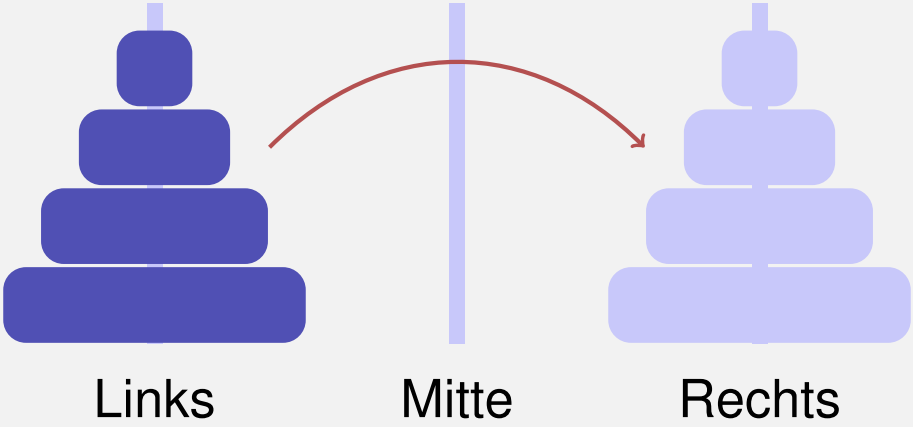
9. Rekursion

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration

Experiment: Die Türme von Hanoi

Links Mitte Rechts

Experiment: Die Türme von Hanoi

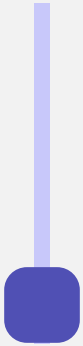


Die Türme von Hanoi - So gehts!

Links Mitte Rechts

Die Türme von Hanoi - So gehts!

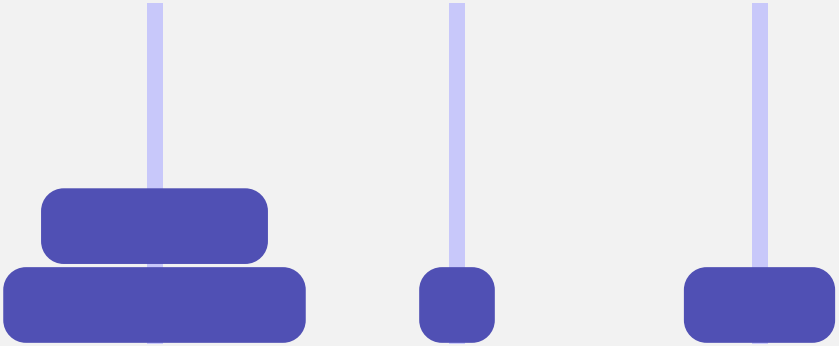
Links



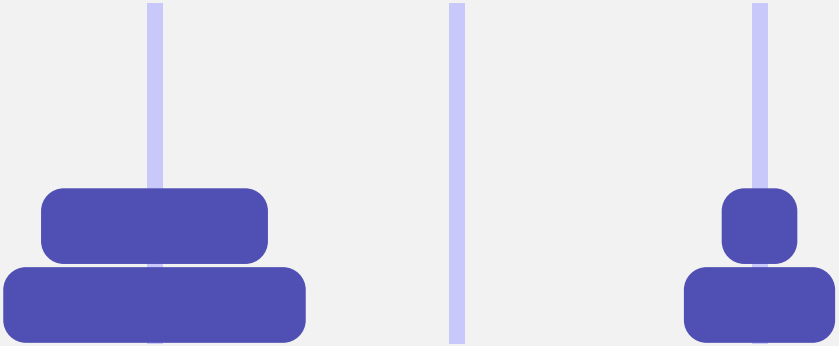
Mitte

Rechts

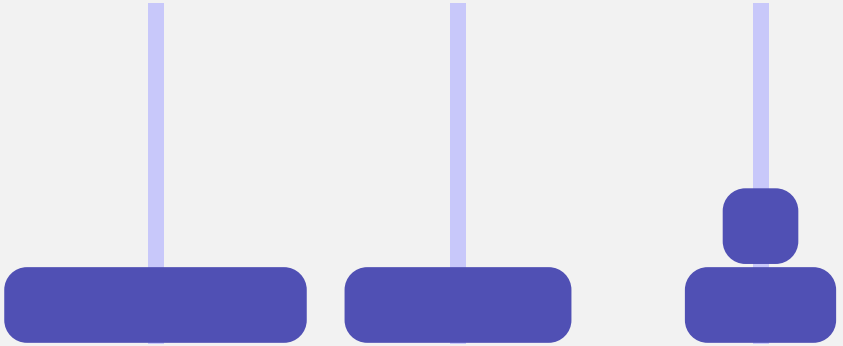
Die Türme von Hanoi - So gehts!



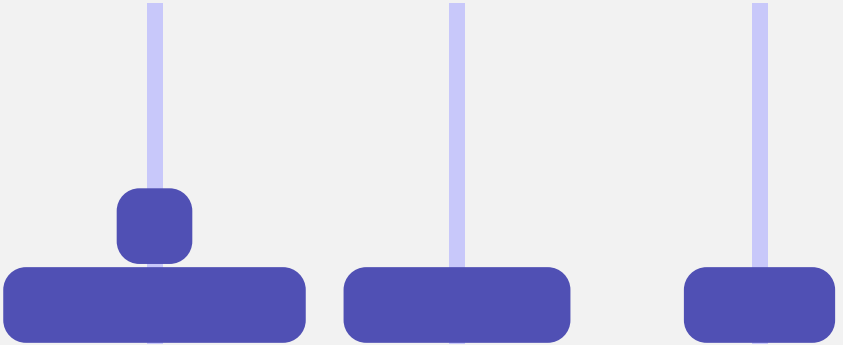
Die Türme von Hanoi - So gehts!



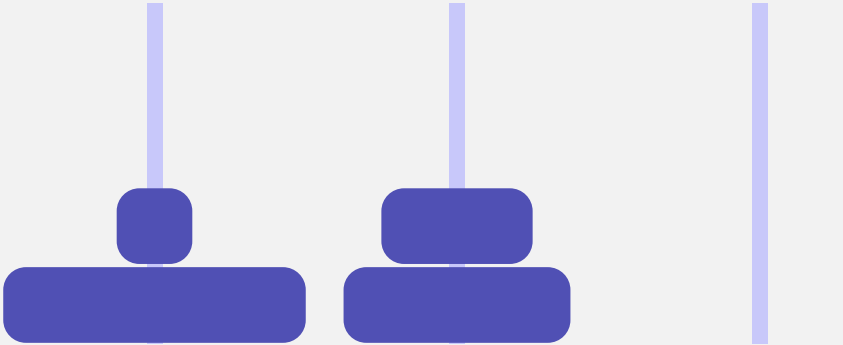
Die Türme von Hanoi - So gehts!



