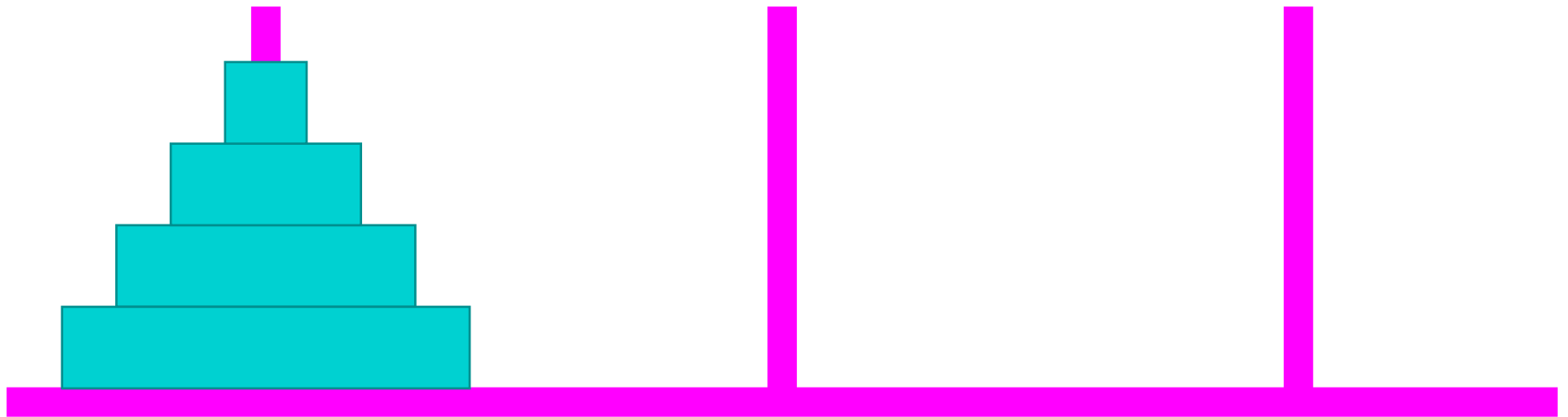
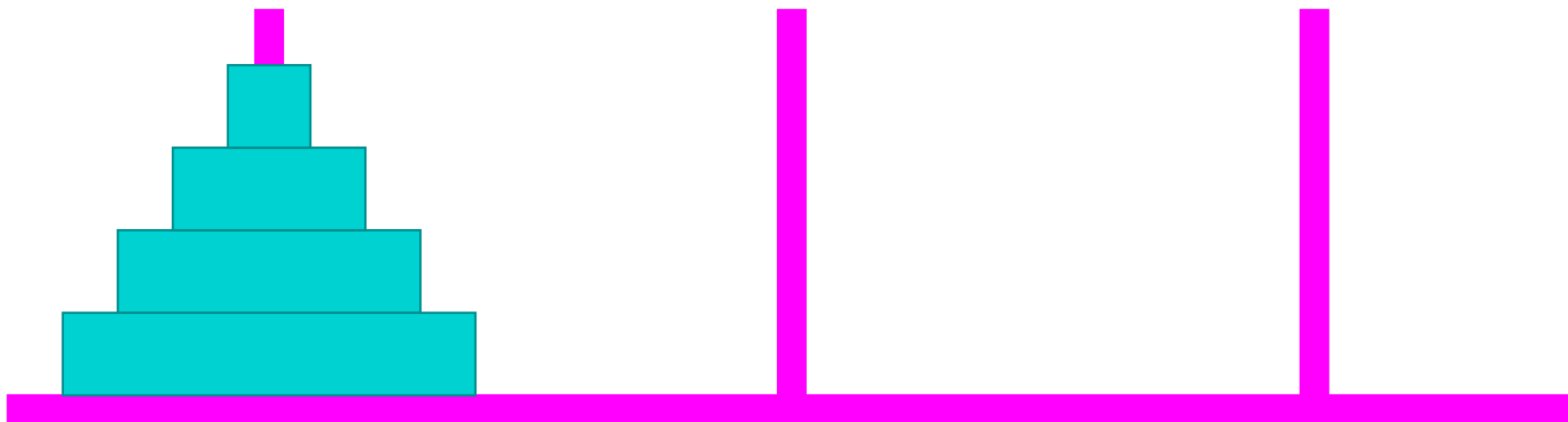


