

6 Eine erste Anwendung: Sortieren

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

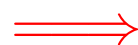
6 Eine erste Anwendung: Sortieren

Gegeben: eine Folge von ganzen Zahlen.

Gesucht: die zugehörige aufsteigend sortierte Folge.

Idee:

- speichere die Folge in einem Feld ab;
- lege ein weiteres Feld an;
- füge der Reihe nach jedes Element des ersten Felds an der richtigen Stelle in das zweite Feld ein!



Sortieren durch **Einfügen** ...

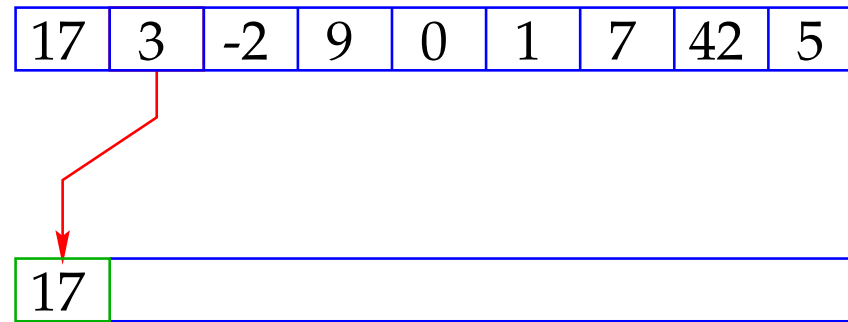
```
public static int[] sort (int[] a) {
    int n = a.length;
    int[] b = new int[n];
    for (int i = 0; i < n; ++i)
        insert (b, a[i], i);
        // b      = Feld, in das eingefügt wird
        // a[i]   = einzufügendes Element
        // i      = Anzahl von Elementen in b
    return b;
} // end of sort ()
```

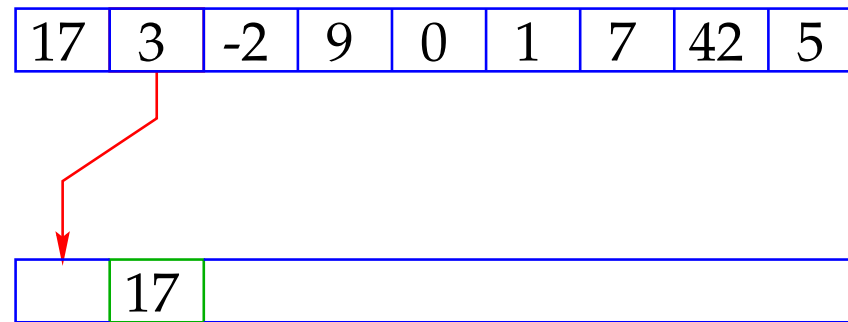
Teilproblem: Wie fügt man ein ???

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---



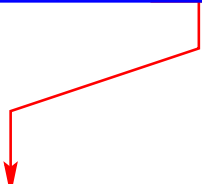
--	--	--	--	--	--	--	--	--





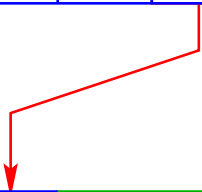
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

3	17							
---	----	--	--	--	--	--	--	--



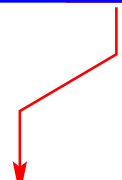
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

	3	17						
--	---	----	--	--	--	--	--	--

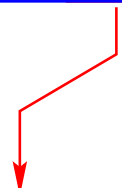


17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	3	17						
----	---	----	--	--	--	--	--	--



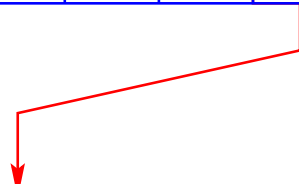
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

