Factorials

- The number **n factorial**, denoted **n!**, is

$$n \times (n - 1) \times \ldots \times 3 \times 2 \times 1$$

- For example:
  - $3! = 3 \times 2 \times 1 = 6$.
  - $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
  - $0! = 1$ (by definition)
- Factorials show up everywhere:
  - Taylor series.
  - Counting ways to shuffle a deck of cards.
  - Determining how quickly computers can sort values (more on that later this quarter).

Thinking Recursively

Factorial Revisited

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Factorial Revisited

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Factorial Revisited

$$5! \;=\; 5 \times \underbrace{4 \times 3 \times 2 \times 1}_{4!}$$

Factorial Revisited

$$5! = 5 \times 4!$$

Factorial Revisited

$5! = 5 \times 4!$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3 \times 2 \times 1$$

Factorial Revisited

$$5! \;=\; 5 \times 4!$$

$$4! \;=\; 4 \times 3 \times 2 \times 1$$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times \underbrace{3 \times 2 \times 1}_{3!}$$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times \underbrace{2 \times 1}_{2!}$$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

Factorial Revisited

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

Factorial Revisited

$$\begin{aligned} 5! &= 5 \times 4! \\ 4! &= 4 \times 3! \\ 3! &= 3 \times 2! \\ 2! &= 2 \times 1! \end{aligned}$$

Factorial Revisited

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= 1 \times 0!
\end{aligned}
$$

Factorial Revisited

$$
\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= 1 \times 0! \\
0! &= 1
\end{aligned}
$$

Another View of Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Another View of Factorials

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n - 1)! & \text{otherwise} \end{cases}$$

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action

```
int main() {
   int n = factorial(5);
   cout << "5! = " << n << endl;
}
```

Recursion in Action

```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
}
```

Recursion in Action

```
int main() {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

int n 5

Recursion in Action

```
int main() {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

int n 5

Recursion in Action

```
int main() {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

int n 5

Recursion in Action

```
int main() {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

int n 5

Recursion in Action

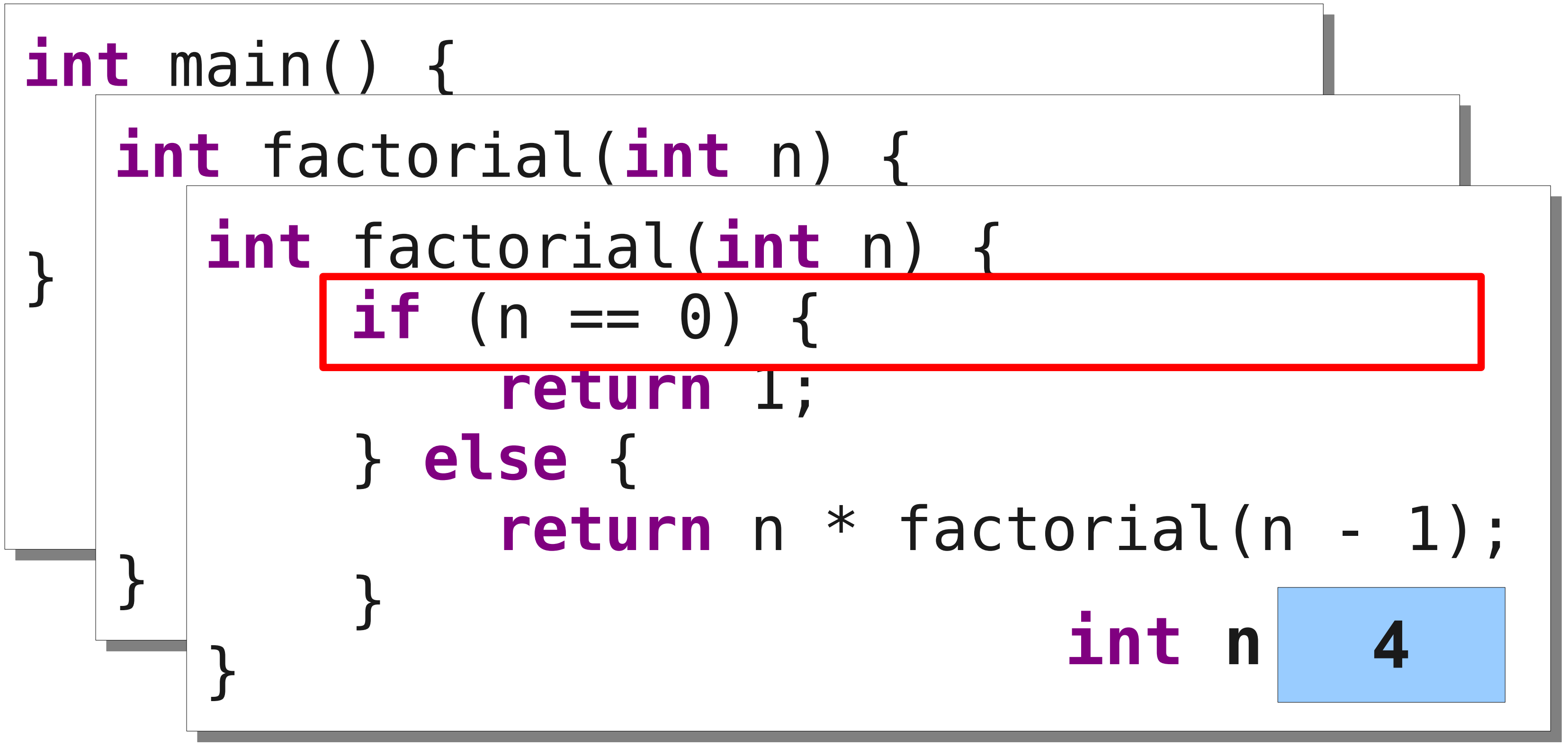
```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

int n | 4

Recursion in Action

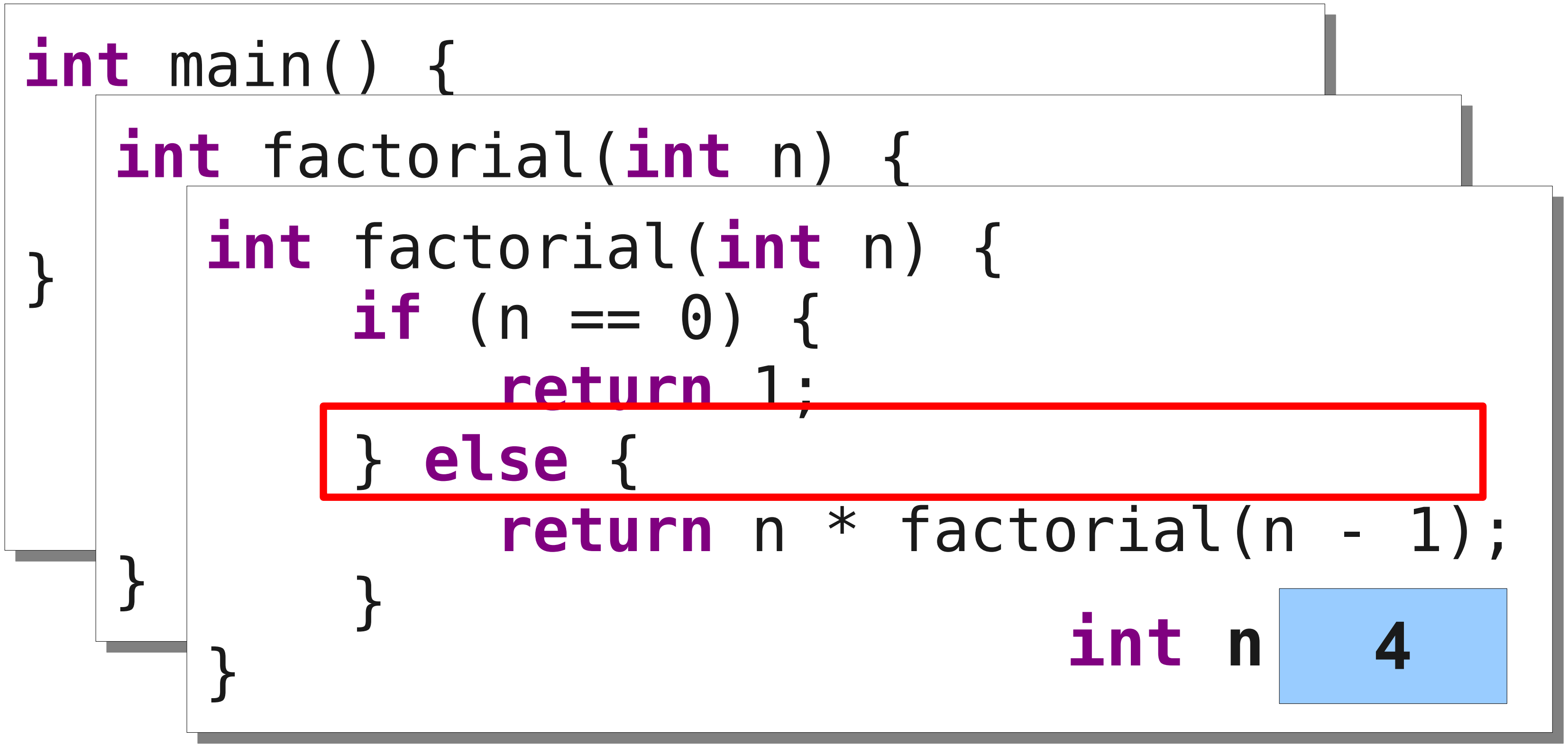


```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action

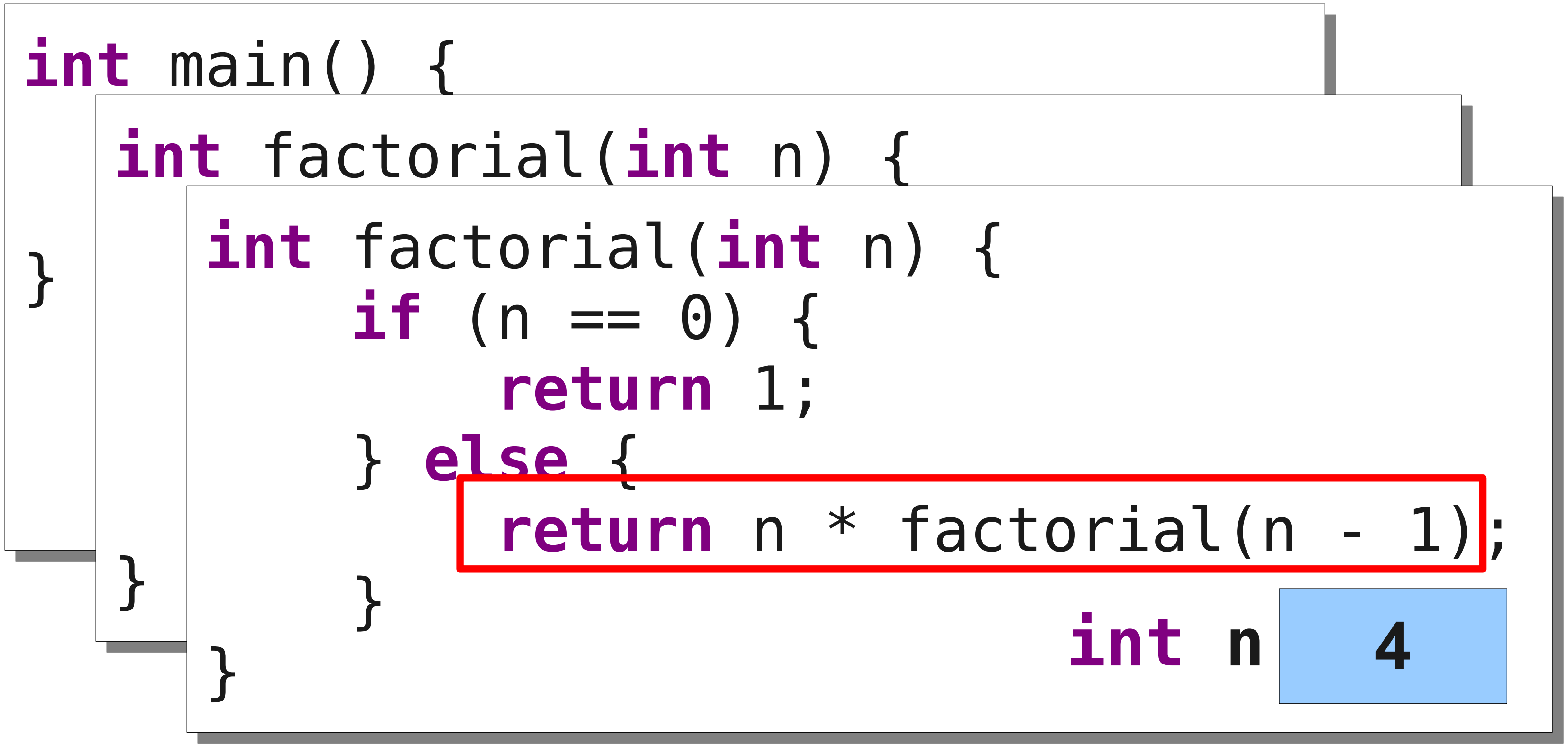


```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action

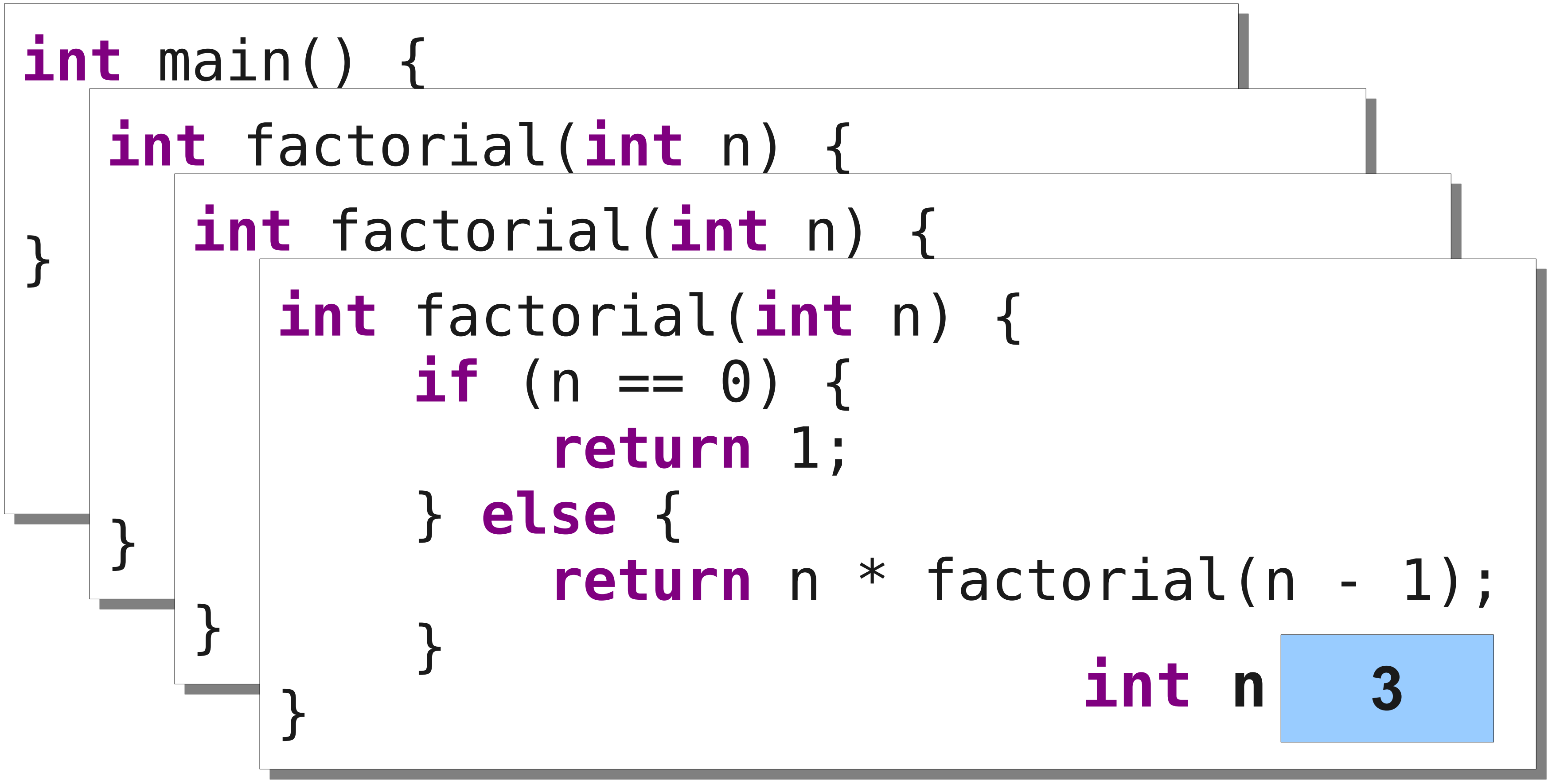


```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



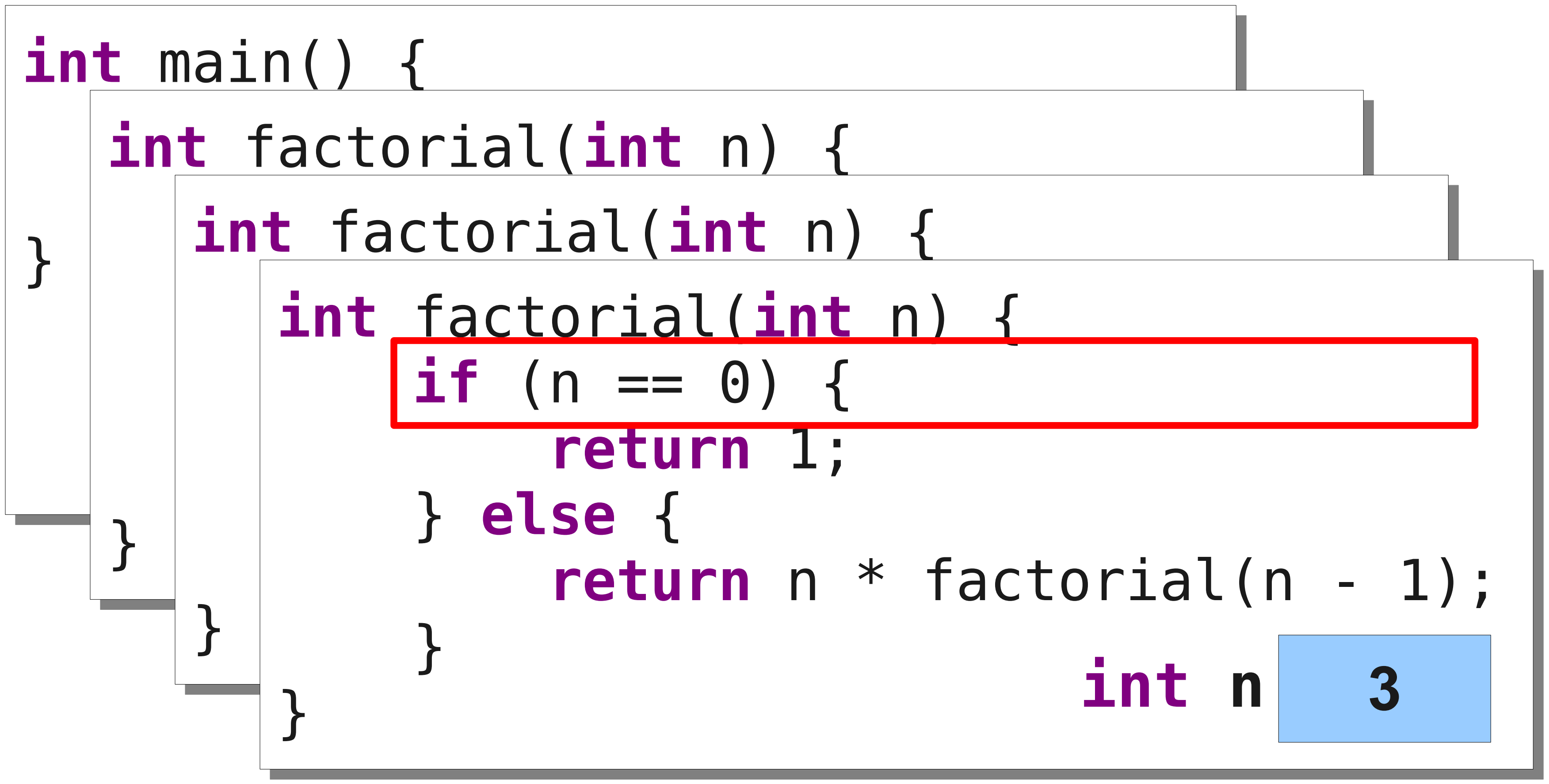
```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {
}
```

```
int factorial(int n) {
}
```

```
int factorial(int n) {
}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action

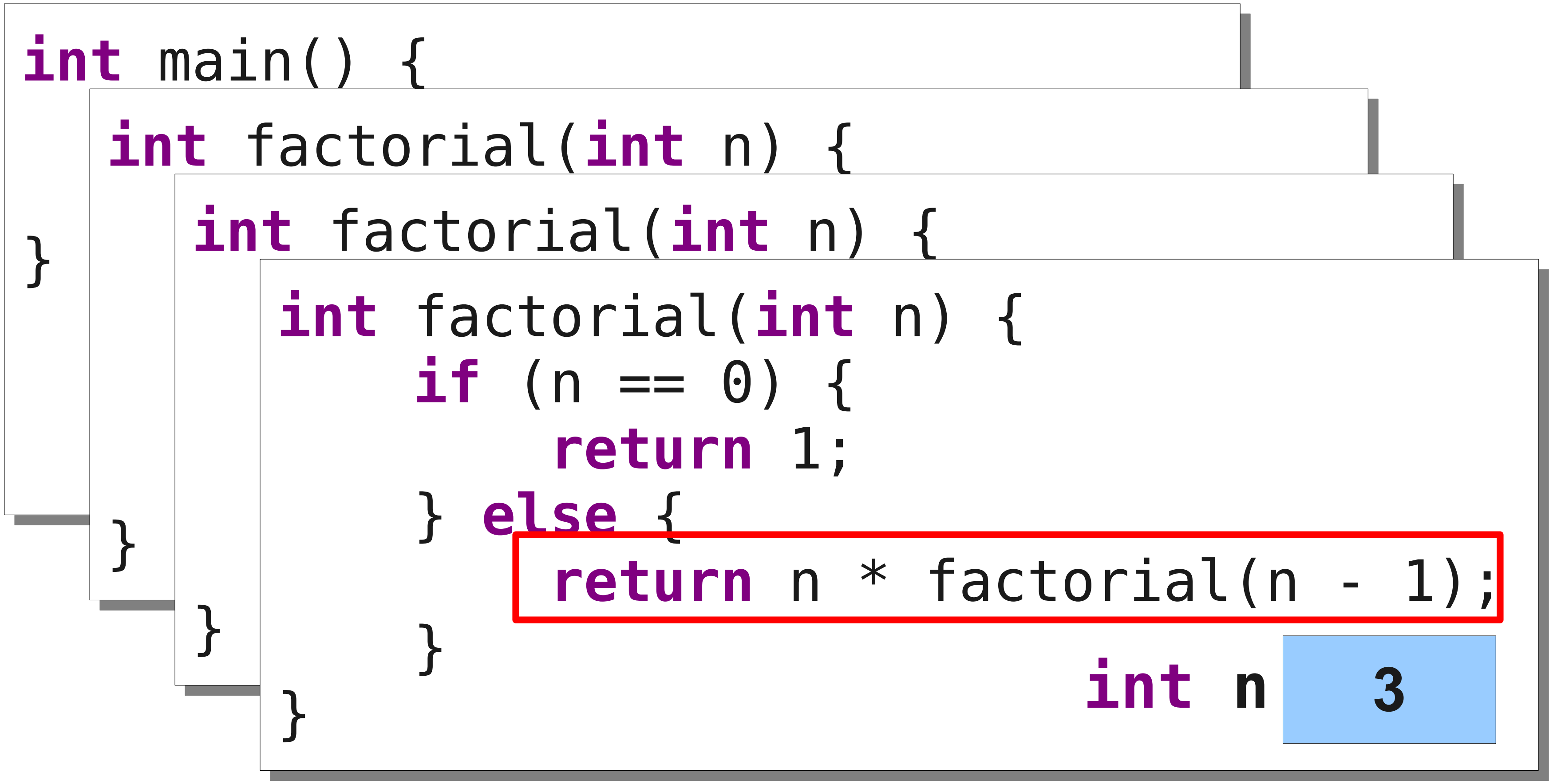
```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
                        int n   3
}
```

Recursion in Action



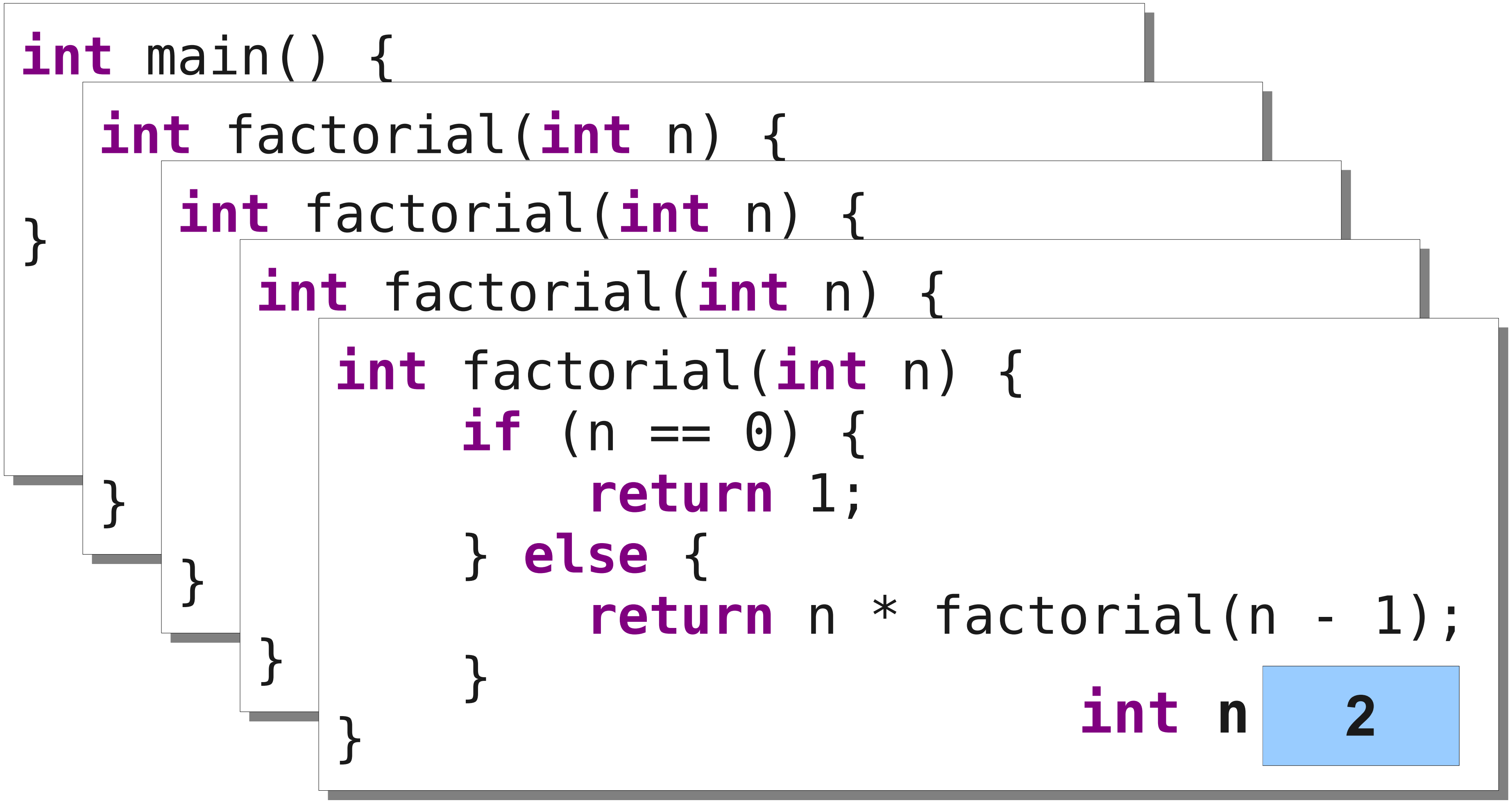
```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