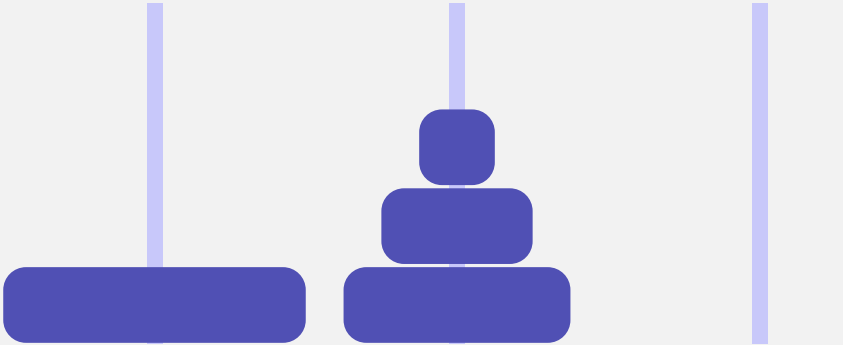
Die Türme von Hanoi - So gehts!



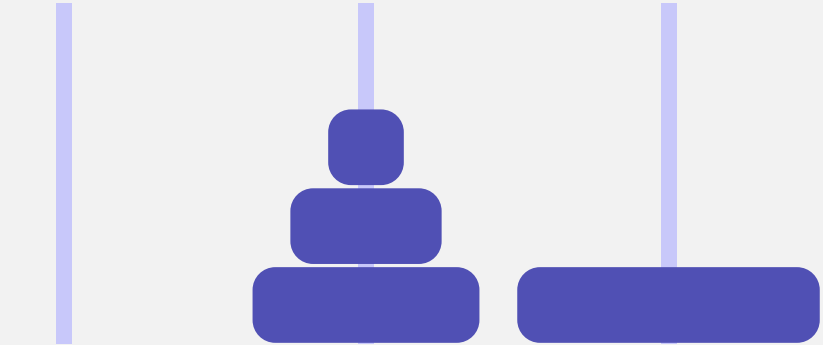
Die Türme von Hanoi - So gehts!



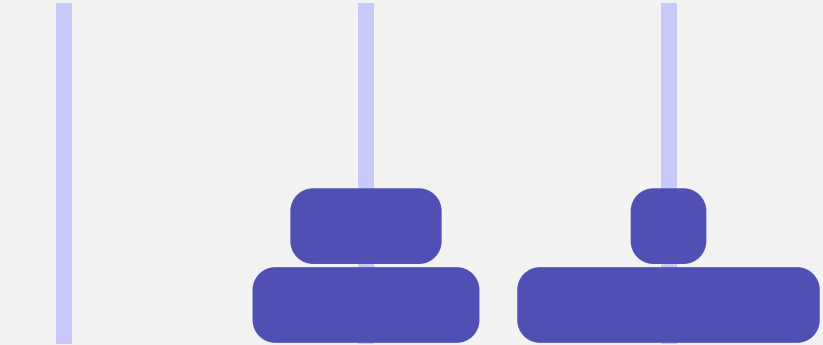
Die Türme von Hanoi - So gehts!



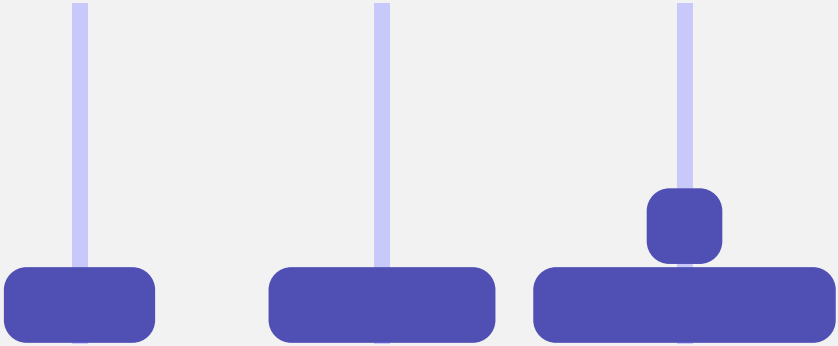
Die Türme von Hanoi - So gehts!



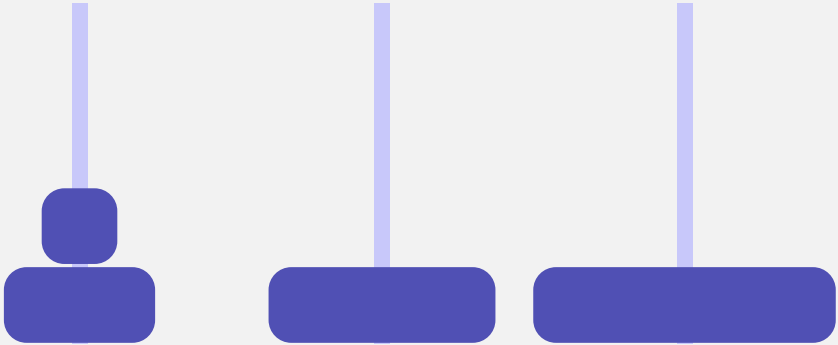
Die Türme von Hanoi - So gehts!



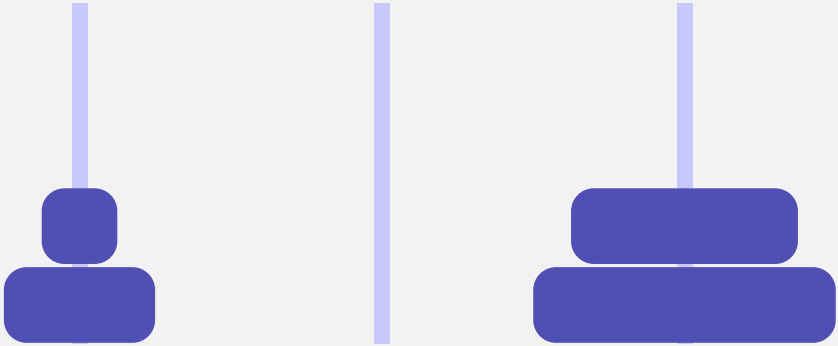
Die Türme von Hanoi - So gehts!



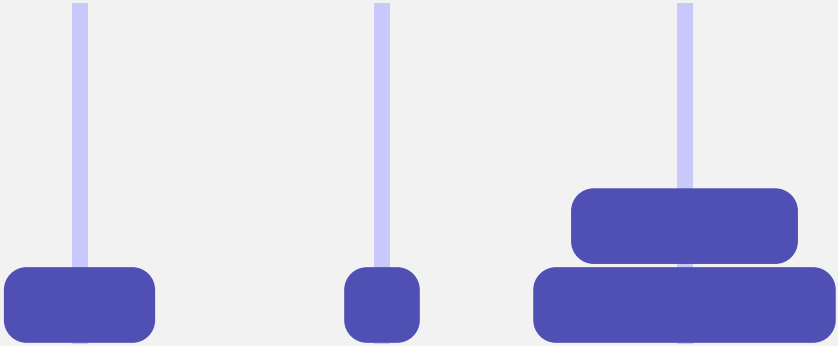
Die Türme von Hanoi - So gehts!



