Script generated by TTT

Title: Seidl: Informatik_1 (07.11.2012)

Date: Wed Nov 07 15:15:30 CET 2012

Duration: 90:13 min

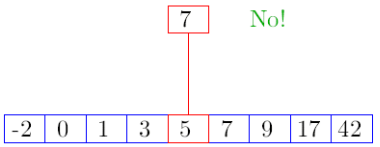
Pages: 54

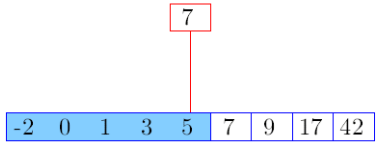
Mo, 19:00
Fragestunde
Anmeldung
beim Tutor

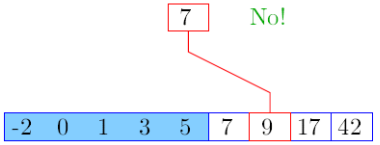
Idee:

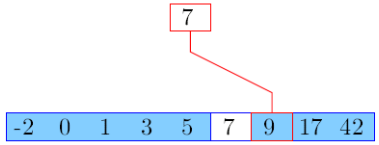
- Sortiere das Feld.
- Vergleiche 7 mit dem Wert, der in der Mitte steht.
- Liegt Gleichheit vor, sind wir fertig.
- Ist 7 kleiner, brauchen wir nur noch links weitersuchen.
- Ist 7 größer, brauchen wir nur noch rechts weiter suchen.

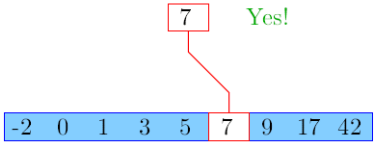
$\Longrightarrow$ binäre Suche ...




7
-2 0 1 3 5 7 9 17 42


7
No!
-2 0 1 3 5 7 9 17 42


7
-2 0 1 3 5 7 9 17 42


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

- 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);
   }
```

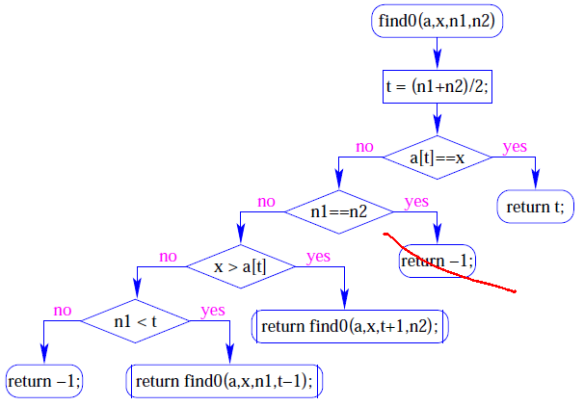


```
public static int find0 (int[] a, int x, int n1, int n2) {
   int t = (n1+n2)/2;
   if (a[t] == x)
      return t;
   else if (n1 == n2)
      return -1;
   else if (x > a[t])
      return find0 (a,x,t+1,n2);
   else if (n1 < t)
      return find0 (a,x,n1,t-1);
   else return -1;
   }
```



```
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.

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.

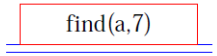
170

Achtung:

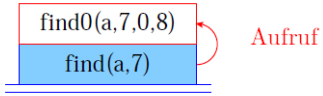
- zwei der `return`-Statements enthalten einen Funktionsaufruf – deshalb die Markierungen an den entsprechenden Knoten.
- (Wir hätten stattdessen auch zwei Knoten und eine Hilfsvariable `result` einführen können)
- `find0()` ruft sich selbst auf.
- Funktionen, die sich selbst (evt. mittelbar) aufrufen,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

170

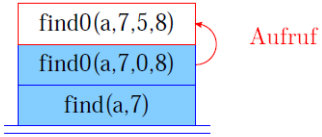
Ausführung:



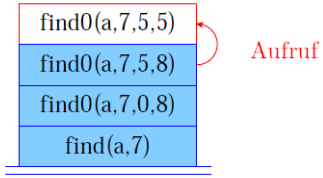
Ausführung:



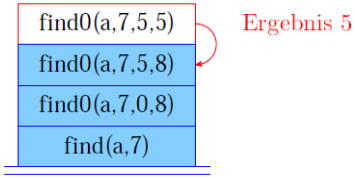
Ausführung:



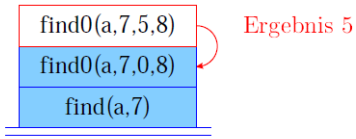
Ausführung:



Ausführung:

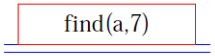


Ausführung:



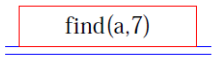
Ergebnis 5

Ausführung:



Ergebnis 5

Ausführung:



Ergebnis 5

- Die Verwaltung der Funktionsaufrufe erfolgt nach dem LIFO-Prinzip (Last-In-First-Out).
- 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 !!!