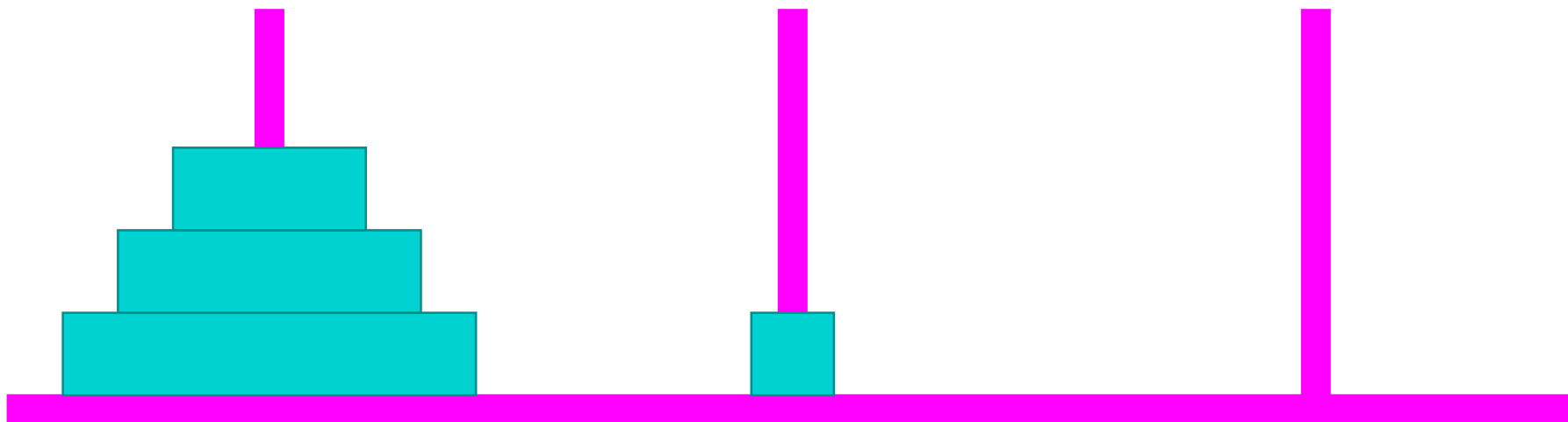
8 Die Türme von Hanoi

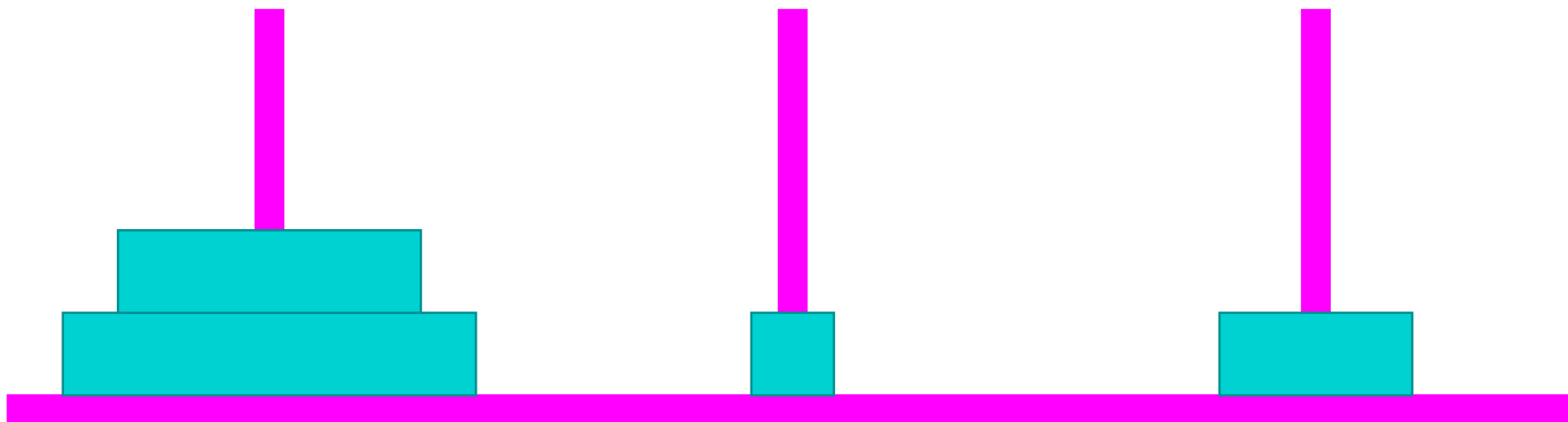
Problem:

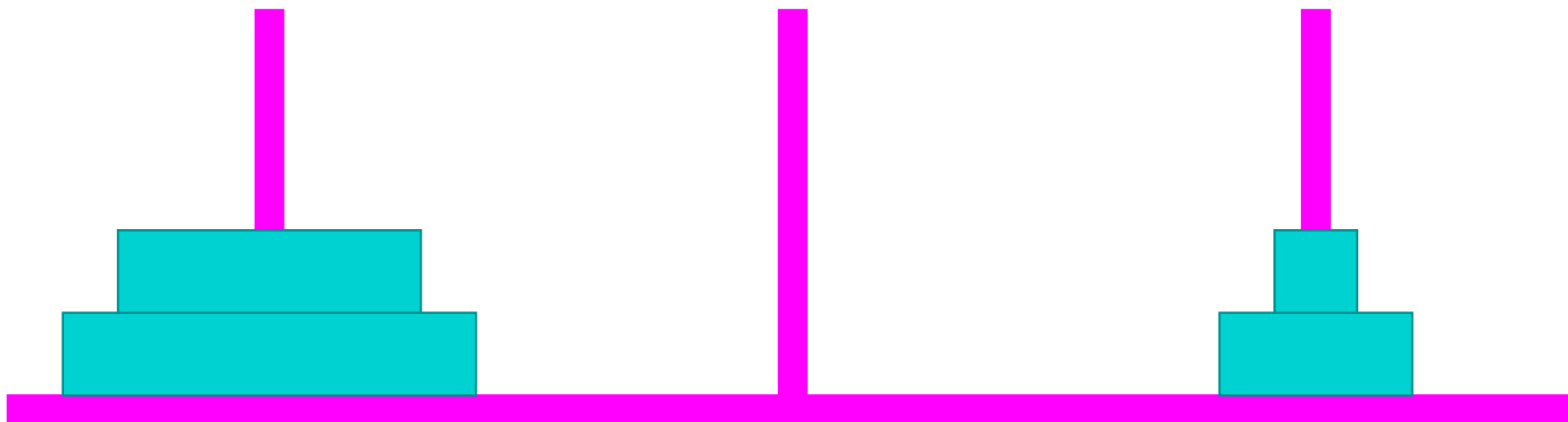


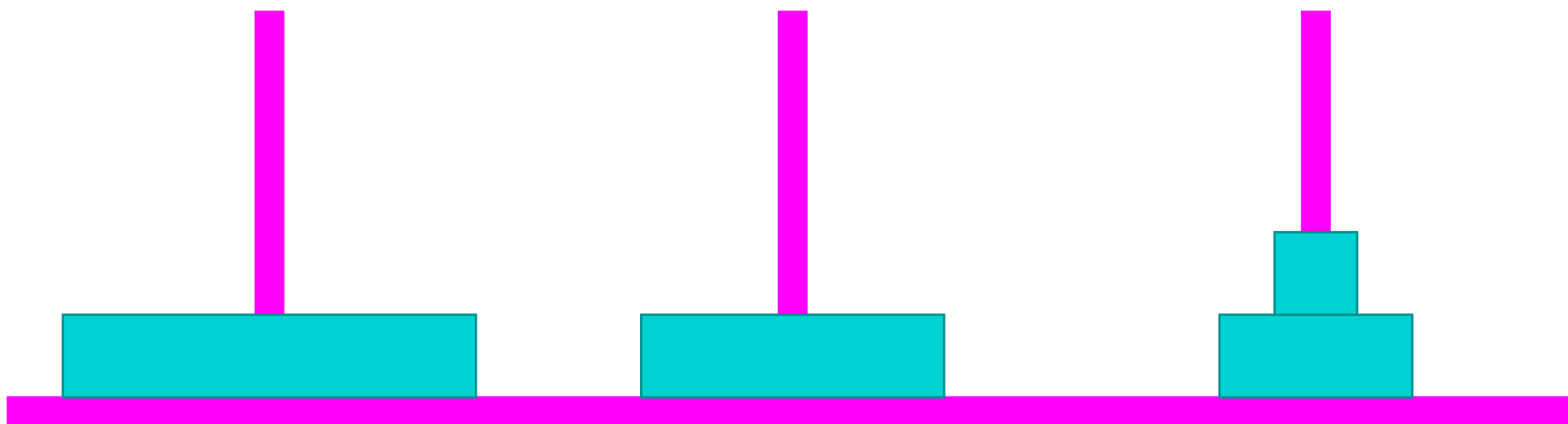
- Bewege den Stapel von links nach rechts!
- In jedem Zug darf genau ein Ring bewegt werden.
- Es darf nie ein größerer Ring auf einen kleineren gelegt werden.

