

8 Anwendung: Suchen

Gegeben: Folge a ganzer Zahlen; Element x

Gesucht: Wo kommt x in a vor?

Naives Vorgehen:

- ▶ Vergleiche x der Reihe nach mit $a[0]$, $a[1]$, usw.
- ▶ Finden wir i mit $a[i] == x$, geben wir i aus.
- ▶ Andernfalls geben wir -1 aus: „Element nicht gefunden“!

Naives Suchen

```
1 public static int find(int[] a, int x) {  
2     int i = 0;  
3     while (i < a.length && a[i] != x)  
4         ++i;  
5     if (i == a.length)  
6         return -1;  
7     else  
8         return i;  
9 }
```

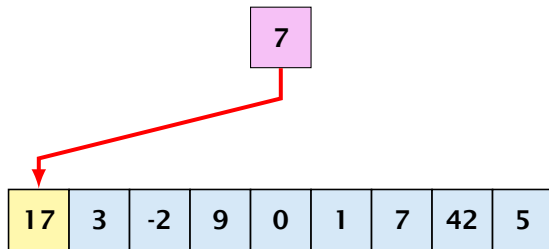
Naives Suchen

Beispiel

7

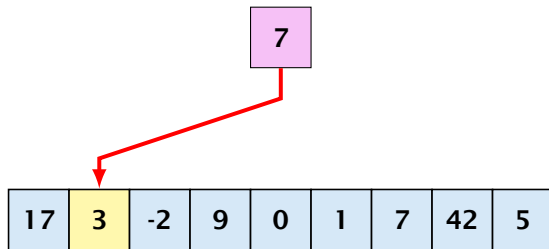
17	3	-2	9	0	1	7	42	5
----	---	----	---	---	---	---	----	---

Beispiel



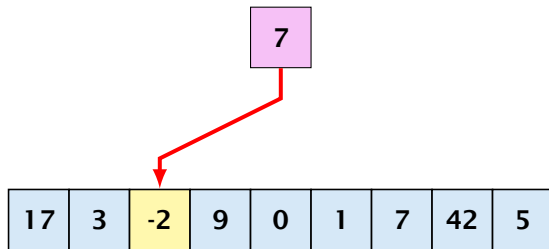
no

Beispiel



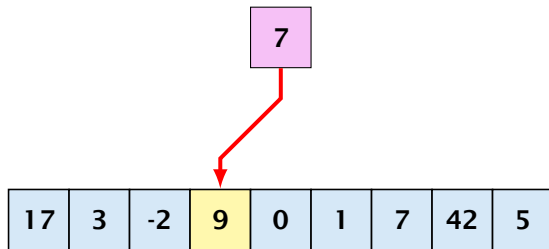
no

Beispiel



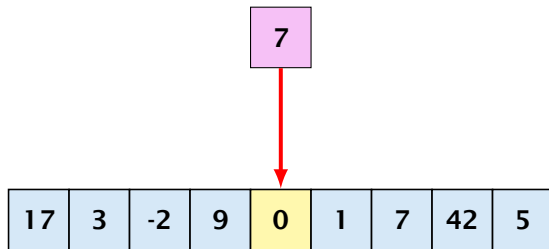
no

Beispiel



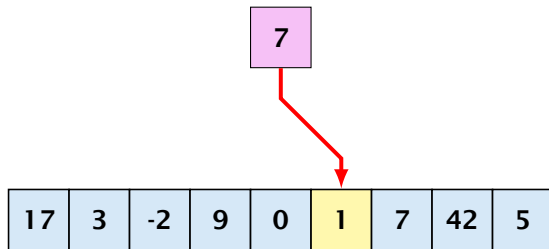
no

Beispiel



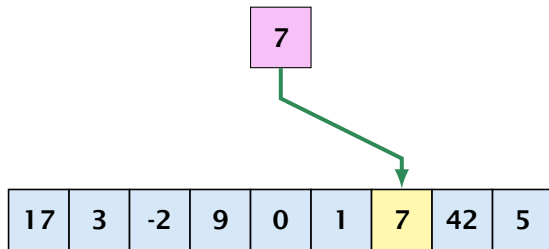
no

Beispiel



no

Beispiel



yes

Naives Suchen

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

...geht das nicht besser?

Binäre Suche

Idee:

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

⇒ binäre Suche

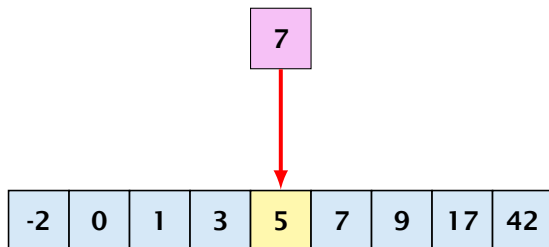
Beispiel

7

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

- ▶ wir benötigen nur **drei** Vergleiche
- ▶ hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Beispiel



no

- ▶ wir benötigen nur **drei** Vergleiche
- ▶ hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Beispiel

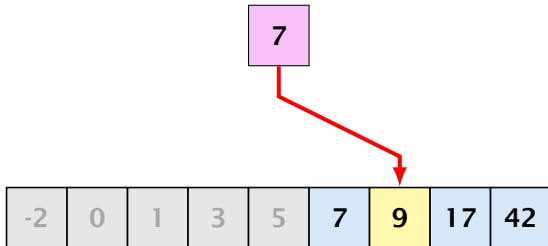
7

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

no

- ▶ wir benötigen nur **drei** Vergleiche
- ▶ hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Beispiel



no

- ▶ wir benötigen nur **drei** Vergleiche
- ▶ hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Beispiel

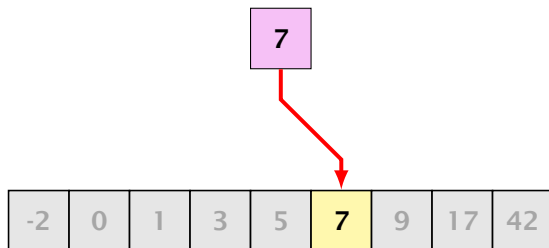
7

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

no

- ▶ wir benötigen nur **drei** Vergleiche
- ▶ hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Beispiel



yes

- ▶ wir benötigen nur **drei** Vergleiche
- ▶ hat das Feld $2^n - 1$ Elemente, benötigen wir maximal n Vergleiche

Implementierung

Idee:

Führe Hilfsfunktion

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

ein, die im Interval $[n1, n2]$ sucht.

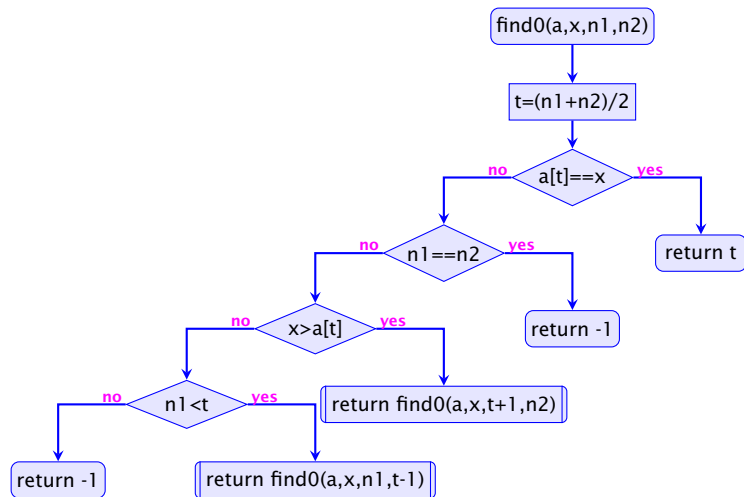
Damit:

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

Implementierung

```
1 public static int find0(int[] a, int x, int n1, int n2) {
2     int t = (n1 + n2) / 2;
3     if (a[t] == x)
4         return t;
5     else if (n1 == n2)
6         return -1;
7     else if (x > a[t])
8         return find0(a, x, t+1, n2);
9     else if (n1 < t)
10        return find0(a, x, n1, t-1);
11    else return -1;
12 }
```