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

1. Wird `find0()` für ein ein-elementiges Intervall `[n,n]` aufgerufen, dann terminiert der Funktionsaufruf direkt.
2. wird `find0()` für ein Intervall `[n1,n2]` 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 `[n1,n2]` enthalten ist, genauer: sogar maximal die Hälfte der Elemente von `[n1,n2]` enthält.

$\Longrightarrow$ ähnliche Technik wird auch für andere rekursive Funktionen angewandt.

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 !

- Die Verwaltung der Funktionsaufrufe erfolgt nach dem LIFO-Prinzip (Last-In-First-Out).
- 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 !!!

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.

Verbesserte Ausführung:

find(a,7)

Verbesserte Ausführung:

find0(a,7,0,8)

Verbesserte Ausführung:

find0(a,7,5,8)

Verbesserte Ausführung:

find0(a,7,5,5) Ergebnis: 5

Verbesserte Ausführung:

find0(a,7,5,5) Ergebnis: 5

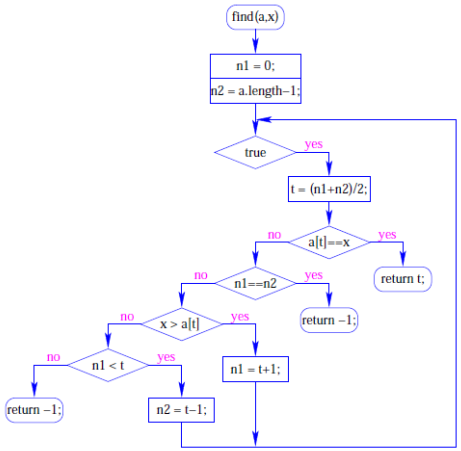
$\Longrightarrow$ end-Rekursion kann durch Iteration (d.h. eine normale Schleife) ersetzt werden ...

```
public static int find (int[] a, int x) {
   int n1 = 0;
   int n2 = a.length-1;
   while (true) {
      int t = (n2+n1)/2;
      if (x == a[t]) return t;
      else if (n1 == n2) return -1;
      else if (x > a[t]) n1 = t+1;
      else if (n1 < t) n2 = t-1;
      else return -1;
      }  // end of while
   }     // end of find
```


$\Longrightarrow$ end-Rekursion kann durch Iteration (d.h. eine normale Schleife) ersetzt werden ...

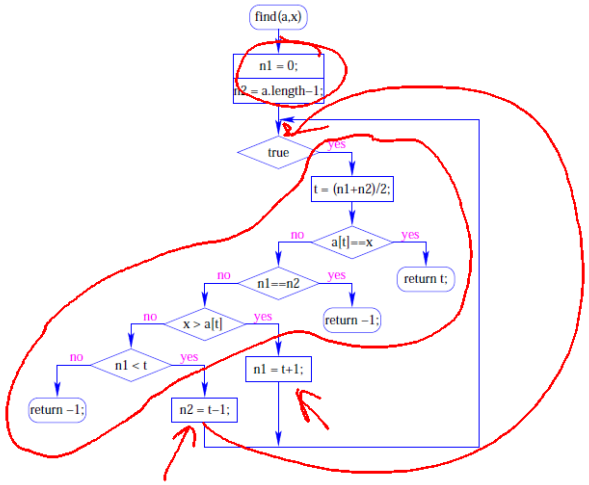
```
public static int find (int[] a, int x) {
   int n1 = 0;
   int n2 = a.length-1;
   while (true) {
      int t = (n2+n1)/2;
      if (x == a[t]) return t;
      else if (n1 == n2) return -1;
      else if (x > a[t]) n1 = t+1;
      else if (n1 < t) n2 = t-1;
      else return -1;
      }  // end of while
   }     // end of find
```

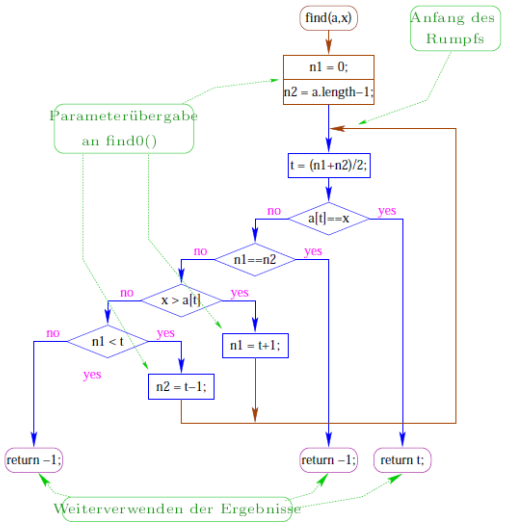

Das Kontrollfluss-Diagramm:



- 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 end-rekursiven Funktion wird ersetzt durch:
  1. Code zur Parameter-Übergabe;
  2. einen Sprung an den Anfang des Rumpfs.
- Aber Achtung, wenn die Funktion an mehreren Stellen benutzt wird !!!
  (Was ist das Problem ?