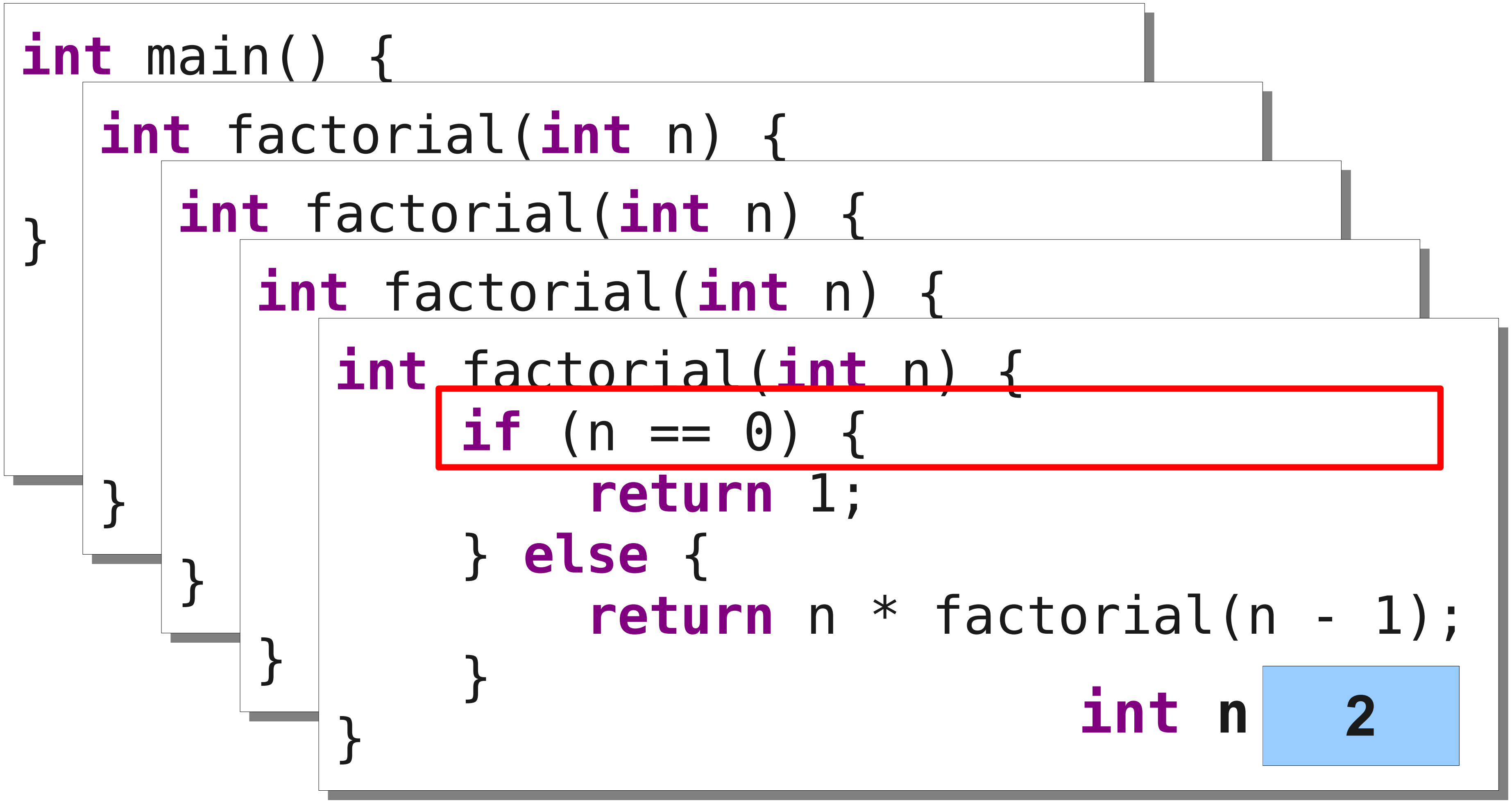
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

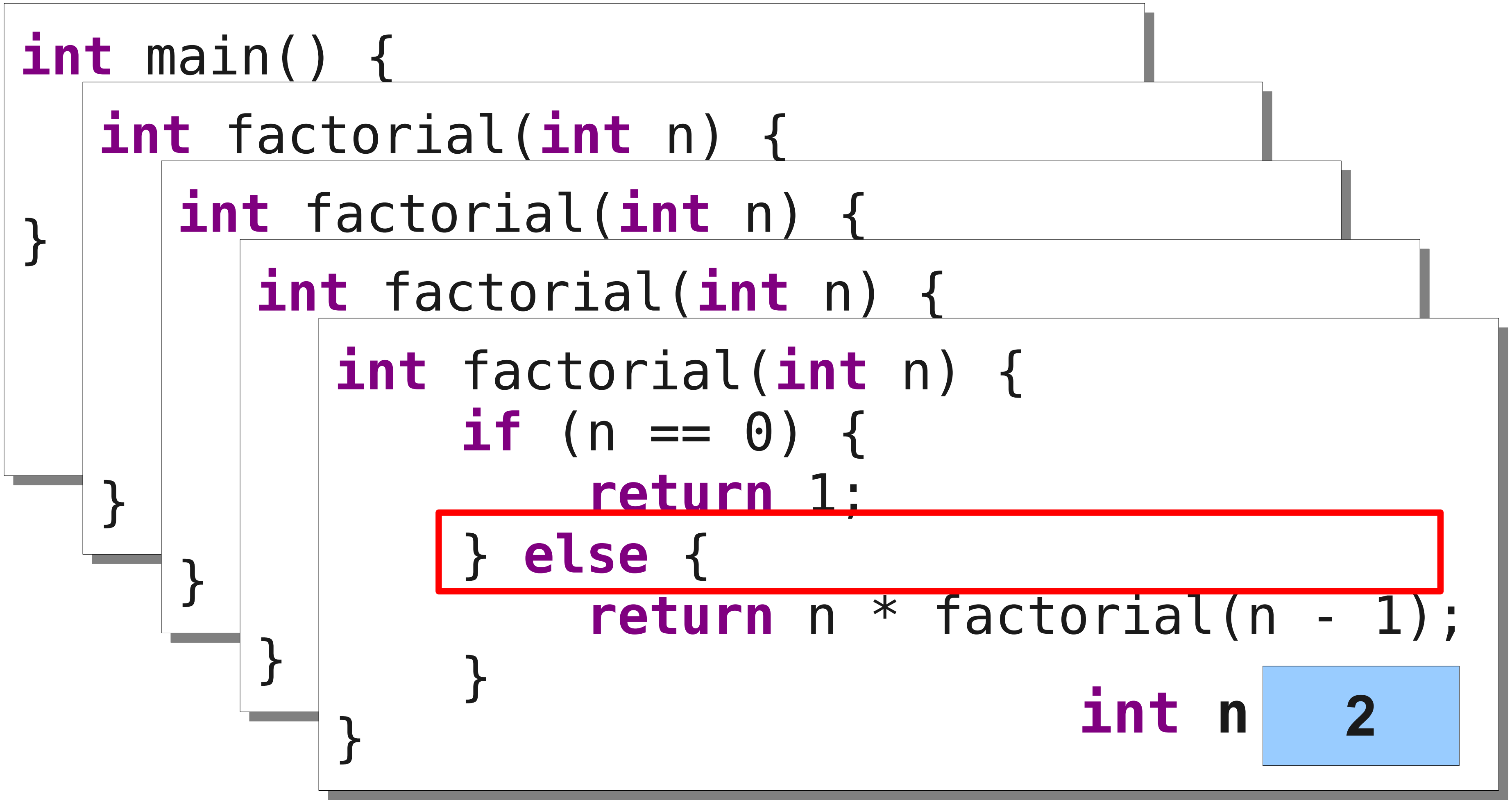
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

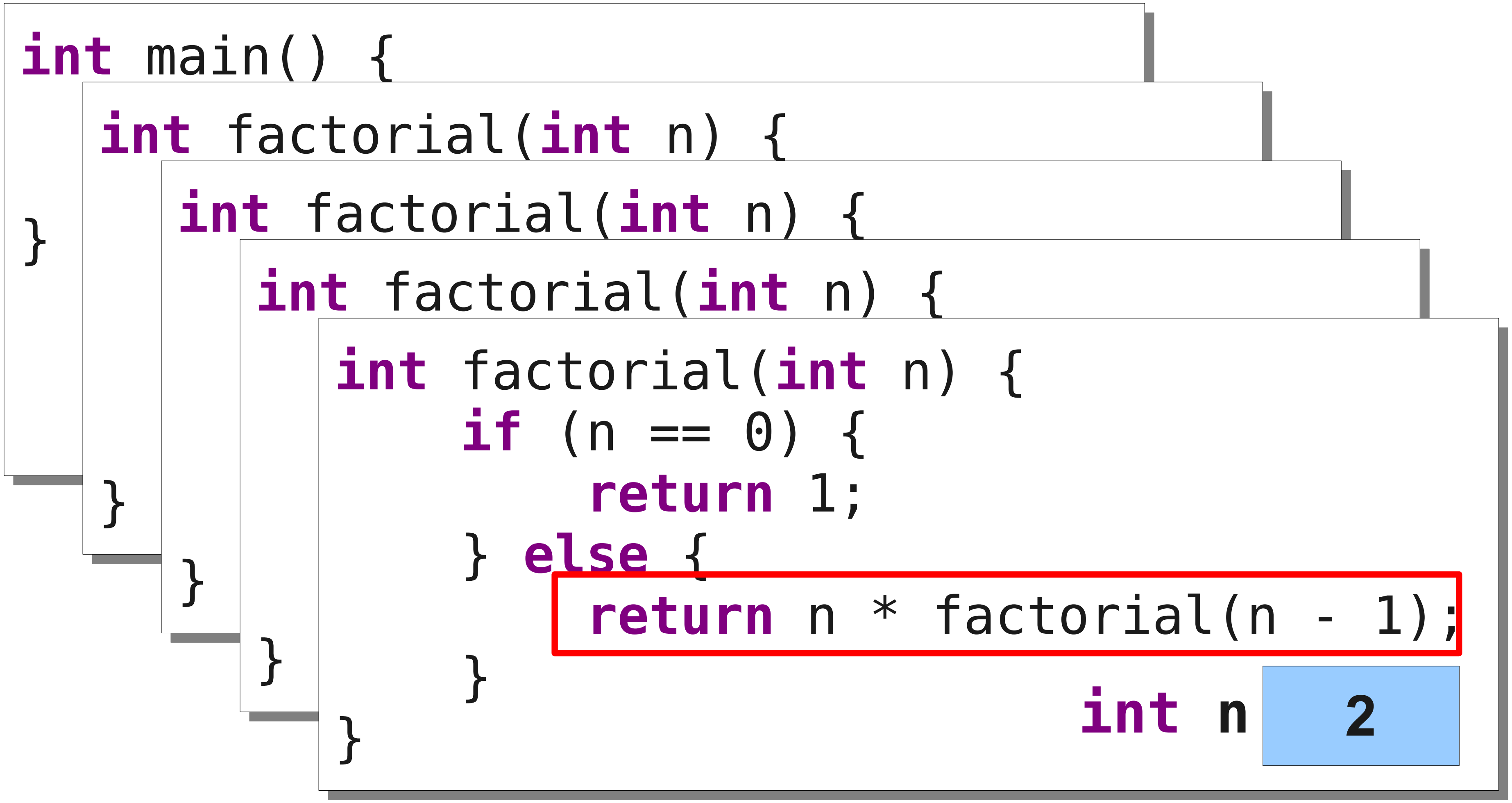
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

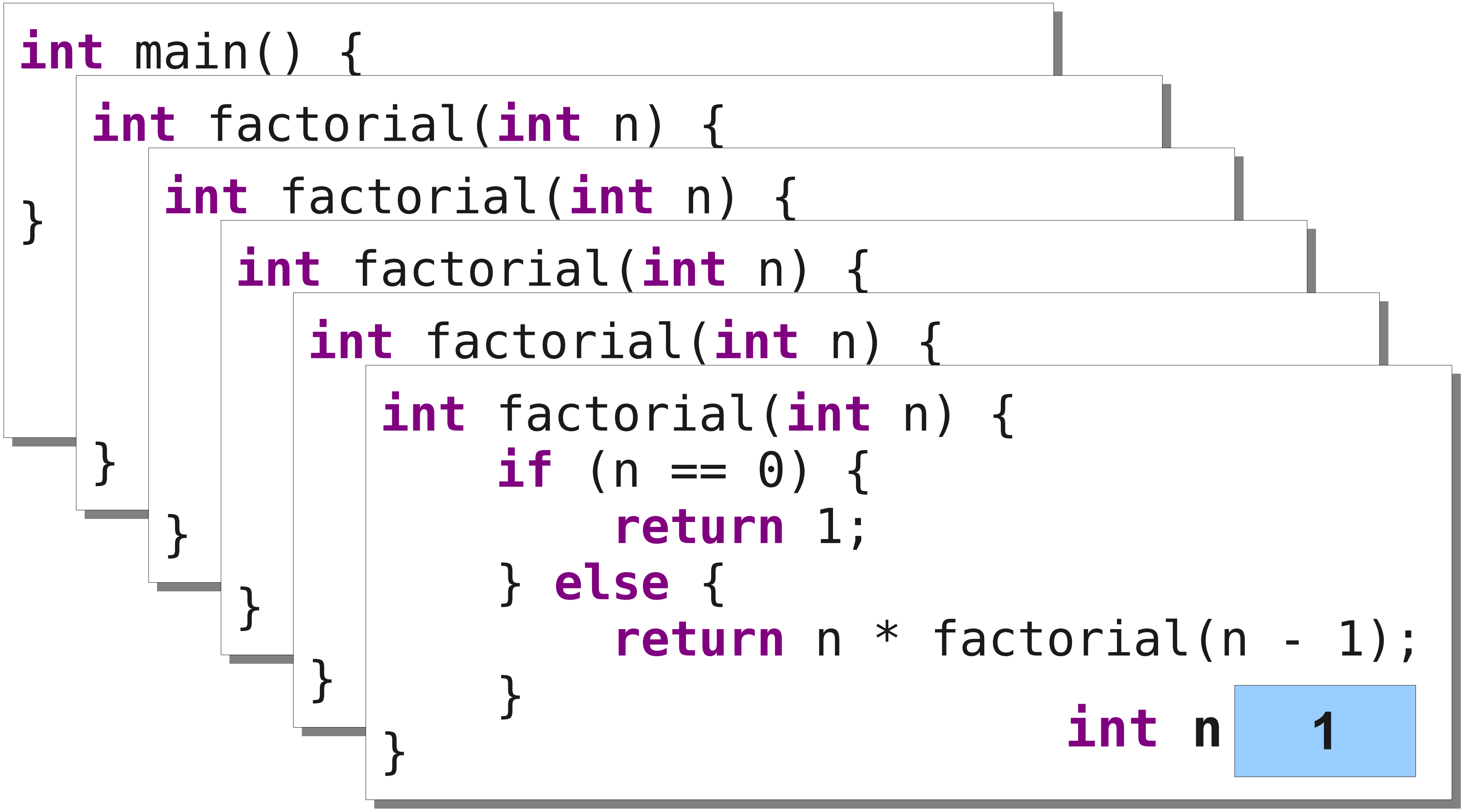
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

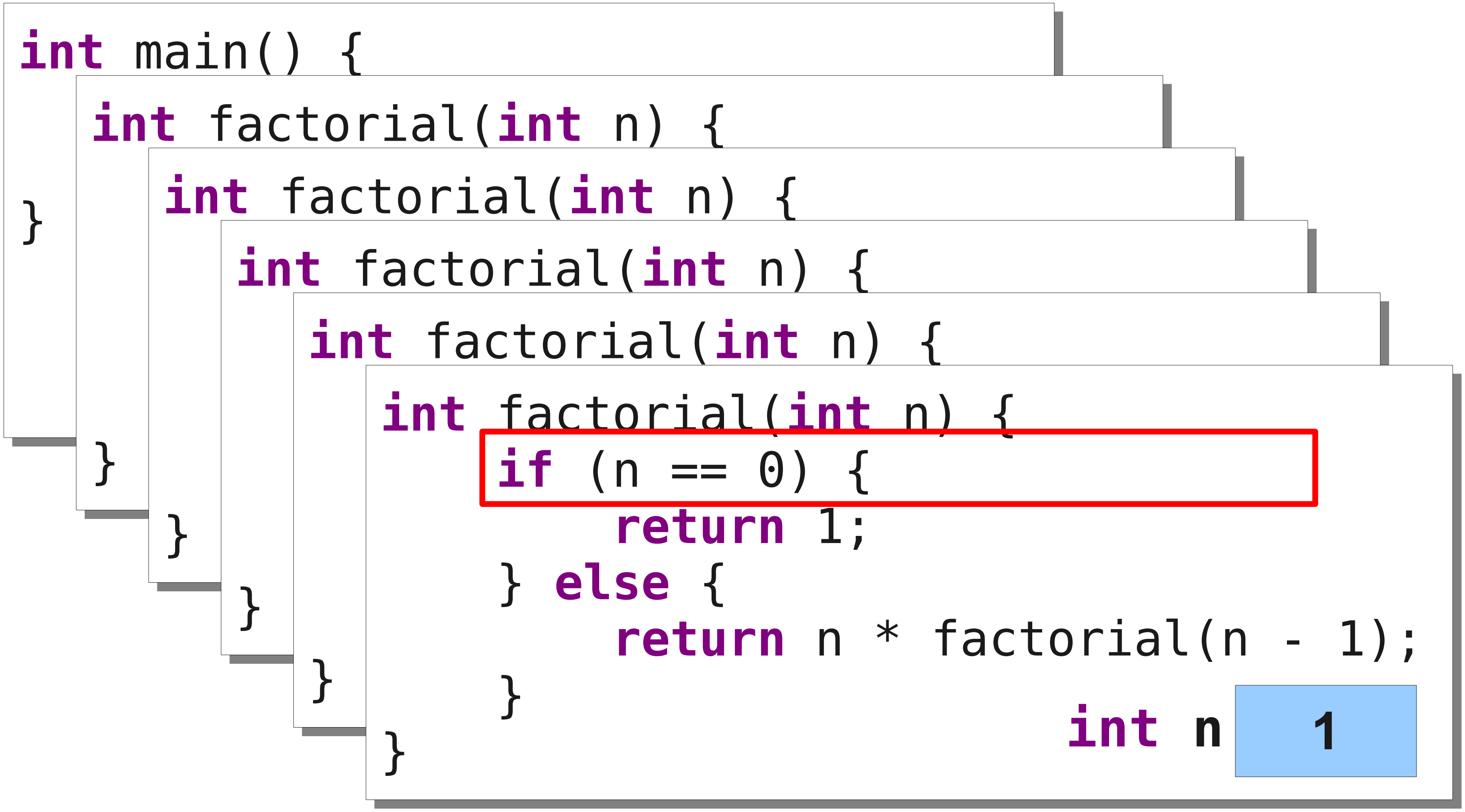
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

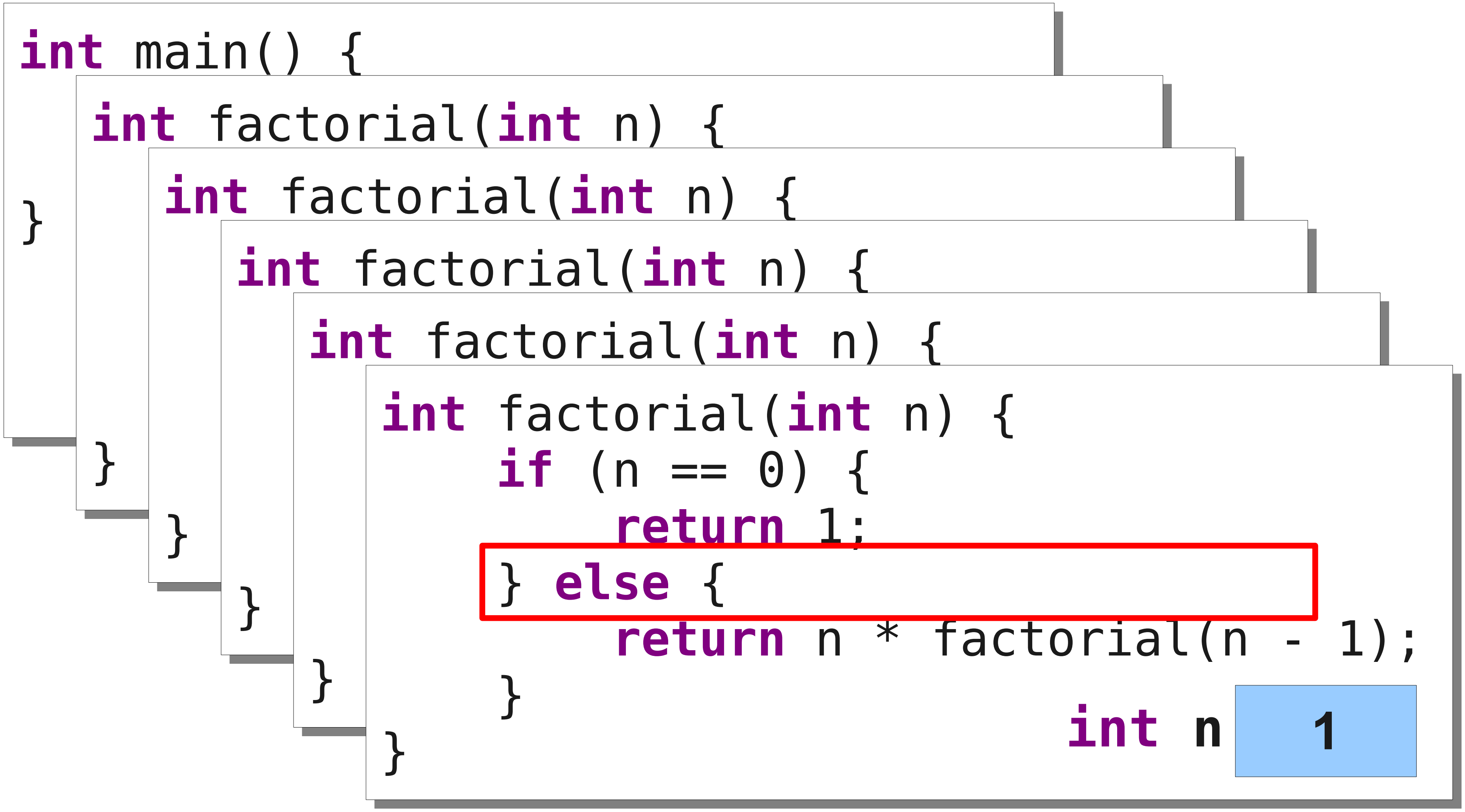
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

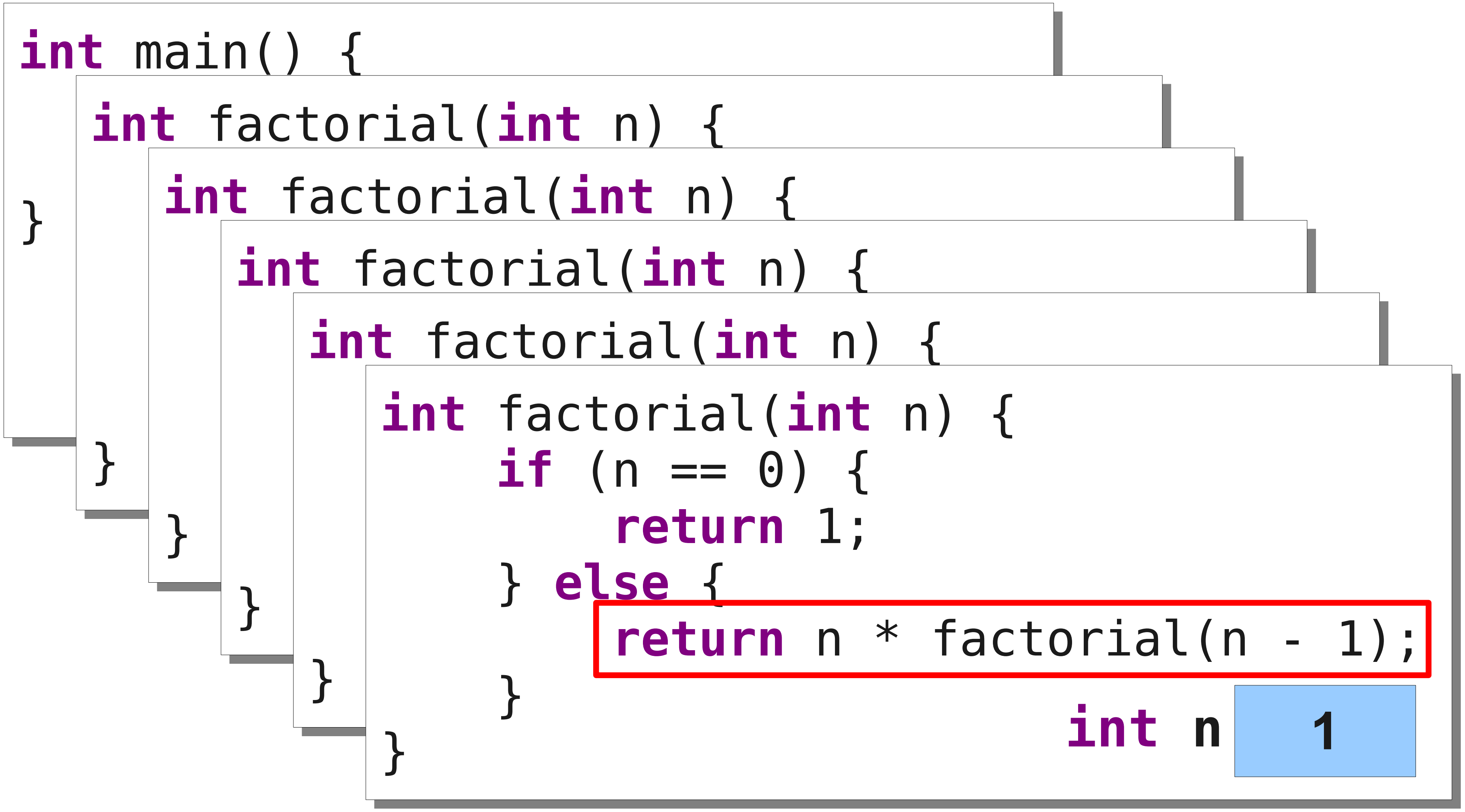
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