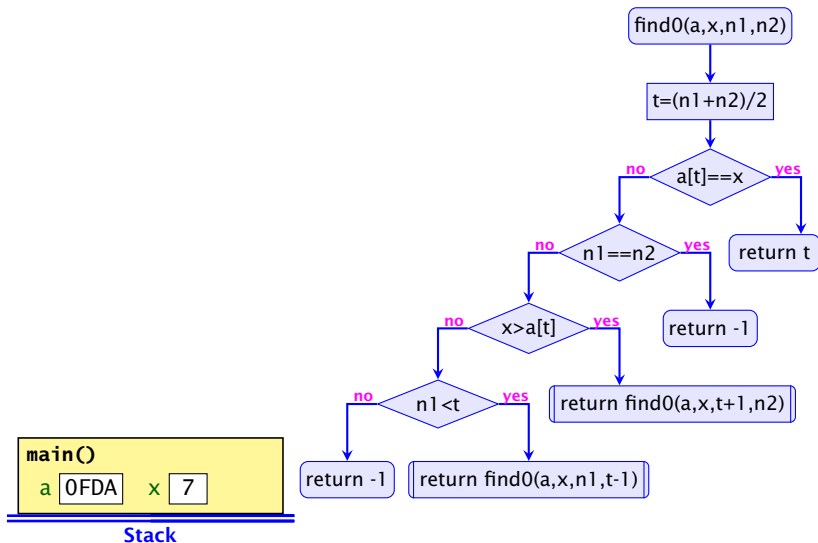
Kontrollflussdiagramm für find0



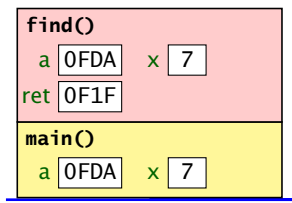
Erläuterungen:

- ▶ 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**.