Das Kontrollfluss-Diagramm:



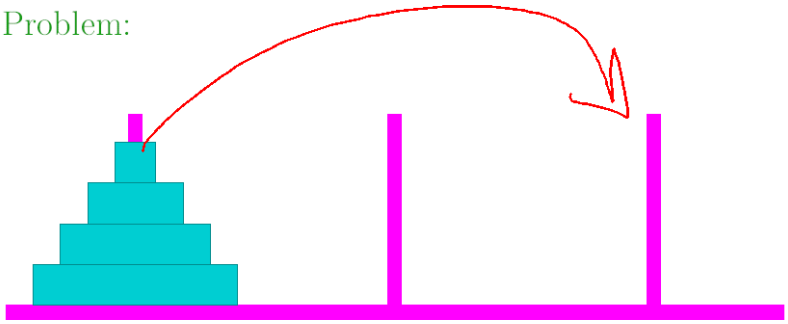


Bemerkung:

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

8 Die Türme von Hanoi

Problem:



- Bewege den Stapel von links nach rechts!

- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer Ring auf einen kleineren gelegt werden.

Idee:

- Versetzen eines Turms der Höhe $h = 0$ ist einfach: wir tun nichts.
- Versetzen eines Turms der Höhe $h > 0$ von Position a nach Position b zerlegen wir in drei Teilaufgaben:
  1. Versetzen der oberen $h - 1$ Scheiben auf den freien Platz;
  2. Versetzen der untersten Scheibe auf die Zielposition;
  3. Versetzen der zwischengelagerten Scheiben auf die Zielposition.
- Versetzen eines Turms der Höhe $h > 0$ erfordert also zweimaliges Versetzen eines Turms der Höhe $h - 1$.

```
public static void move (int h, byte a, byte b) {
  if (h > 0) {
     byte c = free (a,b);
     move (h-1,a,c);
     System.out.print ("\tmove "+a+" to "+b+"\n");
     move (h-1,c,b);
     }
  }
```

Bleibt die Ermittlung des freien Platzes ...

	0	1	2
0		2	1
1	2		0
2	1	0	

Offenbar hängt das Ergebnis nur von der Summe der beiden Argumente ab ...

	0	1	2
0		1	2
1	1		3
2	2	3	

Um solche Tabellen leicht implementieren zu können, stellt Java das `switch`-Statement zur Verfügung:

```
public static byte free (byte a, byte b) {
   switch (a+b) {
   case 1:   return 2;
   case 2:   return 1;
   case 3:   return 0;
   default:  return -1;
   }
}
```


Allgemeine Form eines `switch`-Statements:

```
switch ( expr ) {
case const_0 :   ss_0 ( break; ) ?
case const_1 :   ss_1 ( break; ) ?
    ...
case const_{k-1} :   ss_{k-1} ( break; ) ?
( default:   ss_k ) ?
}
```

- expr sollte eine ganze Zahl (oder ein `char`) sein.
- Die $const_i$ sind ganz-zahlige Konstanten.
- Die ss_i sind die alternativen Statement-Folgen.

Um solche Tabellen leicht implementieren zu können, stellt Java das `switch`-Statement zur Verfügung:

```
public static byte free (byte a, byte b) {
   switch (a+b) {
   case 1:   return 2;
   case 2:   return 1;
   case 3:   return 0;
   default:  return -1;
   }
}
```


Allgemeine Form eines `switch`-Statements:

```
switch ( expr ) {
case const_0 :   ss_0 ( break; ) ?
case const_1 :   ss_1 ( break; ) ?
    ...
case const_{k-1} :   ss_{k-1} ( break; ) ?
( default:   ss_k ) ?
}
```

- expr sollte eine ganze Zahl (oder ein `char`) sein.
- Die $const_i$ sind ganz-zahlige Konstanten.
- Die ss_i sind die alternativen Statement-Folgen.

Um solche Tabellen leicht implementieren zu können, stellt Java das `switch`-Statement zur Verfügung:

```
public static byte free (byte a, byte b) {
   switch (a+b) {
   case 1:   return 2;
   case 2:   return 1;
   case 3:   return 0;
   default:  return -1;
   }
}
```

213

- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren.

Eine einfachere Lösung in unserem Fall ist :

```
public static byte free (byte a, byte b) {
   return (byte) (3-(a+b));
   }
```

	0	1	2
0		2	1
1	2		0
2	1	0	

Offenbar hängt das Ergebnis nur von der Summe der beiden Argumente ab ...

	0	1	2
0		2	2
1	1		3
2	2	3	

- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren.

Eine einfachere Lösung in unserem Fall ist :

```
public static byte free (byte a, byte b) {
   return (byte) (3-(a+b));
   }
```

Typanpassung (type cast)

Für einen Turm der Höhe $h = 4$ liefert das:

```
move 0 to 1
move 0 to 2
move 1 to 2
move 0 to 1
move 2 to 0
move 2 to 1
move 0 to 1
move 0 to 2
move 1 to 2
move 1 to 0
move 2 to 0
move 1 to 2
move 0 to 1
move 0 to 2
move 1 to 2
```