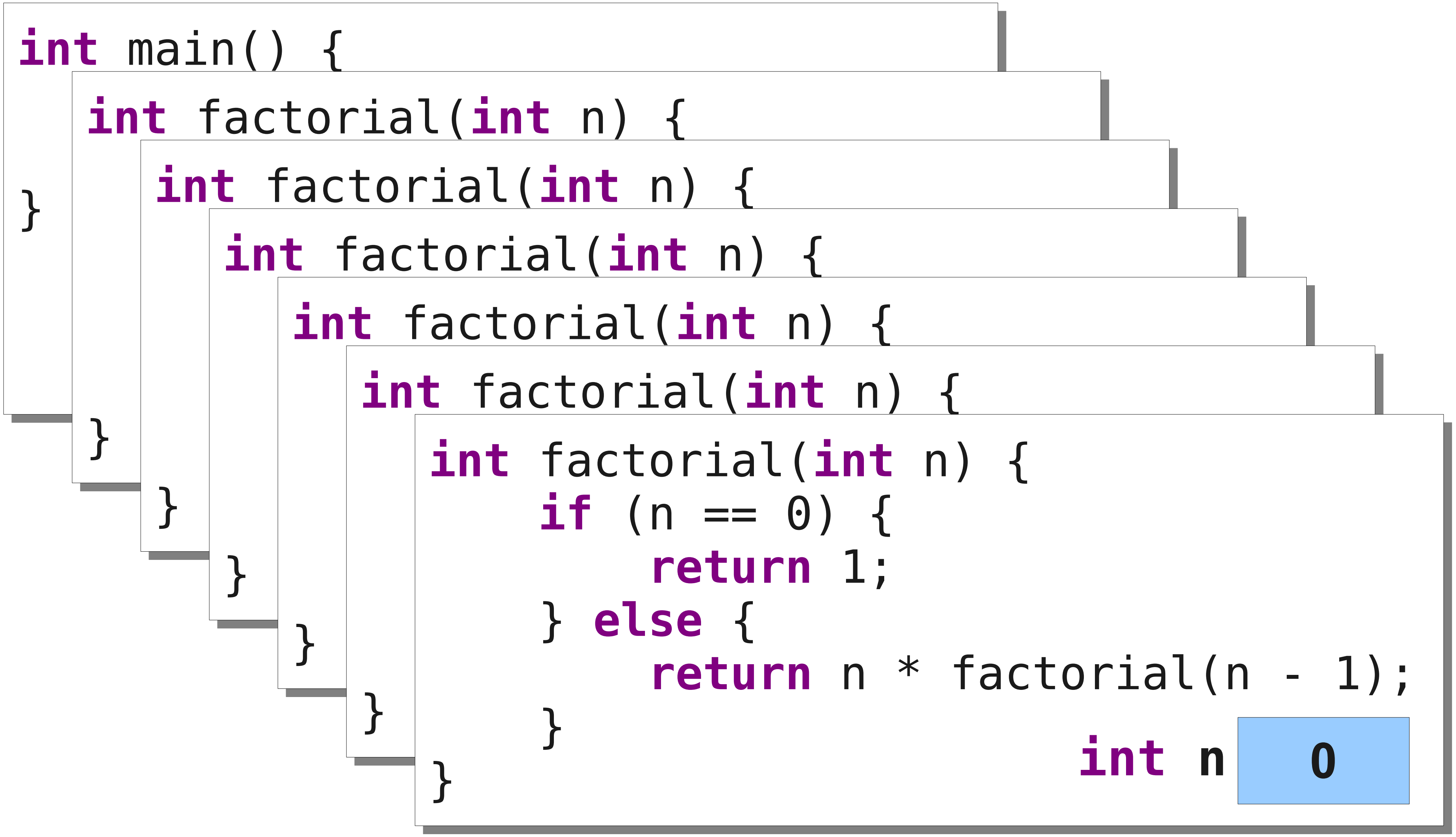
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

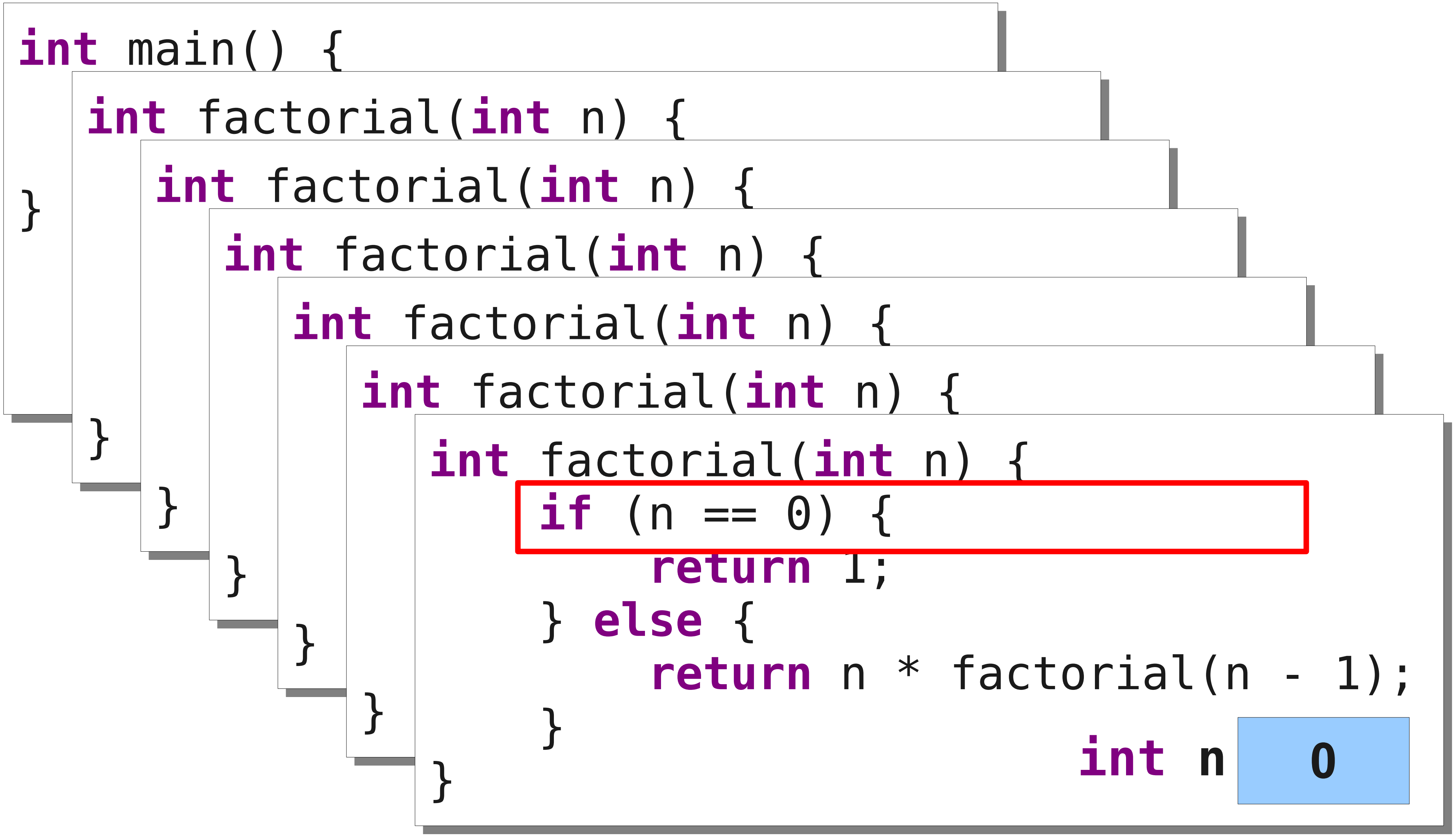
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

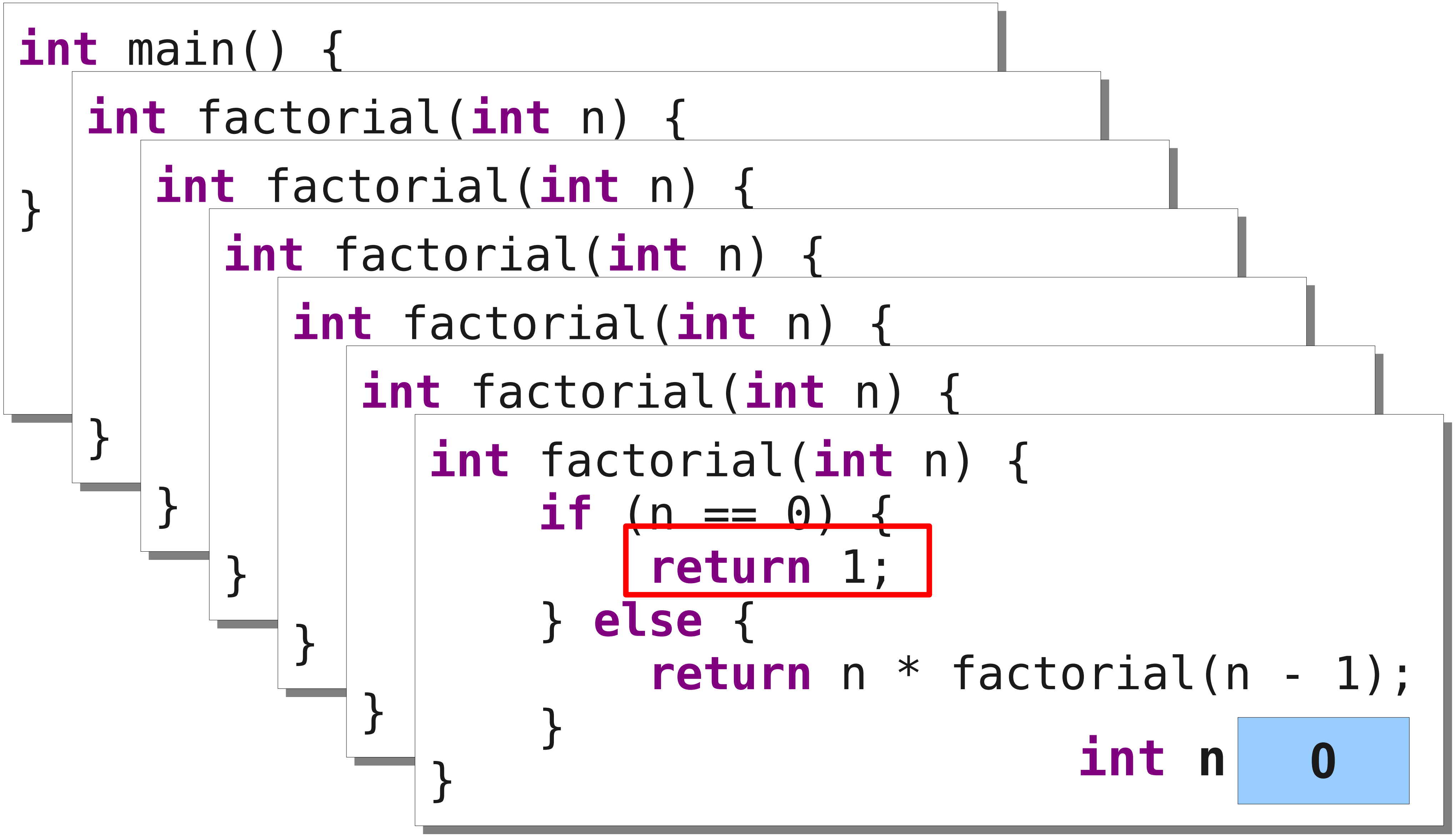
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
                                  int n  0
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

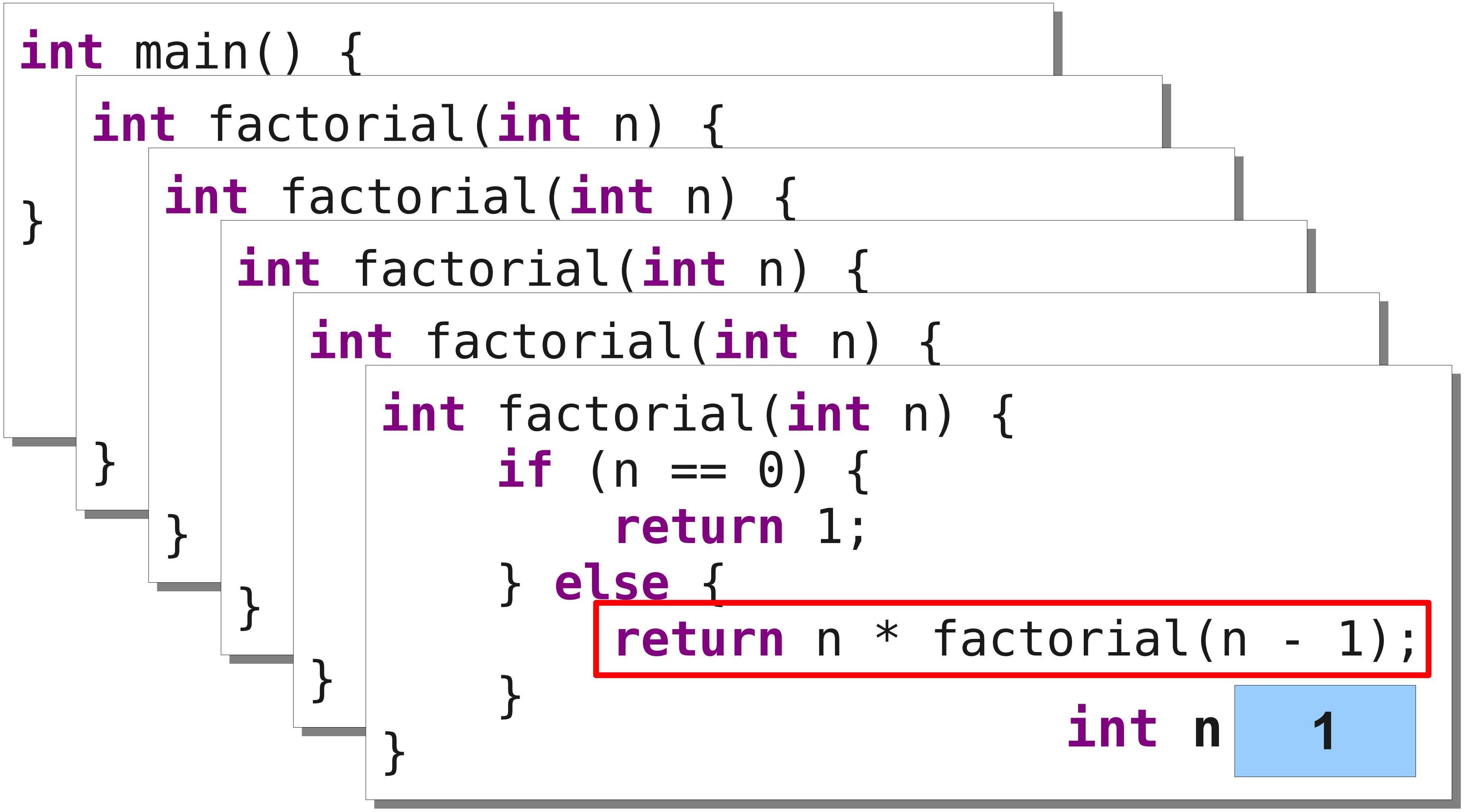
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

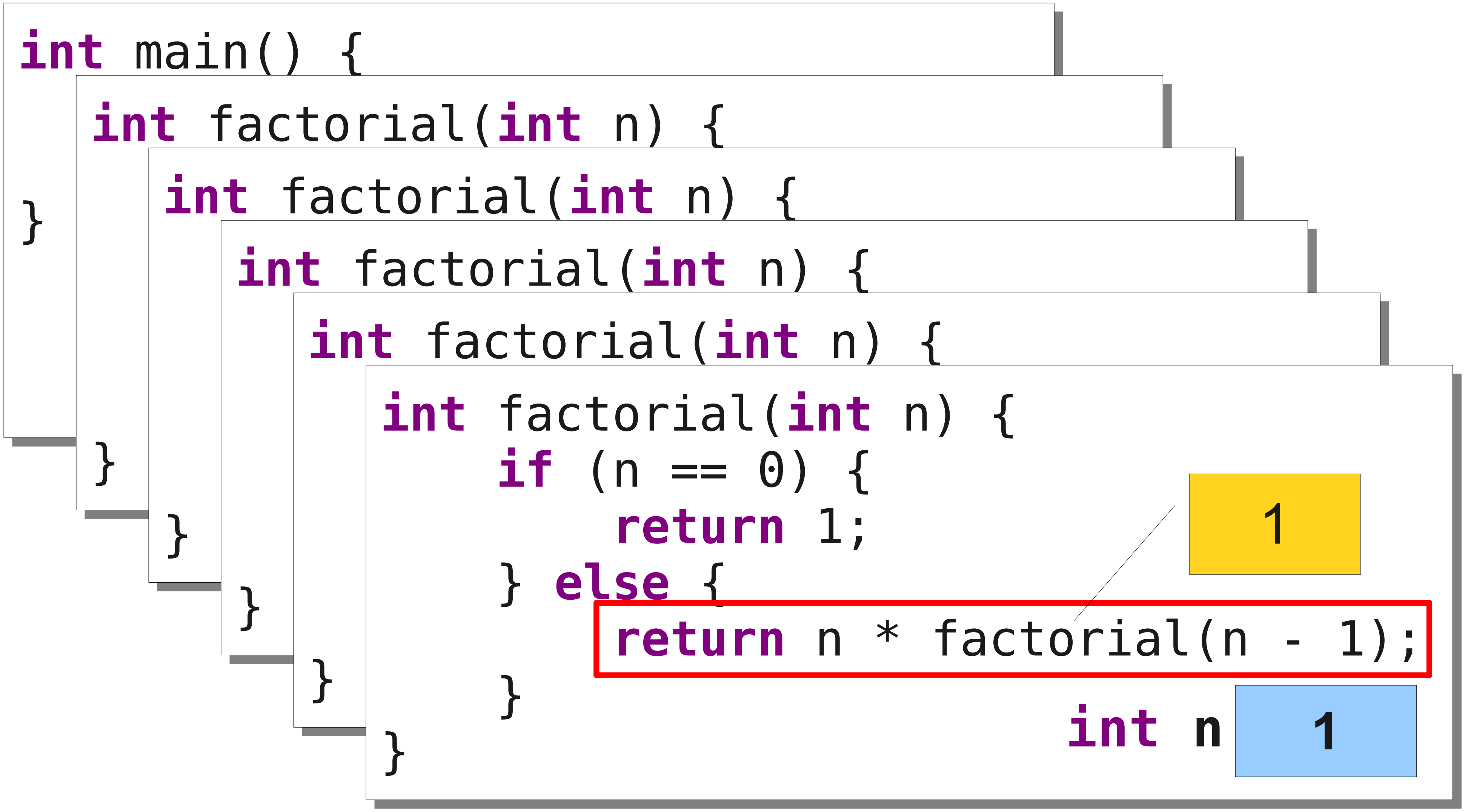
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

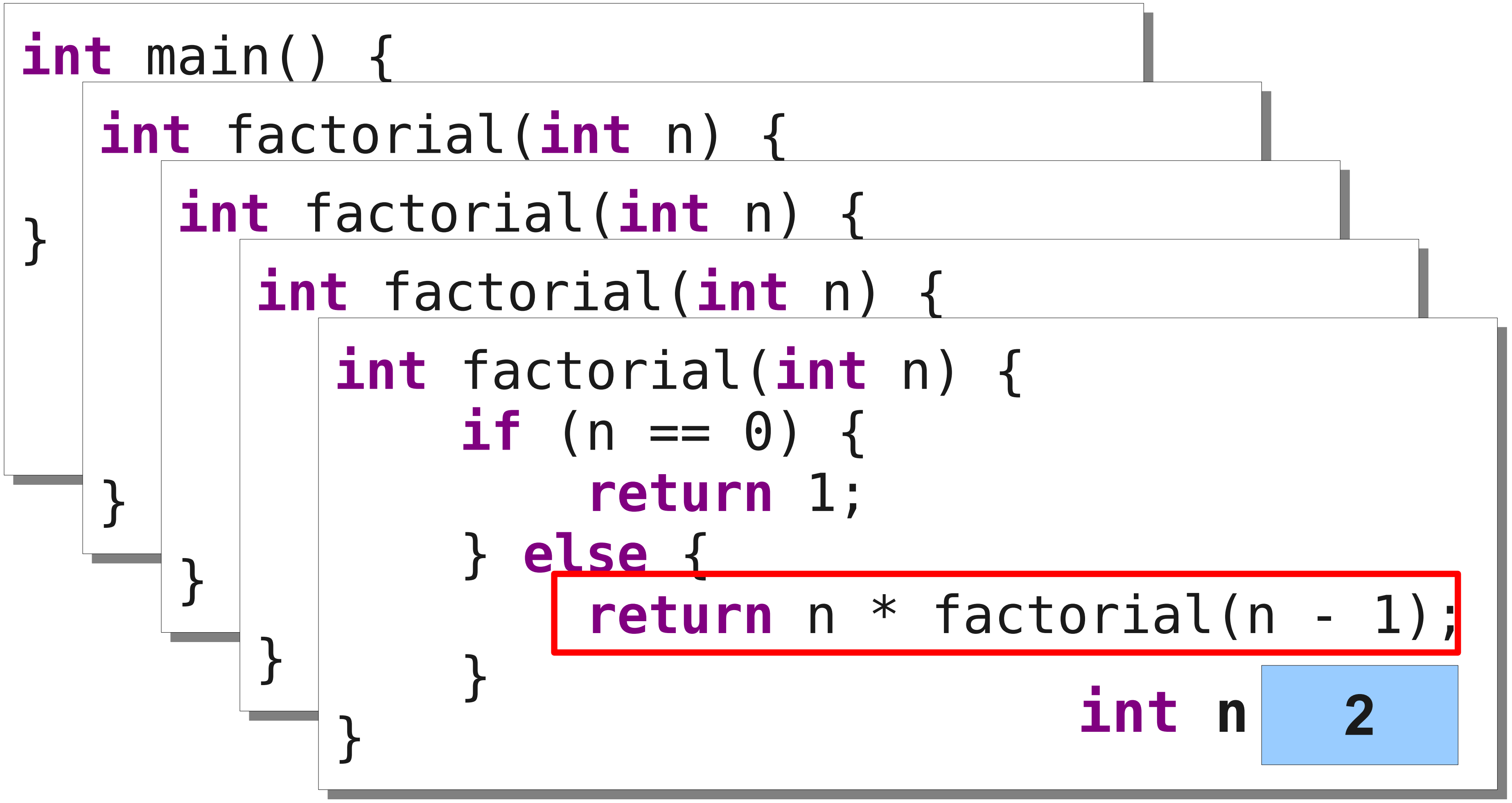
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

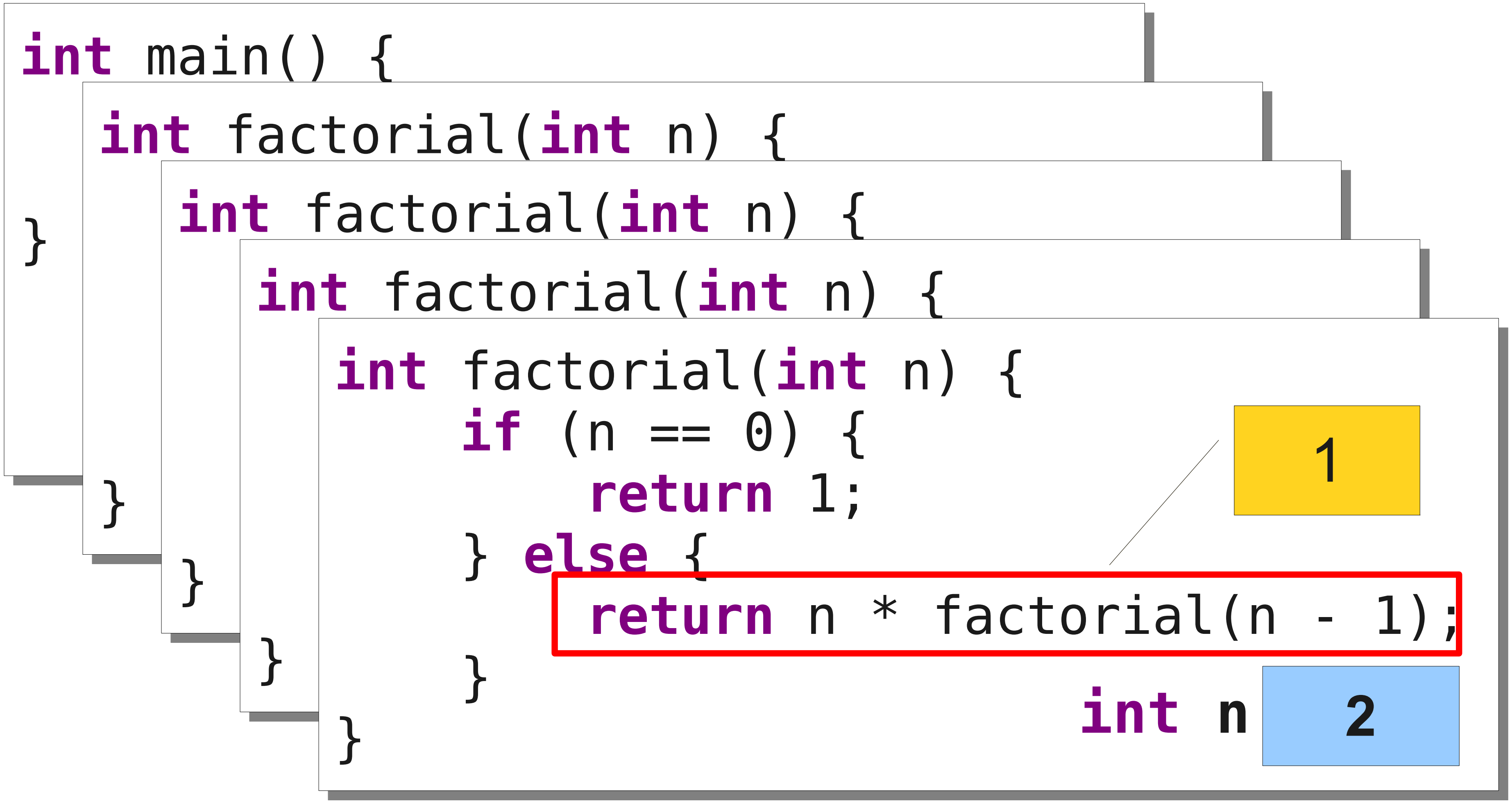
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