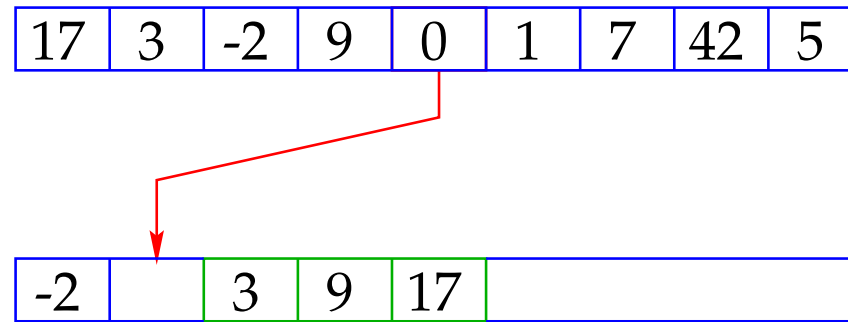


-2	3		17					
----	---	--	----	--	--	--	--	--

17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

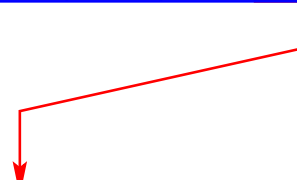
-2	3	9	17					
----	---	---	----	--	--	--	--	--





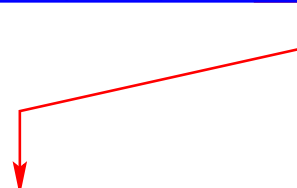
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0	3	9	17				
----	---	---	---	----	--	--	--	--



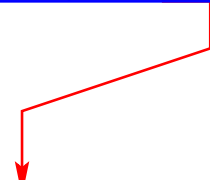
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0		3	9	17			
----	---	--	---	---	----	--	--	--



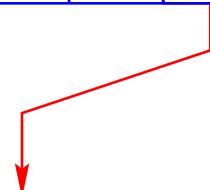
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0	1	3	9	17			
----	---	---	---	---	----	--	--	--



17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

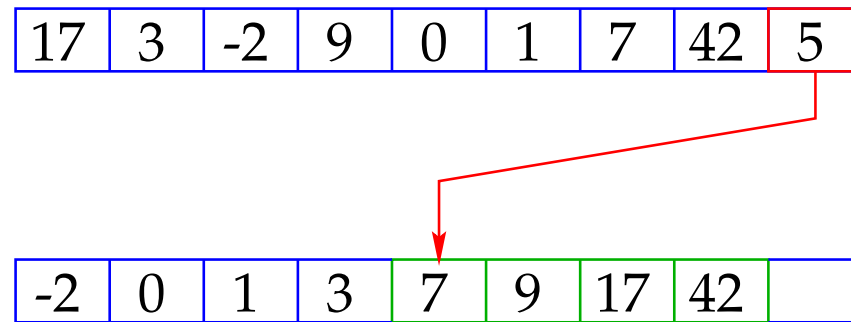
-2	0	1	3		9	17	
----	---	---	---	--	---	----	--

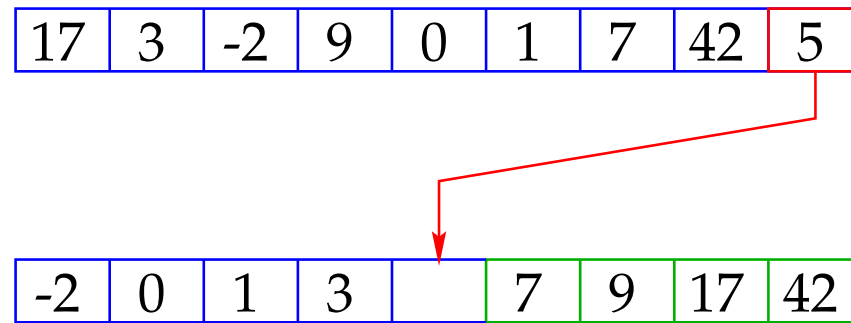


17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---



-2	0	1	3	7	9	17		
----	---	---	---	---	---	----	--	--





17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

-2	0	1	3	5	7	9	17	42
----	---	---	---	---	---	---	----	----

```
public static void insert (int[] b, int x, int i) {  
    int j = locate (b,x,i);  
        // findet die Einfügestelle j für x in b  
    shift (b,j,i);  
        // verschiebt in b die Elemente b[j],...,b[i-1]  
        // nach rechts  
    b[j] = x;  
}
```

Neue Teilprobleme:

- Wie findet man die Einfügestelle?
- Wie verschiebt man nach rechts?

```
public static int locate (int[] b, int x, int i) {
    int j = 0;
    while (j < i && x > b[j]) ++j;
    return j;
}
```

```
public static void shift (int[] b, int j, int i) {
    for (int k = i-1; k >= j; --k)
        b[k+1] = b[k];
}
```

- Warum läuft die Iteration in `shift()` von `i-1` **abwärts** nach `j` ?
- Das zweite Argument des Operators `&&` wird nur ausgewertet, sofern das erste `true` ergibt (**Kurzschluss-Auswertung!**). Sonst würde hier auf eine **uninitialisierte** Variable zugegriffen **!!!**

- Das Feld `b` ist (ursprünglich) eine **lokale** Variable von `sort()`.
- Lokale Variablen sind nur im eigenen Funktionsrumpf sichtbar, nicht in den aufgerufenen Funktionen !
- Damit die aufgerufenen Hilfsfunktionen auf `b` zugreifen können, muss `b` explizit als Parameter übergeben werden !

Achtung:

Das Feld wird nicht kopiert. Das Argument ist der Wert der Variablen `b`, also nur eine **Referenz** !

- Deshalb benötigen weder `insert()`, noch `shift()` einen separaten Rückgabewert :-)
- Weil das Problem so **klein** ist, würde eine **erfahrene** Programmiererin hier keine Unterprogramme benutzen ...

```

public static int[] sort (int[] a) {
    int[] b = new int[a.length];
    for (int i = 0; i < a.length; ++i) {
        // begin of insert
        int j = 0;
        while (j < i && a[i] > b[j]) ++j;
        // end of locate
        for (int k = i-1; k >= j; --k)
            b[k+1] = b[k];
        // end of shift
        b[j] = a[i];
        // end of insert
    }
    return b;
} // end of sort

```


Diskussion:

- Die Anzahl der ausgeführten Operationen wächst quadratisch in der Größe des Felds a :-(
 $O(n^2)$
- Glücklicherweise gibt es Sortier-Verfahren, die eine bessere Laufzeit haben (↑ [Algorithmen und Datenstrukturen](#)).

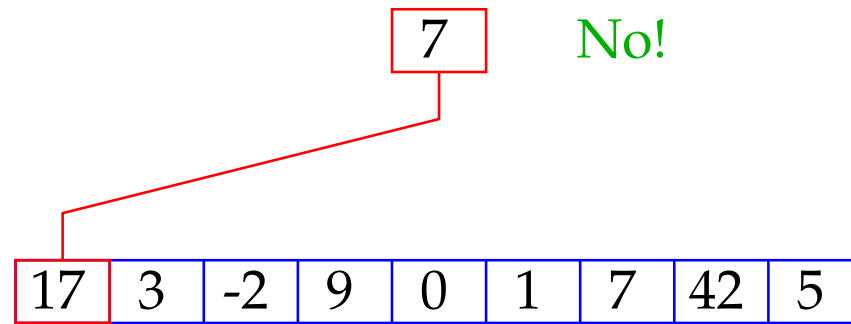
7 Eine zweite Anwendung: Suchen

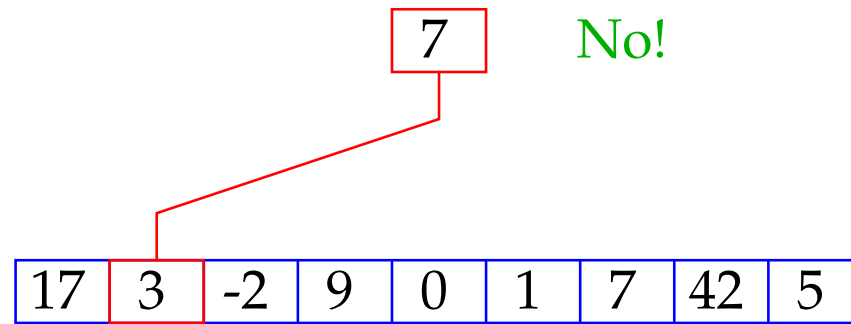
Nehmen wir an, wir wollen herausfinden, ob das Element 7 in unserem Feld a enthalten ist.