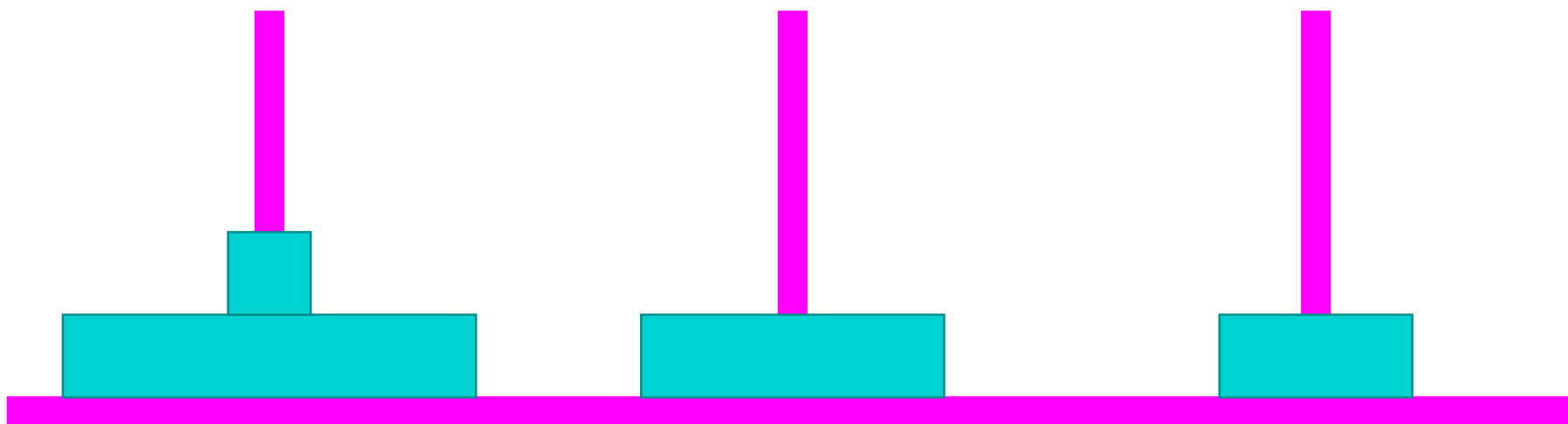


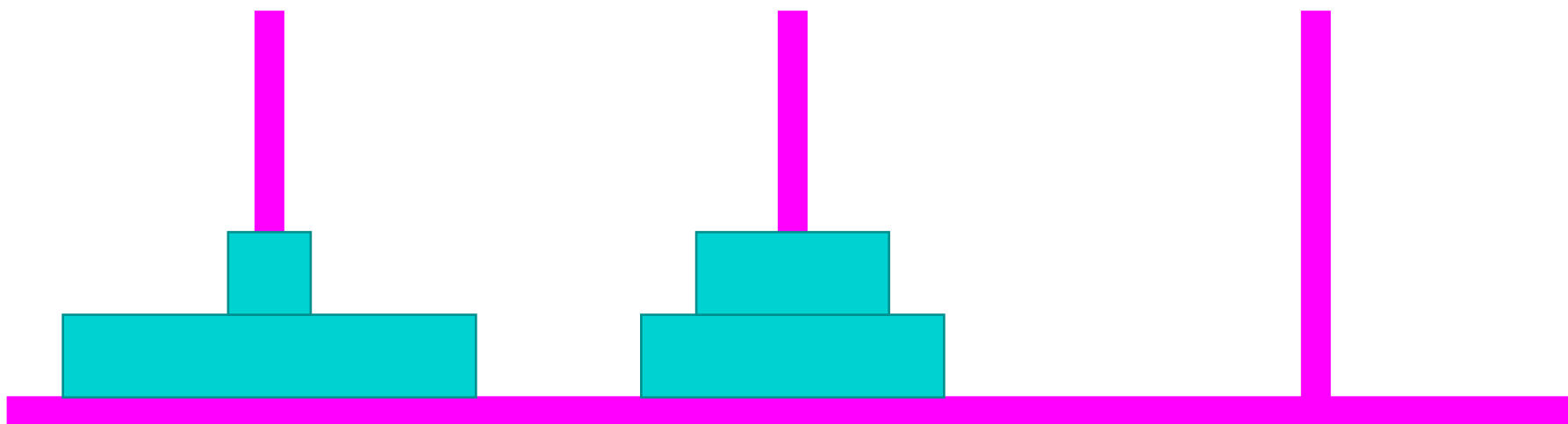


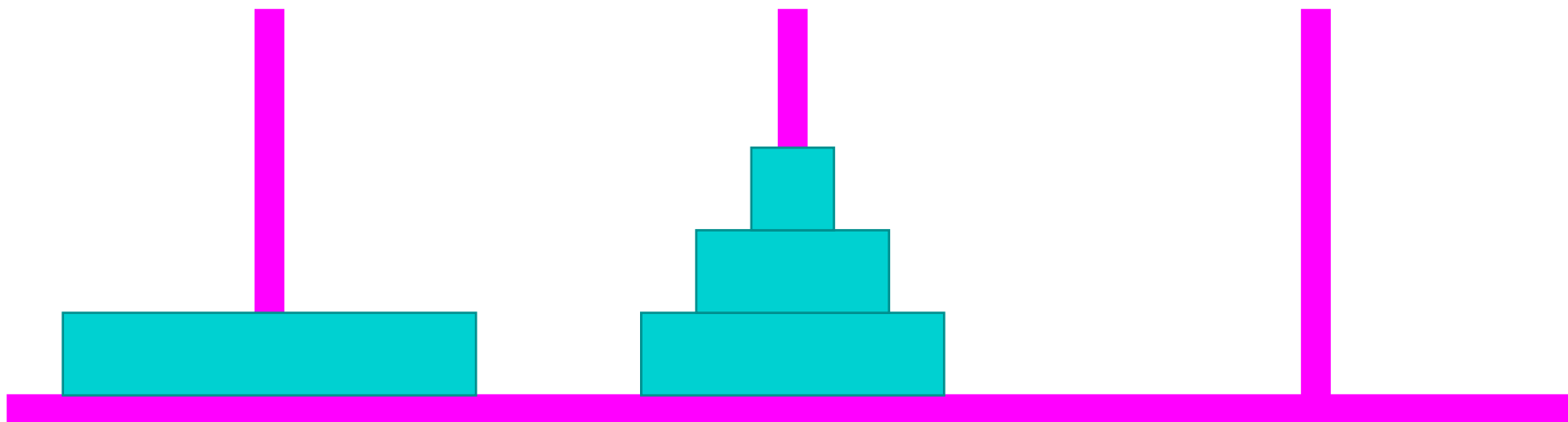


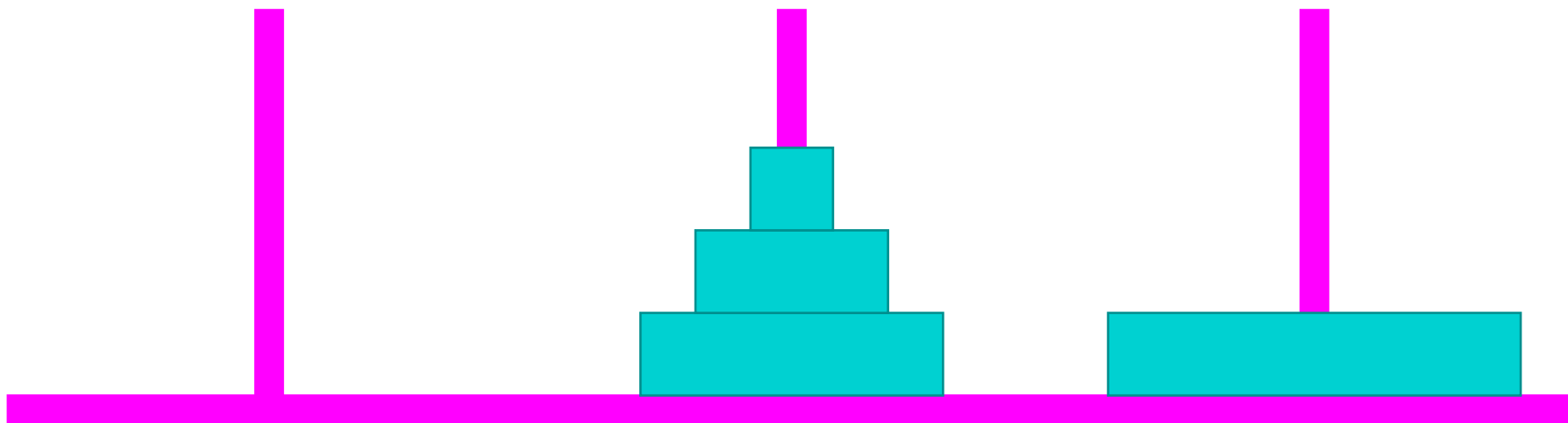


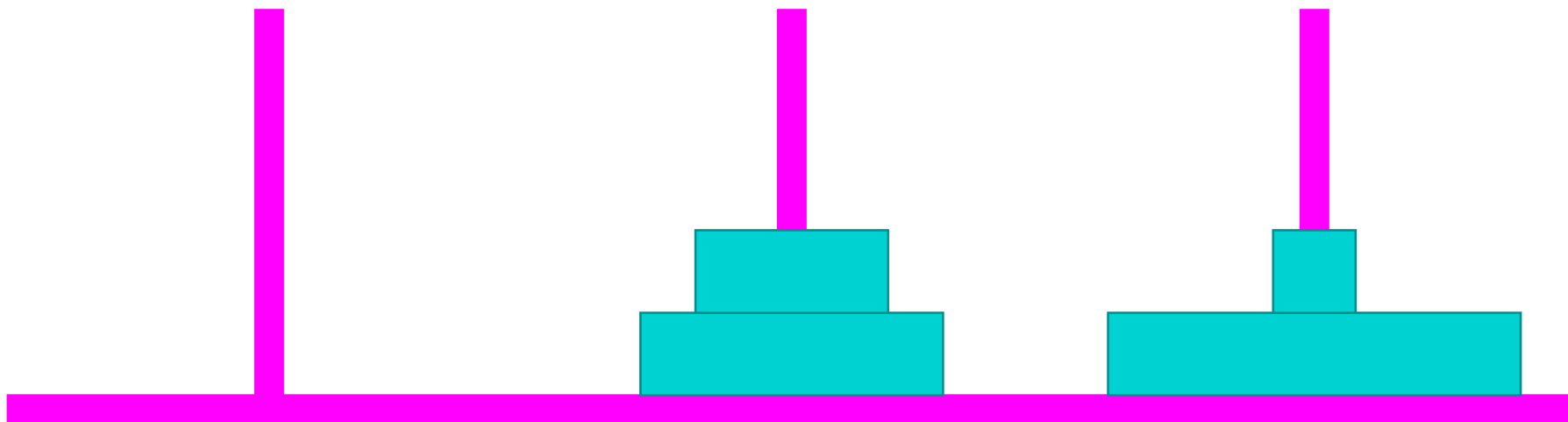


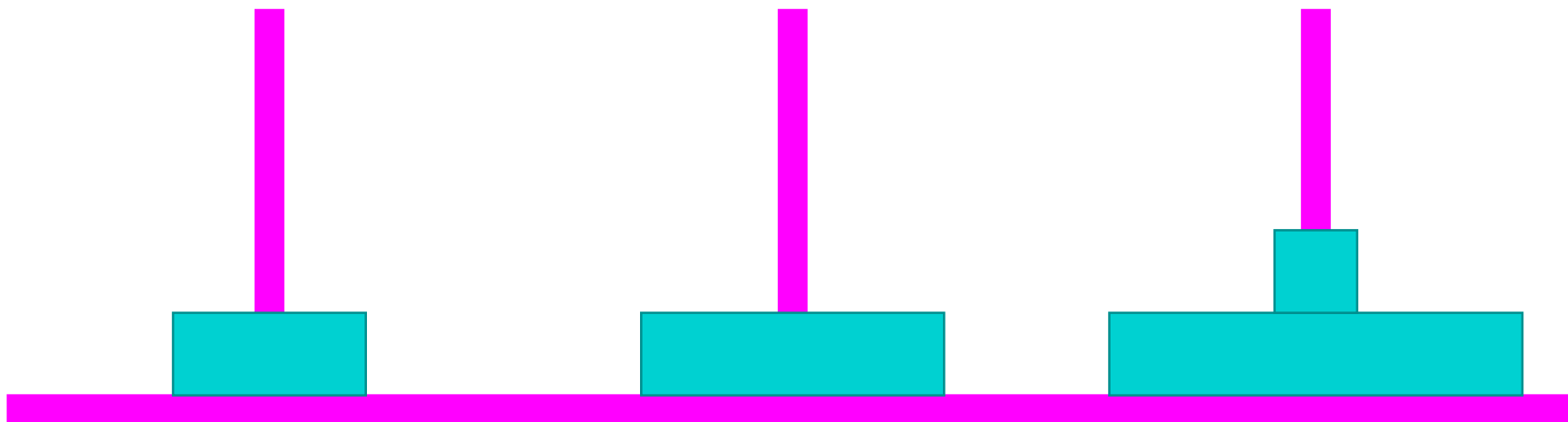


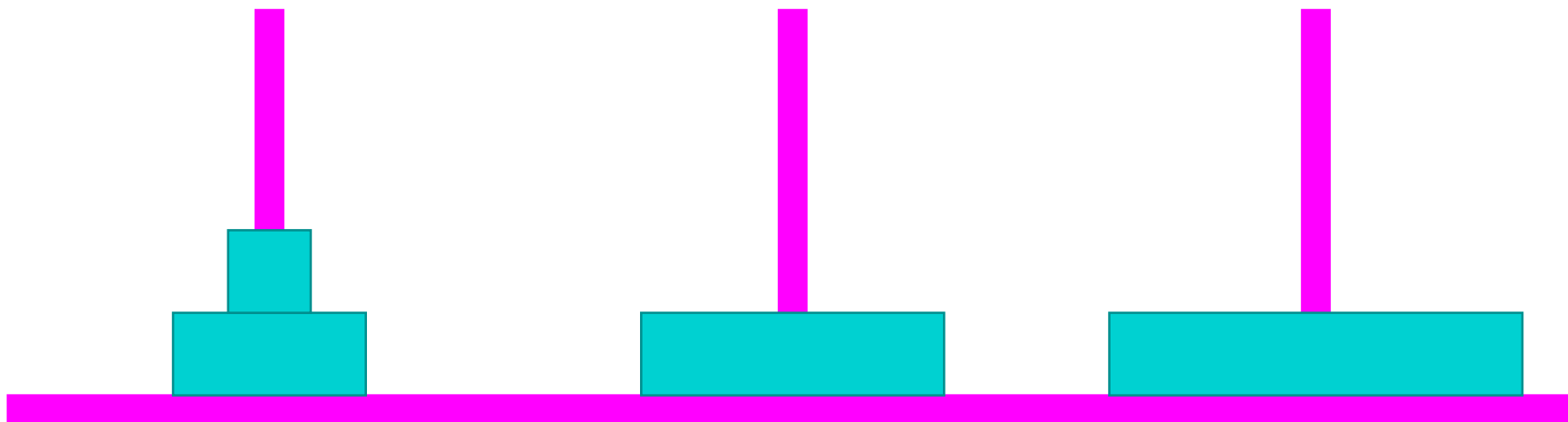


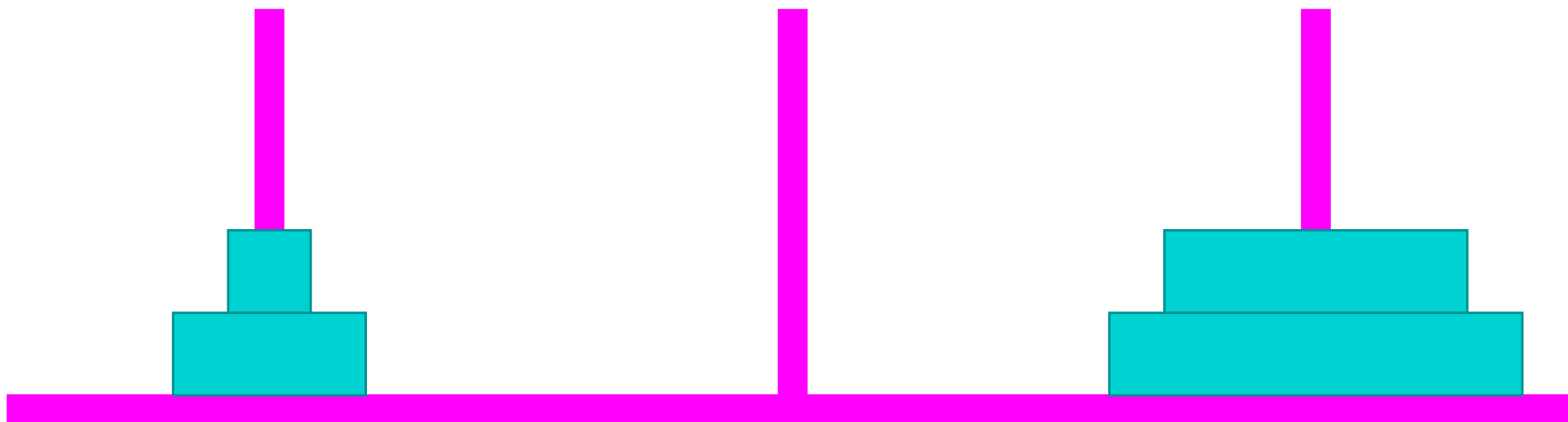


