort

- Effizienter Sortieralgorithmus von Tony Hoare (1959)
- Ebenso Divide & Conquer
- Array wird anhand von *Pivot-Element* rekursiv geteilt.
- Als Pivot wird oft das letzte Element herangezogen (aber auch andere Varianten sind möglich).
- Trotz Worst-Case-Laufzeit von $O(n^2)$ in der Praxis schneller als Mergesort.
- Der Average-Case $O(n \log n)$ tritt so gut wie immer ein!

Quicksort

```
1  Function Quicksort(left, right)
2      if left < right then
3          pidx = partition(left, right);
4          quicksort(left, pidx - 1);
5          quicksort(pidx + 1, right);
```

```
1  Function Partition(left, right)
2      pivot = a[right];
3      i = left;
4      j = right - 1;
5      while i < j do
6          while a[i] < pivot do
7              i++;
8          while j > left && a[j] ≥ pivot do
9              j- -;
10         if i < j then
11             vertausche a[i] mit a[j];
12             i++;
13             j- -;
14     if i == j && a[j] < pivot then
15         i++;
16     if a[i] != pivot then
17         vertausche a[i] mit a[right];
18     return i;
```

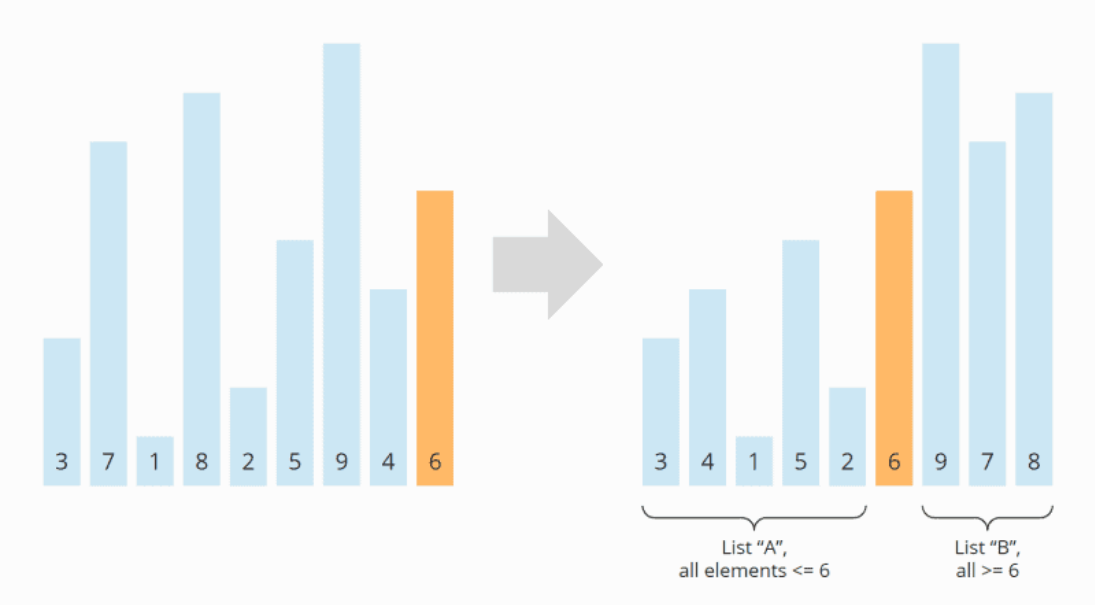
Ablauf (Pivot in []) mit der Zahlenfolge 3, 7, 1, 8, 2, 5, 9, 4, 6:

```
3  7  1  8  2  5  9  4  6

3  7  1  8  2  5  9  4 [6]
3  4  1  8  2  5  9  7  6
3  4  1  5  2  8  9  7  6
3  4  1  5  2  6  9  7  8
3  4  1  5  2 [6] 9  7  8

3  4  1  5 [2] 6  9  7  8
1  4  3  5  2  6  9  7  8
1  2  3  5  4  6  9  7  8
1 [2] 3  5  4  6  9  7  8

1  2  3  5 [4] 6  9  7  8
1  2  3  4  5  6  9  7  8
1  2  3 [4] 5  6  9  7  8

1  2  3  4  5  6  9  7 [8]
1  2  3  4  5  6  7  9  8
1  2  3  4  5  6  7  8  9
1  2  3  4  5  6  7 [8] 9
```