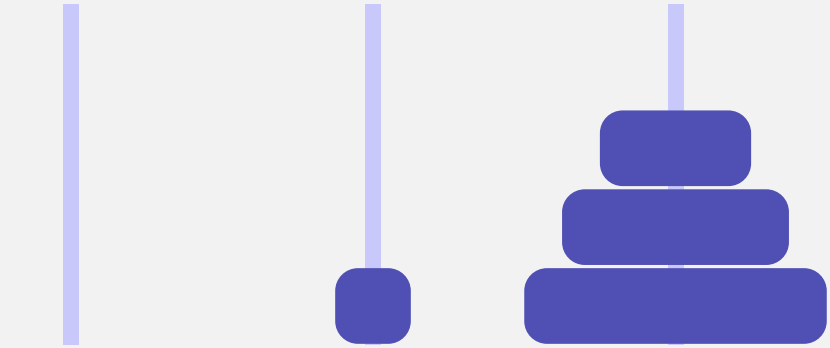
Die Türme von Hanoi - So gehts!



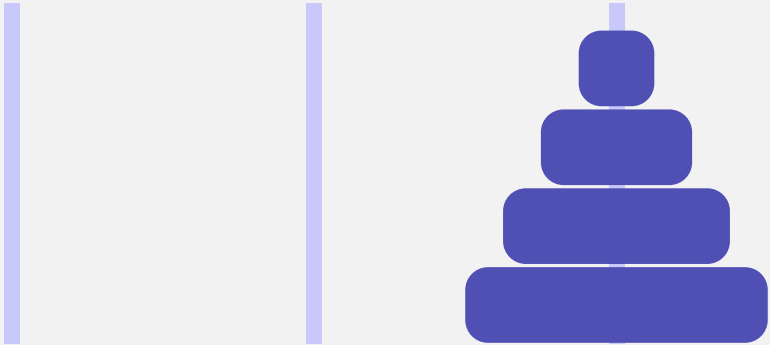
Die Türme von Hanoi - So gehts!



Die Türme von Hanoi - So gehts!



Die Türme von Hanoi - So gehts!

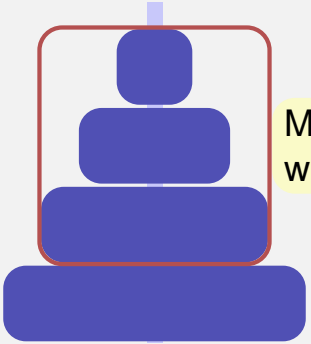


Links Mitte Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz

Links Mitte Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz

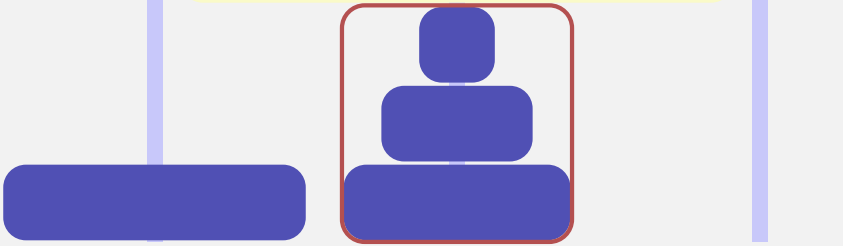


Mal *angenommen*, wir wüssten wie man ...

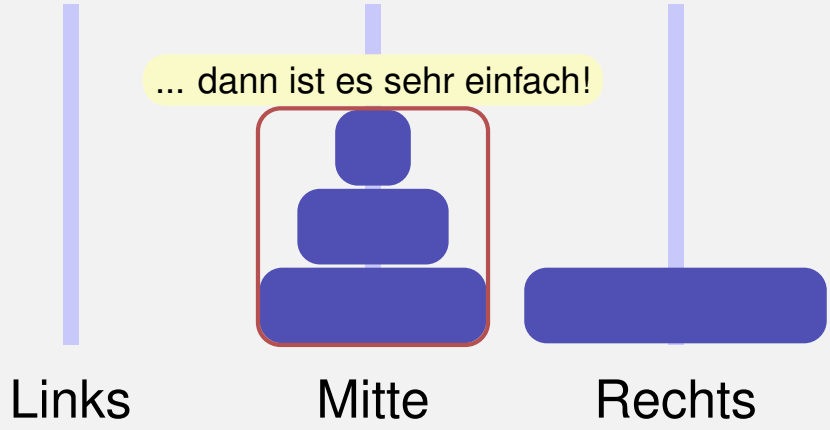
Links Mitte Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz

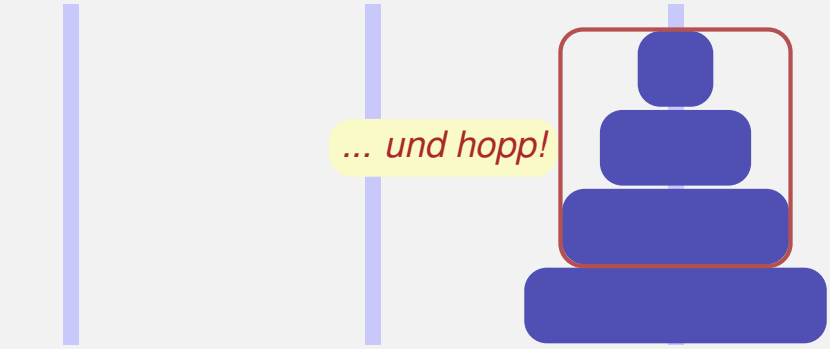
... *drei* Scheiben von einem Stapel zum anderen bringt ...



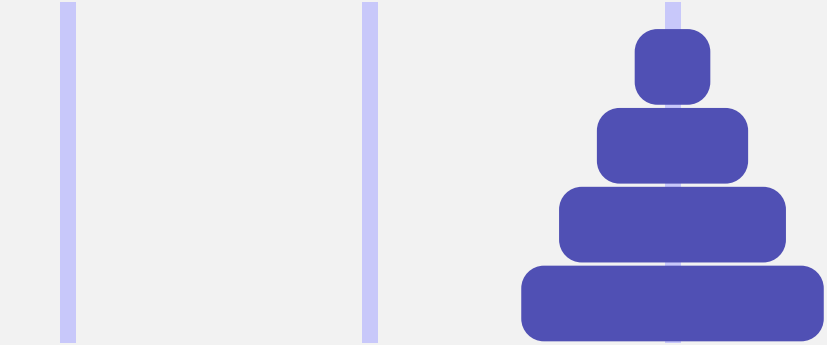
Die Türme von Hanoi - Rekursiver Lösungsansatz



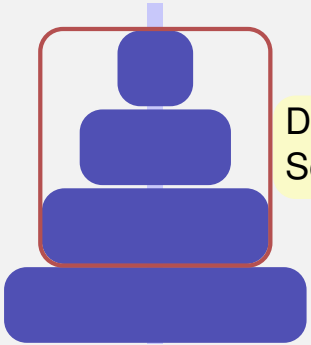
Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



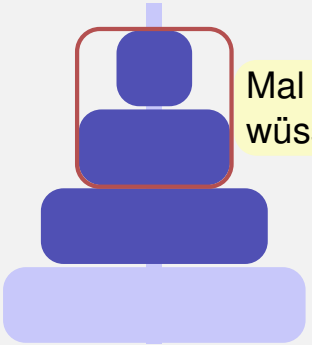
Die Türme von Hanoi - Rekursiver Lösungsansatz



Doch *wie* können wir drei Scheiben bewegen?