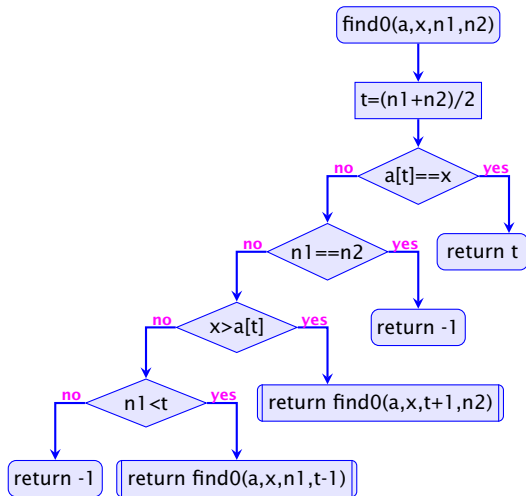
Ausführung



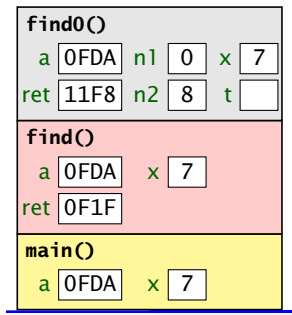
Ausführung



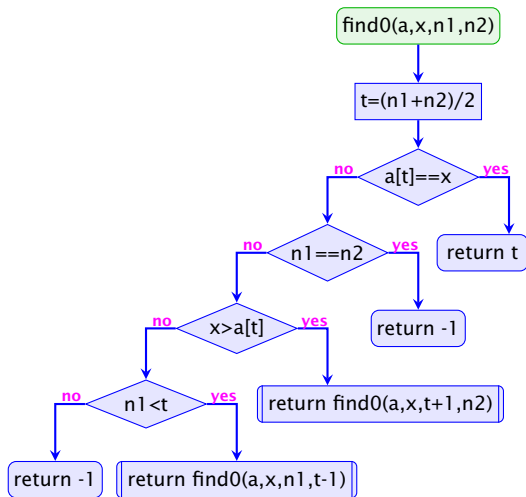
Stack



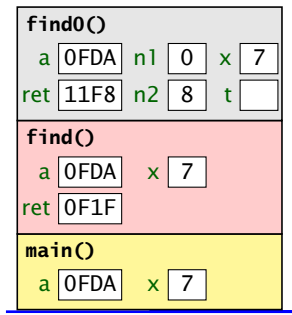
Ausführung



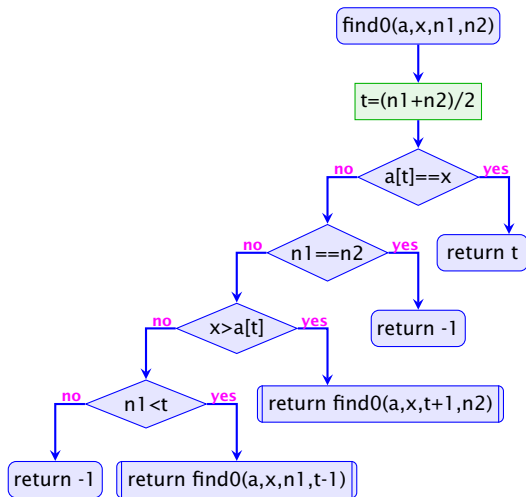
Stack



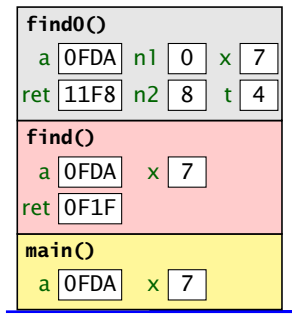
Ausführung



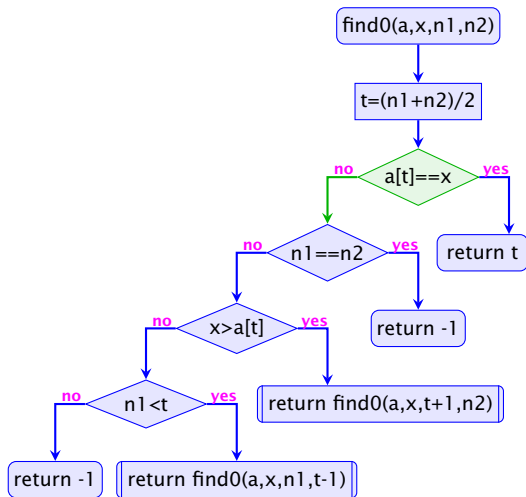
Stack



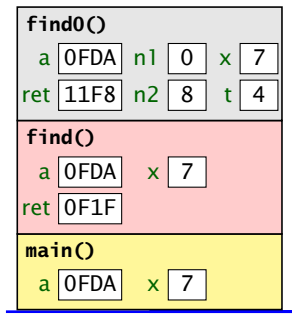
Ausführung



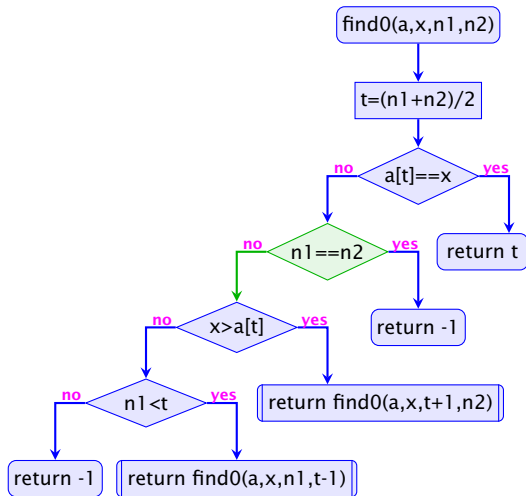
Stack



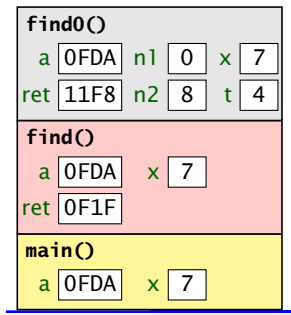
Ausführung



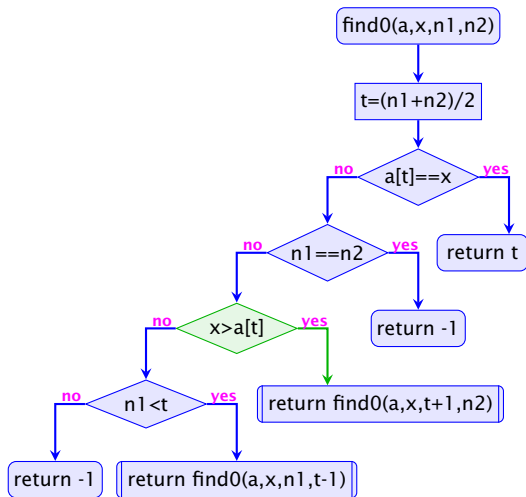
Stack



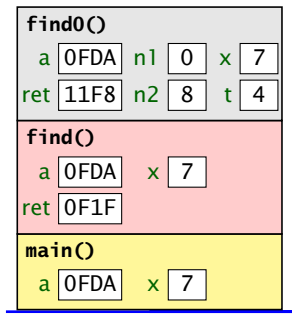
Ausführung



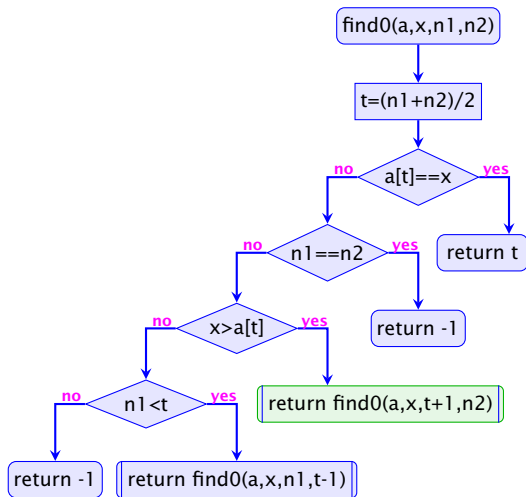
Stack



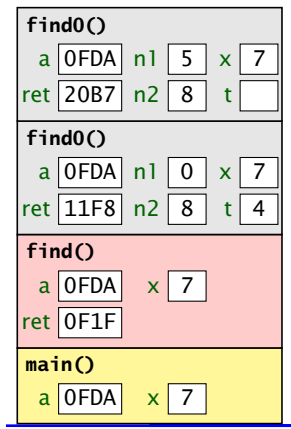
Ausführung



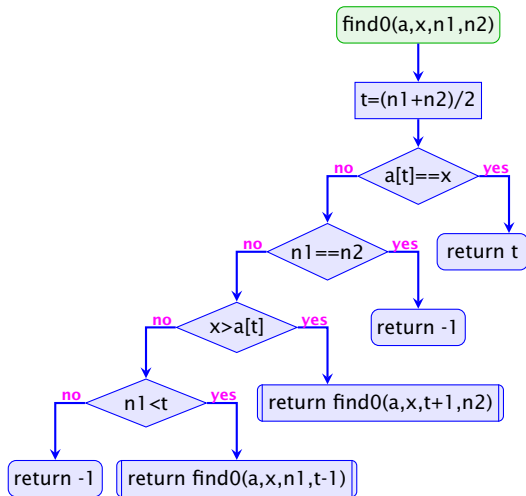
Stack



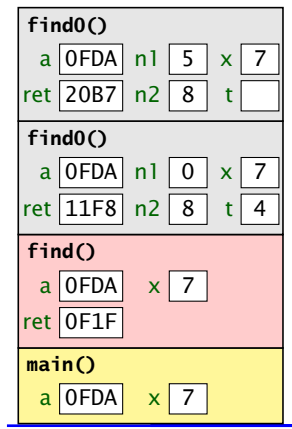
Ausführung



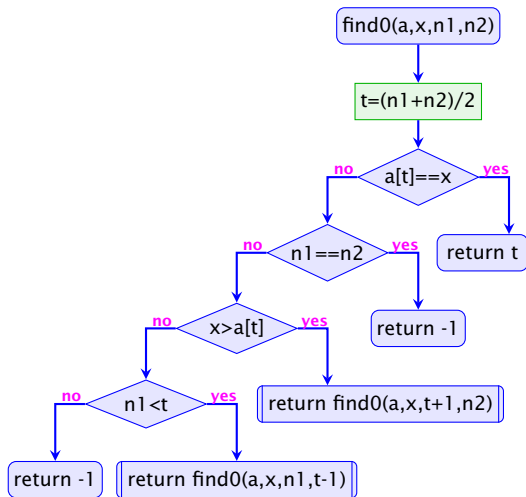
Stack



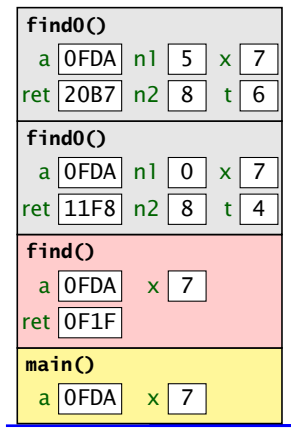
Ausführung



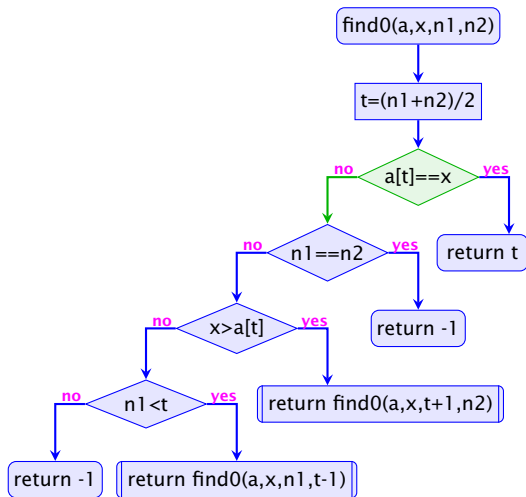
Stack



Ausführung



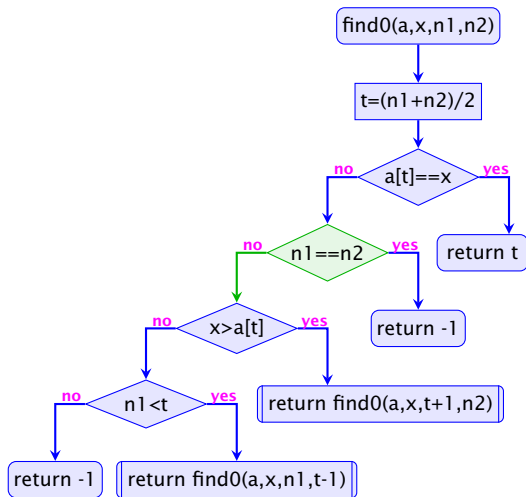
Stack



Ausführung

find0()					
a	0FDA	n1	5	x	7
ret	20B7	n2	8	t	6
find0()					
a	0FDA	n1	0	x	7
ret	11F8	n2	8	t	4
find()					
a	0FDA	x	7		
ret	0F1F				
main()					
a	0FDA	x	7		

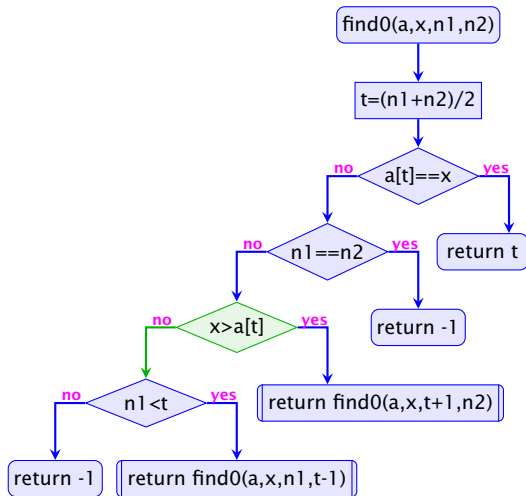
Stack



Ausführung

find0()					
a	0FDA	n1	5	x	7
ret	20B7	n2	8	t	6
find0()					
a	0FDA	n1	0	x	7
ret	11F8	n2	8	t	4
find()					
a	0FDA	x	7		
ret	0F1F				
main()					
a	0FDA	x	7		

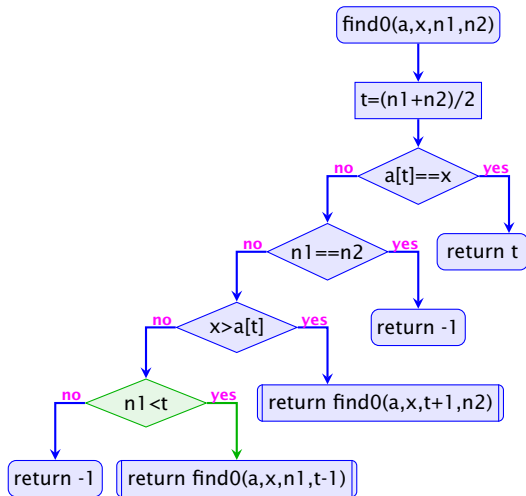
Stack



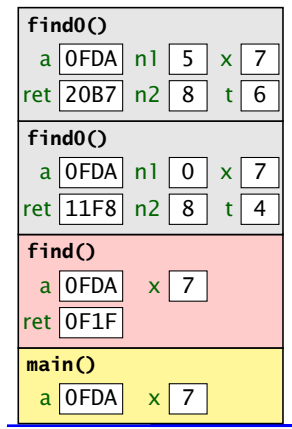
Ausführung

find0()					
a	0FDA	n1	5	x	7
ret	20B7	n2	8	t	6
find0()					
a	0FDA	n1	0	x	7
ret	11F8	n2	8	t	4
find()					
a	0FDA	x	7		
ret	0F1F				
main()					
a	0FDA	x	7		