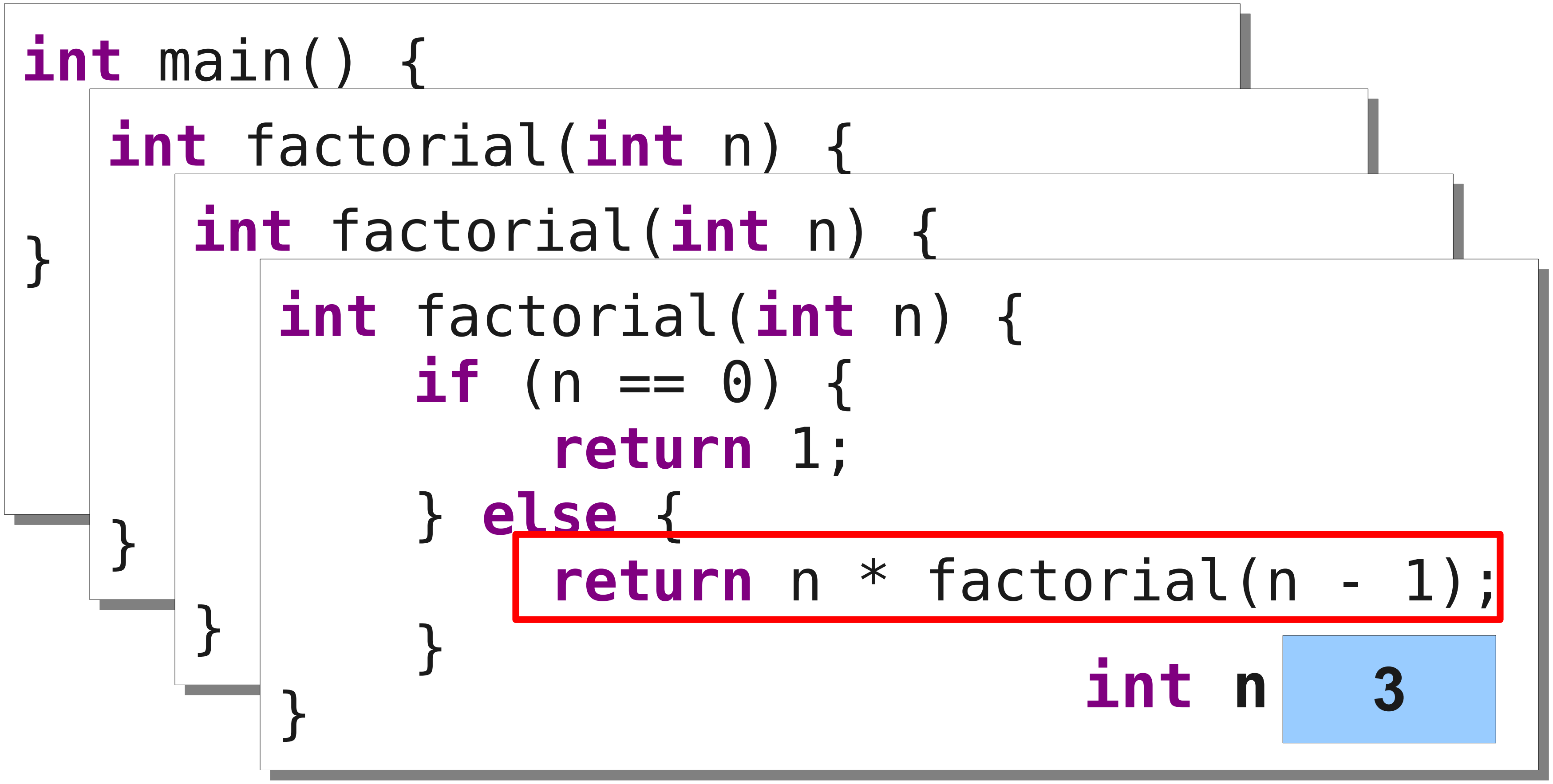
```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



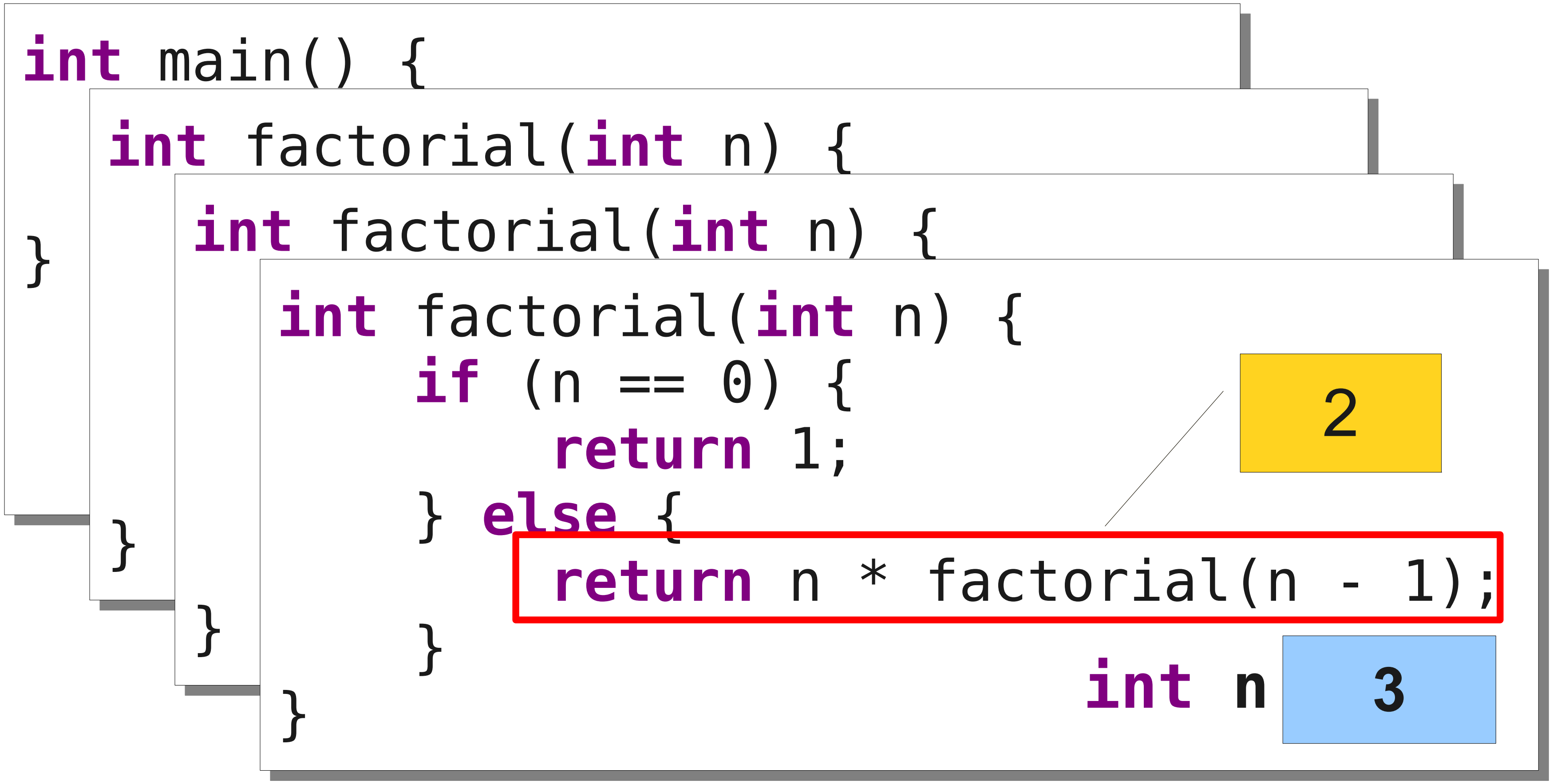
```
int main() {
}
```

```
int factorial(int n) {
}
```

```
int factorial(int n) {
}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action

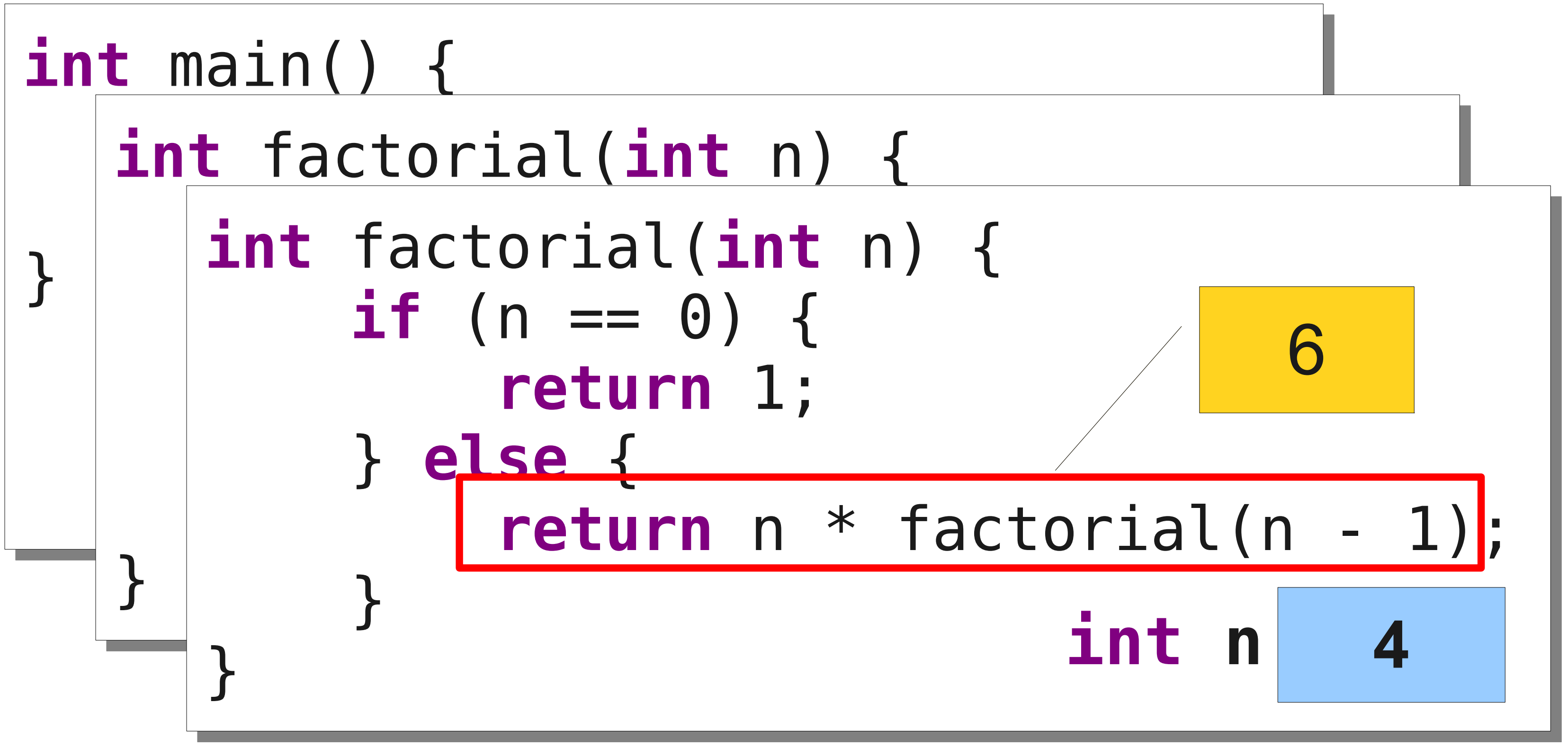
```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

int n 4

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

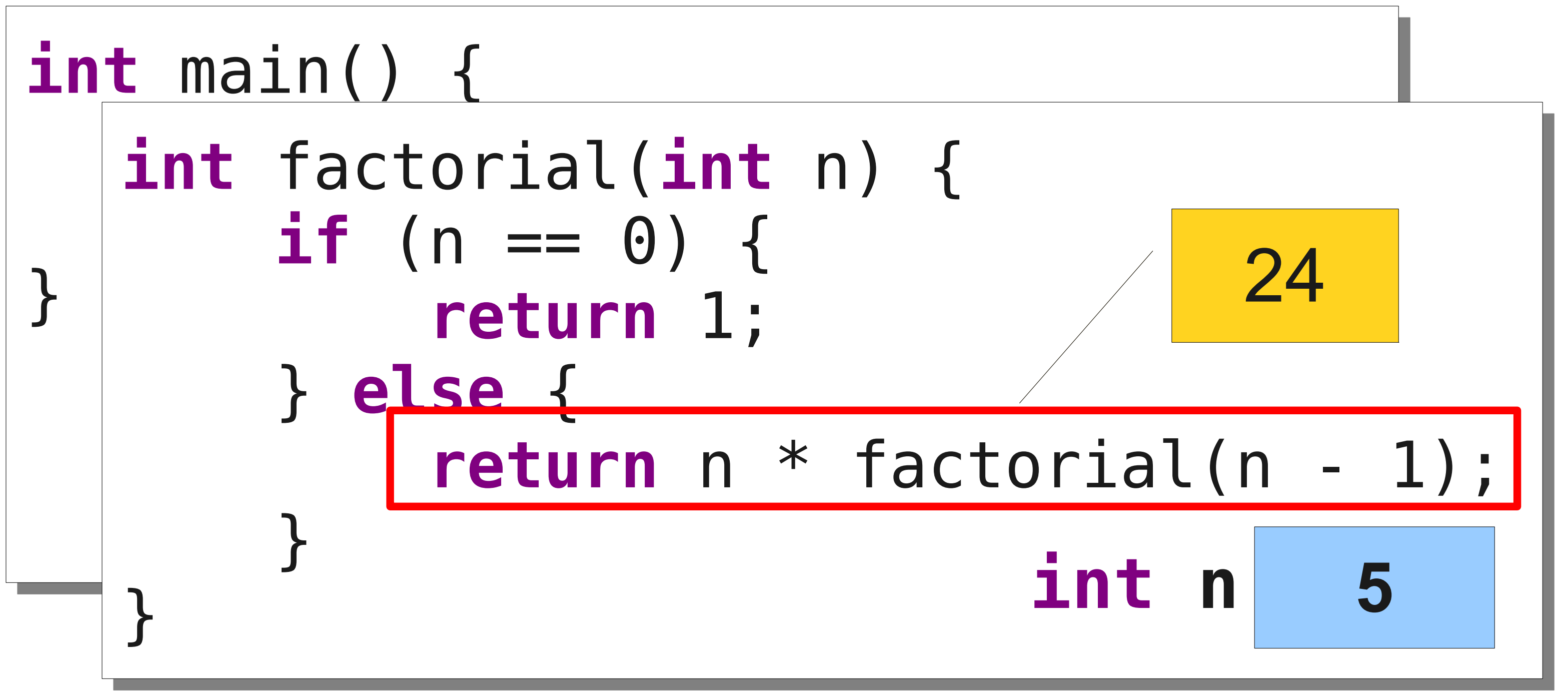
Recursion in Action

```
int main() {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

int n 5

Recursion in Action



```
int main() {

}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Recursion in Action

```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
}
```

Recursion in Action

```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
}
```

int n 120

Recursion in Action

```
int main() {
    int n = factorial(5);
    cout << "5! = " << n << endl;
}
```

int n **120**

Thinking Recursively

- Solving a problem with recursion requires two steps.
- First, determine how to solve the problem for simple cases.
  - This is called the **base case**.
- Second, determine how to break down larger cases into smaller instances.
  - This is called the **recursive decomposition**.

Thinking Recursively

```
if (problem is sufficiently simple) {
    Directly solve the problem.
    Return the solution.
} else {
    Split the problem up into one or more smaller
        problems with the same structure as the original.
    Solve each of those smaller problems.
    Combine the results to get the overall solution.
    Return the overall solution.
}
```

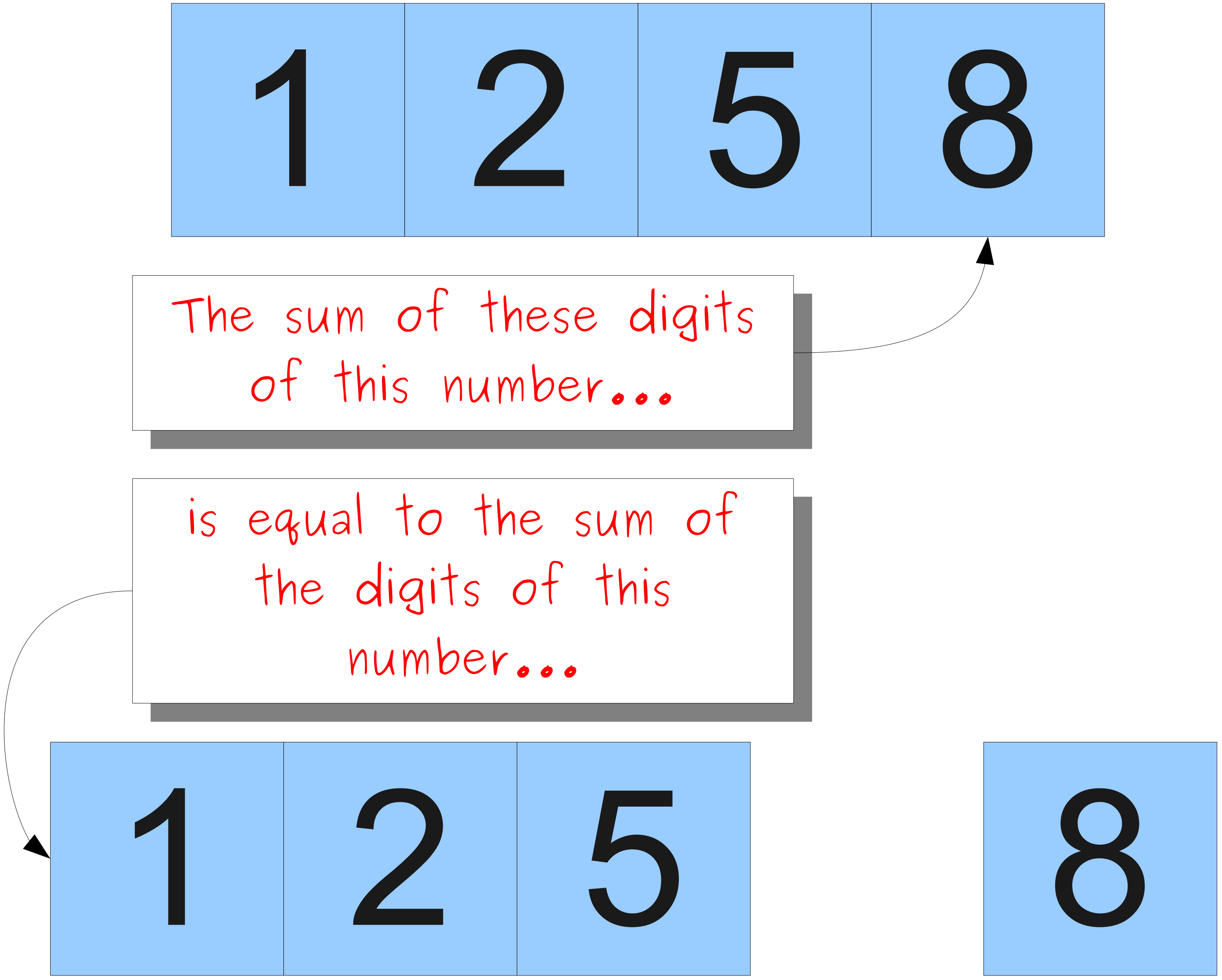
Summing Up Digits

- One way to compute the sum of the digits of a number is shown here:

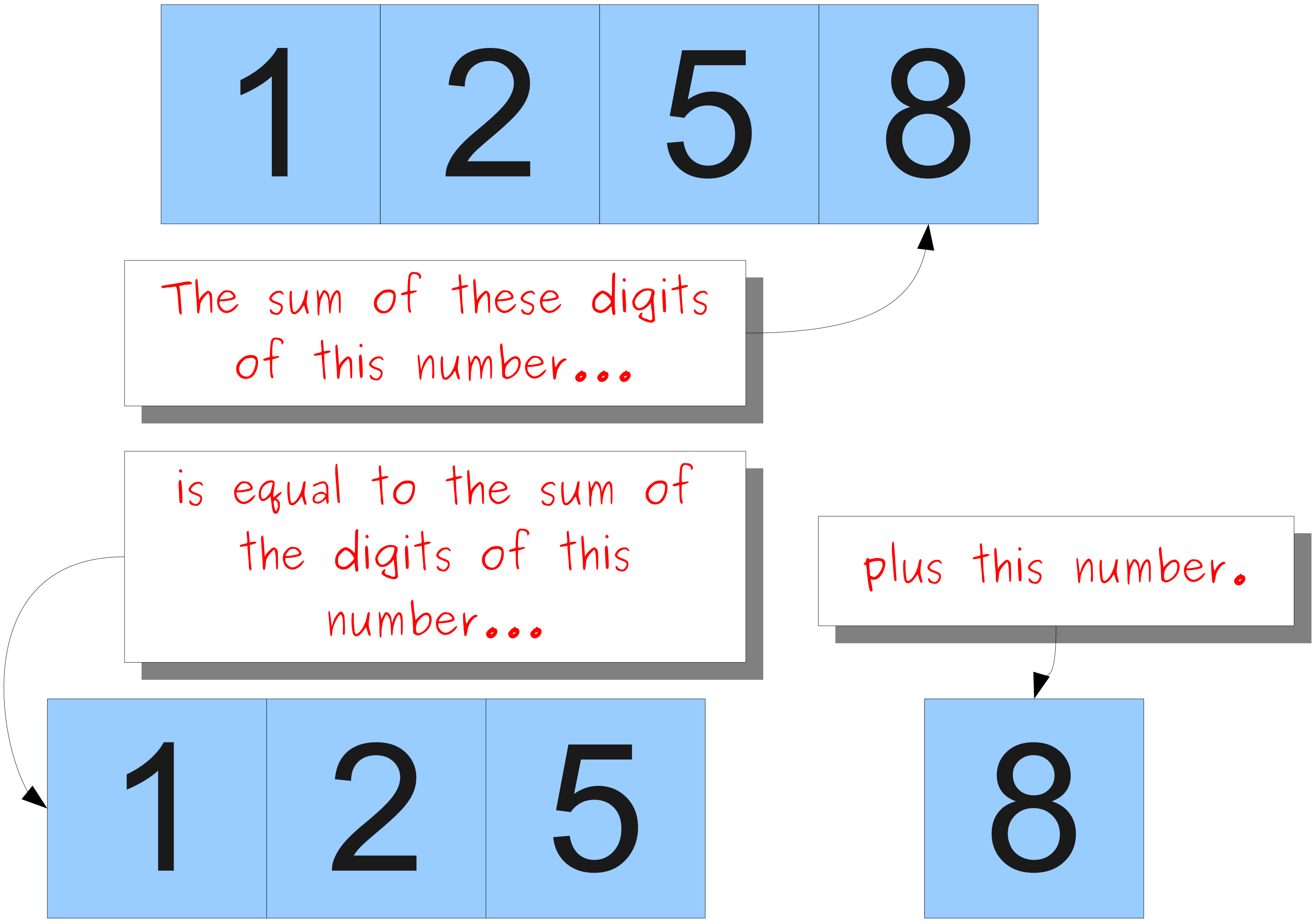
```
int sumOfDigits(int n) {
    int result = 0;
    while (n != 0) {
        result += n % 10;
        n /= 10;
    }
    return result;
}
```

- How would we rewrite this function recursively?

Summing Up Digits


1 2 5 8
The sum of these digits of this number...
is equal to the sum of the digits of this number...
1 2 5
8

Summing Up Digits


1 2 5 8
The sum of these digits of this number...
is equal to the sum of the digits of this number...
plus this number.
1 2 5
8

Summing Up Digits

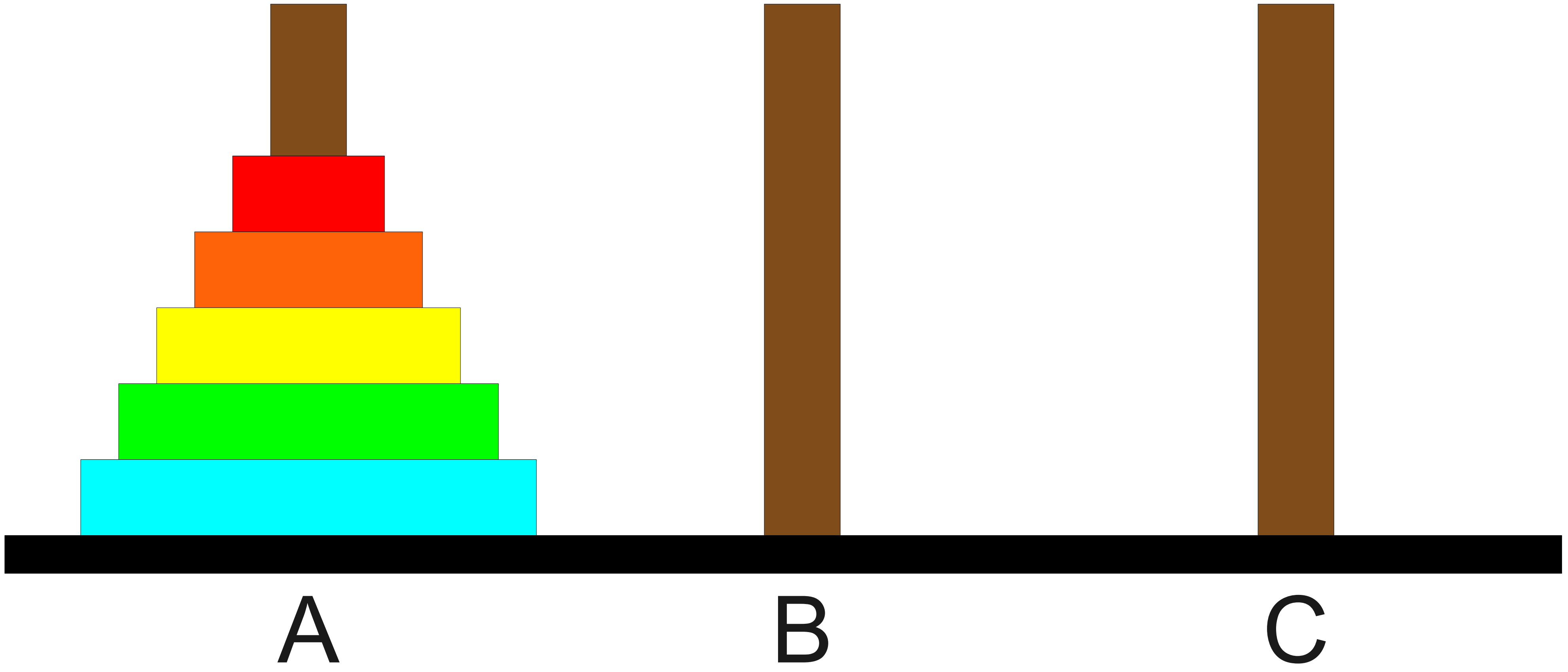
- A recursive implementation of `sumOfDigits` is shown here:

```
int sumOfDigits(int n) {
    if (n < 10) {
        return n;
    } else {
        return (n % 10) + sumOfDigits(n / 10);
    }
}
```

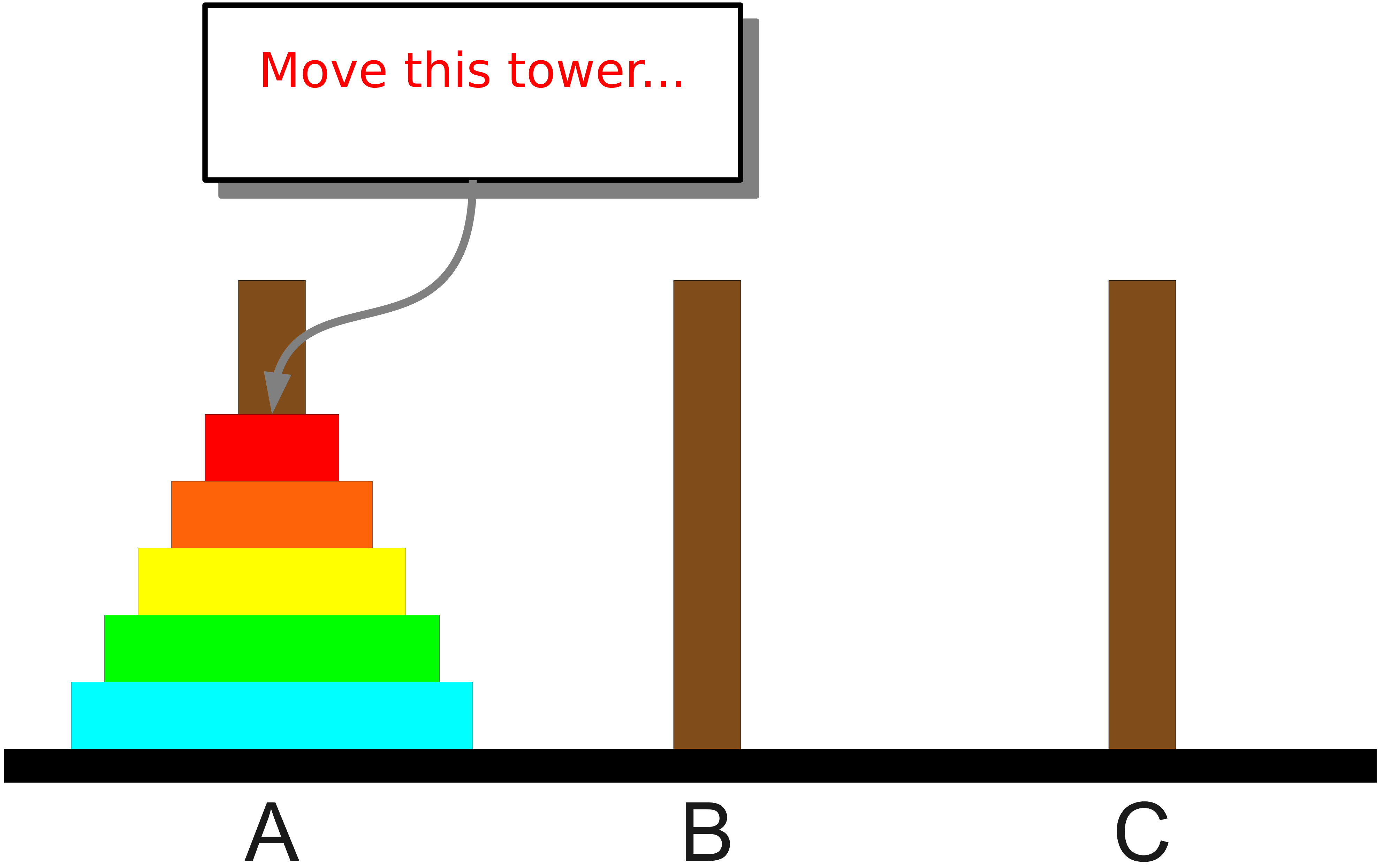
- Notice the structure:
    - If the problem is sufficiently simple, solve it directly.
    - Otherwise, reduce it to a smaller instance and solve that one.

The Towers of Hanoi Problem

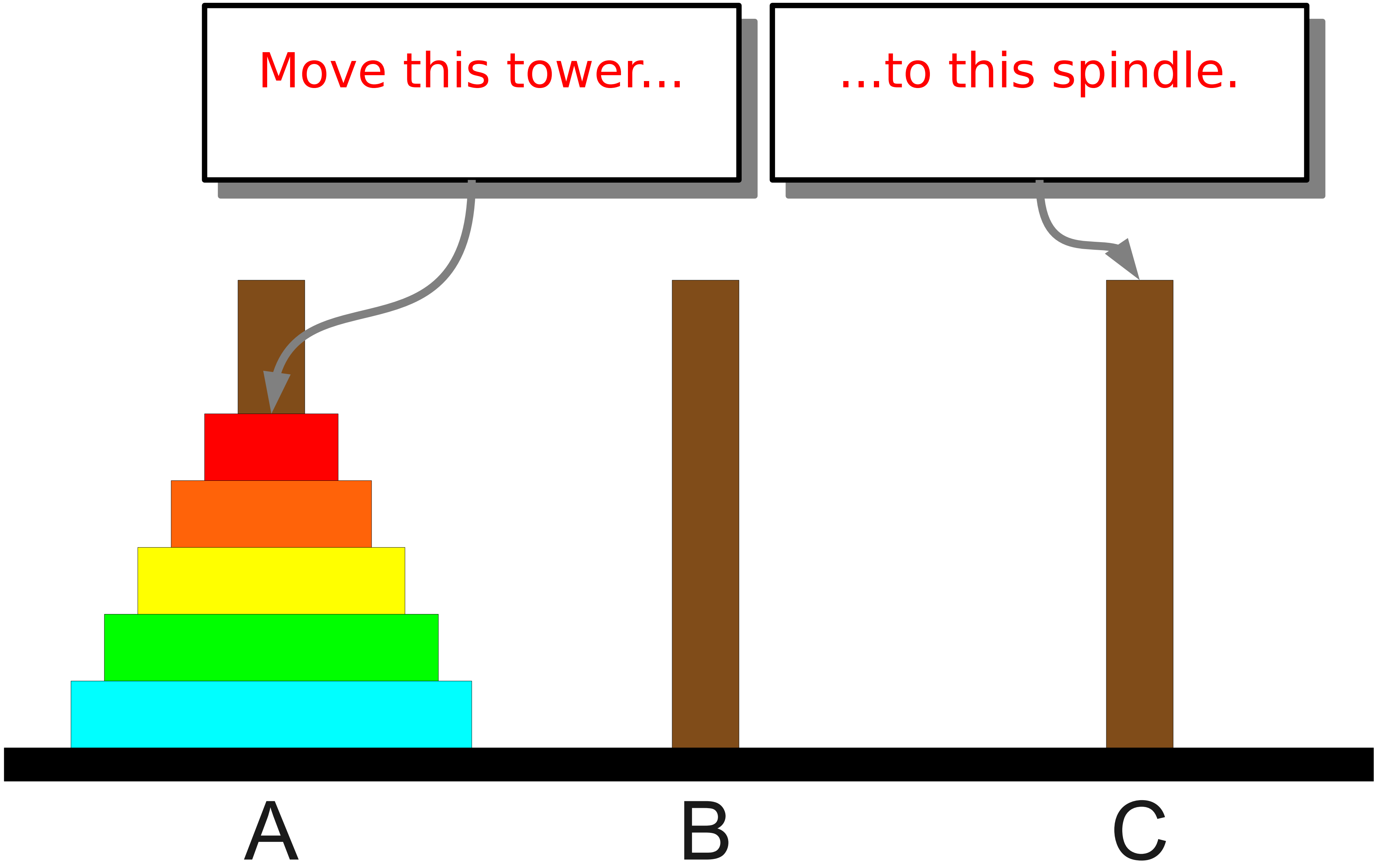
Towers of Hanoi



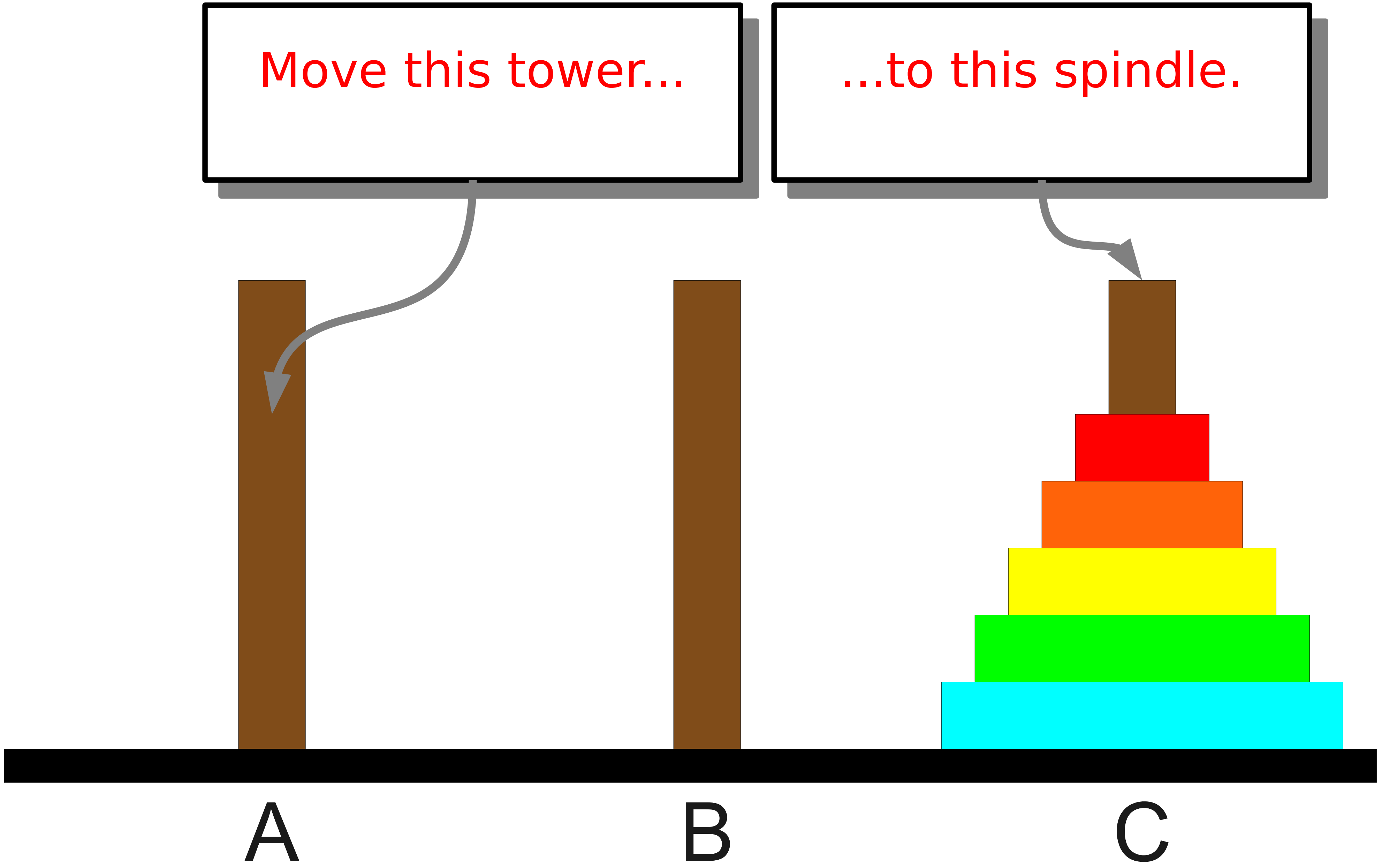
Towers of Hanoi



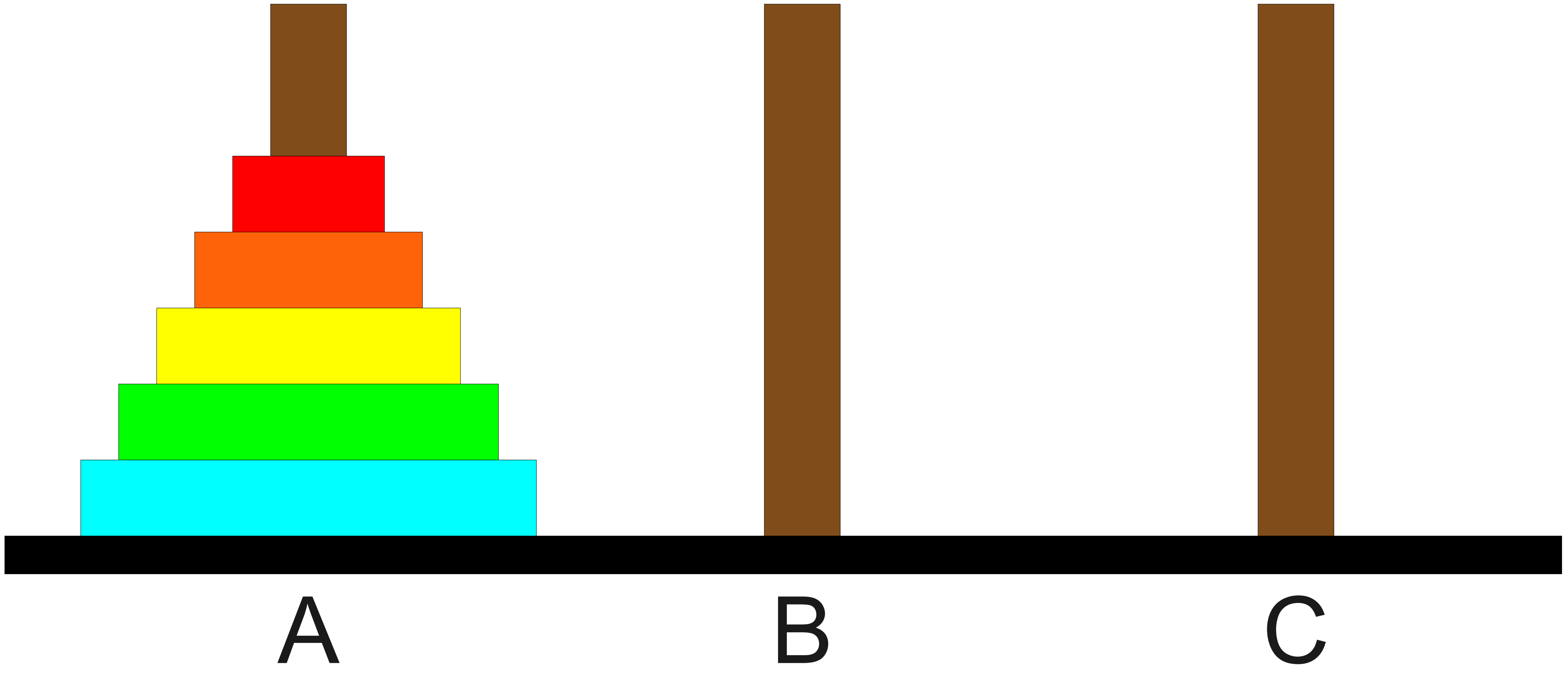
Towers of Hanoi



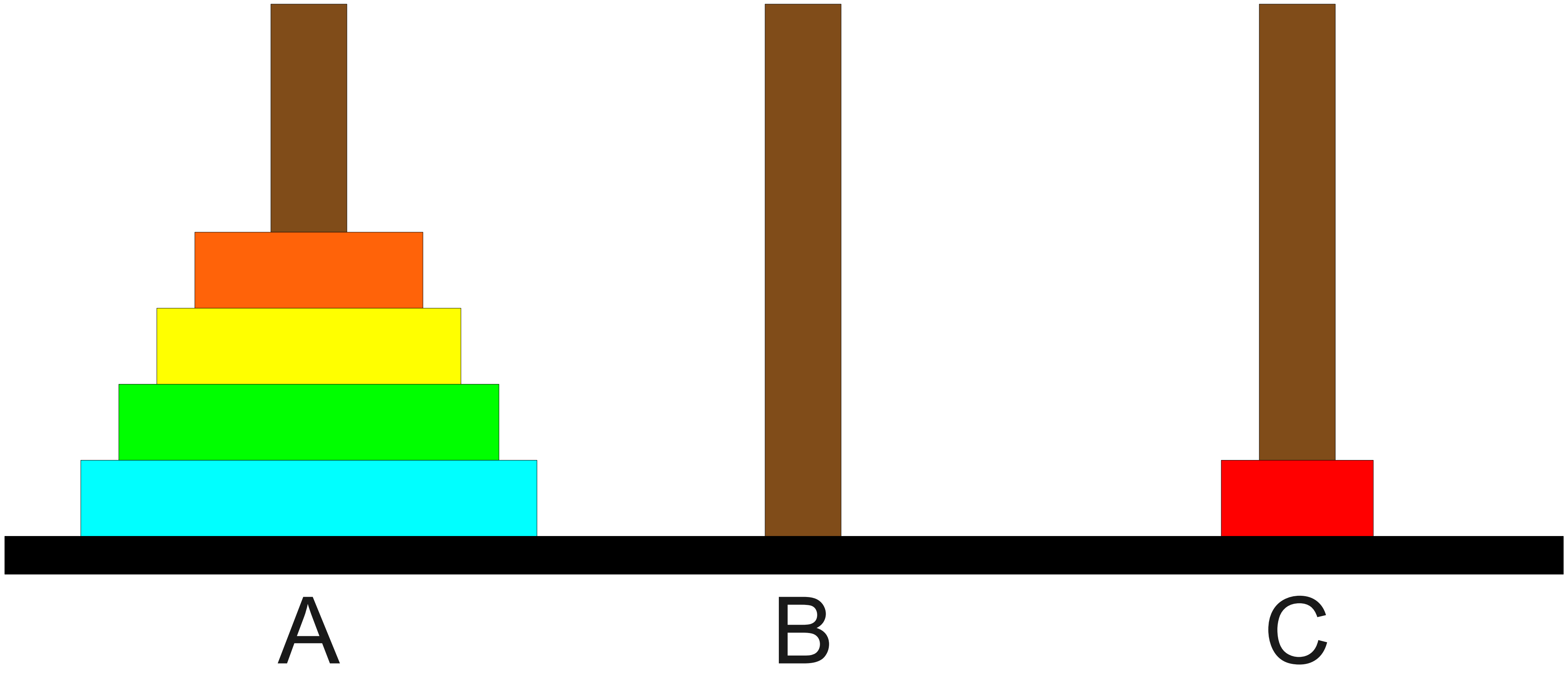