Links Mitte Rechts

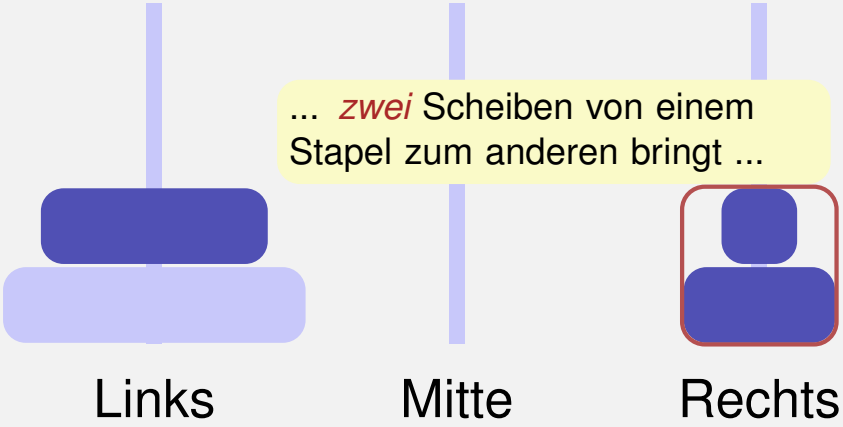
Die Türme von Hanoi - Rekursiver Lösungsansatz



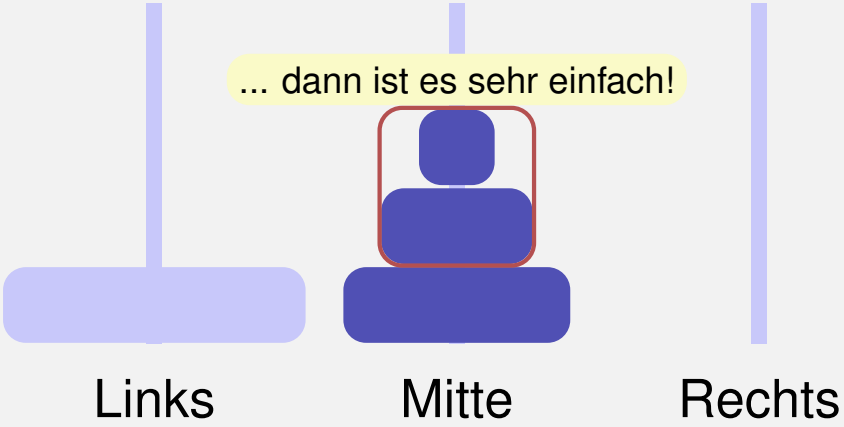
Mal *angenommen*, wir wüssten wie man ...

Links Mitte Rechts

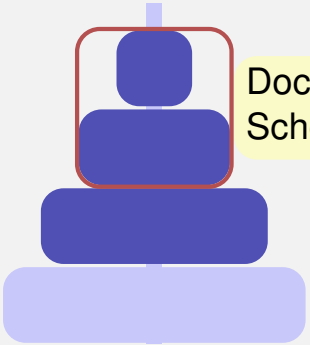
Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz

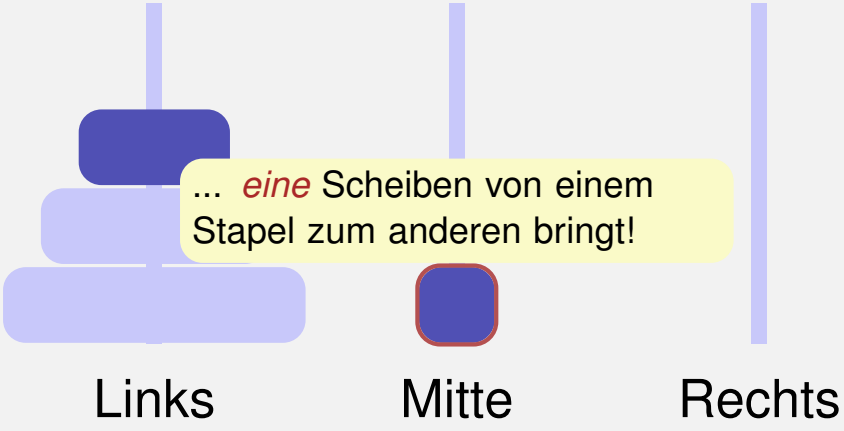


Doch *wie* können wir zwei Scheiben bewegen?

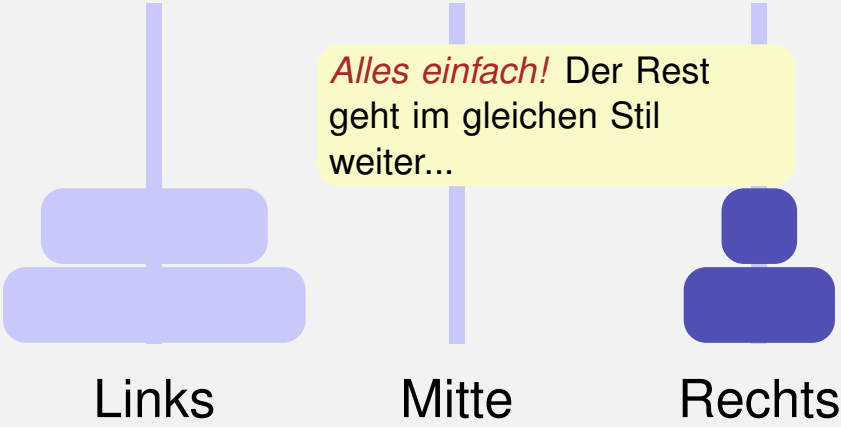
Links Mitte Rechts

Die Türme von Hanoi - Rekursiver Lösungsansatz

Die Türme von Hanoi - Rekursiver Lösungsansatz



Die Türme von Hanoi - Rekursiver Lösungsansatz



Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar.

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar.
- Das heisst, die Funktion erscheint in ihrer eigenen Definition.

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

Rekursion in Java: Genauso!

$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

```
public static int fakultaet(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fakultaet(n-1);
    }
}
```

Rekursion in Java: Genauso!