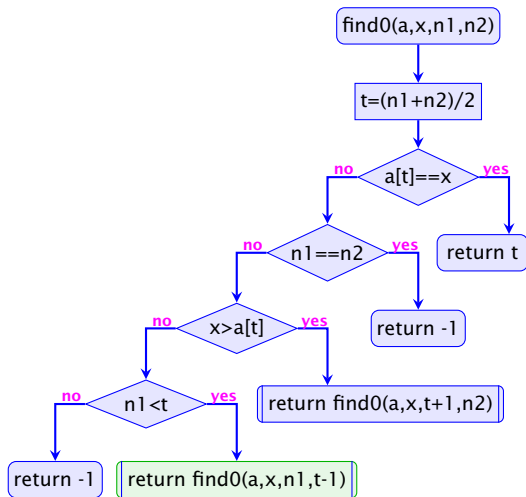
Stack



Ausführung



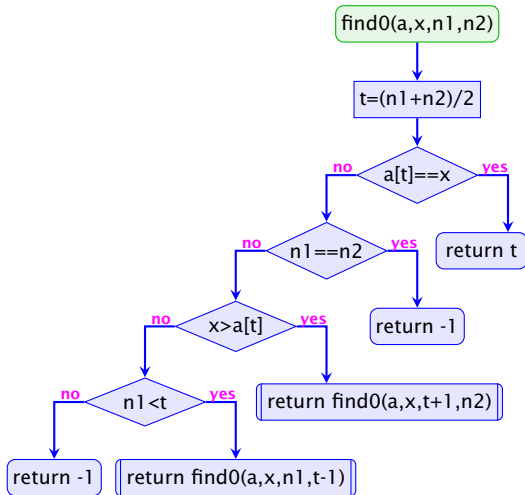
Stack



Ausführung

find0()					
a	0FDA	n1	5	x	7
ret	20C9	n2	5	t	
find0()					
a	0FDA	n1	5	x	7
ret	20B7	n2	8	t	6
find0()					
a	0FDA	n1	0	x	7
ret	11F8	n2	8	t	4
find()					
a	0FDA	x	7		
ret	0F1F				
main()					
a	0FDA	x	7		

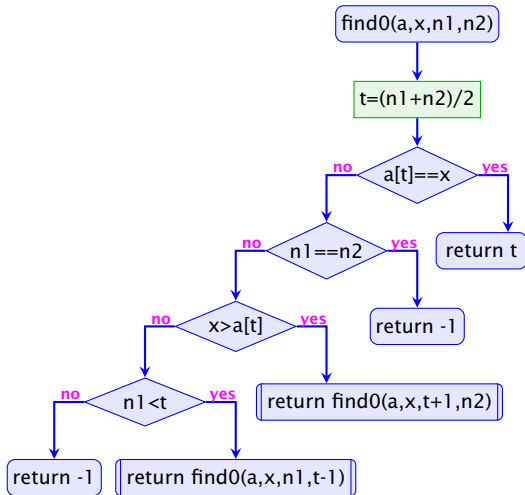
Stack



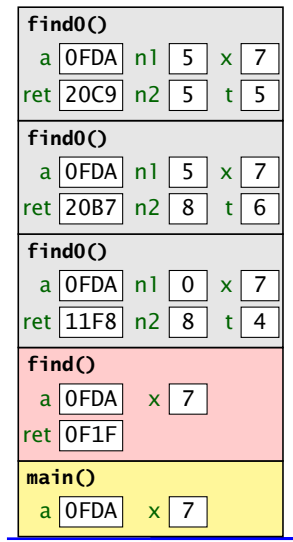
Ausführung

find0()					
a	0FDA	n1	5	x	7
ret	20C9	n2	5	t	
find0()					
a	0FDA	n1	5	x	7
ret	20B7	n2	8	t	6
find0()					
a	0FDA	n1	0	x	7
ret	11F8	n2	8	t	4
find()					
a	0FDA	x	7		
ret	0F1F				
main()					
a	0FDA	x	7		