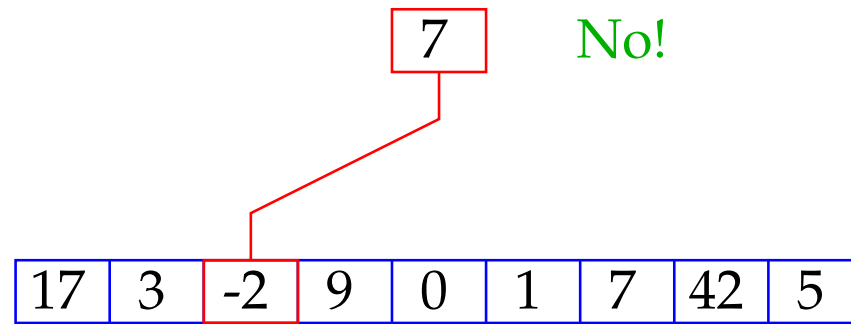
Naives Vorgehen:

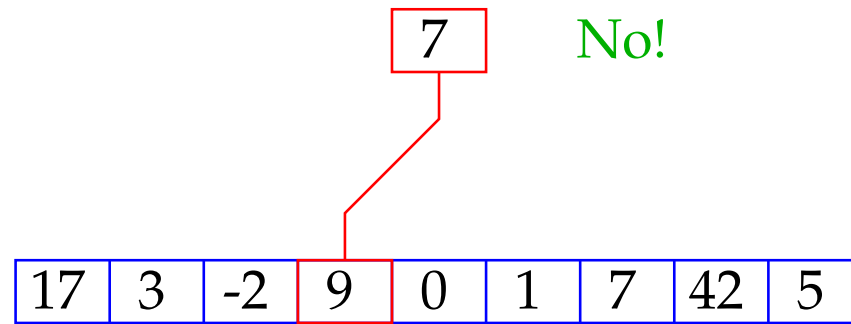
- Wir vergleichen 7 der Reihe nach mit den Elementen $a[0]$, $a[1]$, usw.
- Finden wir ein i mit $a[i] == 7$, geben wir i aus.
- Andernfalls geben wir -1 aus: "Sorry, gibt's leider nicht :-(")

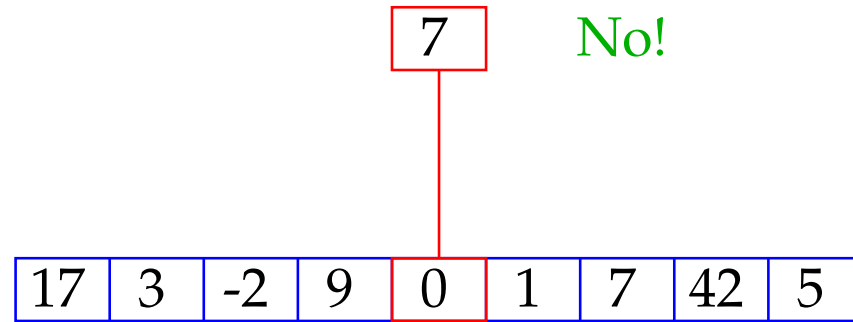
```
public static int find (int[] a, int x) {  
    int i = 0;  
    while (i < a.length && a[i] != x)  
        ++i;  
    if (i == a.length)  
        return -1;  
    else  
        return i;  
}
```