Towers of Hanoi



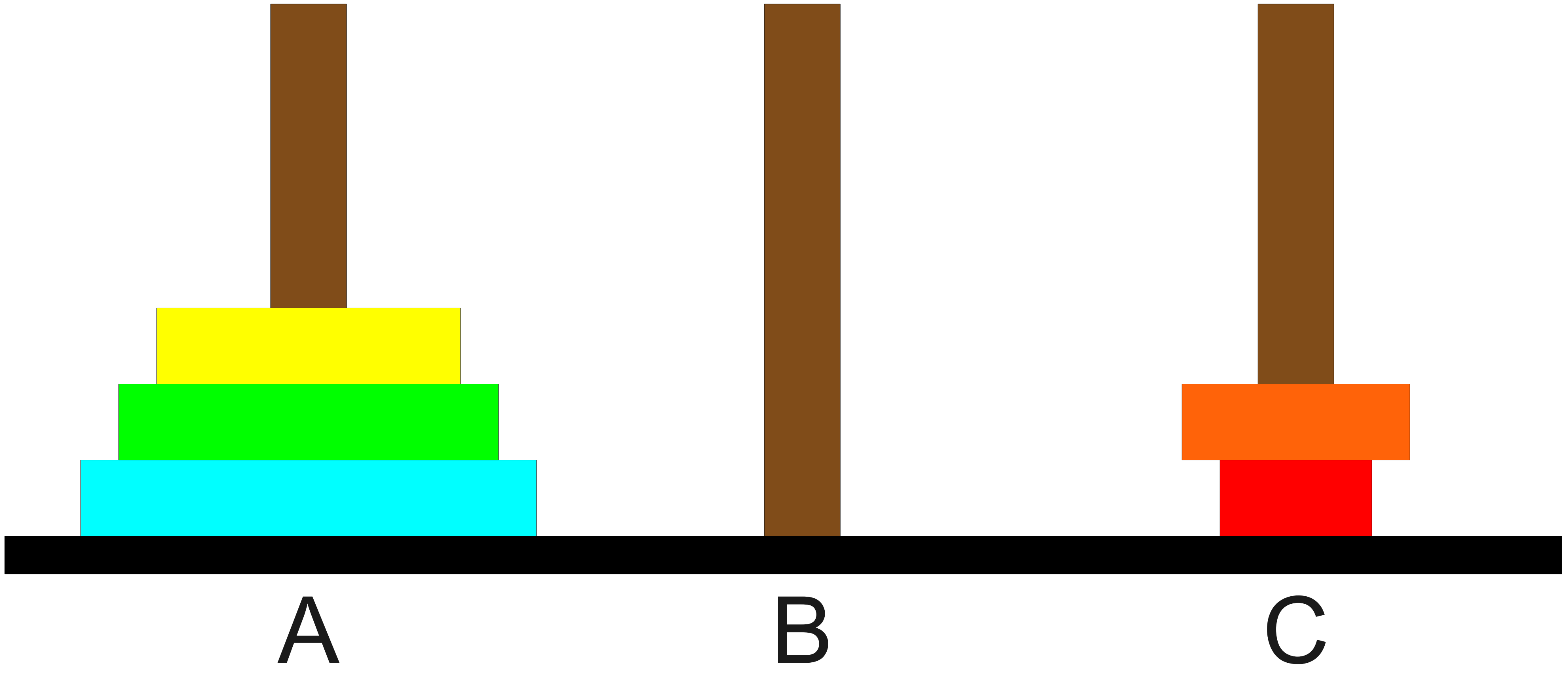
Towers of Hanoi



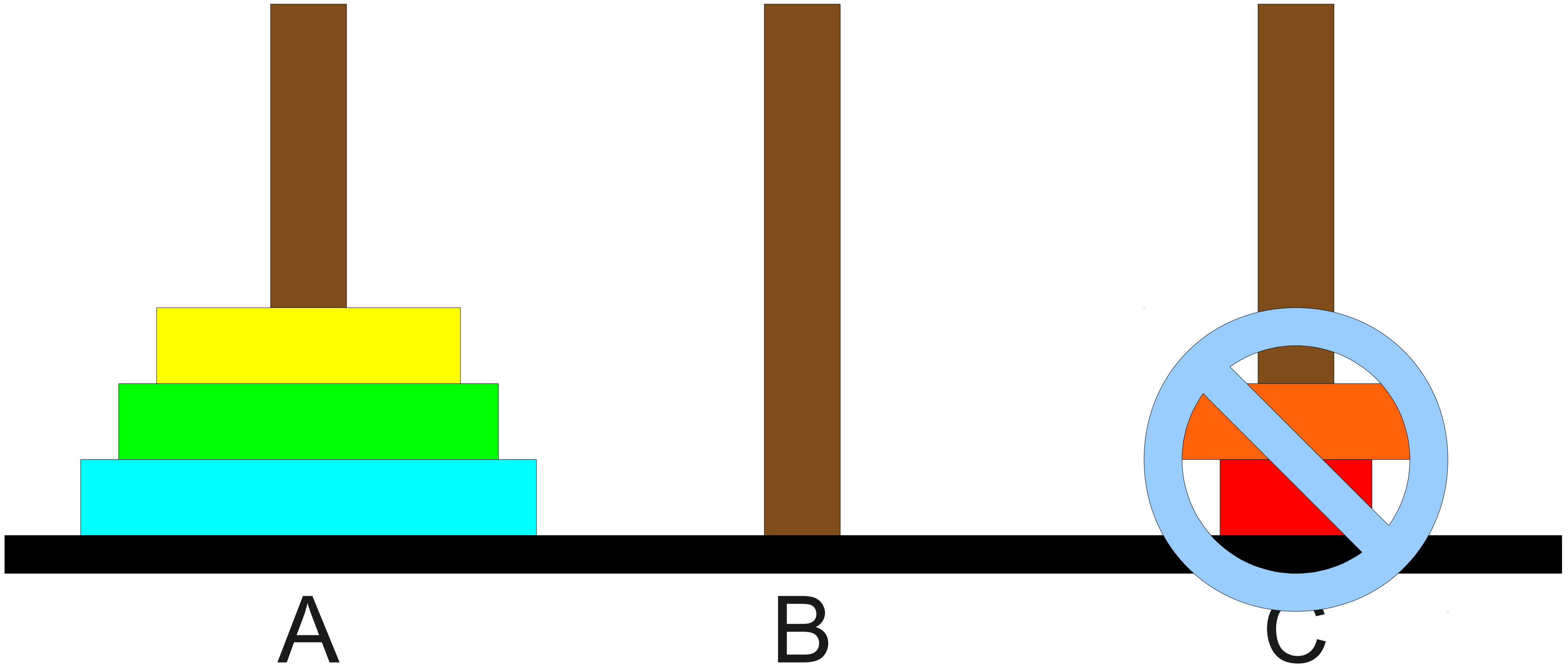
Towers of Hanoi



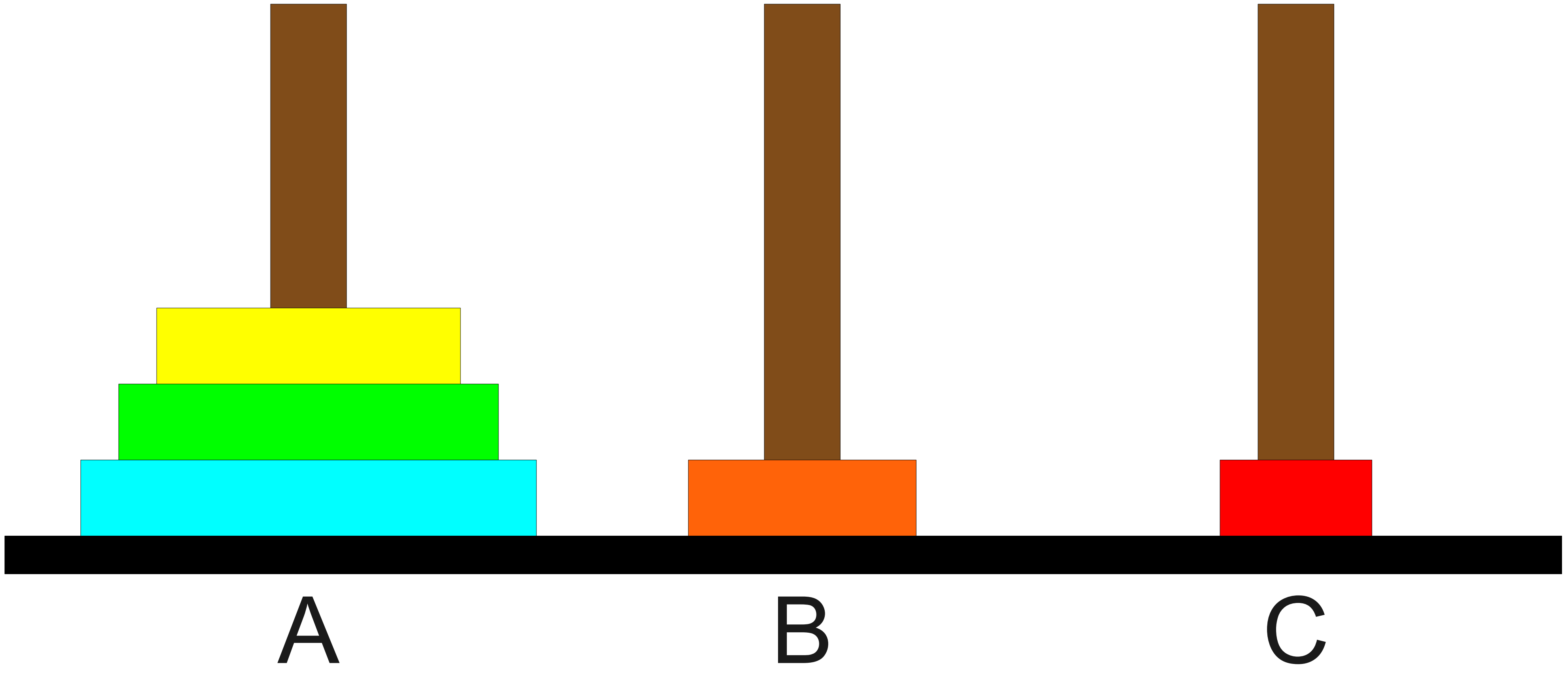
Towers of Hanoi



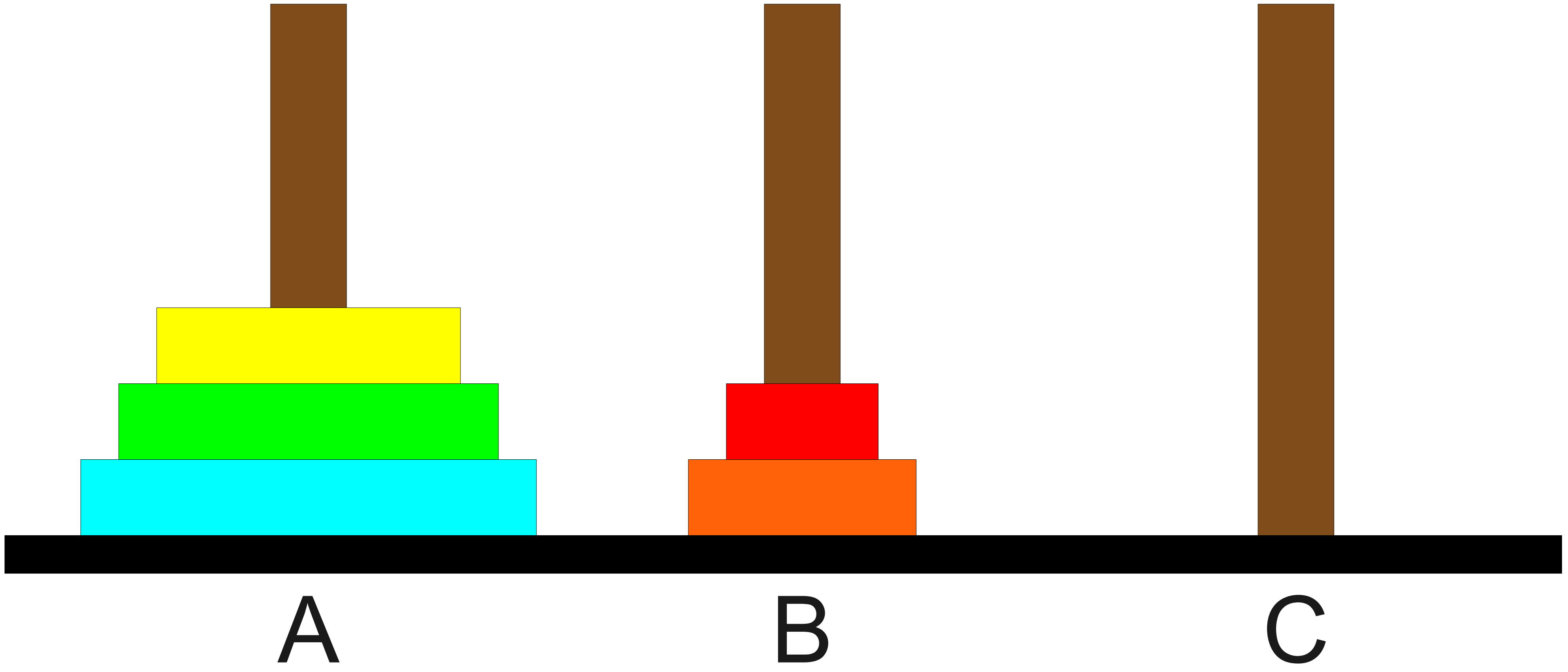
Towers of Hanoi



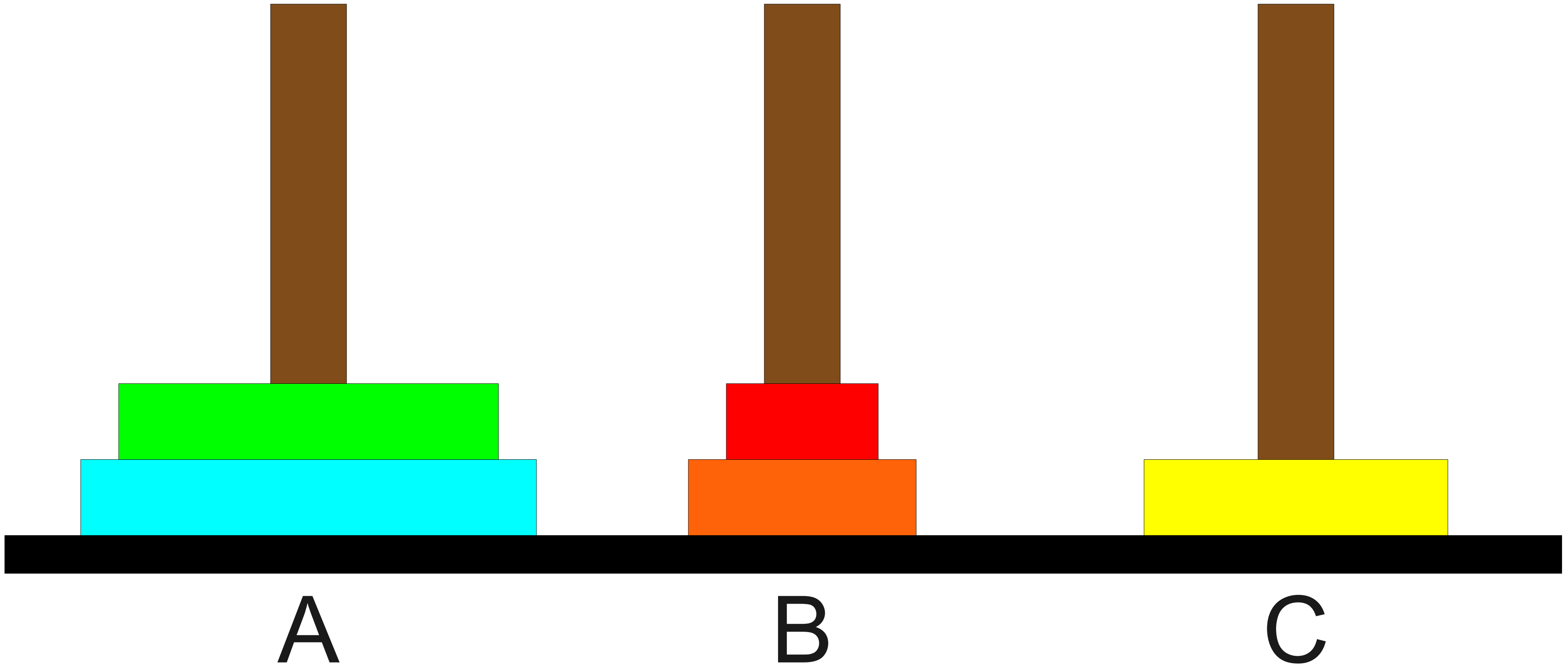
Towers of Hanoi



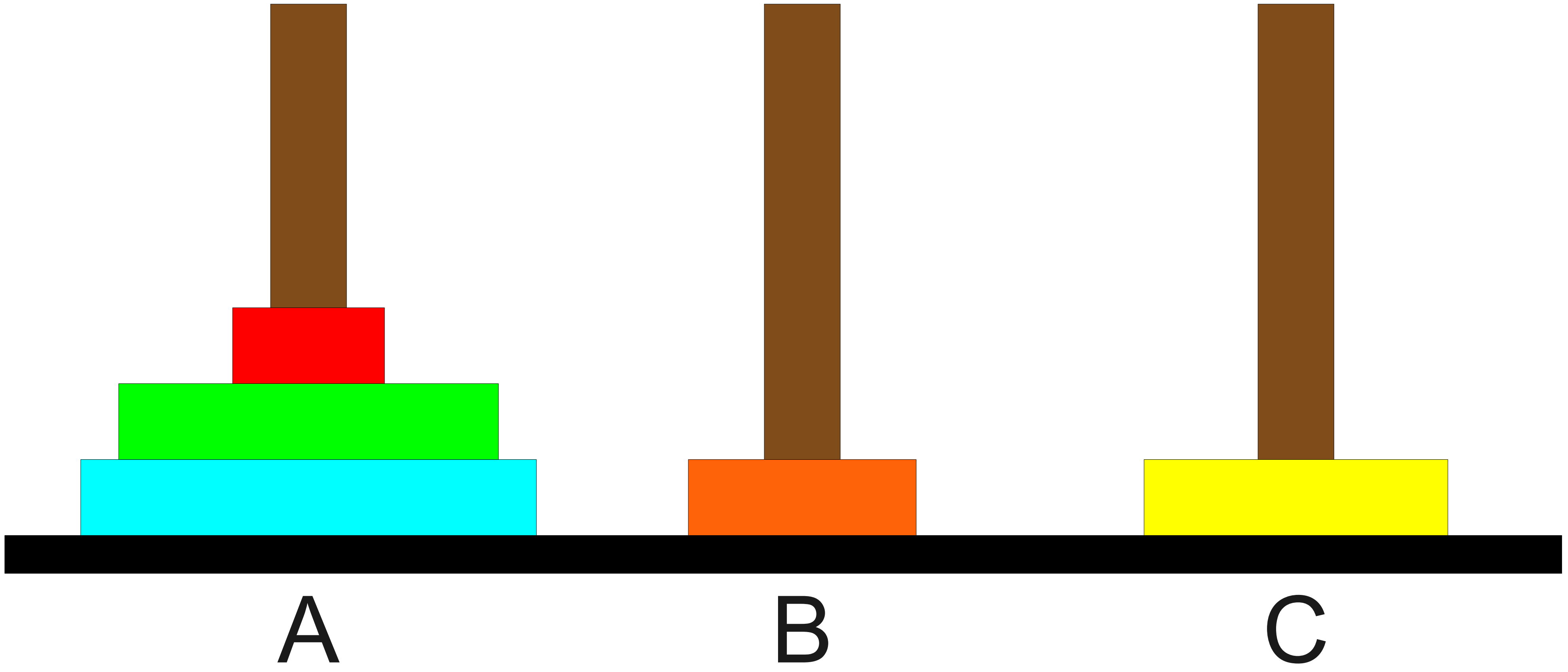
Towers of Hanoi



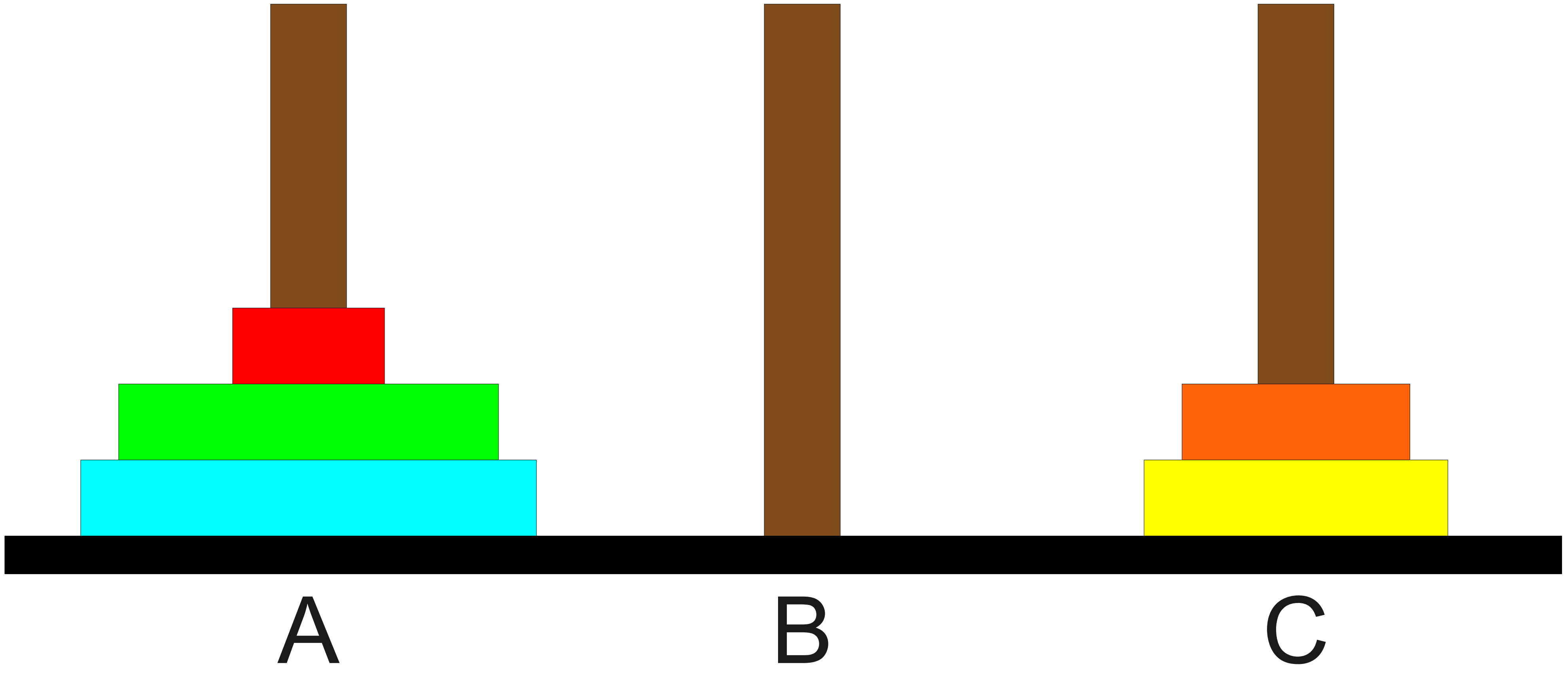
Towers of Hanoi



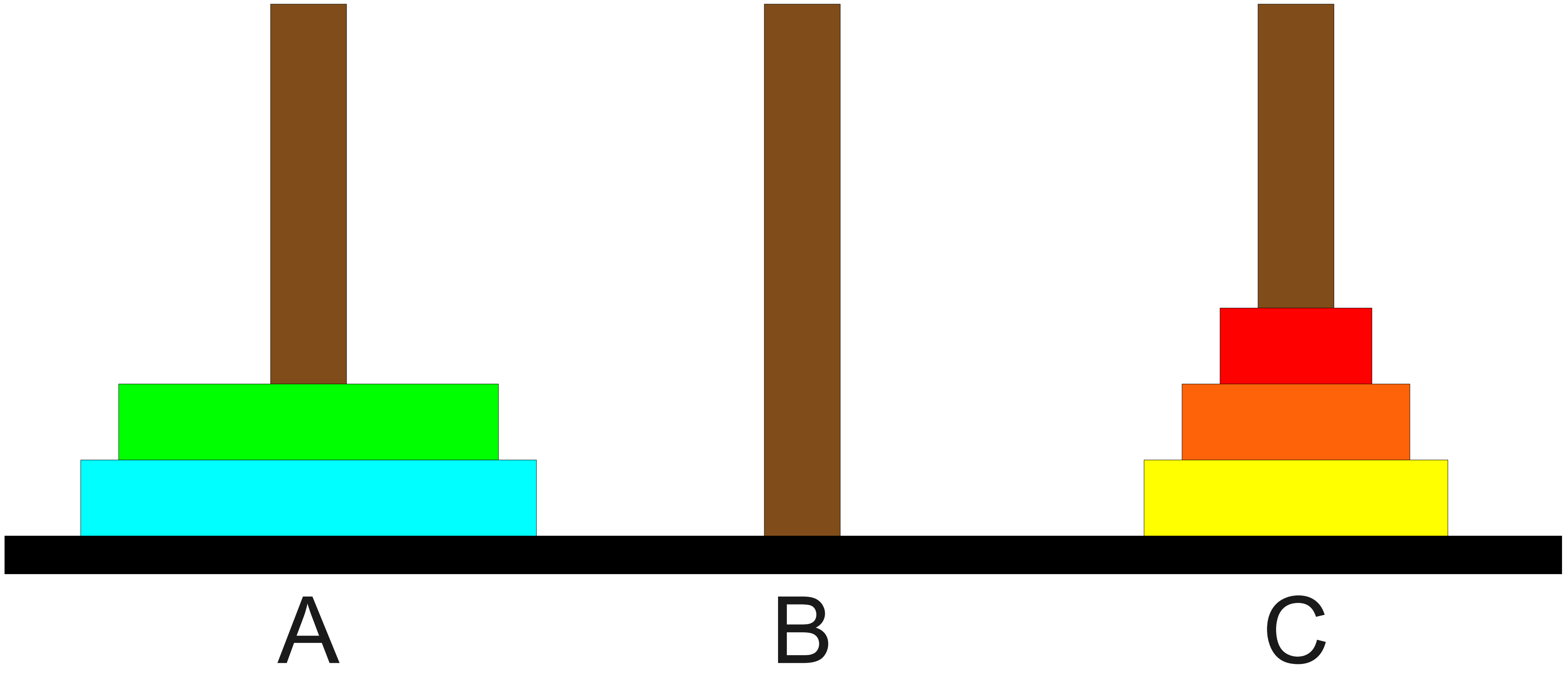
Towers of Hanoi



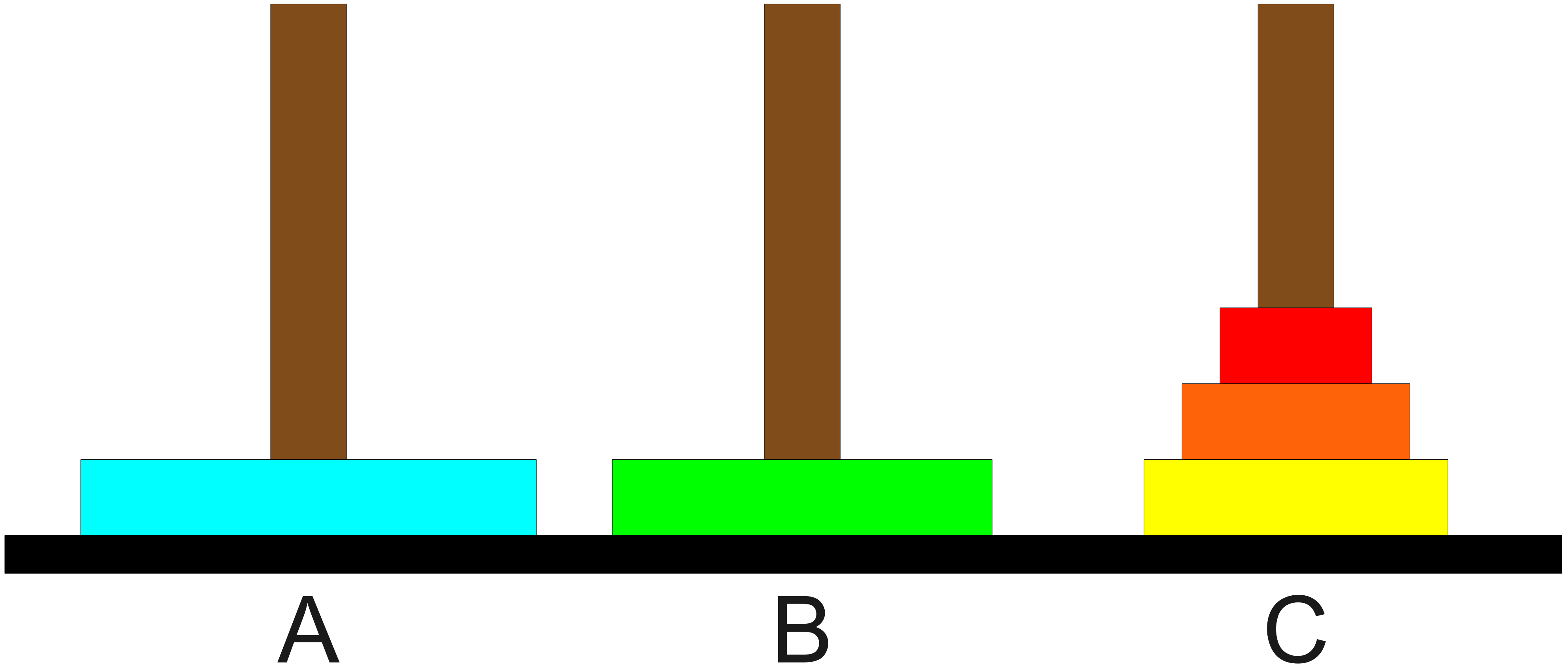
Towers of Hanoi