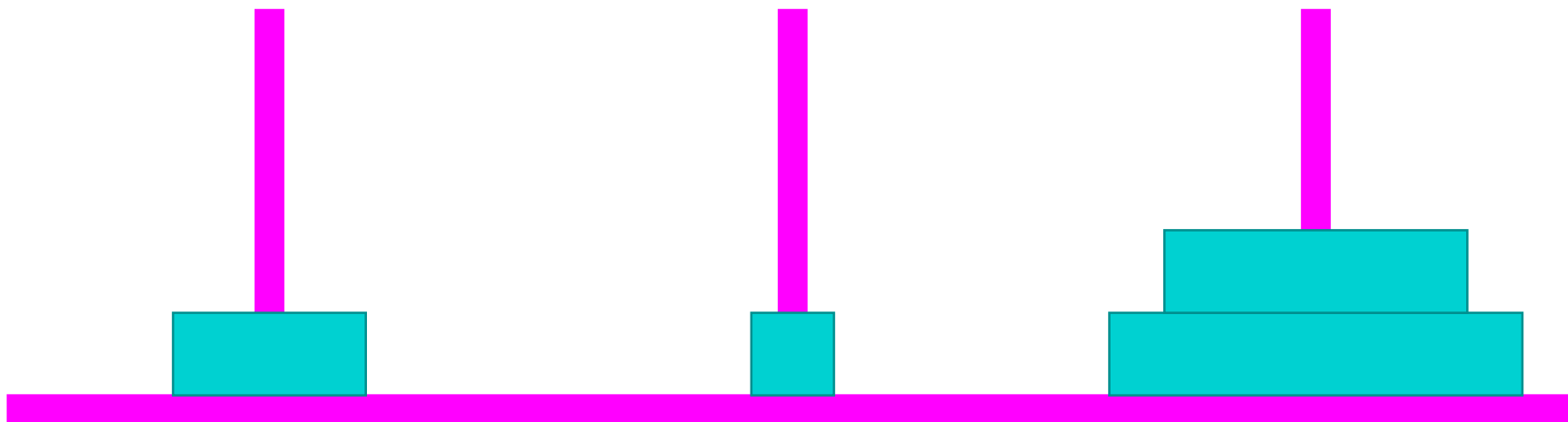


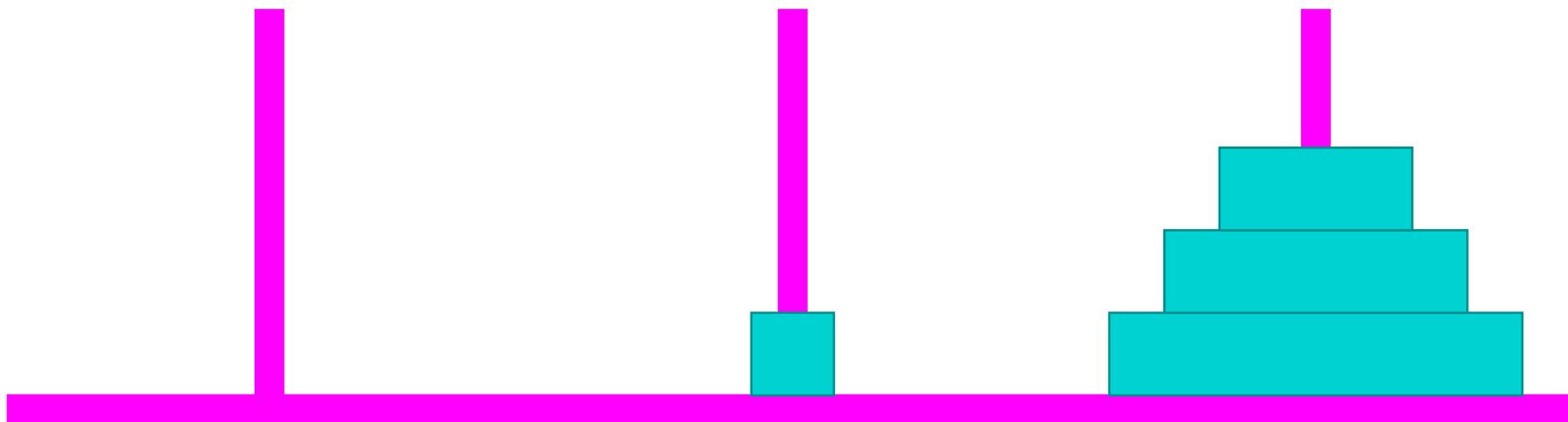


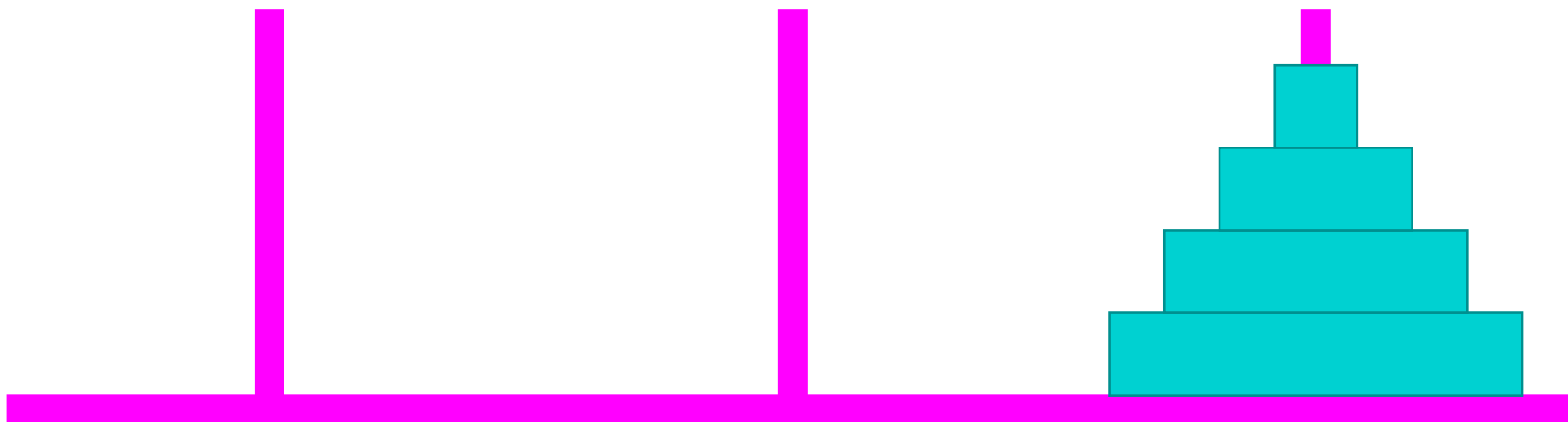












Idee:

- 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 const0 :   ss0 ( break; ) ?  
  case const1 :   ss1 ( break; ) ?  
    ...  
  case constk-1 :  ssk-1 ( break; ) ?  
  ( default:   ssk ) ?  
}
```

- **expr** sollte eine ganze Zahl (oder ein char) sein.
- Die **const**_i sind ganz-zahlige Konstanten.
- Die **ss**_i sind die alternativen Statement-Folgen.

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

- `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));  
}
```