Bestimmung aller Permutationen

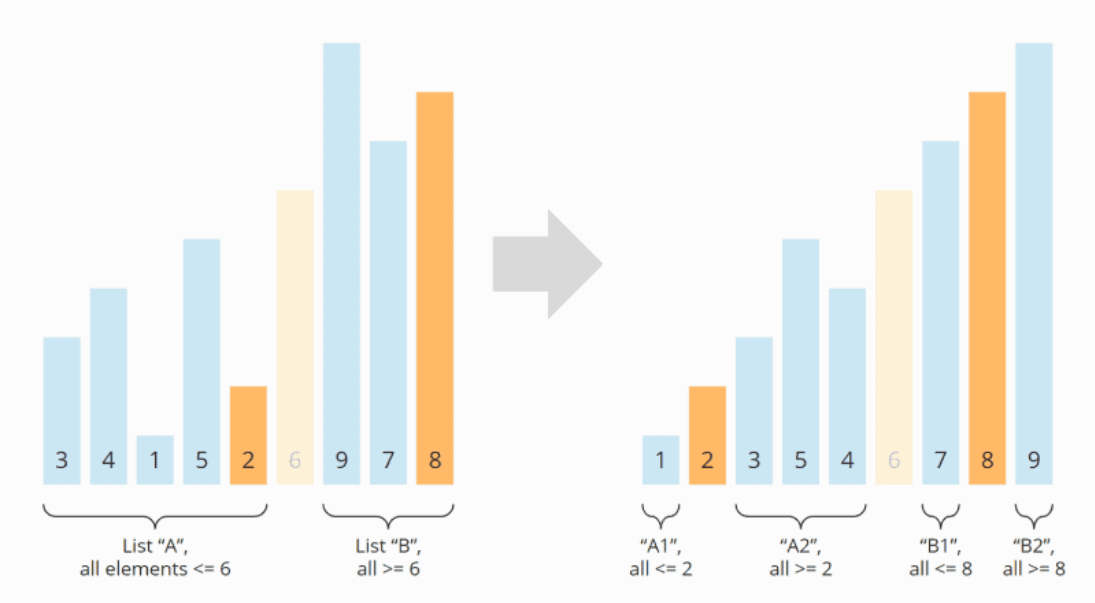
```
1 Function permutate(left, right)
2     if left < right then
3         for i=l; ij=r; i++ do
4             quicksort(left, pidx - 1);
5             quicksort(pidx + 1, right);
```

Quicksort



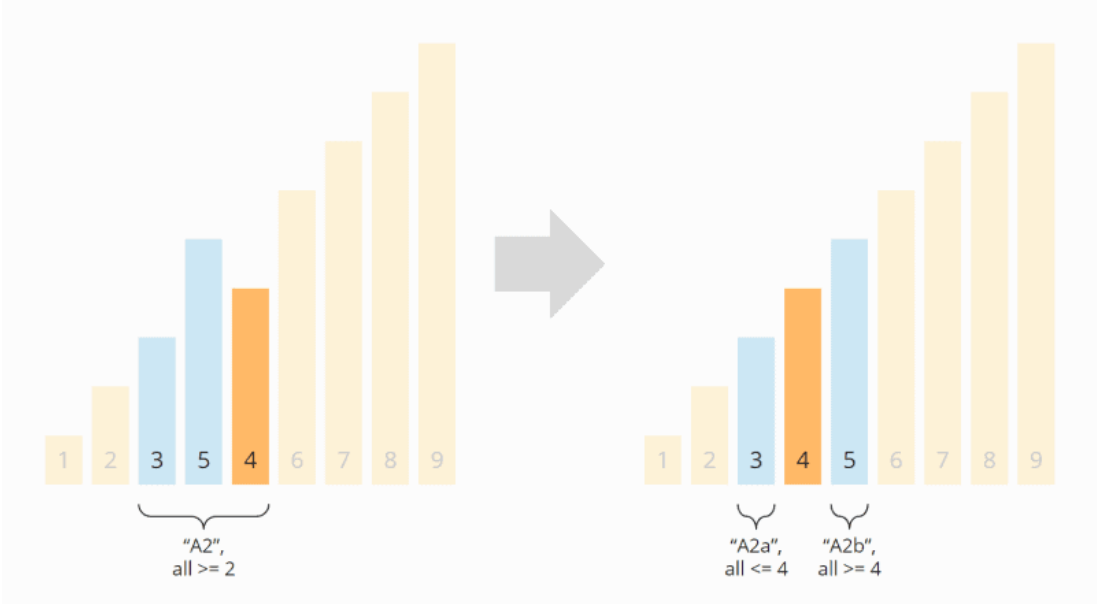
Quelle: `https://www.happycoders.eu/de/algorithmen/quicksort/`