$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

```
public static int fakultaet(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fakultaet(n−1);
    }
}
```

$n! \Leftrightarrow$ `fakultaet(n)`

$n - 1! \Leftrightarrow$ `fakultaet(n−1)`

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife. . .

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife. . .
- . . . nur noch schlimmer (“verbrennt” Zeit *und* Speicher)

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife...
- ...nur noch schlimmer ("verbrennt" Zeit *und* Speicher)

Beispiel: f() → f() → f() → f() → ... stack overflow

```
public static void f() {
    f();
}
```

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife. . .
- . . . nur noch schlimmer (“verbrennt” Zeit *und* Speicher)

Beispiel: f() → f() → f() → f() → . . . stack overflow

```
public static void f() {
    f();
}
```

Mir san mir.

Bayerisches Lebensmotto

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fakultaet(n)`
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument < n aufgerufen.

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fakultaet(n)`
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument < n aufgerufen.

“n wird mit jedem Aufruf kleiner.”

Rekursive Funktionen: Auswertung

Beispiel: `fakultaet(4)`

```
public static int fakultaet (int n){

    if (n <= 1) return 1;
    return n * fakultaet(n-1); // n > 1
}
```

Aufruf von `fakultaet(4)`

Rekursive Funktionen: Auswertung

Beispiel: `fakultaet(4)`

```
public static int fakultaet (int n){
    // n = 4
    if (n <= 1) return 1;
    return n * fakultaet(n-1); // n > 1
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fakultaet(4)`

```
public static int fakultaet (int n){
    // n = 4
    if (n <= 1) return 1;
    return n * fakultaet(n-1); // n > 1
}
```

Auswertung des Rückgabeausdrucks

Rekursive Funktionen: Auswertung

Beispiel: `fakultaet(4)`

```
public static int fakultaet (int n){
    // n = 4
    if (n <= 1) return 1;
    return n * fakultaet(n-1); // n > 1
}
```

Rekursiver Aufruf mit Argument $n - 1$, also 3

Rekursive Funktionen: Auswertung

Beispiel: `fakultaet(4)`

```
public static int fakultaet (int n){
    // n = 3
    if (n <= 1) return 1;
    return n * fakultaet(n-1); // n > 1
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fakultaet(4)`

```
public static int fakultaet (int n){
    // n = 3
    if (n <= 1) return 1;
    return n * fakultaet(n-1); // n > 1
}
```

Es gibt jetzt zwei n. Das von `fac(4)` und das von `fac(3)`

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fakultaet(4)`

```
public static int fakultaet (int n){
    ⟵ Es wird mit dem n des aktuellen Aufrufs gearbeitet: n = 3
    if (n <= 1) return 1;
    return n * fakultaet(n-1); // n > 1
}
```

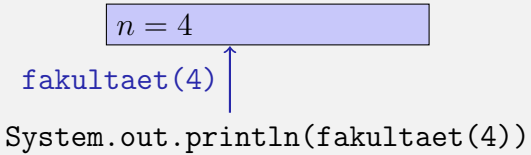
Initialisierung des formalen Arguments

Der Aufrufstapel

```
System.out.println(fakultaet(4))
```

Der Aufrufstapel

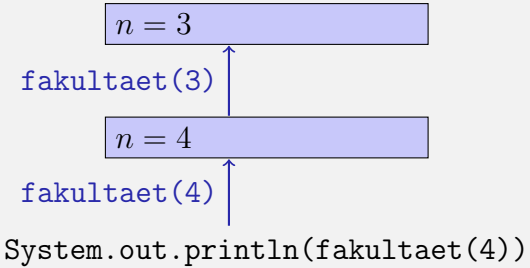
Bei jedem Funktionsaufruf:



Der Aufrufstapel

Bei jedem Funktionsaufruf:

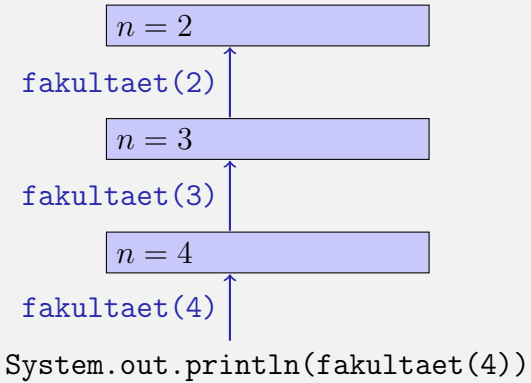
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

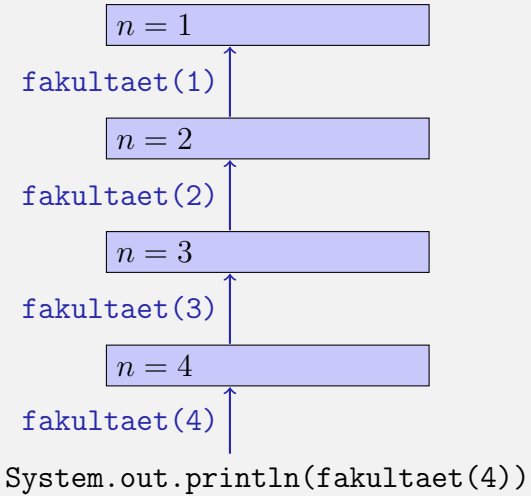
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

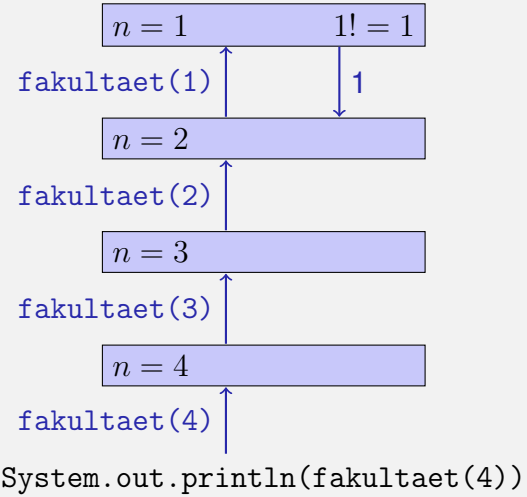
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

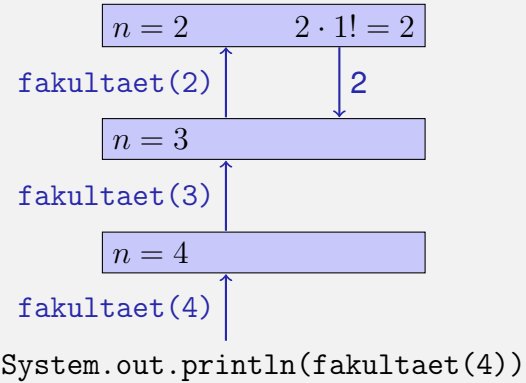
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet



Der Aufrufstapel

Bei jedem Funktionsaufruf:

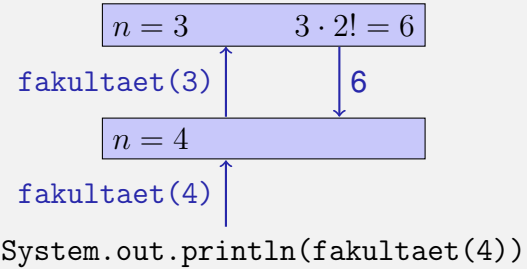
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

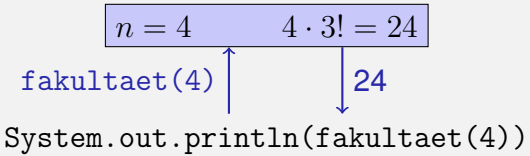
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\gcd(a, b)$ zweier natürlicher Zahlen a und b

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\gcd(a, b)$ zweier natürlicher Zahlen a und b
- basiert auf folgender mathematischen Rekursion:

$$\gcd(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

Euklidischer Algorithmus in Java

$$\gcd(a,b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
public static int gcd(int a, intb){
    if (b == 0) {
        return a;
    } else {
        return gcd(b, a%b);
    }
}
```

Euklidischer Algorithmus in Java

$$\gcd(a,b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
public static int gcd(int a, intb){
    if (b == 0) {
        return a;
    } else {
        return gcd(b, a%b);
    }
}
```

Terminierung: $a \bmod b < b$, also wird b in jedem rekursiven Aufruf kleiner.

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

Resultat: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$

Fibonacci-Zahlen in Zürich

Fibonacci-Zahlen in Java

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

```
public static int fib(int n){
    if (n == 0 || n == 1){
        return n;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Fibonacci-Zahlen in Java

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

```
public static int fib(int n){
    if (n == 0 || n == 1){
        return n;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Korrektheit und Terminierung sind klar, aber...

Fibonacci-Zahlen in Java

Laufzeit

`fib(50)` dauert „ewig”, denn es berechnet
F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal, F_{45} 8-mal, F_{44} 13-mal,
F_{43} 21-mal ... F_1 ca. 10^9 mal (!)

```
public static int fib(int n){
    if (n == 0 || n == 1){
        return n;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen `a` und `b`)!

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen `a` und `b`)!
- Berechne die nächste Zahl als Summe von `a` und `b`!

Schnelle Fibonacci-Zahlen in Java