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
```

Bemerkungen:

- `move()` ist rekursiv, aber nicht end-rekursiv.
- Sei $N(h)$ die Anzahl der ausgegebenen Moves für einen Turm der Höhe $h \geq 0$. Dann ist

$$N(0) = 0 \quad \text{und für } h > 0,$$

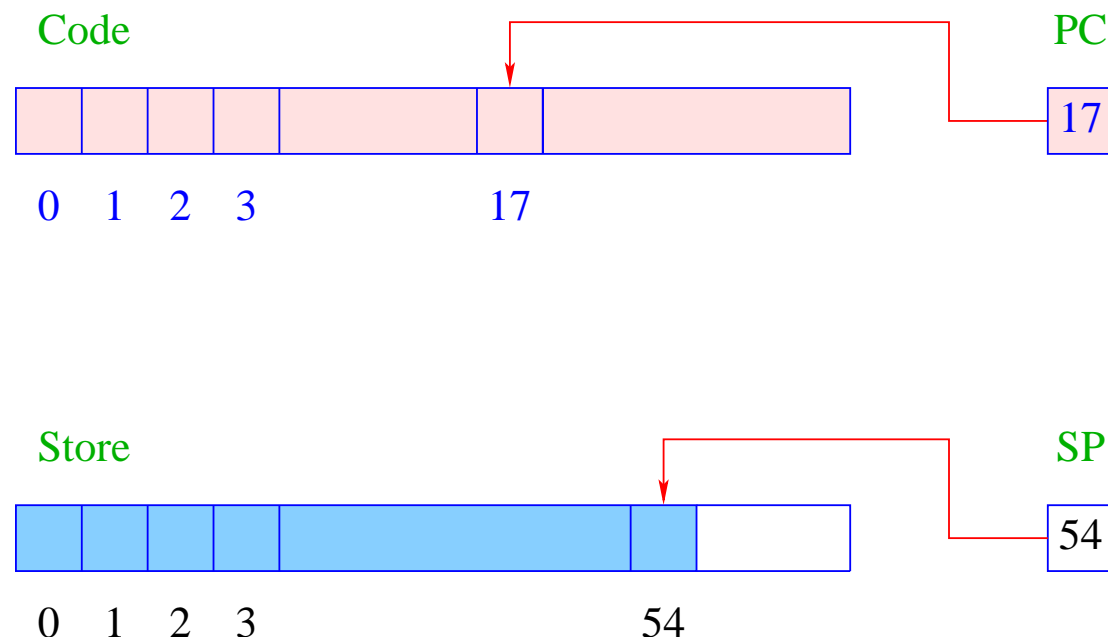
$$N(h) = 1 + 2 \cdot N(h - 1)$$

- Folglich ist $N(h) = 2^h - 1$.
- Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden ... (wie könnte der aussehen? :-)

Hinweis: Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter ...

9 Von MiniJava zur JVM

Architektur der JVM:



Code = enthält JVM-Programm;
jede Zelle enthält einen Befehl;

PC = Program Counter –
zeigt auf nächsten auszuführenden Befehl;

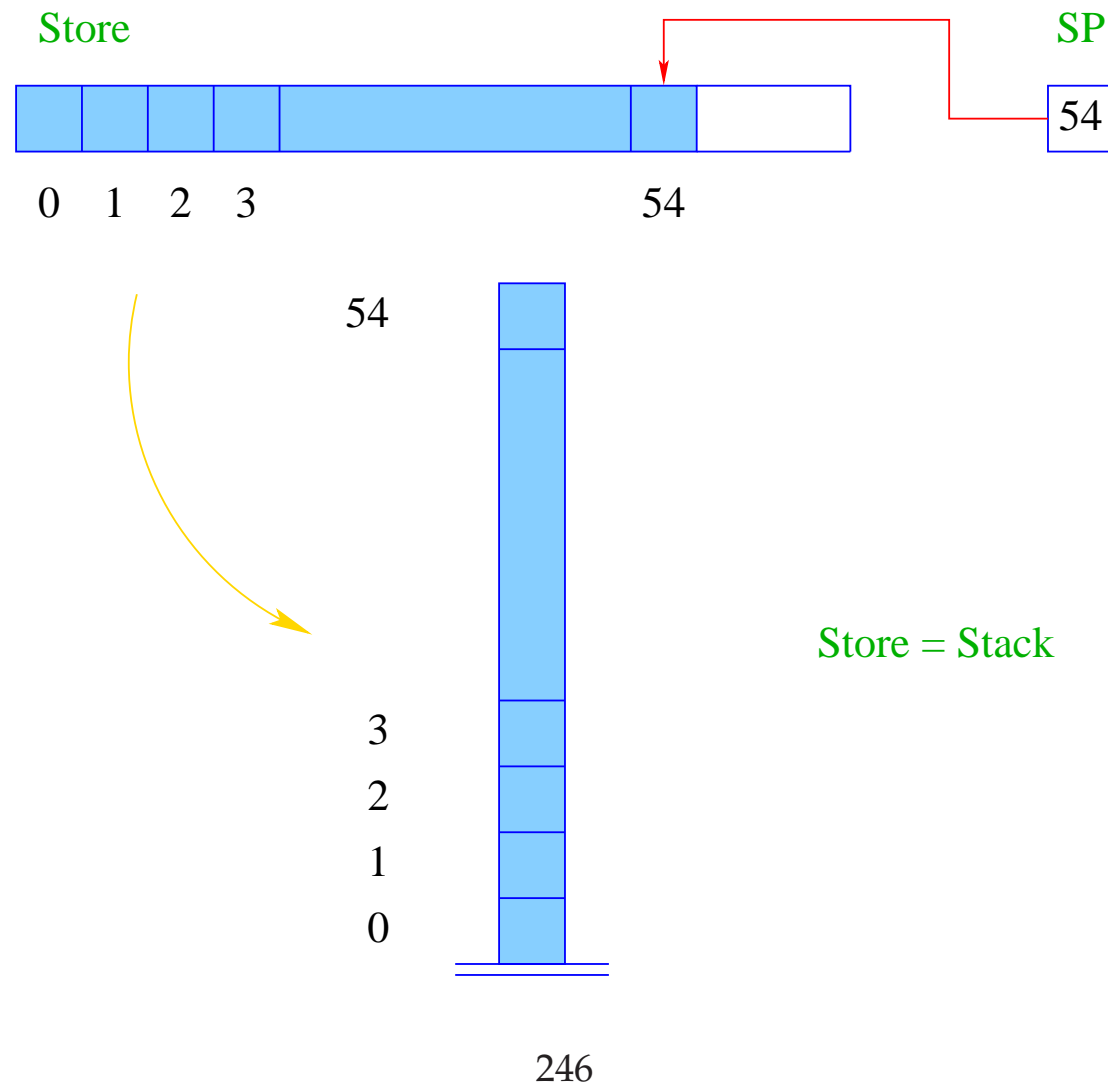
Store = Speicher für Daten;
jede Zelle kann einen Wert aufnehmen;

SP = Stack-Pointer –
zeigt auf oberste belegte Zelle.

Achtung:

- Programm wie Daten liegen im Speicher – aber in verschiedenen Abschnitten.
- Programm-Ausführung holt nacheinander Befehle aus **Code** und führt die entsprechenden Operationen auf **Store** aus.