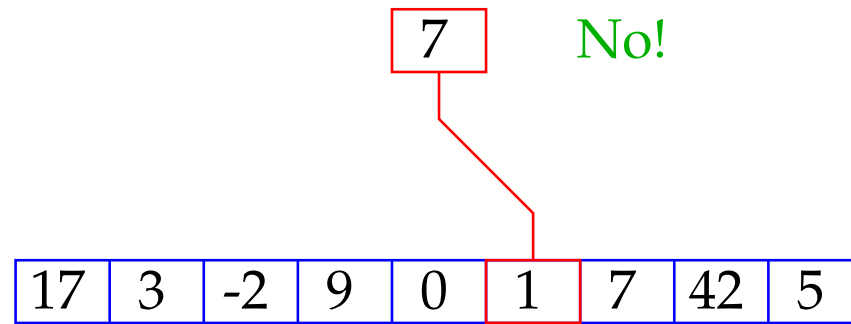


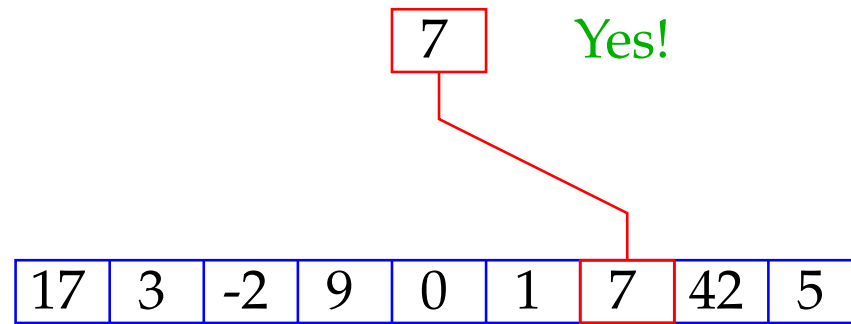












- Im Beispiel benötigen wir 7 Vergleiche.
- Im schlimmsten Fall benötigen wir bei einem Feld der Länge n sogar n Vergleiche :-((
- Kommt 7 tatsächlich im Feld vor, benötigen wir selbst im **Durchschnitt** $(n + 1)/2$ viele Vergleiche :-(((

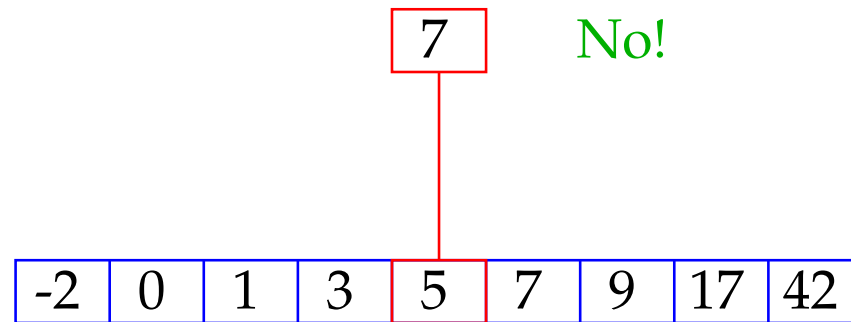
Geht das nicht besser ???

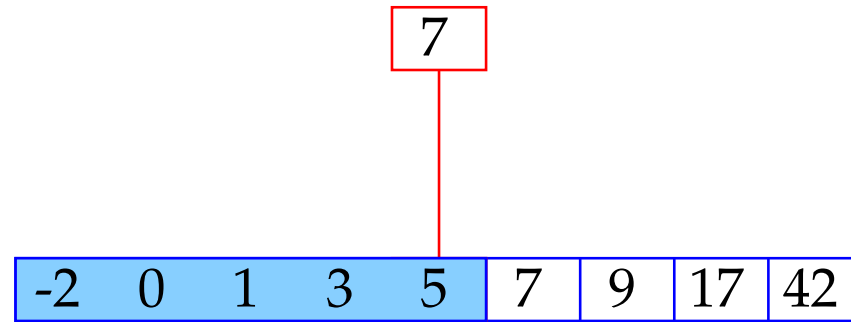
Idee:

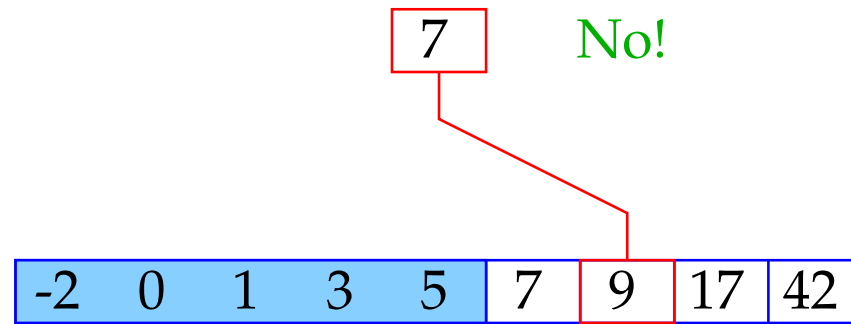
- Sortiere das Feld.
- Vergleiche 7 mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist 7 kleiner, brauchen wir nur noch links weitersuchen.
- Ist 7 größer, brauchen wir nur noch rechts weiter suchen.