```
public static int fib (int n)\{
    if (n == 0) return 0;
    if (n <= 2) return 1;
    int a = 1;                          // F(1)
    int b = 1;                          // F(2)
    for (int i = 3; i <= n; ++i){
        int a_old = a;                  // F(i-2)
        a = b;                          // F(i-1)
        b += a_old;                     // F(i-1) += F(i-2) -> F(i)
    }
    return b;
}
```

10. Suchen

Das Suchproblem

Gegeben

- Menge von Datensätzen.

Beispiele

Telefonverzeichnis, Wörterbuch, Symboltabelle

- Jeder Datensatz hat einen Schlüssel k.
- Schlüssel sind vergleichbar: eindeutige Antwort auf Frage $k_1 \leq k_2$ für Schlüssel k_1, k_2.

Aufgabe: finde Datensatz nach Schlüssel k.

Suche in Array

Gegeben

- Array A mit n Elementen $(A[1], \ldots, A[n])$.
- Schlüssel b

Gesucht: Index k, $1 \leq k \leq n$ mit $A[k] = b$ oder ”nicht gefunden”.

22	20	32	10	35	24	42	38	28	41
1	2	3	4	5	6	7	8	9	10

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls* n Vergleiche.

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls* n Vergleiche.
- Annahme: Jede Anordnung der n Schlüssel ist gleichwahrscheinlich. *Erwartete* Anzahl Vergleiche:

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls* n Vergleiche.
- Annahme: Jede Anordnung der n Schlüssel ist gleichwahrscheinlich. *Erwartete* Anzahl Vergleiche:

Lineare Suche

Durchlaufen des Arrays von $A[1]$ bis $A[n]$.

- *Bestenfalls* 1 Vergleich.
- *Schlimmstenfalls* n Vergleiche.
- Annahme: Jede Anordnung der n Schlüssel ist gleichwahrscheinlich. *Erwartete* Anzahl Vergleiche:

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}.$$

Suche in sortierten Array

Gegeben

- Sortiertes Array A mit n Elementen $(A[1], \ldots, A[n])$ mit $A[1] \leq A[2] \leq \cdots \leq A[n]$.
- Schlüssel b

Gesucht: Index k, $1 \leq k \leq n$ mit $A[k] = b$ oder "nicht gefunden".

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

Divide and Conquer!

Suche $b = 23$.

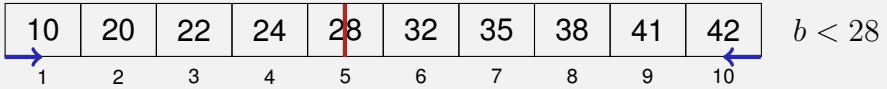
Divide and Conquer!

Suche $b = 23$.

10	20	22	24	28	32	35	38	41	42
1	2	3	4	5	6	7	8	9	10

Divide and Conquer!

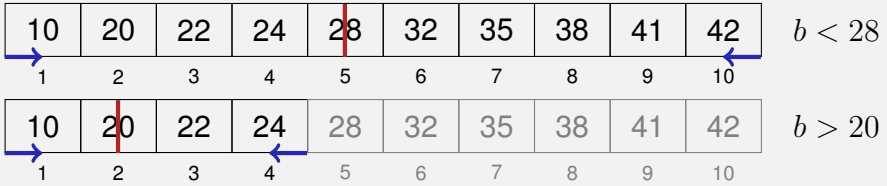
Suche $b = 23$.



$b < 28$

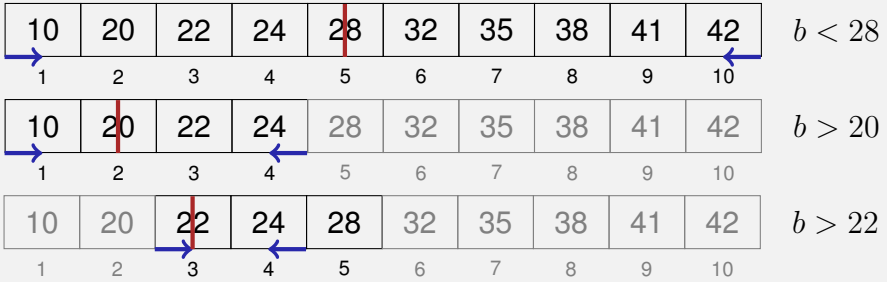
Divide and Conquer!

Suche $b = 23$.



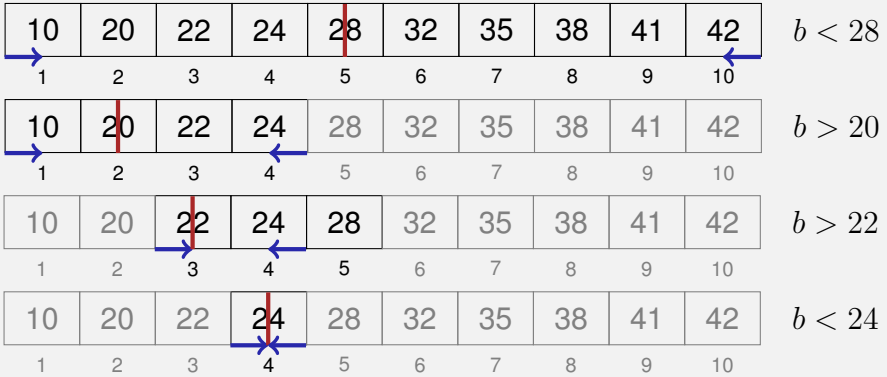
Divide and Conquer!

Suche $b = 23$.



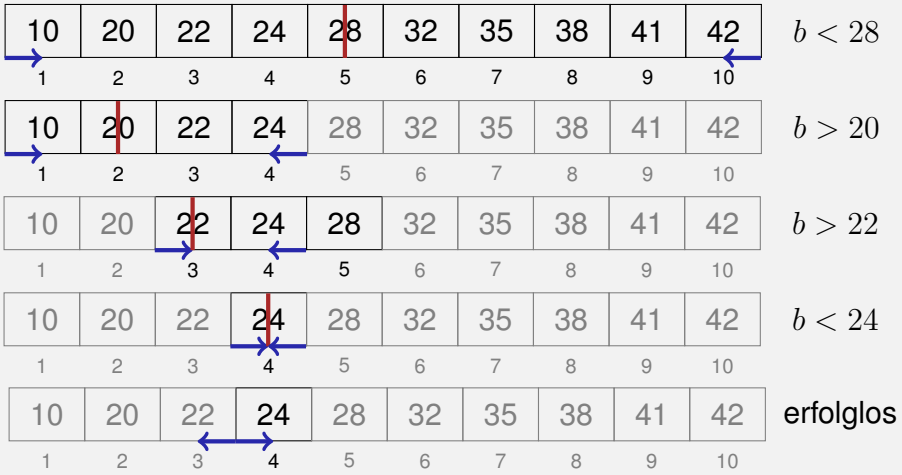
Divide and Conquer!

Suche $b = 23$.



Divide and Conquer!

Suche $b = 23$.



Binärer Suchalgorithmus `BSearch(A,b,l,r)`

Input : Sortiertes Array A von n Schlüsseln. Schlüssel b. Bereichsgrenzen $1 \leq l \leq r \leq n$ oder $l > r$ beliebig.
Output : Index des gefundenen Elements. 0, wenn erfolglos.

$m \leftarrow \lfloor (l+r)/2 \rfloor$
if $l > r$ **then** // erfolglose Suche
    return 0
else if $b = A[m]$ **then**// gefunden
    return m
else if $b < A[m]$ **then**// Element liegt links
    return BSearch$(A, b, l, m-1)$
else // $b > A[m]$: Element liegt rechts
    return BSearch$(A, b, m+1, r)$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c$$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \end{aligned}$$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c. \end{aligned}$$

Analyse (Schlimmster Fall)

Rekurrenz ($n = 2^k$)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Teleskopieren:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + 2c \\ &= T\left(\frac{n}{2^i}\right) + i \cdot c \\ &= T\left(\frac{n}{n}\right) + \log_2 n \cdot c. \end{aligned}$$

$\Rightarrow$ Annahme: $T(n) = d + c \log_2 n$

Analyse (Schlimmster Fall)

$$T(n) = \begin{cases} d & \text{falls } n = 1, \\ T(n/2) + c & \text{falls } n > 1. \end{cases}$$

Vermutung : $T(n) = d + c \cdot \log_2 n$

Beweis durch Induktion:

- Induktionsanfang: $T(1) = d$.
- Hypothese: $T(n/2) = d + c \cdot \log_2 n/2$
- Schritt ($n/2 \to n$)

$$T(n) = T(n/2) + c = d + c \cdot (\log_2 n - 1) + c = d + c \log_2 n.$$

Resultat

Theorem

Der Algorithmus zur binären sortierten Suche benötigt $\Theta(\log n)$ *Elementarschritte.*

Iterativer binärer Suchalgorithmus

Input : Sortiertes Array A von n Schlüsseln. Schlüssel b.
Output : Index des gefundenen Elements. 0, wenn erfolglos.

$l \leftarrow 1$; $r \leftarrow n$
while $l \leq r$ **do**
    $m \leftarrow \lfloor (l + r)/2 \rfloor$
    if $A[m] = b$ **then**
        return m
    else if $A[m] < b$ **then**
        $l \leftarrow m + 1$
    else
        $r \leftarrow m - 1$
return 0;

Korrektheit

Algorithmus bricht nur ab, falls A leer oder b gefunden.

Invariante: Falls b in A, dann im Bereich $A[l, ..., r]$

Beweis durch Induktion

- Induktionsanfang: $b \in A[1, .., n]$ (oder nicht)
- Hypothese: Invariante gilt nach i Schritten
- Schritt:
  $b < A[m] \Rightarrow b \in A[l, .., m-1]$
  $b > A[m] \Rightarrow b \in A[m+1, .., r]$

11. Auswählen

Min und Max

? Separates Finden von Minimum und Maximum in $(A[1], \ldots, A[n])$ benötigt insgesamt $2n$ Vergleiche. (Wie) geht es mit weniger als $2n$ Vergleichen für beide gemeinsam?

Min und Max

? Separates Finden von Minimum und Maximum in $(A[1], \ldots, A[n])$ benötigt insgesamt $2n$ Vergleiche. (Wie) geht es mit weniger als $2n$ Vergleichen für beide gemeinsam?

! Es geht mit $\frac{3}{2}N$ Vergleichen: Vergleiche jeweils 2 Elemente und deren kleineres mit Min und grösseres mit Max.

Das Auswahlproblem

Eingabe

- Unsortiertes Array $A = (A_1, \ldots, A_n)$ paarweise verschiedener Werte
- Zahl $1 \leq k \leq n$.

Ausgabe: $A[i]$ mit $|\{j : A[j] < A[i]\}| = k - 1$

Spezialfälle

$k = 1$: Minimum: Algorithmus mit n Vergleichsoperationen trivial.
$k = n$: Maximum: Algorithmus mit n Vergleichsoperationen trivial.
$k = \lfloor n/2 \rfloor$: Median.

Ansätze

Ansätze

- Wiederholt das Minimum entfernen / auslesen: $\mathcal{O}(k \cdot n)$.
  Median: $\mathcal{O}(n^2)$

Ansätze

- Wiederholt das Minimum entfernen / auslesen: $\mathcal{O}(k \cdot n)$.
  Median: $\mathcal{O}(n^2)$
- Sortieren (kommt bald): $\mathcal{O}(n \log n)$

Ansätze

- Wiederholt das Minimum entfernen / auslesen: $\mathcal{O}(k \cdot n)$.
  Median: $\mathcal{O}(n^2)$
- Sortieren (kommt bald): $\mathcal{O}(n \log n)$
- Pivotieren $\mathcal{O}(n)$!

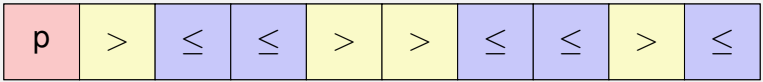
Pivotieren

Pivotieren

1. Wähle ein Element p als Pivotelement

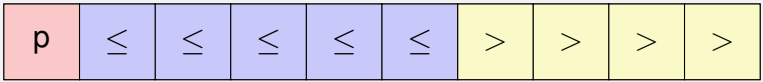
Pivotieren

1. Wähle ein Element p als Pivotelement
2. Teile A in zwei Teile auf, den Rang von p bestimmend.



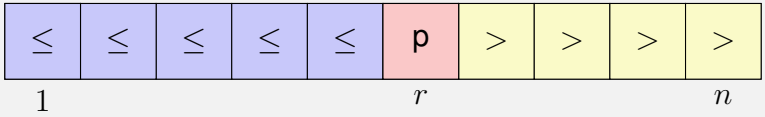
Pivotieren

1. Wähle ein Element p als Pivotelement
2. Teile A in zwei Teile auf, den Rang von p bestimmend.



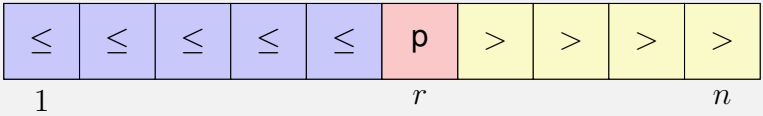
Pivotieren

1. Wähle ein Element p als Pivotelement
2. Teile A in zwei Teile auf, den Rang von p bestimmend.



Pivotieren

1. Wähle ein Element p als Pivotelement
2. Teile A in zwei Teile auf, den Rang von p bestimmend.
3. Rekursion auf dem relevanten Teil. Falls $k = r$, dann gefunden.



Algorithmus Partition($A[l..r], p$)

Input : Array A, welches den Sentinel p im Intervall $[l, r]$ mindestens einmal enthält.
Output : Array A partitioniert in $[l..r]$ um p. Rückgabe der Position von p.