Konvention:



Befehle der JVM:

int-Operatoren:	NEG, ADD, SUB, MUL, DIV, MOD
boolean-Operatoren:	NOT, AND, OR
Vergleichs-Operatoren:	LESS, LEQ, EQ, NEQ
Laden von Konstanten:	CONST i, TRUE, FALSE
Speicher-Operationen:	LOAD i, STORE i
Sprung-Befehle:	JUMP i, FJUMP i
IO-Befehle:	READ, WRITE
Reservierung von Speicher:	ALLOC i
Beendung des Programms:	HALT

Ein Beispiel-Programm:

	ALLOC 2	LOAD 0	B: LOAD 0
	READ	LOAD 1	LOAD 1
	STORE 0	LESS	SUB
	READ	FJUMP B	STORE 0
	STORE 1	LOAD 1	C: JUMP A
A: LOAD 0		LOAD 0	D: LOAD 1
LOAD 1		SUB	WRITE
NEQ		STORE 1	HALT
FJUMP D		JUMP C	

- Das Programm berechnet den GGT :-)
- Die Marken (**Labels**) A, B, C, D bezeichnen symbolisch die Adressen der zugehörigen Befehle:

A = 5

B = 18

C = 22

D = 23

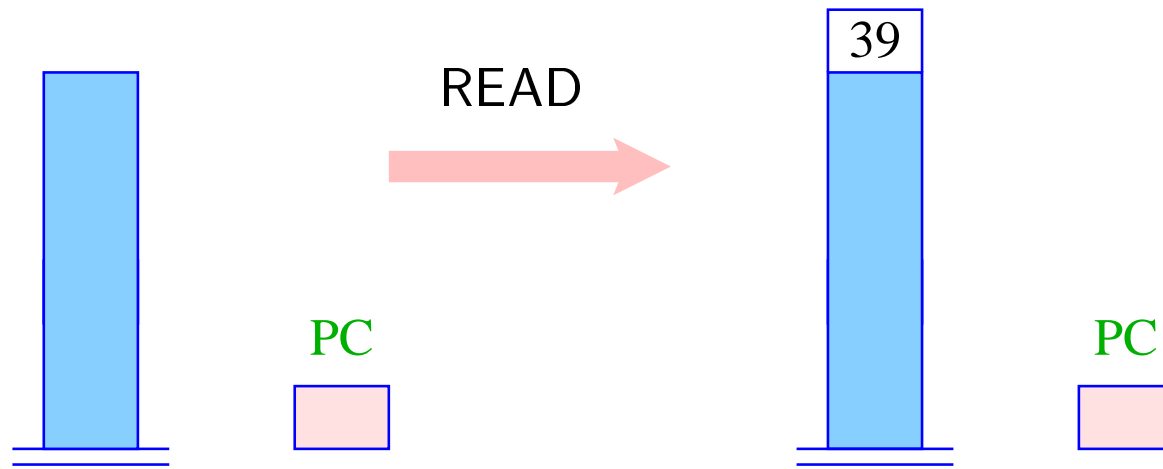
- ... können vom Compiler **leicht** in die entsprechenden Adressen umgesetzt werden (wir benutzen sie aber, um uns besser im Programm zurechtzufinden :-)

Bevor wir erklären, wie man **MiniJava** in **JVM**-Code übersetzt, erklären wir, was die einzelnen Befehle bewirken.