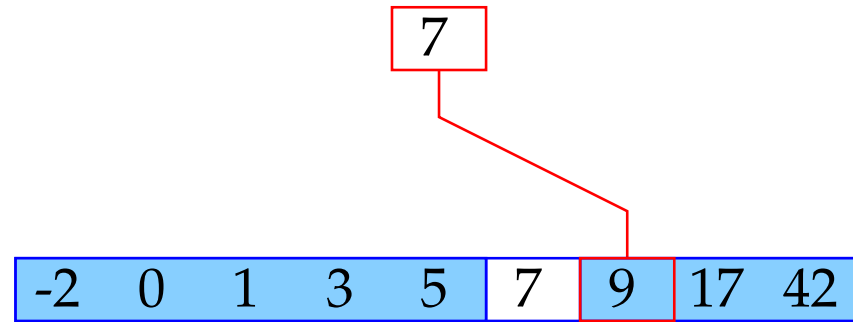


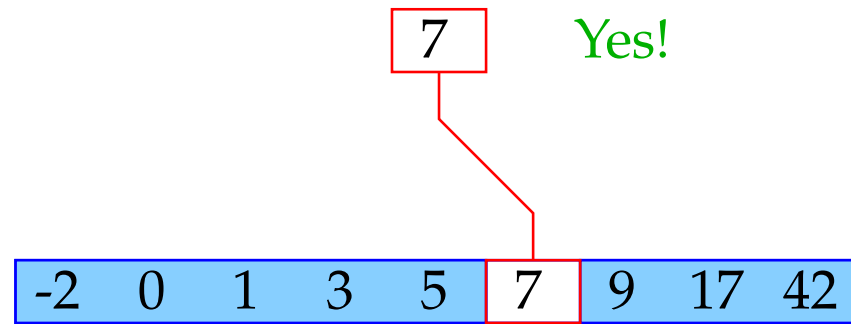
binäre Suche ...











- D.h. wir benötigen gerade mal **drei** Vergleiche.
- Hat das sortierte Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche.

Idee:

Wir führen eine Hilfsfunktion

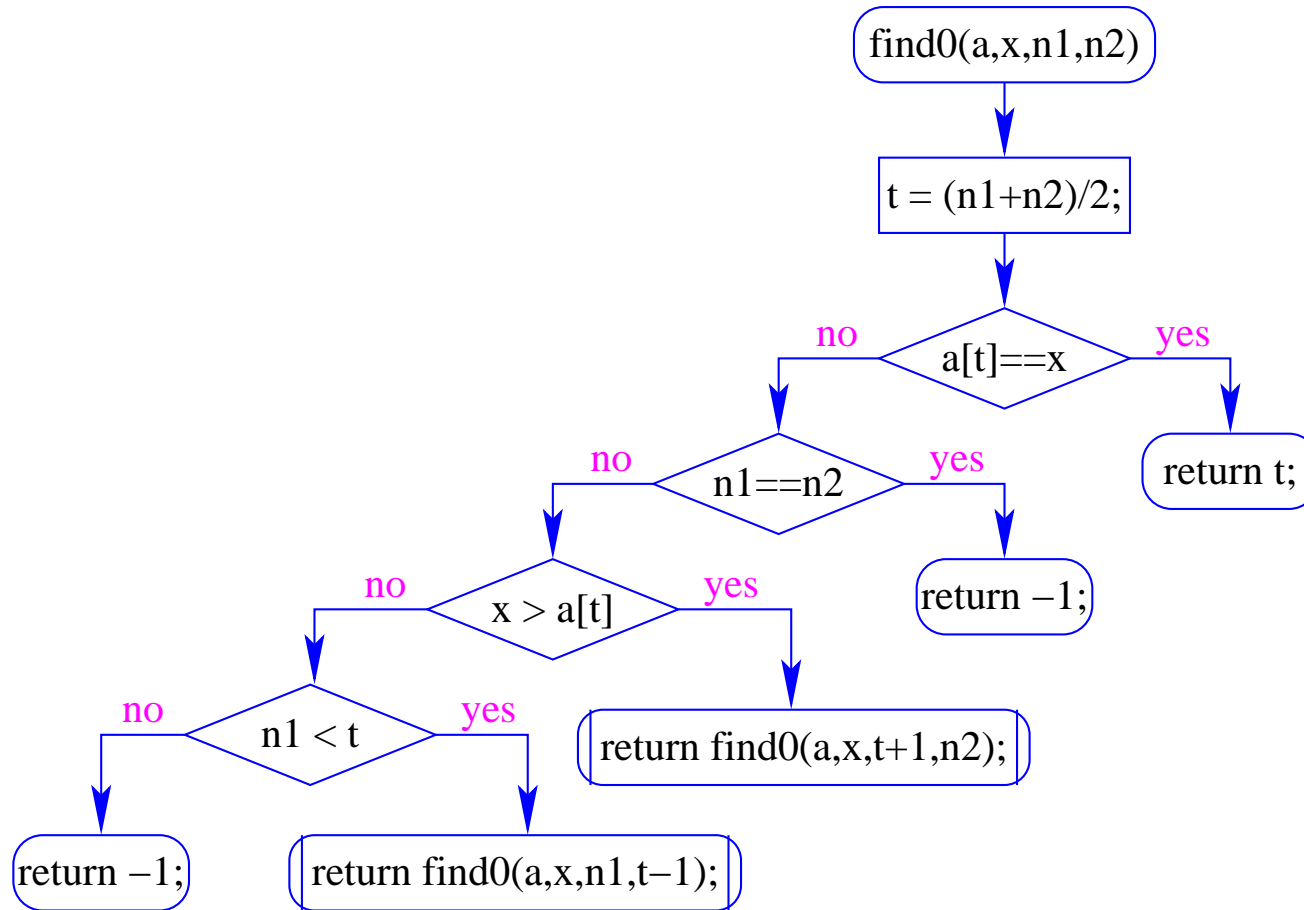
```
public static int find0 (int[] a, int x, int n1, int n2)
```

ein, die im Intervall $[n1, n2]$ sucht. Damit:

```
public static int find (int[] a, int x) {  
    return find0 (a, x, 0, a.length-1);  
}
```

```
public static int find0 (int[] a, int x, int n1, int n2) {  
    int t = (n1+n2)/2;  
    if (a[t] == x)  
        return t;  
    else if (n1 == n2)  
        return -1;  
    else if (x > a[t])  
        return find0 (a,x,t+1,n2);  
    else if (n1 < t)  
        return find0 (a,x,n1,t-1);  
    else return -1;  
}
```

Das Kontrollfluss-Diagramm für `find0()`:



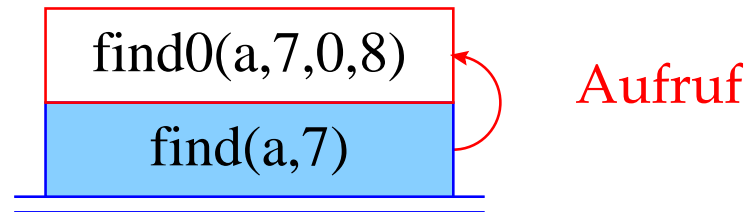
Achtung:

- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können :-)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen, heißen **rekursiv**.

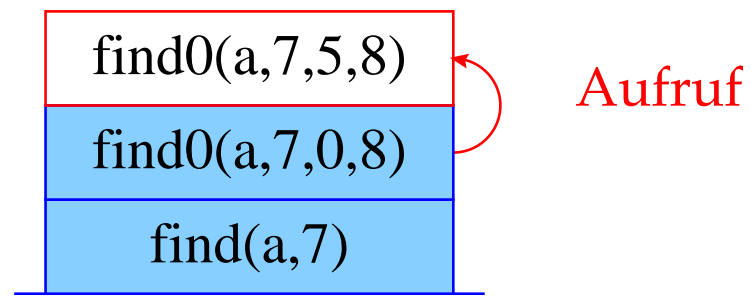
Ausführung:

find(a,7)