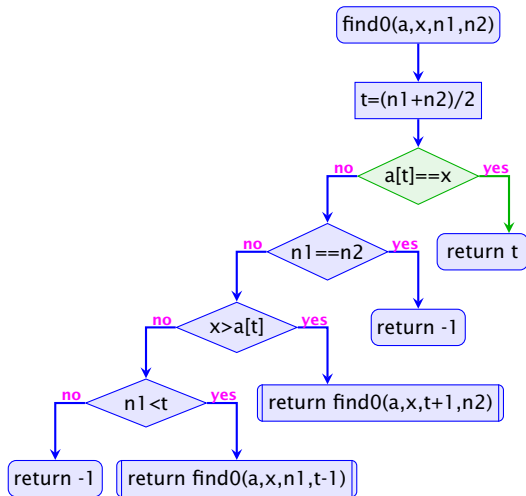
Stack



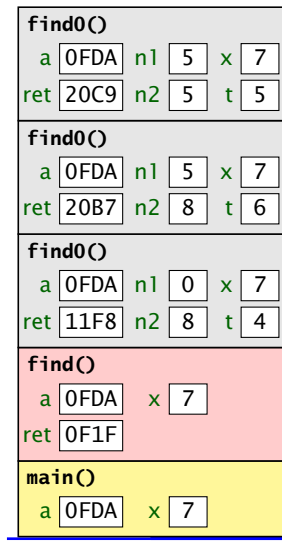
Ausführung



Stack

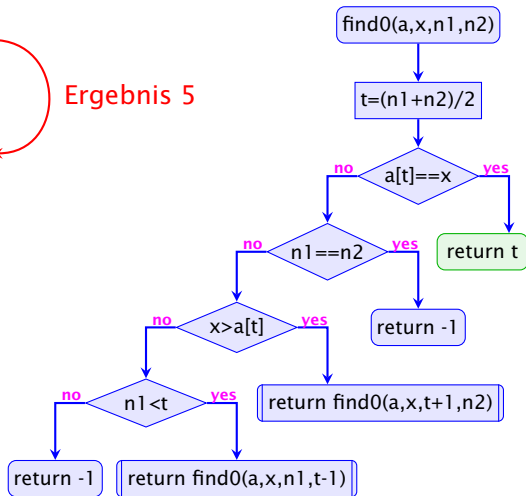


Ausführung

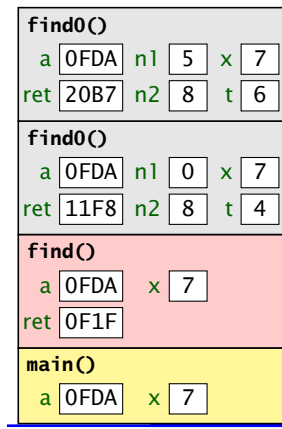


Stack

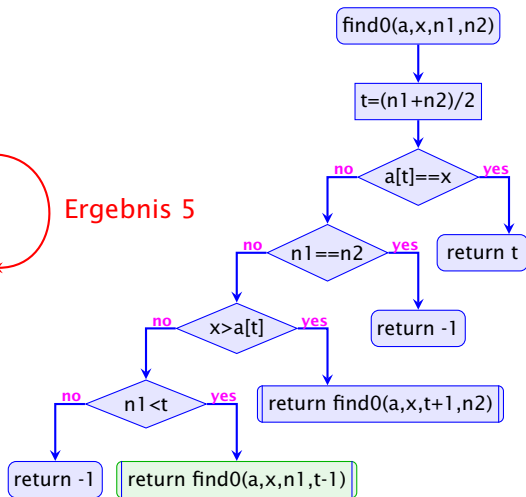
Ergebnis 5



Ausführung

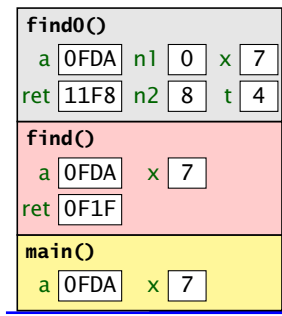


Stack

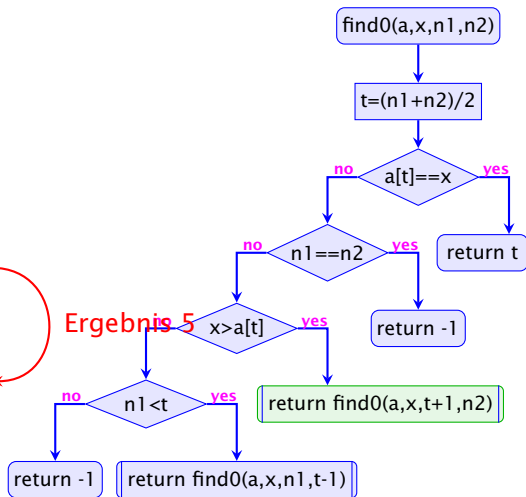


Ergebnis 5

Ausführung

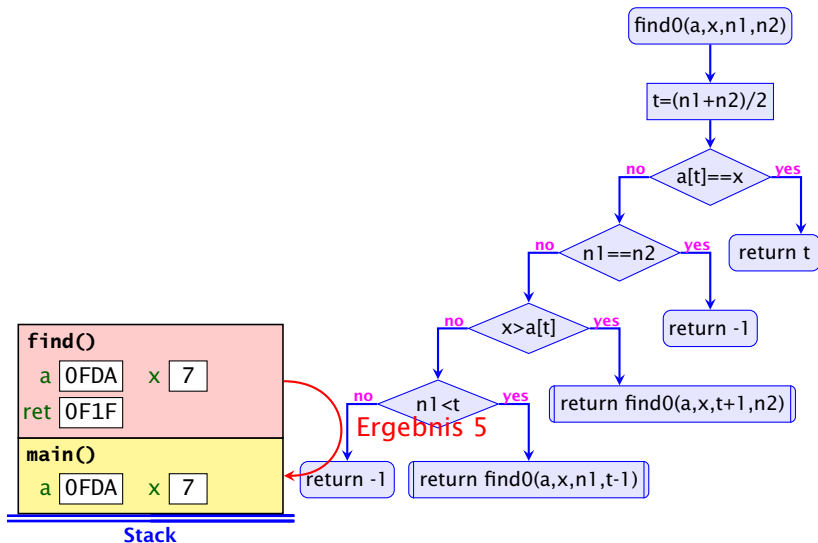


Stack



Ergebnis 5

Ausführung



Terminierung

Um zu **beweisen**, dass `find0()` terminiert, beobachten wir:

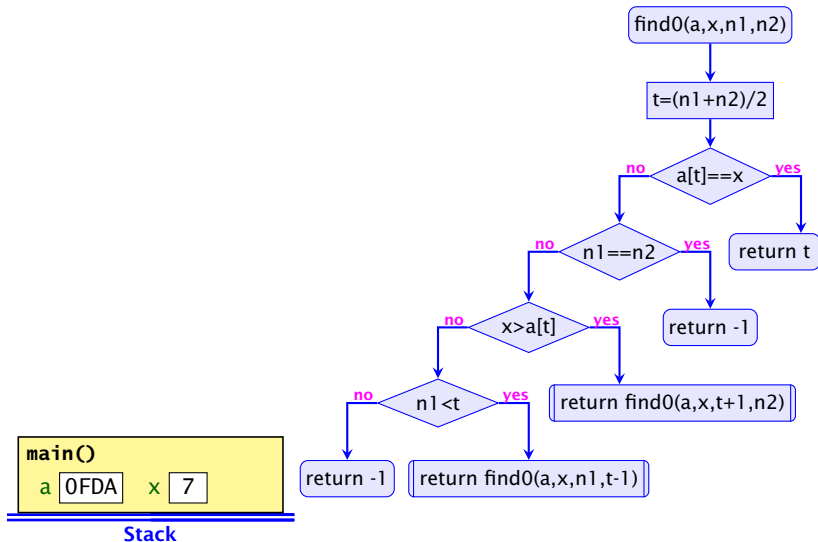
1. Wird `find0()` für ein einelementiges Intervall $[n, n]$ aufgerufen, dann terminiert der Funktionsaufruf direkt.
2. wird `find0()` für ein Intervall $[n_1, n_2]$ aufgerufen mit mehr als einem Element, dann terminiert der Aufruf entweder direkt (weil `x` gefunden wurde), oder `find0()` wird mit einem Intervall aufgerufen, das **echt** in $[n_1, n_2]$ enthalten ist, genauer: sogar maximal die Hälfte der Elemente von $[n_1, n_2]$ enthält.

Ähnliche Beweistechnik wird auch für andere rekursive Funktionen verwendet.

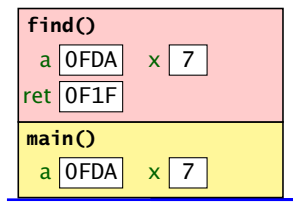
Beobachtung

- ▶ Das Ergebnis eines Aufrufs von `find0()` liefert **direkt** das Ergebnis auch für die aufrufende Funktion!
- ▶ Solche Rekursion heißt **End-** oder **Tail-Rekursion**.
- ▶ End-Rekursion kann auch ohne Aufrufkeller implementiert werden. . .
- ▶ **Idee:** lege den neuen Aufruf von `find0()` nicht oben auf den Stapel drauf, sondern **ersetze** den bereits dort liegenden Aufruf!

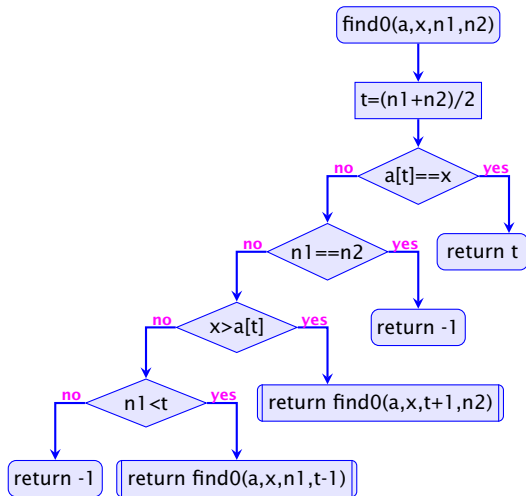
Verbesserte Ausführung



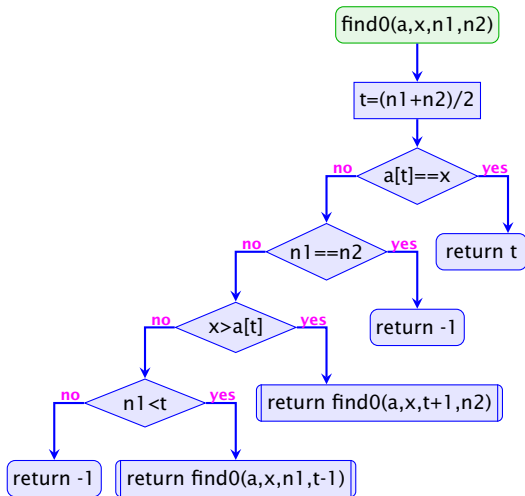
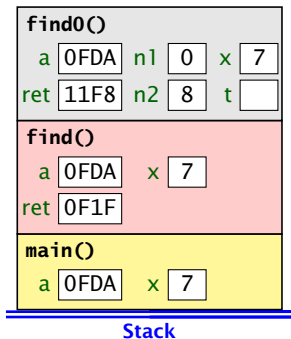
Verbesserte Ausführung