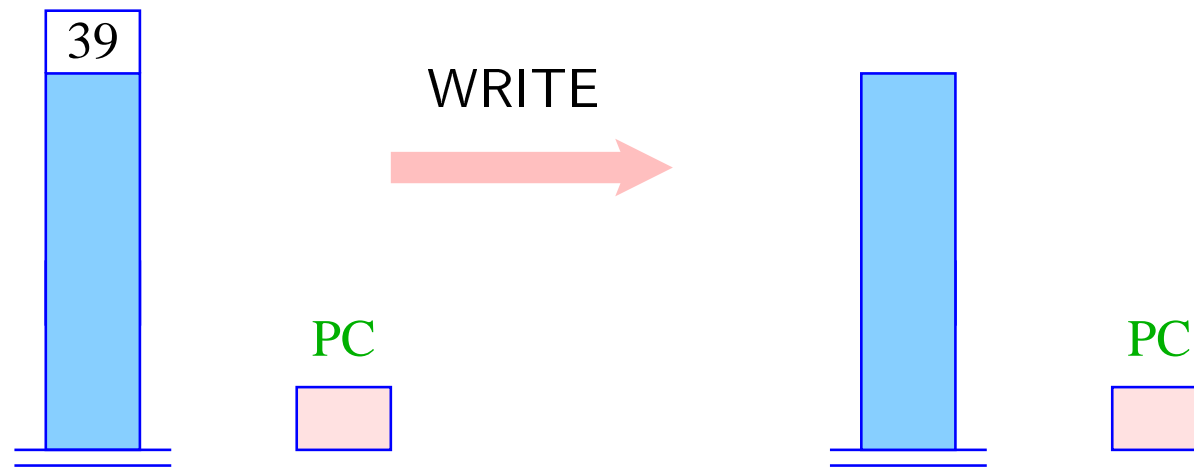
Idee:

- Befehle, die Argumente benötigen, erwarten sie am oberen Ende des Stack.
- Nach ihrer Benutzung werden die Argumente vom Stack herunter geworfen.
- Mögliche Ergebnisse werden oben auf dem Stack abgelegt.

Betrachten wir als Beispiele die IO-Befehle **READ** und **WRITE**.



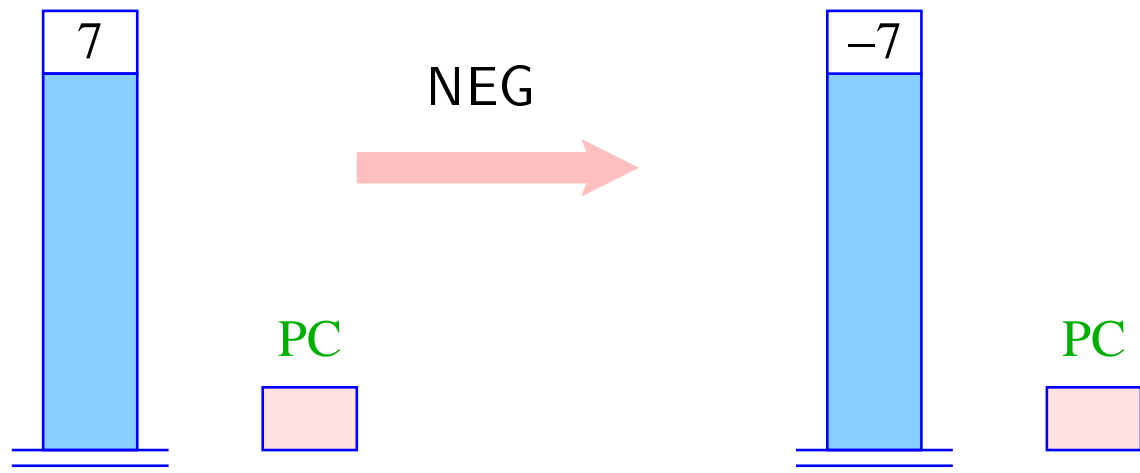
... falls 39 eingegeben wurde

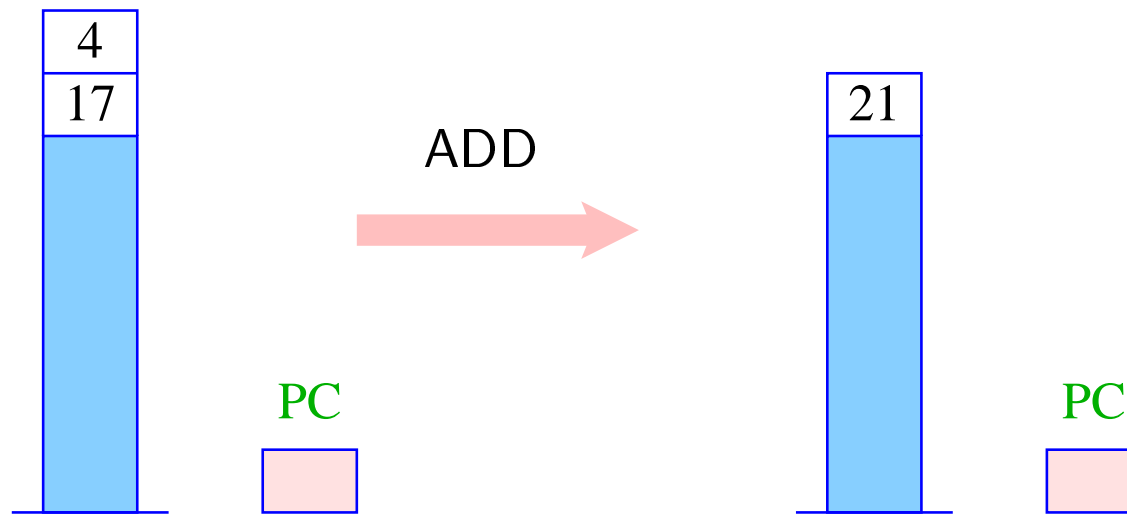


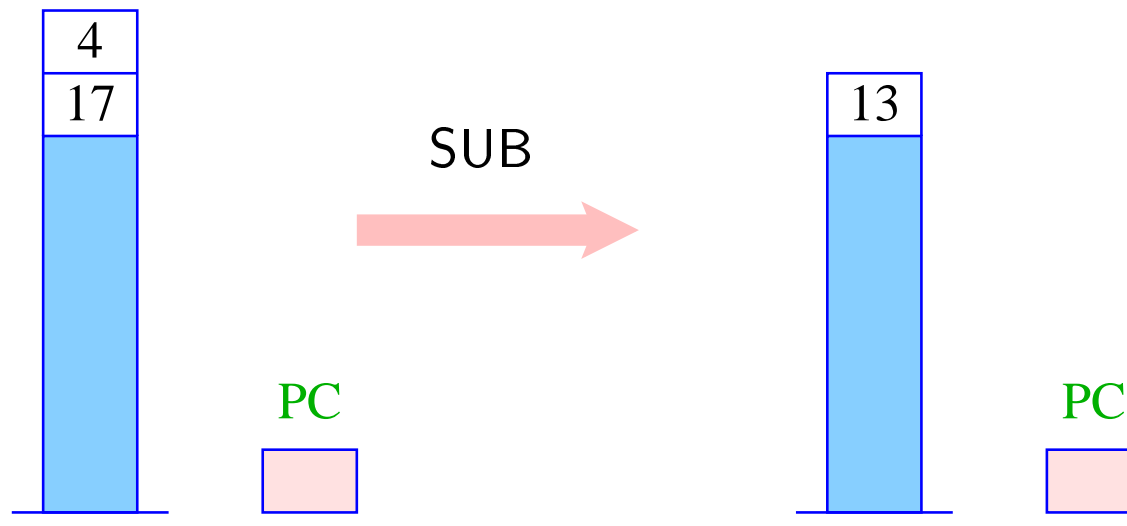
... wobei 39 ausgegeben wird

Arithmetik

- Unäre Operatoren modifizieren die oberste Zelle.
- Binäre Operatoren verkürzen den Stack.







- Die übrigen arithmetischen Operationen MUL, DIV, MOD funktionieren völlig analog.
- Die logischen Operationen NOT, AND, OR ebenfalls – mit dem Unterschied, dass sie statt mit ganzen Zahlen, mit Intern-Darstellungen von true und false arbeiten (hier: “tt” und “ff”).
- Auch die Vergleiche arbeiten so – nur konsumieren sie ganze Zahlen und liefern einen logischen Wert.