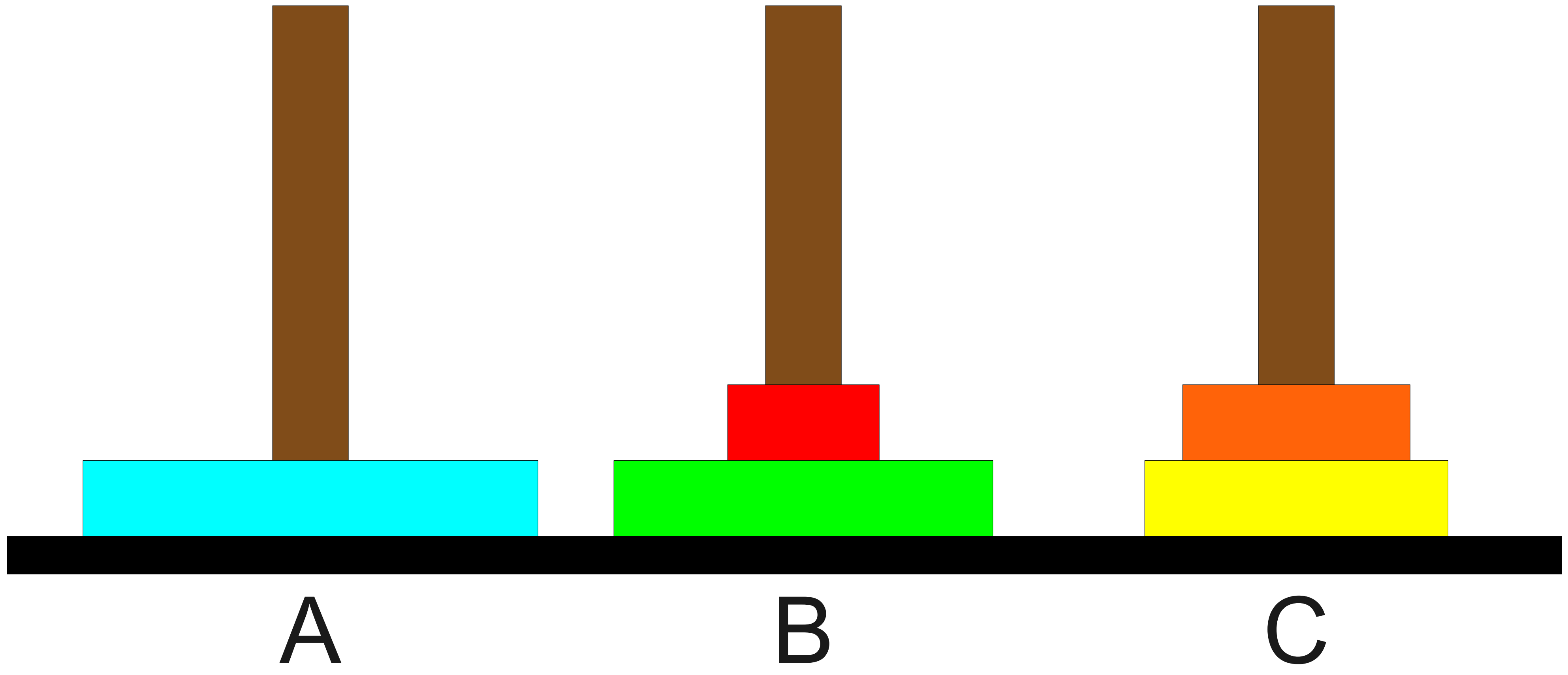
Towers of Hanoi



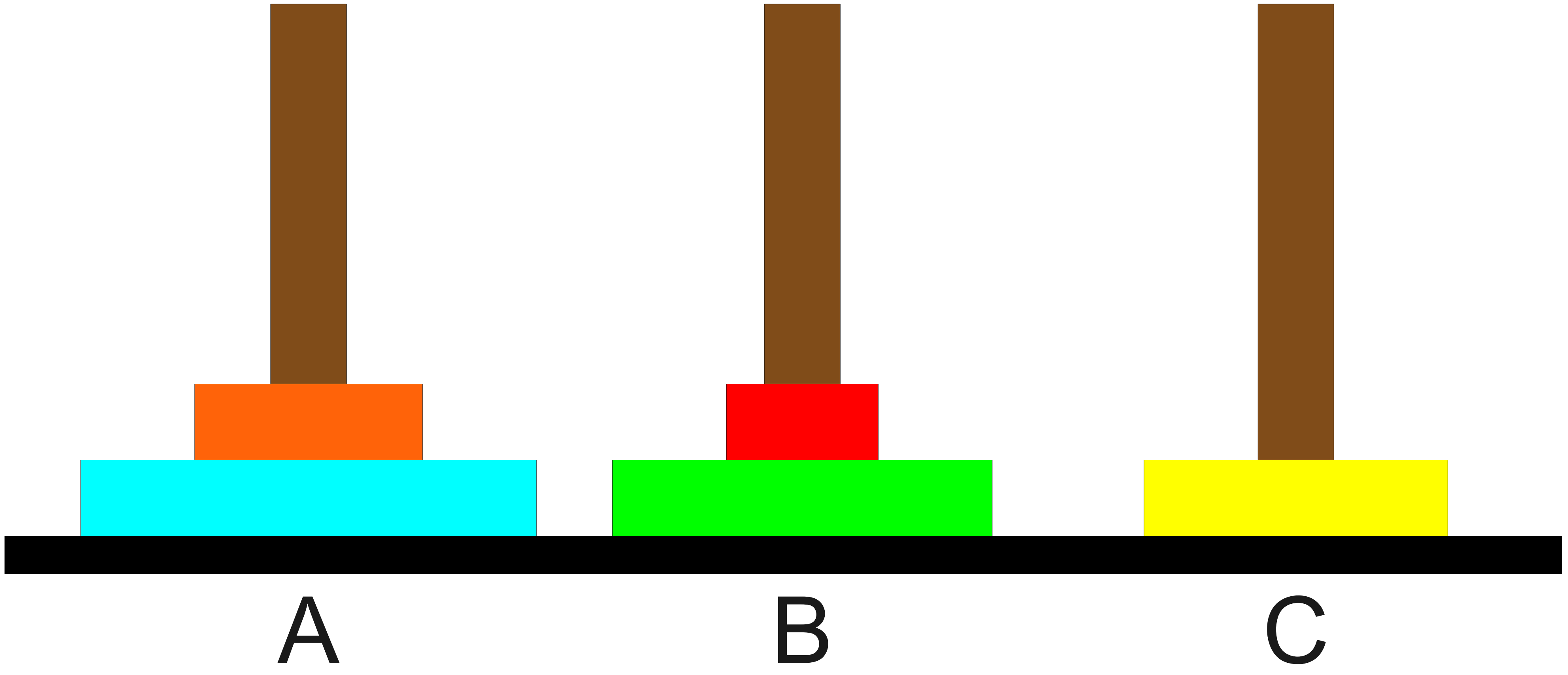
Towers of Hanoi



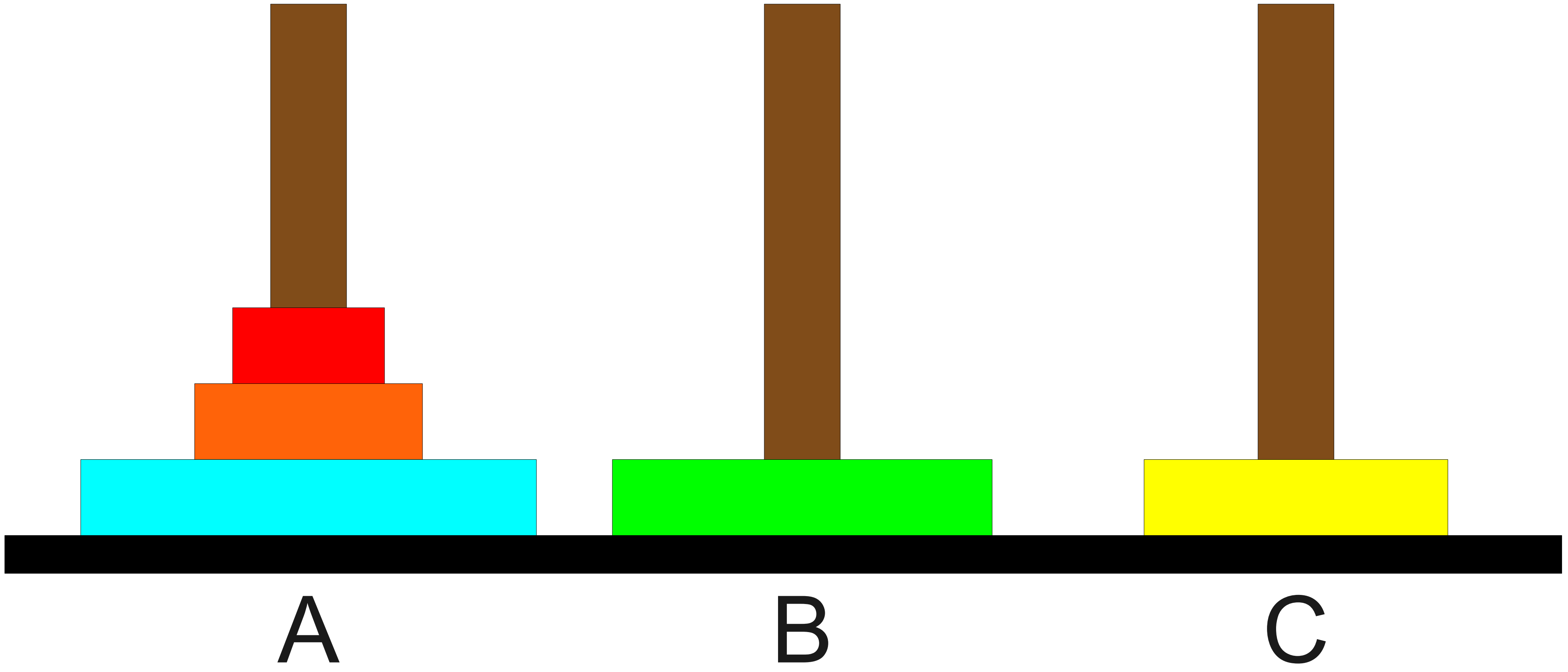
Towers of Hanoi



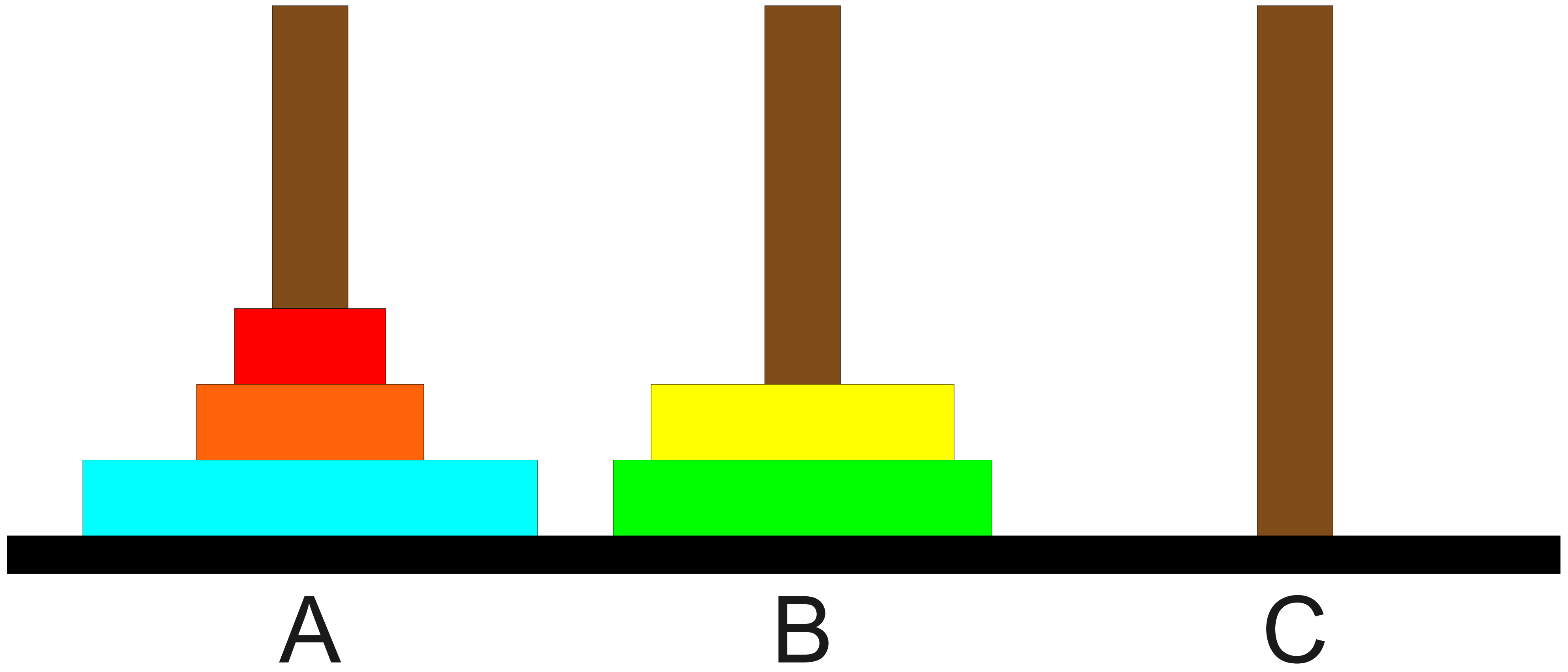
Towers of Hanoi



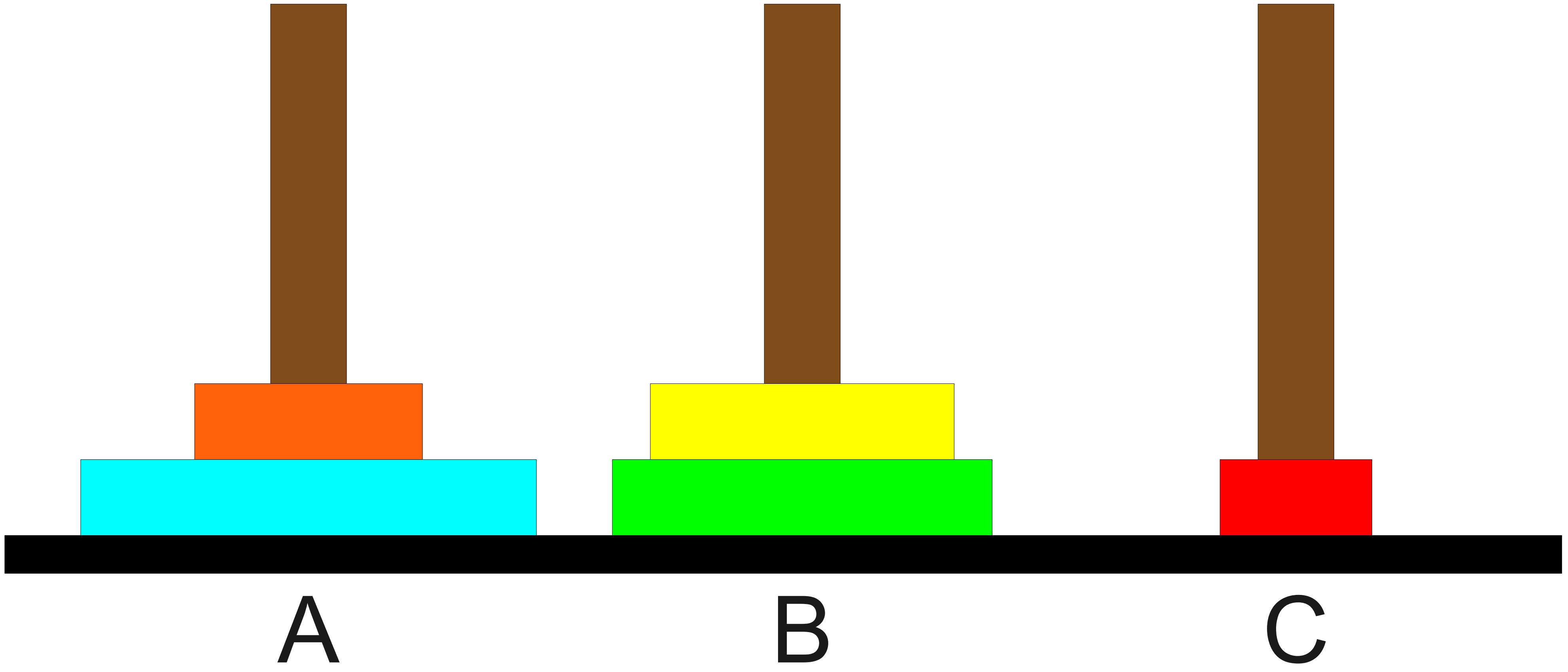
Towers of Hanoi



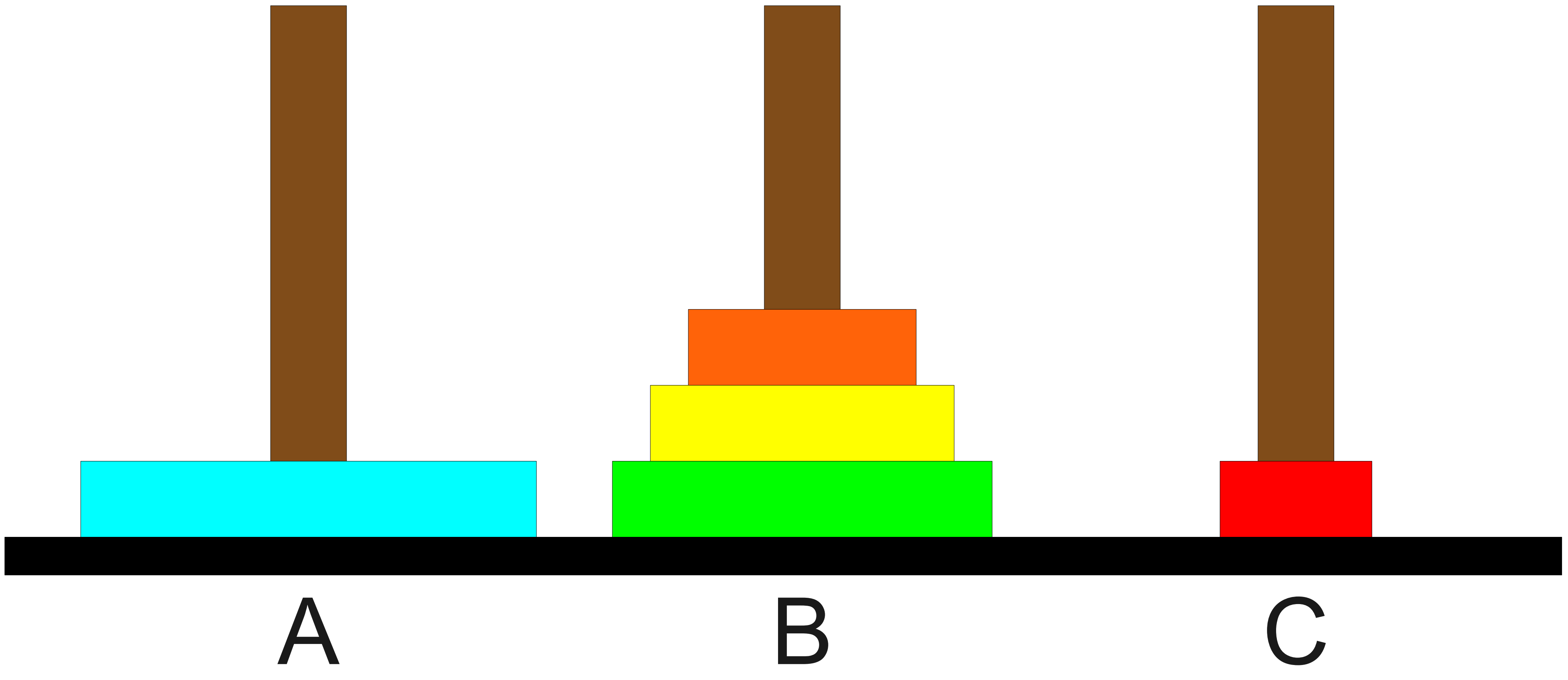
Towers of Hanoi



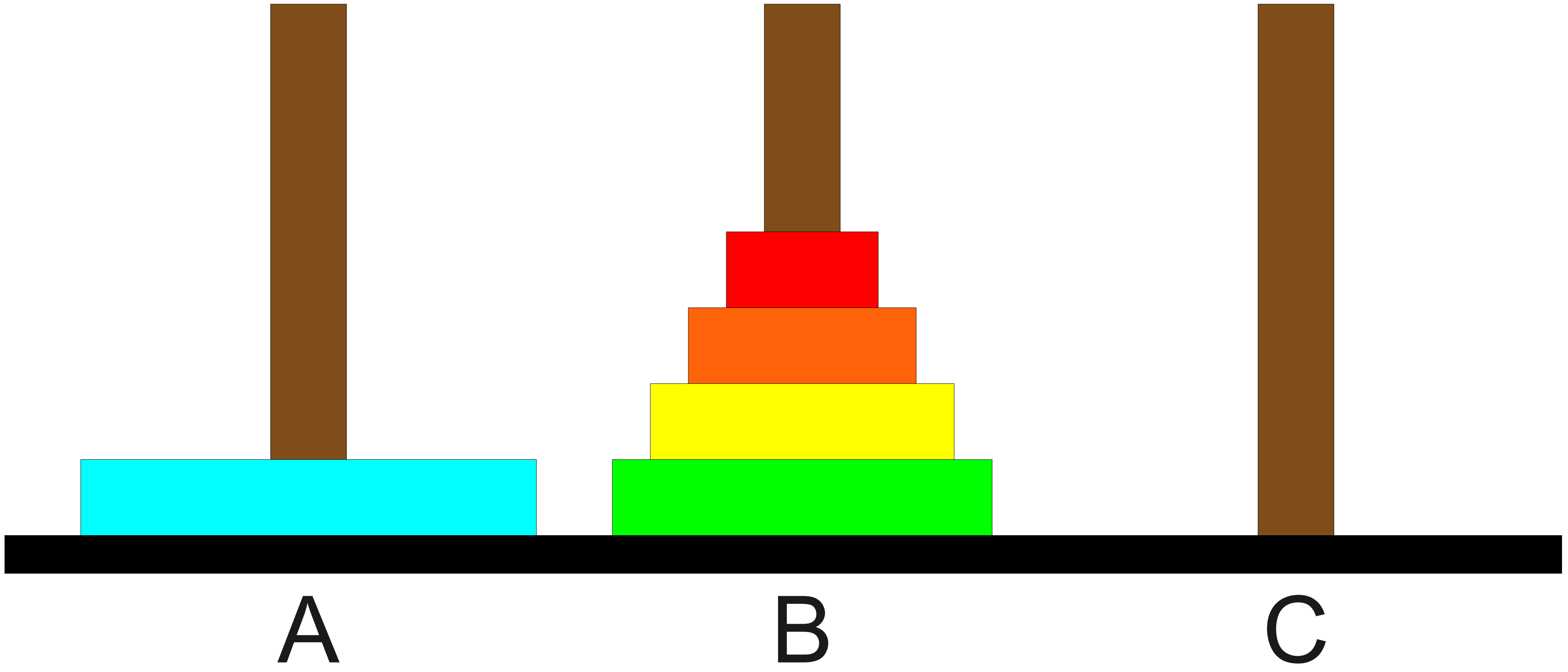
Towers of Hanoi



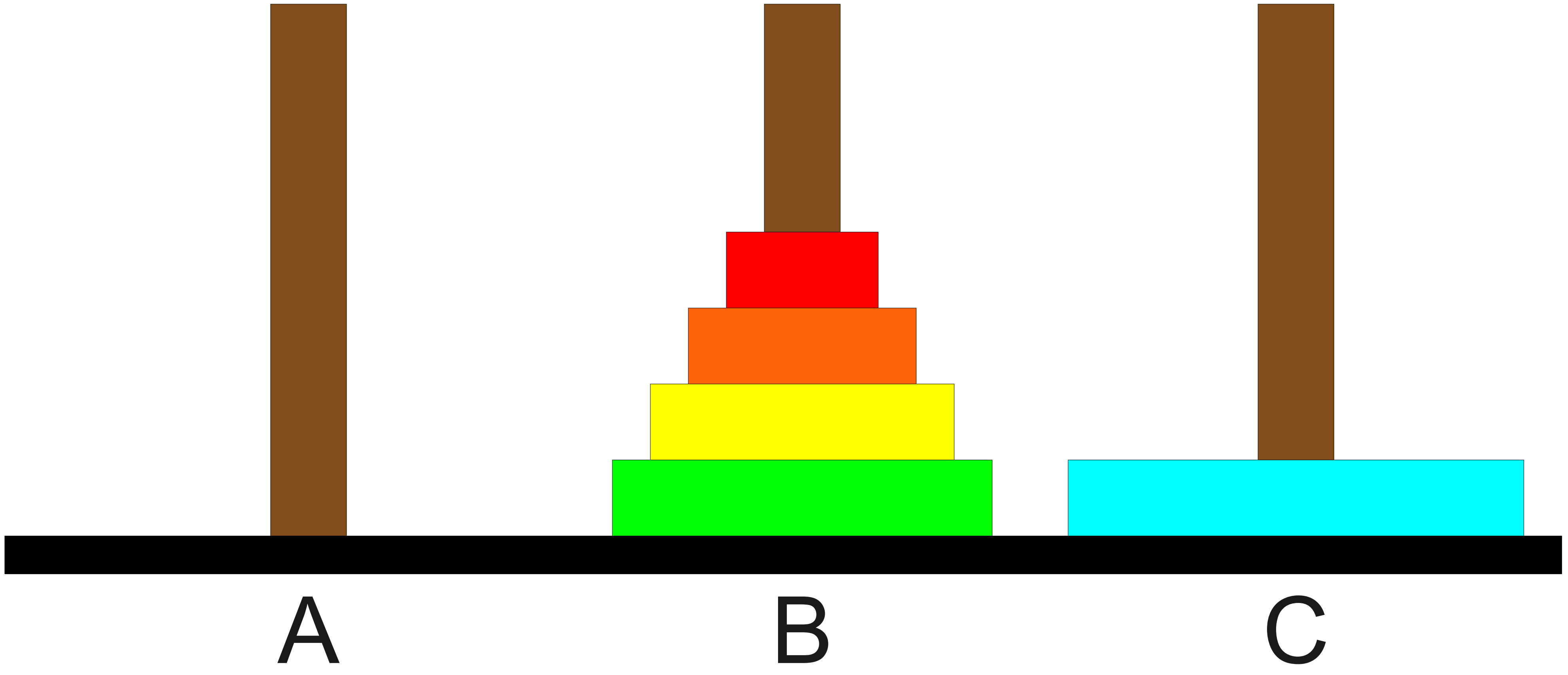
Towers of Hanoi



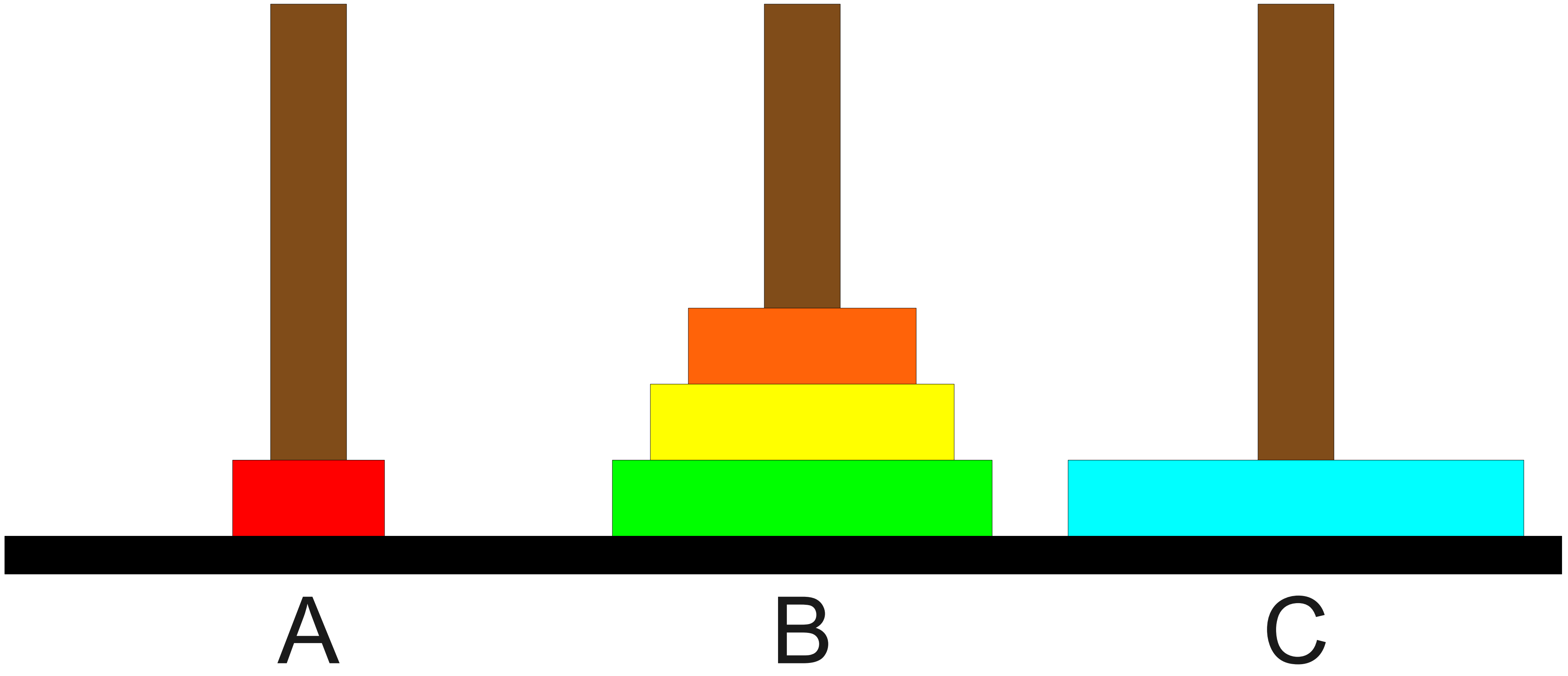
Towers of Hanoi



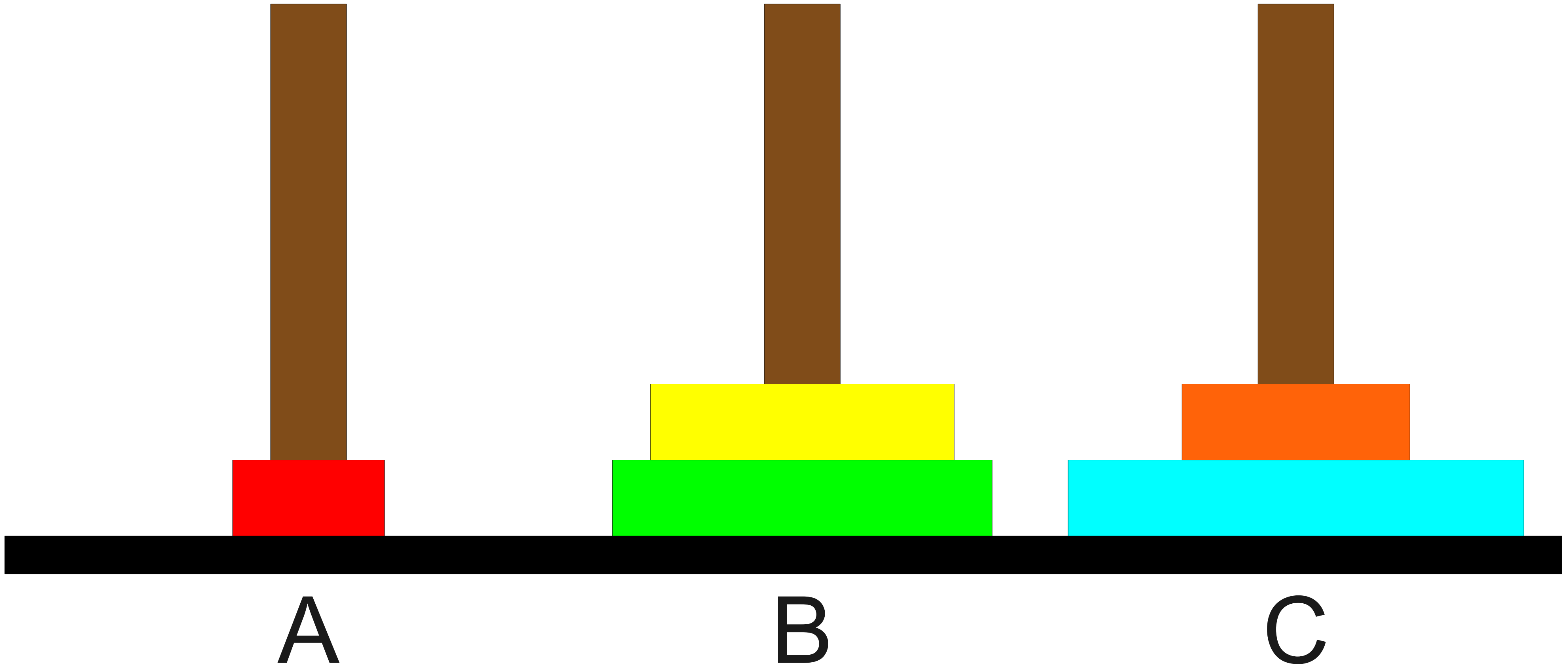