Quicksort



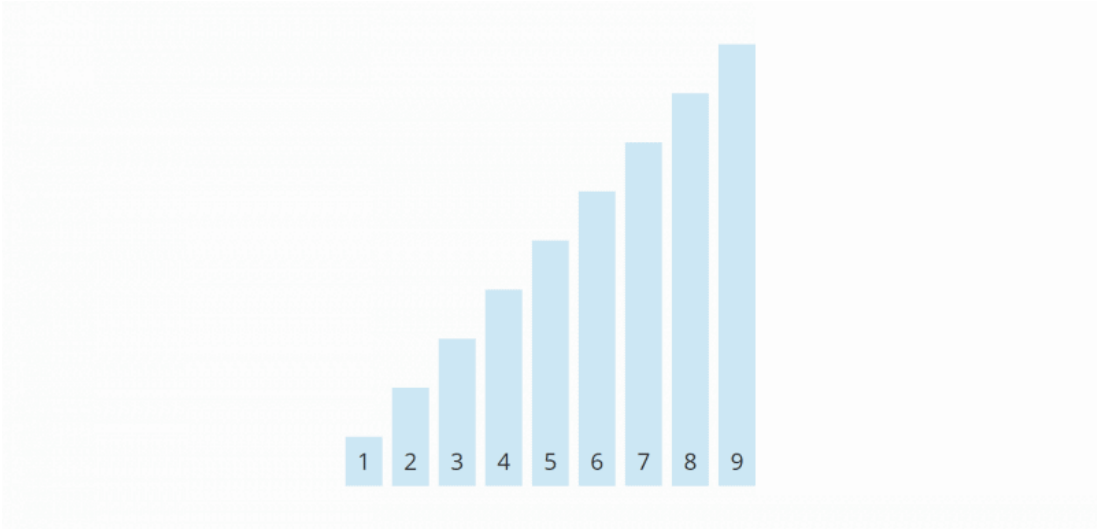
Quelle: `https://www.happycoders.eu/de/algorithmen/quicksort/`

Quicksort



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

Quicksort



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

Programmierbeispiel 12.1.1

Gegeben ist ein 2-dimensionales Boolean-Array, befüllt mit *wahr* und *falsch*. Ermitteln Sie mit einem Programm die Größe des größten zusammenhängenden Gebietes mit Wert wahr.

Beispiel: hier wird falsch mit einem '.' und wahr mit einem 'X' dargestellt.

```
...XXX..XX..
.XXX.XXXX...
X.XX..XX..XX
X...XXX..XXX
XXX....XXX..
XX.XX..X..XX
...X..X.XXX.
.....XXXX...
```

In diesem Beispiel ist die Größe 19.

Zusatzaufgabe 12.1.2

Füllen Sie ein zweidimensionales Boolean-Array (Größe 300x300) mit Zufallswerten, wobei mit einer Wahrscheinlichkeit von $1/3$ der Wert wahr, und mit $2/3$ der Wert falsch gesetzt werden soll. Wenden Sie den Algorithmus aus dem vorigen Beispiel an. Ist das Ergebnis plausibel? Was passiert wenn man die Wahrscheinlichkeiten tauscht?

Programmierbeispiel 12.2.1

Implementieren Sie Mergesort in Java.

Programmierbeispiel 12.2.2

Implementieren Sie Quicksort in Java.

```
1   private static boolean[][] werte = {
2                    {false,false,false,true,true,true,false,false,true,true,false,false},
3                    {false,true,true,true,false,true,true,true,true,false,false,false},
4                    {true,false,true,true,false,false,true,true,false,false,true,true},
5                    {true,false,false,false,true,true,true,false,false,true,true,true},
6                    {true,true,true,false,false,false,false,true,true,true,false,false},
7                    {true,true,false,true,true,false,false,true,false,false,true,true},
8                    {false,false,false,true,false,false,true,false,true,true,true,false},
9                    {false,false,false,false,false,true,true,true,true,false,false,false}
10                   };
```