```
while l < r do
    while A[l] < p do
        l ← l + 1
    while A[r] > p do
        r ← r − 1
    swap(A[l], A[r])
    if A[l] = A[r] then
        l ← l + 1
return l-1
```

Korrektheit: Invariante

Invariante I: $A_i \leq p \, \forall i \in [0, l)$, $A_i > p \, \forall i \in (r, n]$, $\exists k \in [l, r] : A_k = p$.

while $l < r$ **do**

  I

  while $A[l] < p$ **do**
    $l \leftarrow l + 1$

  I und $A[l] \geq p$

  while $A[r] > p$ **do**
    $r \leftarrow r - 1$

  I und $A[r] \leq p$

  swap($A[l]$, $A[r]$)

  I und $A[l] \leq p \leq A[r]$

  if $A[l] = A[r]$ **then**
    $l \leftarrow l + 1$

  I

return l-1

Korrektheit: Fortschritt

```
while l < r do
    while A[l] < p do          Fortschritt wenn A[l] < p
        l ← l + 1
    while A[r] > p do          Fortschritt wenn A[r] > p
        r ← r − 1
    swap(A[l], A[r])           Fortschritt wenn A[l] > p oder A[r] < p
    if A[l] = A[r] then        Fortschritt wenn A[l] = A[r] = p
        l ← l + 1
return l-1
```

Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(?)$

Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(?)$

Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(?)$

Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(?)$

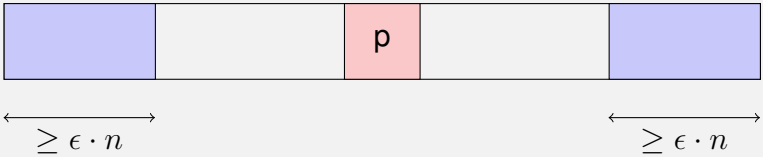
Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(n^2)$

Wahl des Pivots

Das Minimum ist ein schlechter Pivot: worst Case $\Theta(n^2)$

Ein guter Pivot hat linear viele Elemente auf beiden Seiten.



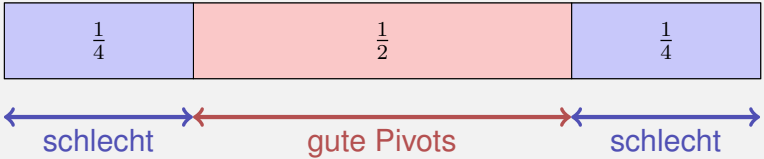
Analyse

Unterteilung mit Faktor q ($0 < q < 1$): zwei Gruppen mit $q \cdot n$ und $(1-q) \cdot n$ Elementen (ohne Einschränkung $q \geq 1 - q$).

$$
\begin{aligned}
T(n) &\leq T(q \cdot n) + c \cdot n \\
&= c \cdot n + q \cdot c \cdot n + T(q^2 \cdot n) = \ldots = c \cdot n \sum_{i=0}^{\log_q(n)-1} q^i + T(1) \\
&\leq c \cdot n \underbrace{\sum_{i=0}^{\infty} q^i}_{\text{geom. Reihe}} = c \cdot n \cdot \frac{1}{1-q} = \mathcal{O}(n)
\end{aligned}
$$

Wie bekommen wir das hin?

Der Zufall hilft uns (Tony Hoare, 1961). Wähle in jedem Schritt einen zufälligen Pivot.



Wahrscheinlichkeit für guten Pivot nach einem Versuch: $\frac{1}{2} =: \rho$.

Wahrscheinlichkeit für guten Pivot nach k Versuchen: $(1-\rho)^{k-1} \cdot \rho$.

Erwartungswert der geometrischen Verteilung: $1/\rho = 2$

Algorithmus Quickselect ($A[l..r], i$)

Input : Array A der Länge n. Indizes $1 \leq l \leq i \leq r \leq n$, so dass für alle $x \in A[l..r]$ gilt, dass $|\{j|A[j] \leq x\}| \geq l$ und $|\{j|A[j] \leq x\}| \leq r$.
Output : Partitioniertes Array A, so dass $|\{j|A[j] \leq A[i]\}| = i$

```
if l=r then return;
repeat
    wähle zufälligen Pivot x ∈ A[l..r]
    p ← l
    for j = l to r do
        if A[j] ≤ x then p ← p + 1
until (l+r)/4 ≤ p ≤ 3(l+r)/4
m ← Partition(A[l..r], x)
if i < m then
    quickselect(A[l..m], i)
else
    quickselect(A[m..r], i)
```