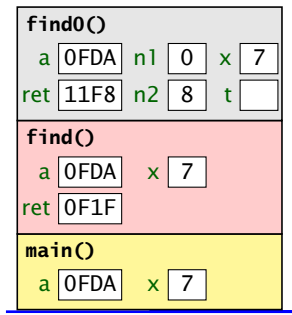
Stack



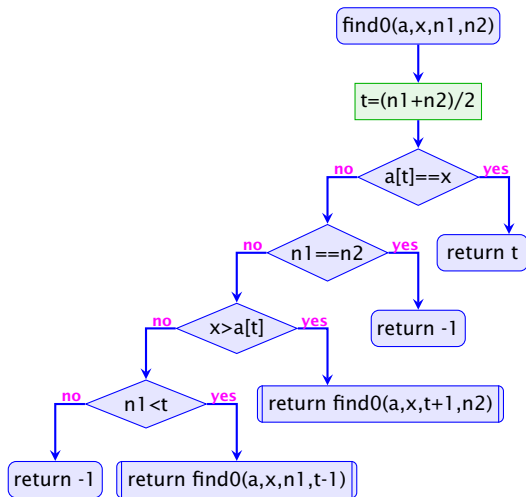
Verbesserte Ausführung



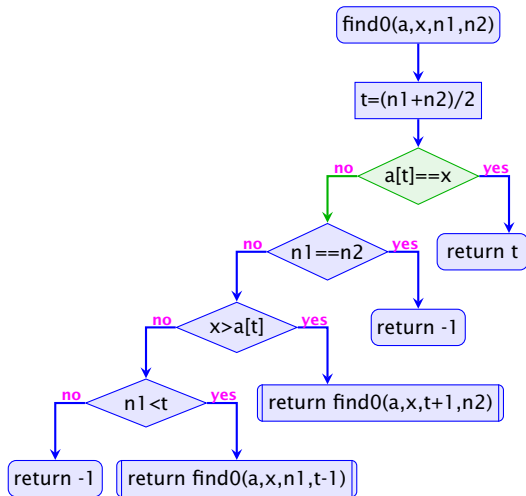
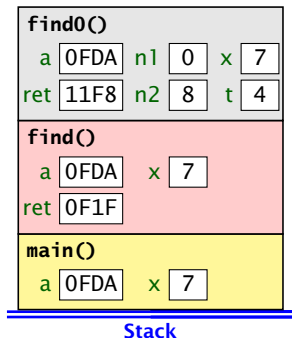
Verbesserte Ausführung



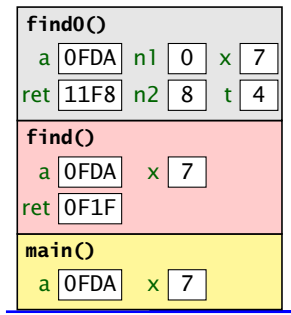
Stack



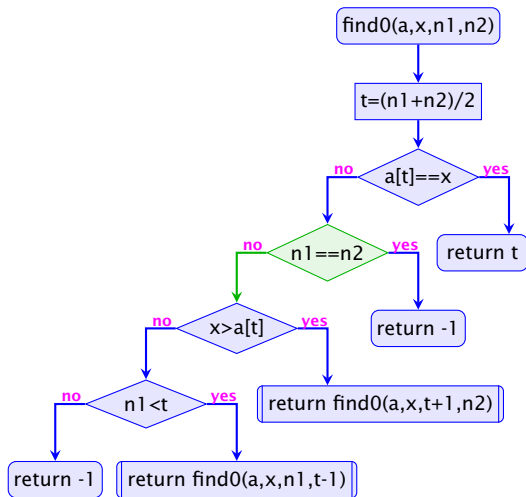
Verbesserte Ausführung



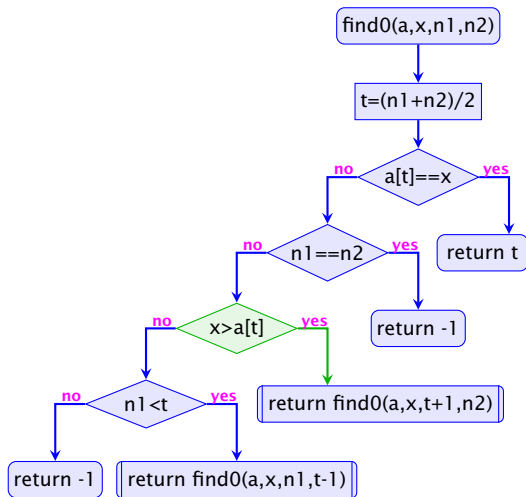
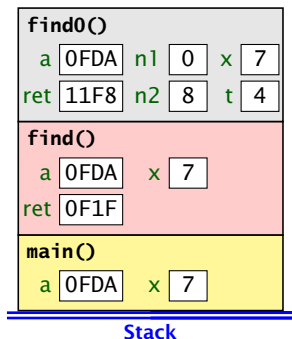
Verbesserte Ausführung



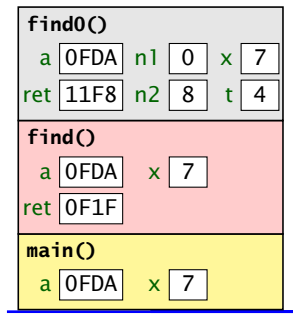
Stack



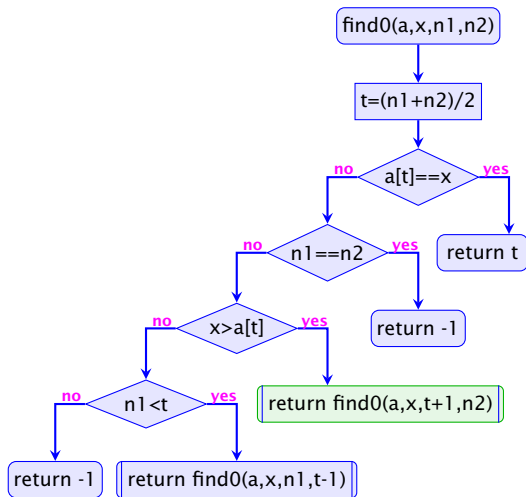
Verbesserte Ausführung



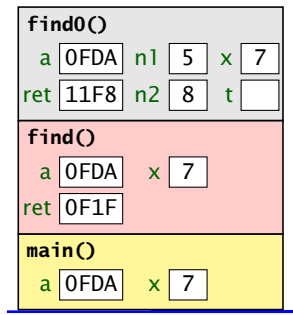
Verbesserte Ausführung



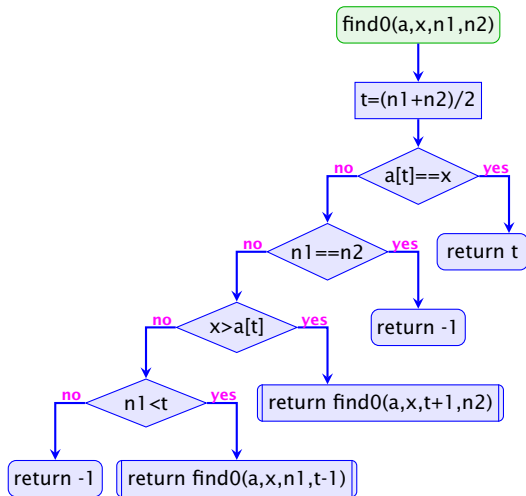
Stack



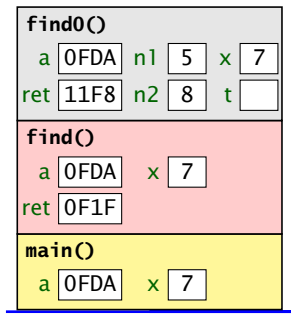
Verbesserte Ausführung



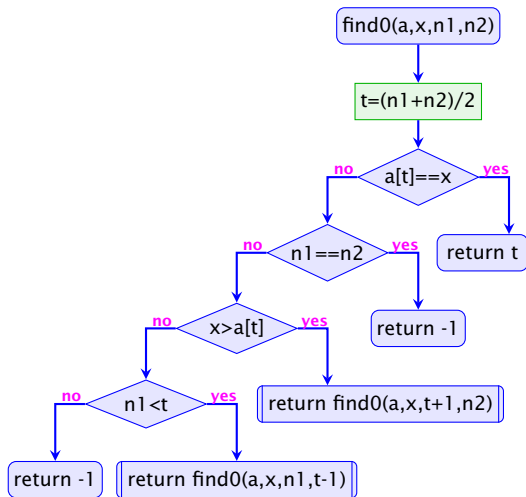
Stack



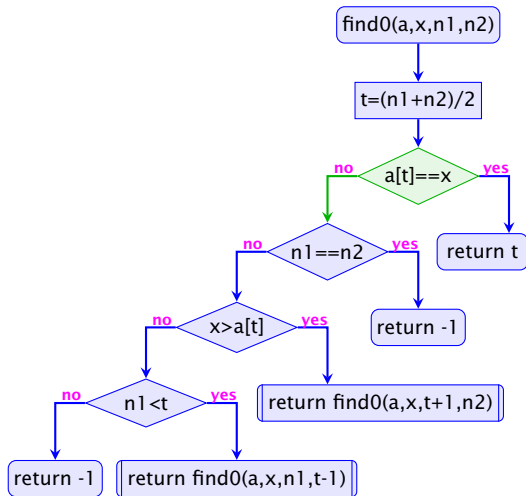
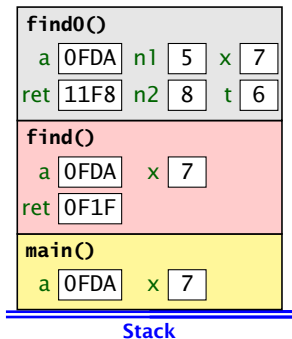
Verbesserte Ausführung