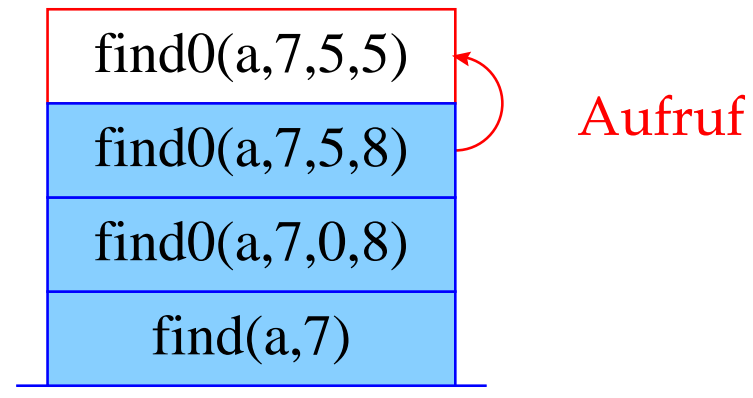
Ausführung:



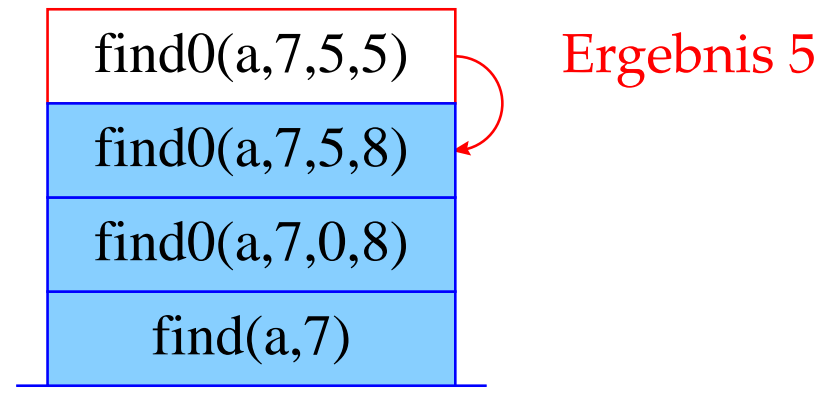
Ausführung:



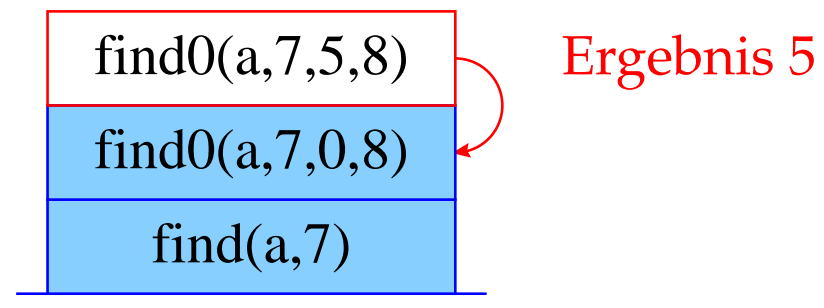
Ausführung:



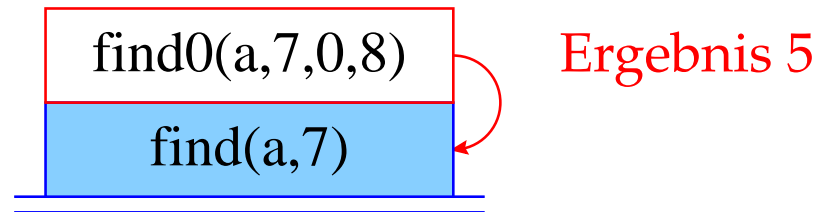
Ausführung:



Ausführung:



Ausführung:



Ausführung:

find(a,7)

Ergebnis 5

- Die Verwaltung der Funktionsaufrufe erfolgt nach dem **LIFO**-Prinzip (**L**ast-**I**n-**F**irst-**O**ut).
- Eine Datenstruktur, die nach diesem Stapel-Prinzip verwaltet wird, heißt auch **Keller** oder **Stack**.
- Aktiv ist jeweils nur der oberste/letzte Aufruf.
- **Achtung:** es kann zu einem Zeitpunkt mehrere weitere **inaktive** Aufrufe der selben Funktion geben !!!