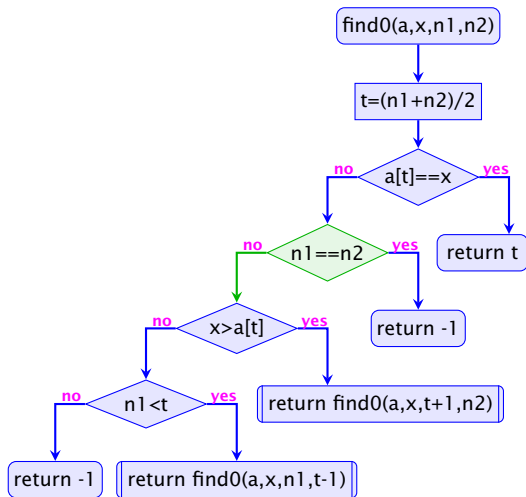
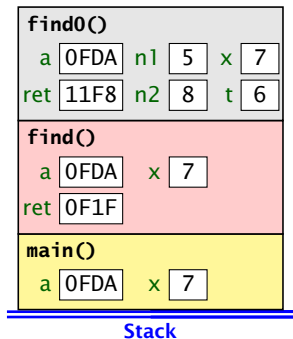
Stack



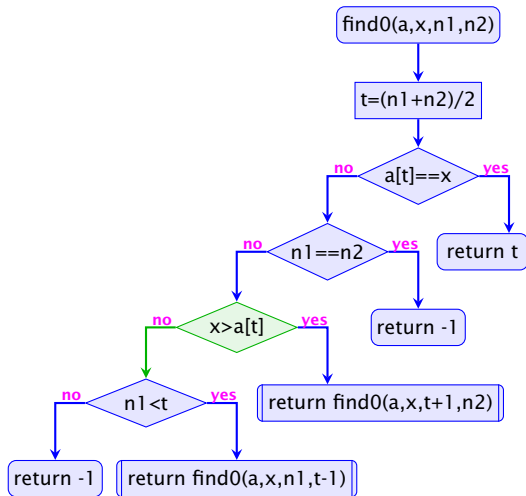
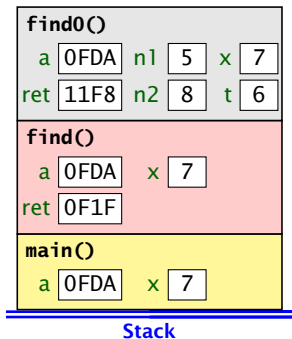
Verbesserte Ausführung



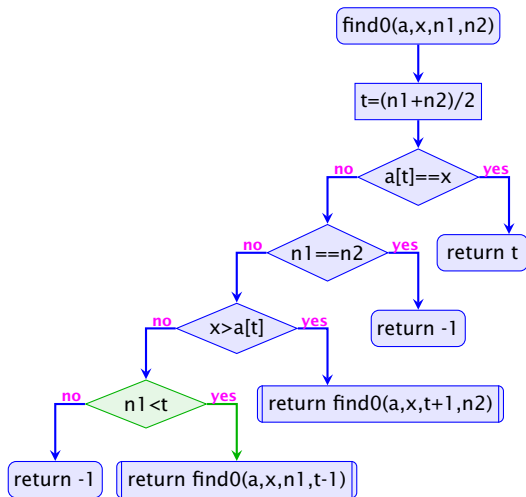
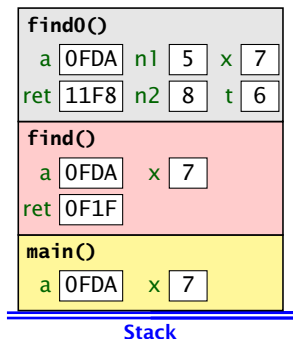
Verbesserte Ausführung



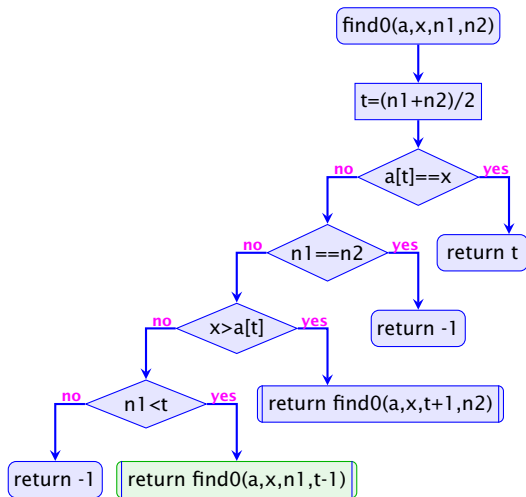
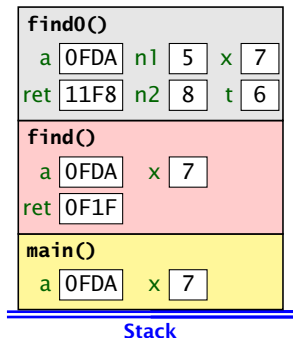
Verbesserte Ausführung



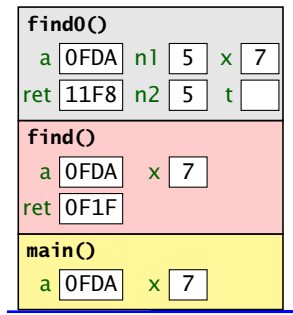
Verbesserte Ausführung



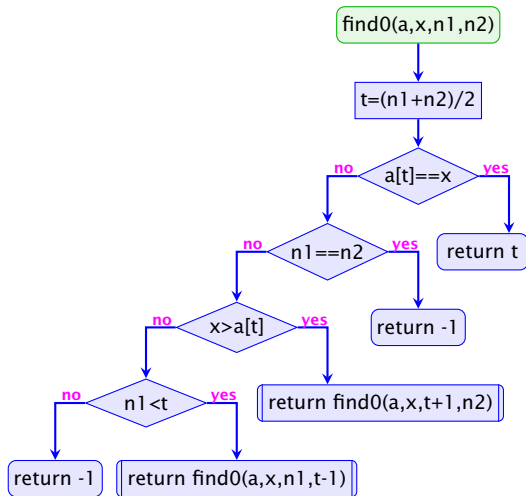
Verbesserte Ausführung



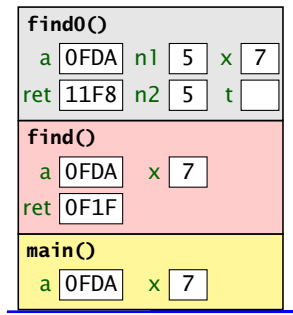
Verbesserte Ausführung



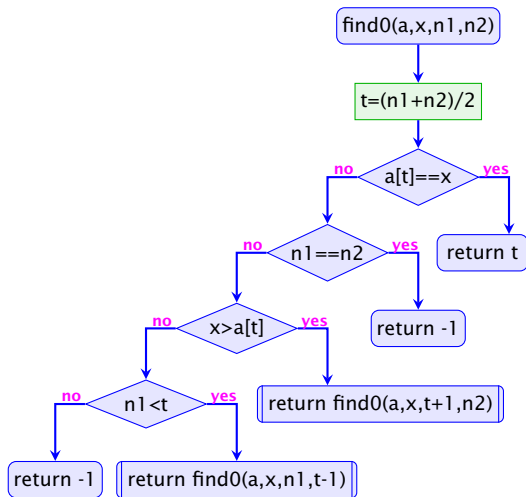
Stack



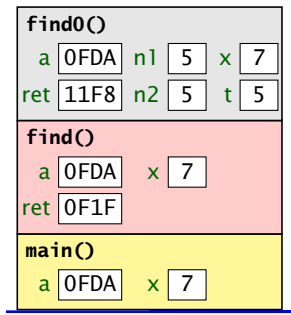
Verbesserte Ausführung



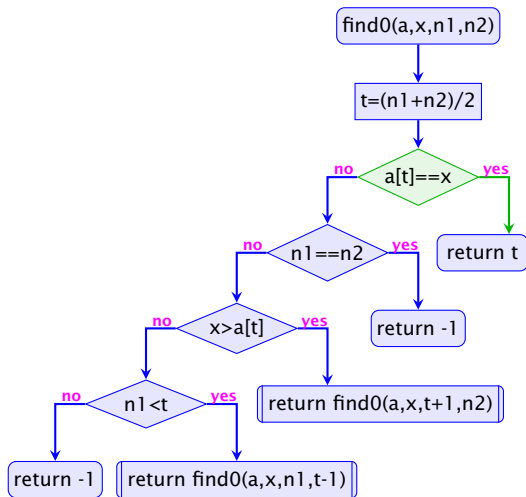
Stack



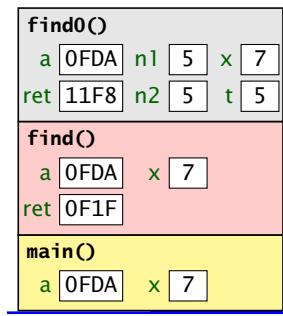
Verbesserte Ausführung



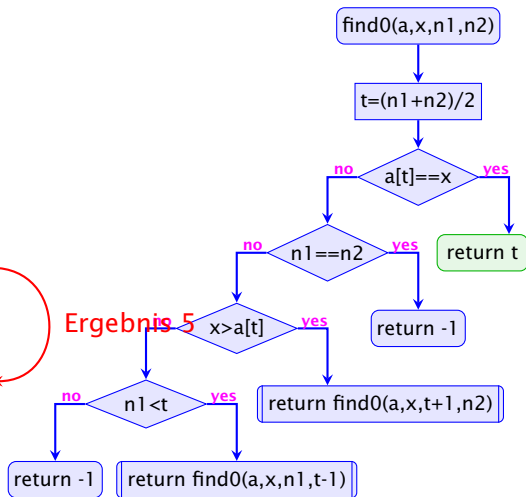
Stack



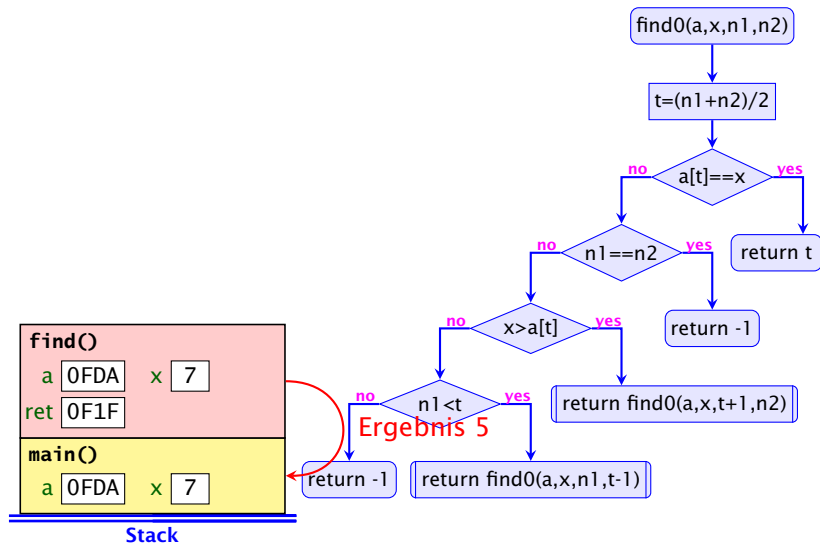
Verbesserte Ausführung



Stack



Verbesserte Ausführung

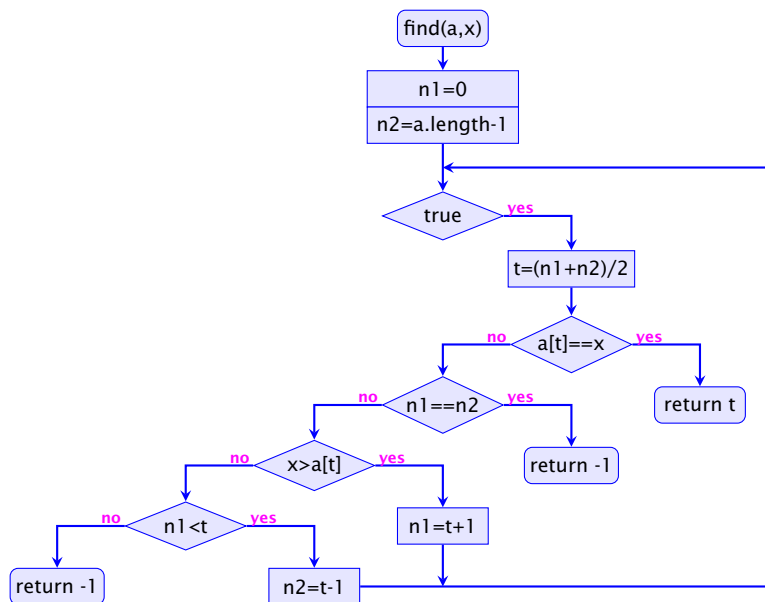


Endrekursion

Endrekursion kann durch **Iteration** ersetzt werden...

```
1 public static int find(int[] a, int x) {
2     int n1 = 0;
3     int n2 = a.length-1;
4     while (true) {
5         int t = (n2 + n1) / 2;
6         if (x == a[t]) return t;
7         else if (n1 == n2) return -1;
8         else if (x > a[t]) n1 = t+1;
9         else if (n1 < t) n2 = t-1;
10        else return -1;
11    } // end of while
12 } // end of find
```

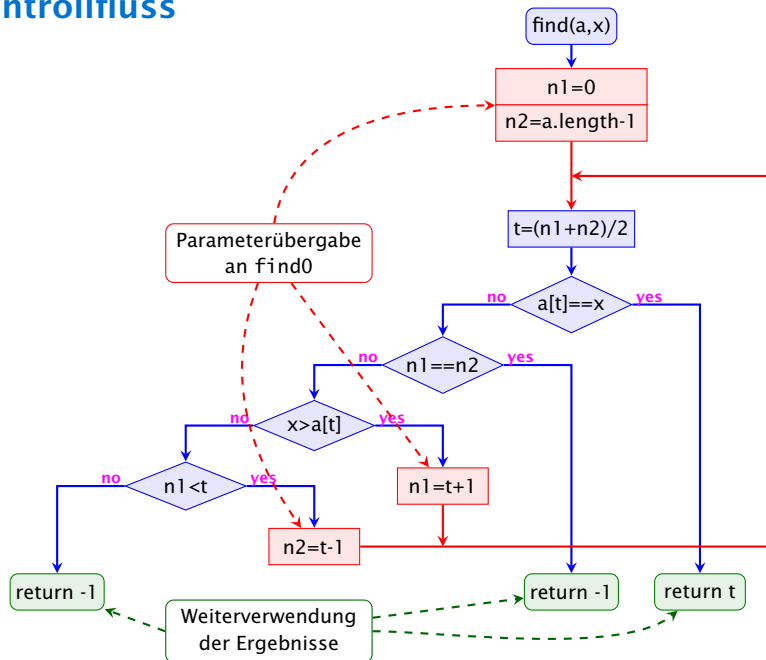
Kontrollfluss



Verlassen von Schleifen

- ▶ Die Schleife wird hier alleine durch die **return**-Anweisungen verlassen.
- ▶ Offenbar machen Schleifen mit **mehreren** Ausgängen Sinn.
- ▶ Um eine Schleife zu verlassen, ohne gleich ans Ende der Funktion zu springen, kann man das **break**-Statement benutzen.
- ▶ Der Aufruf der endrekursiven Funktion wird ersetzt durch:
 1. Code zur Parameter-Übergabe;
 2. einen **Sprung** an den Anfang des Rumpfs.

Kontrollfluss



Bemerkung

- ▶ Jede Rekursion läßt sich beseitigen, indem man den Aufruf-Keller **explizit** verwaltet.
- ▶ Nur im Falle von Endrekursion kann man auf den Keller verzichten.
- ▶ Rekursion ist trotzdem nützlich, weil rekursive Programme oft **leichter zu verstehen** sind als äquivalente Programme ohne Rekursion. . .