Towers of Hanoi



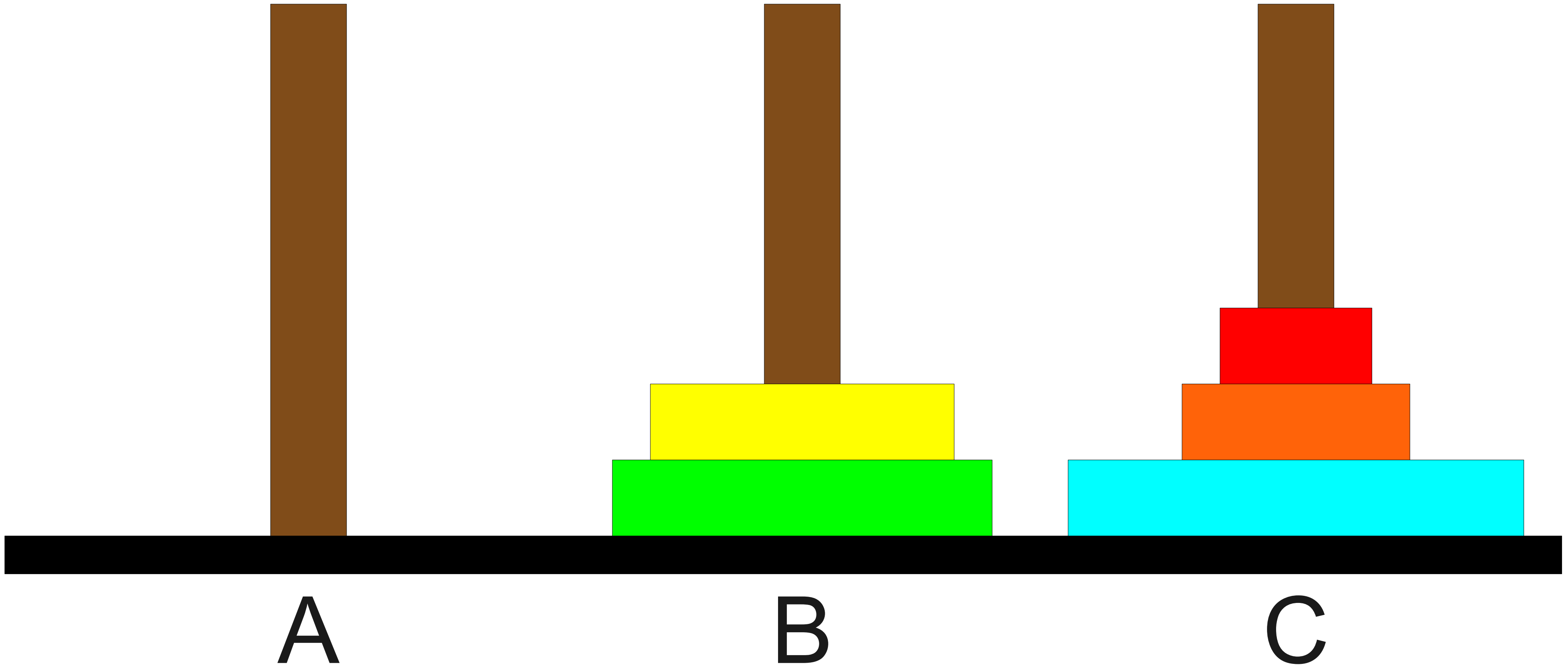
Towers of Hanoi



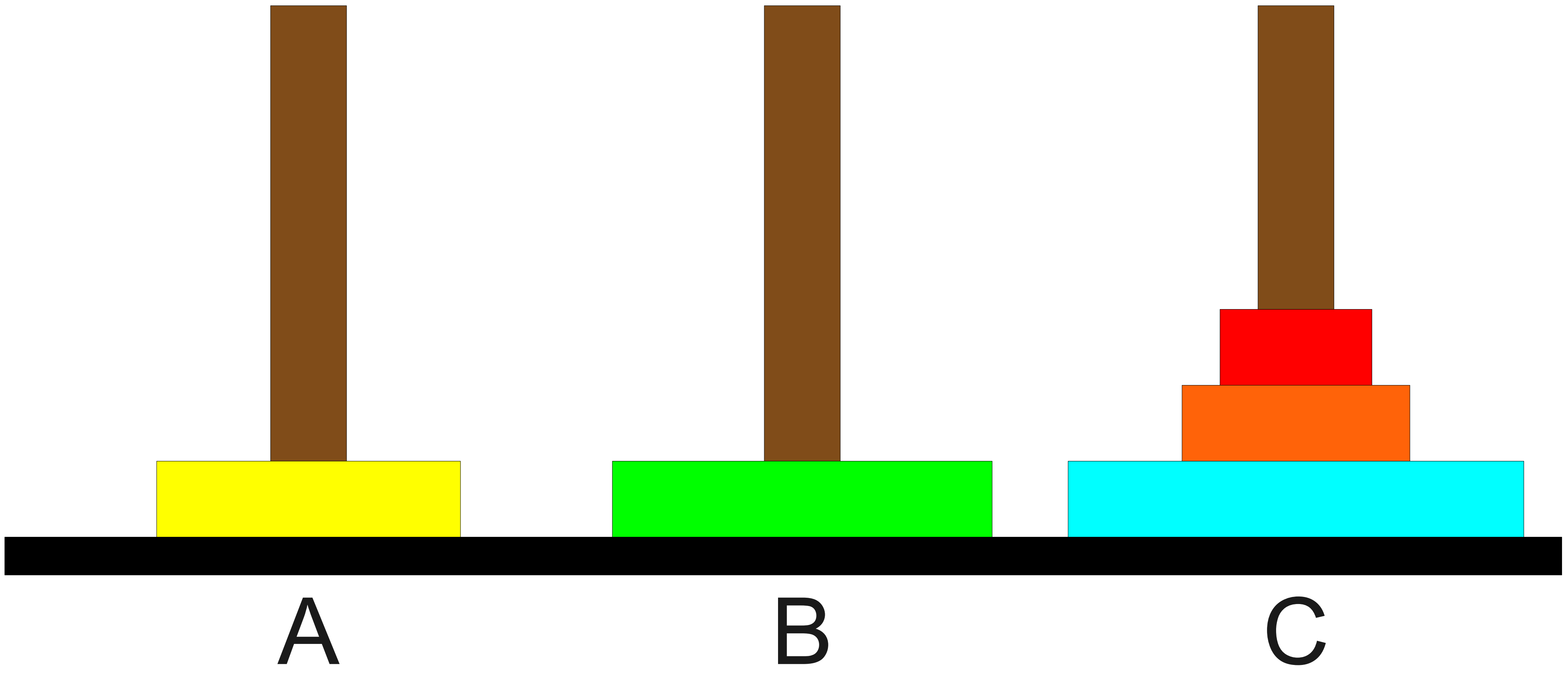
Towers of Hanoi



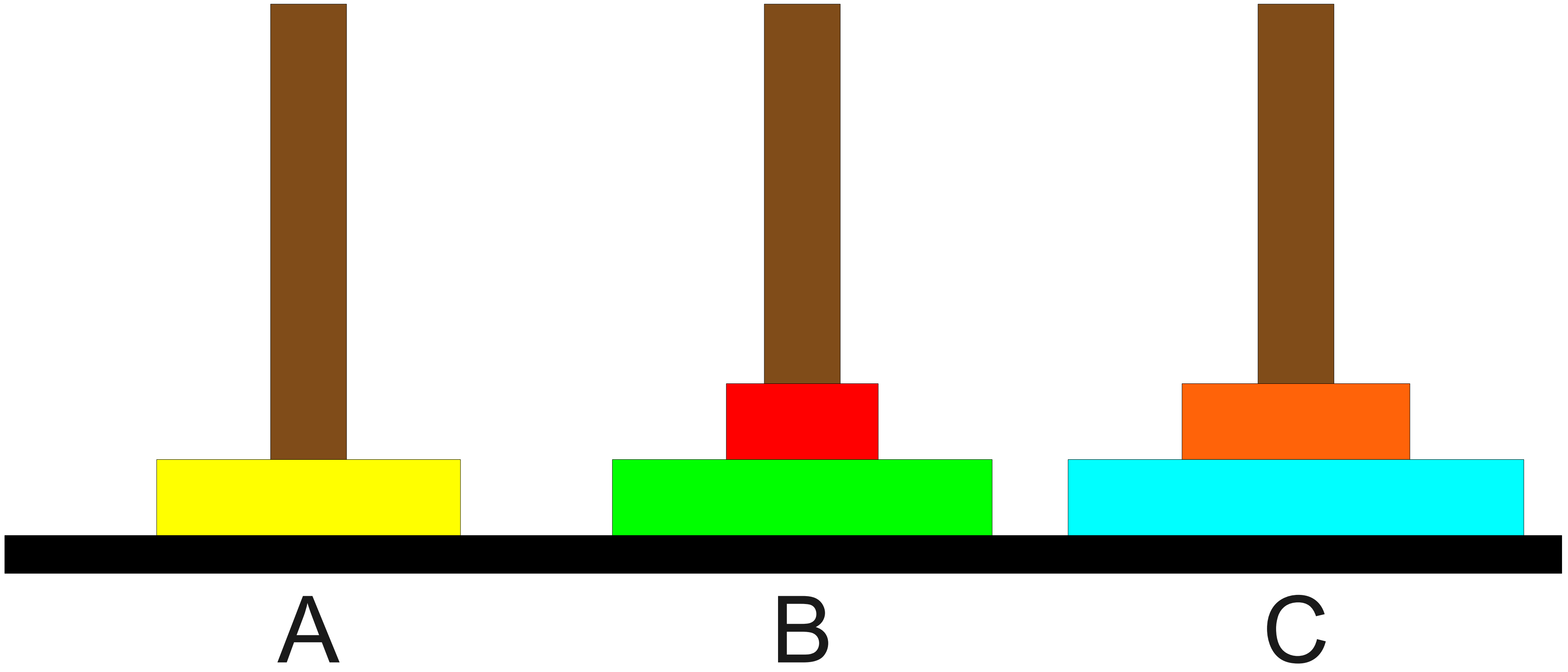
Towers of Hanoi



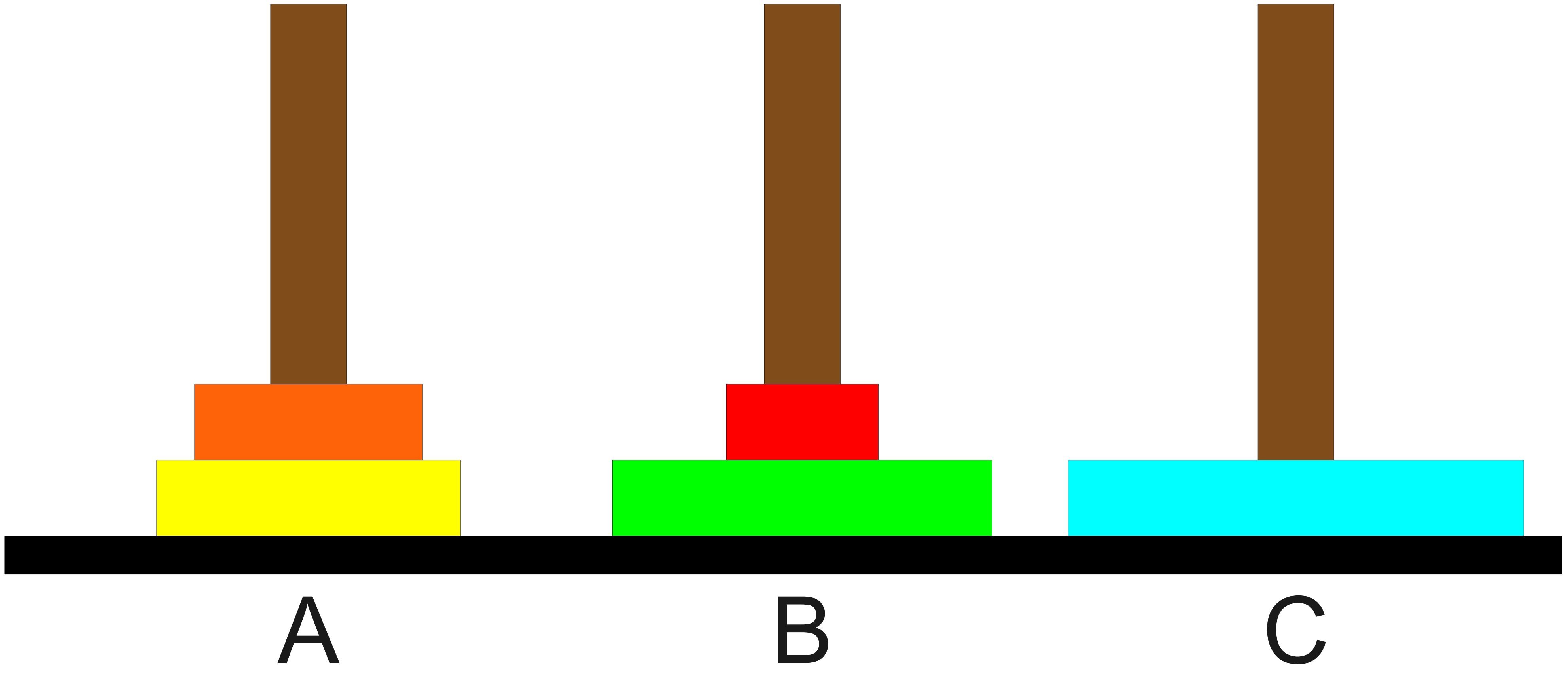
Towers of Hanoi



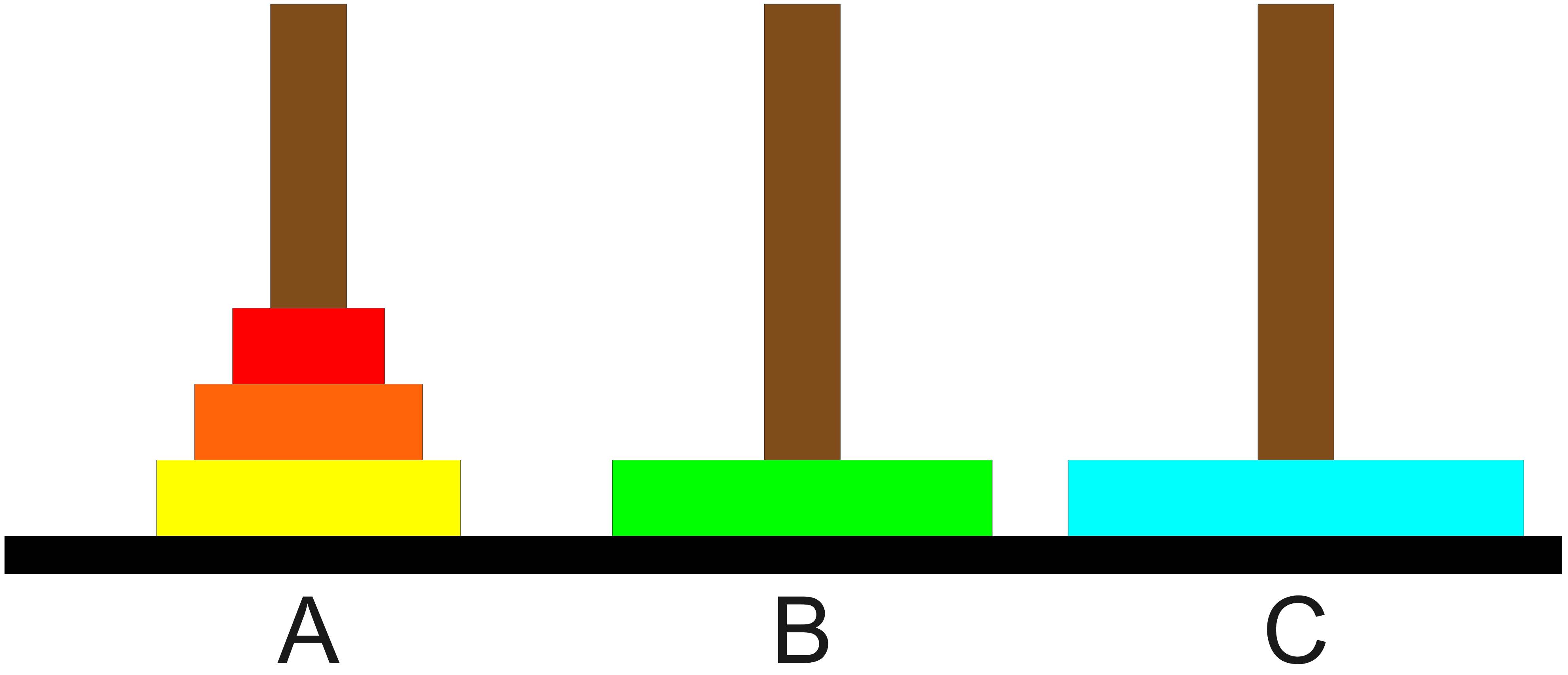
Towers of Hanoi



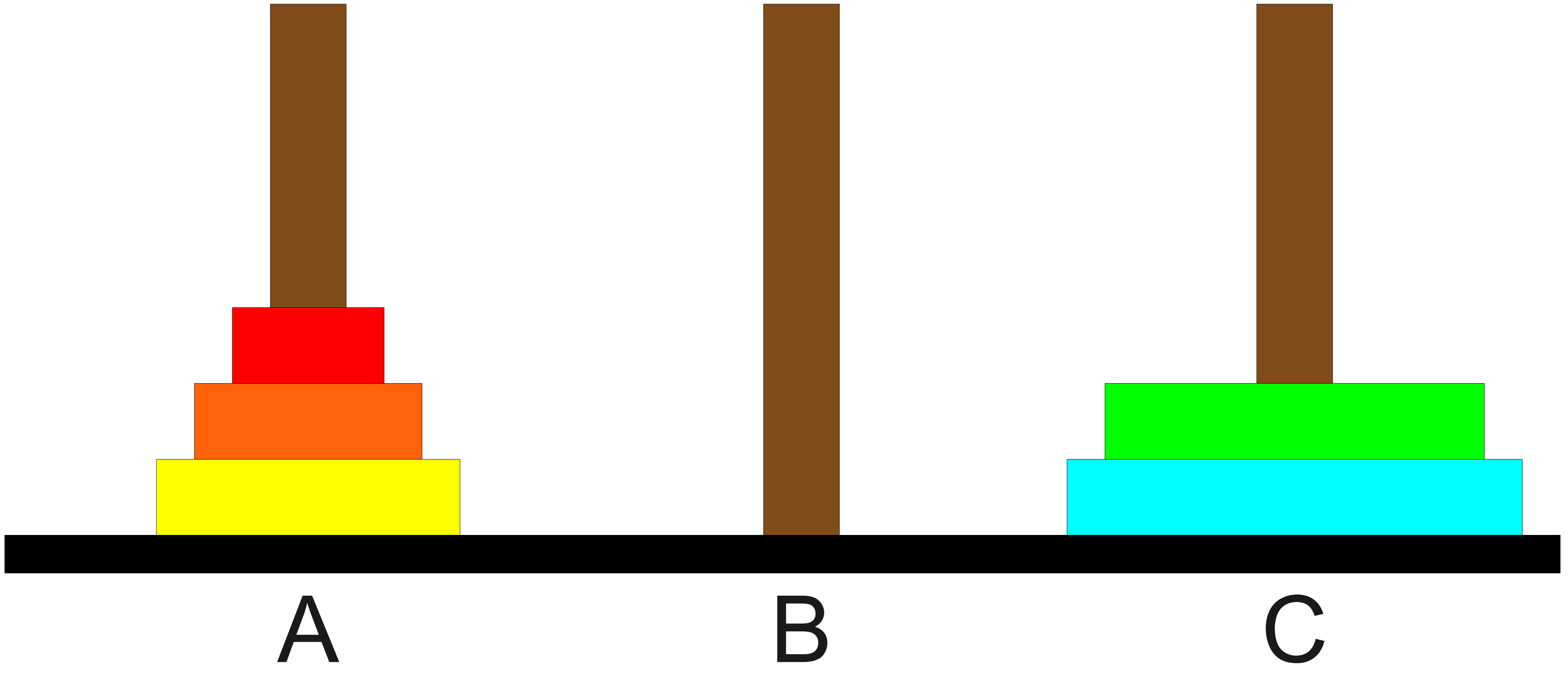
Towers of Hanoi



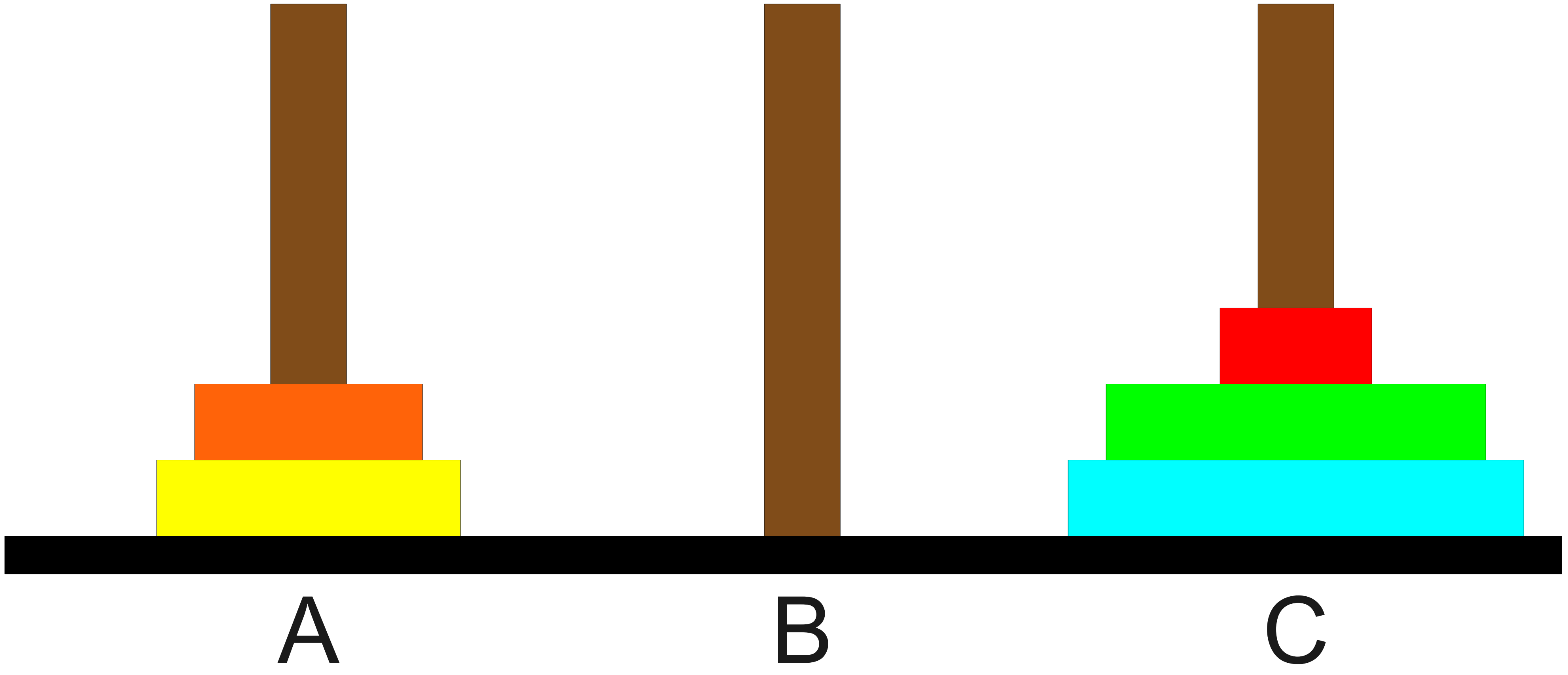
Towers of Hanoi



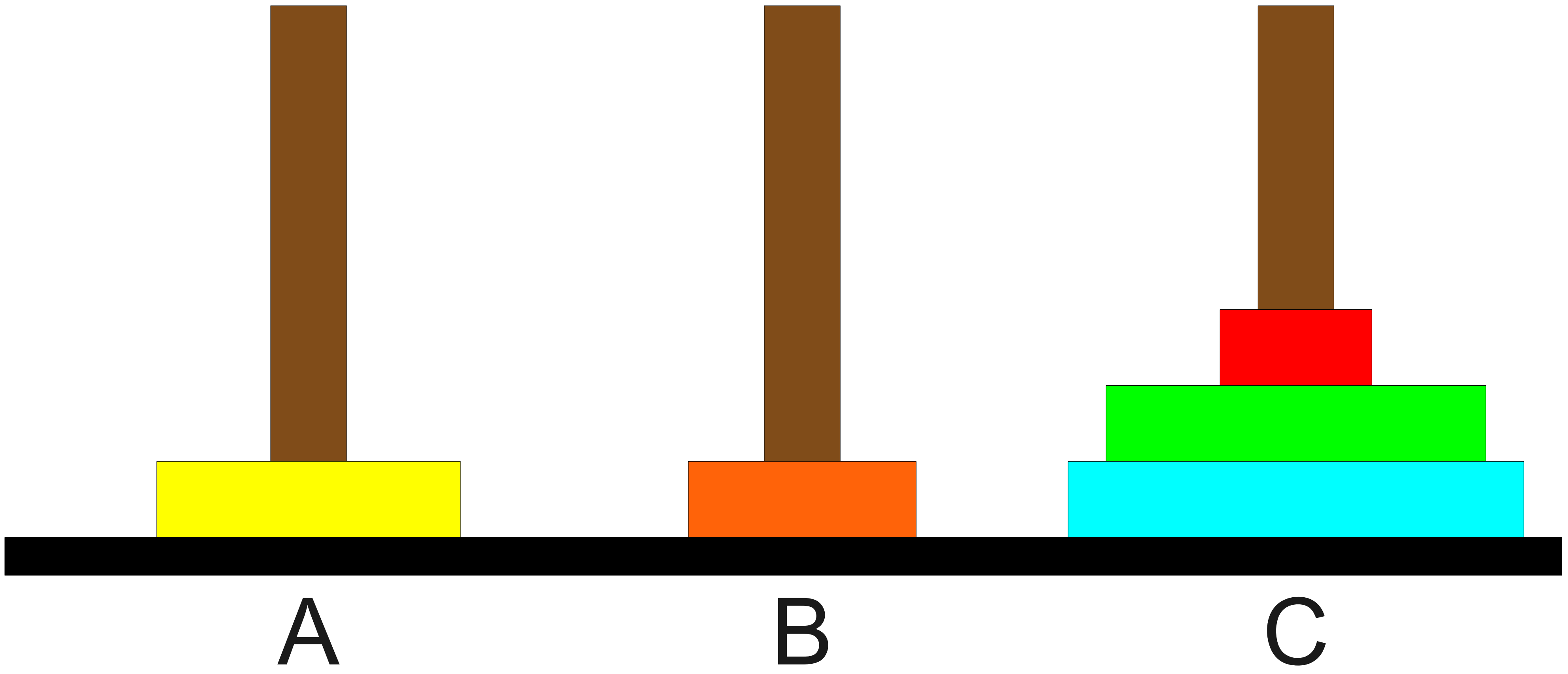
Towers of Hanoi



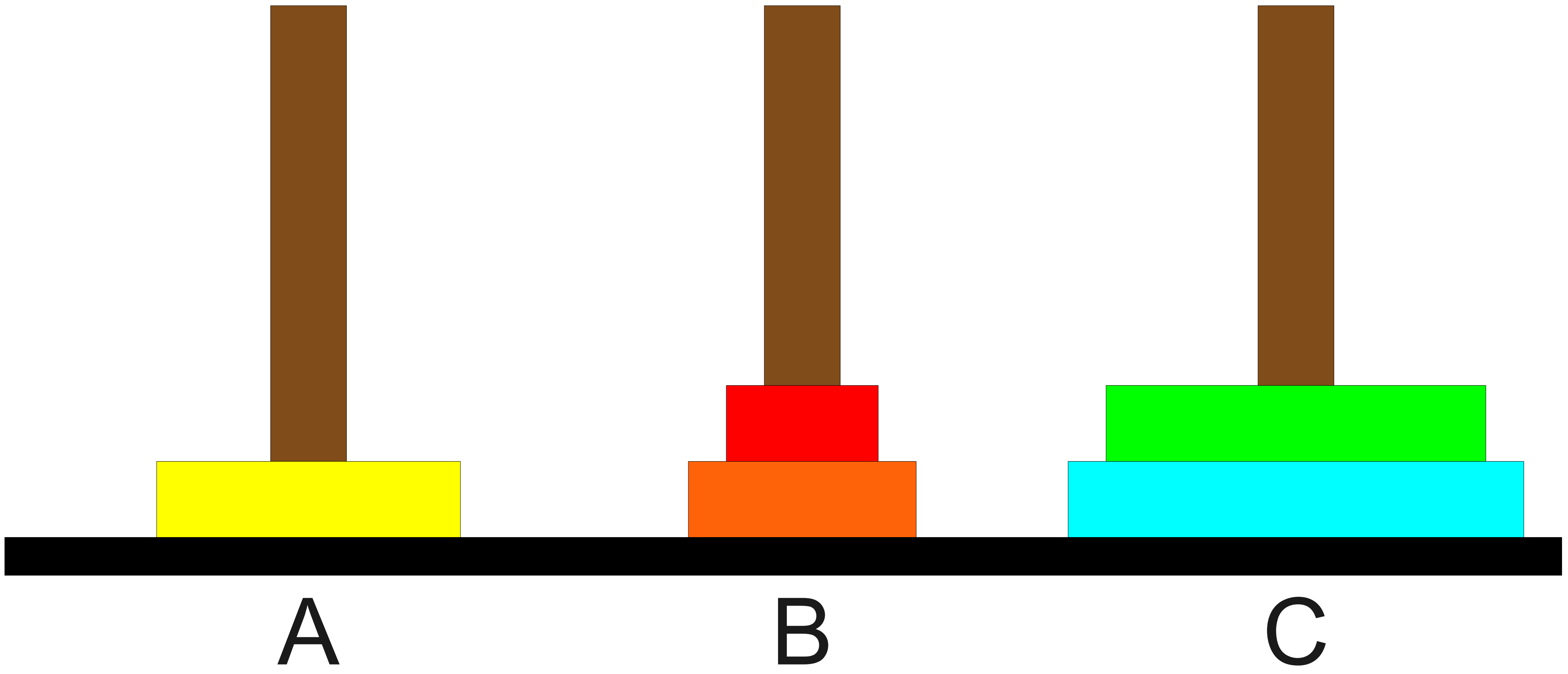
Towers of Hanoi



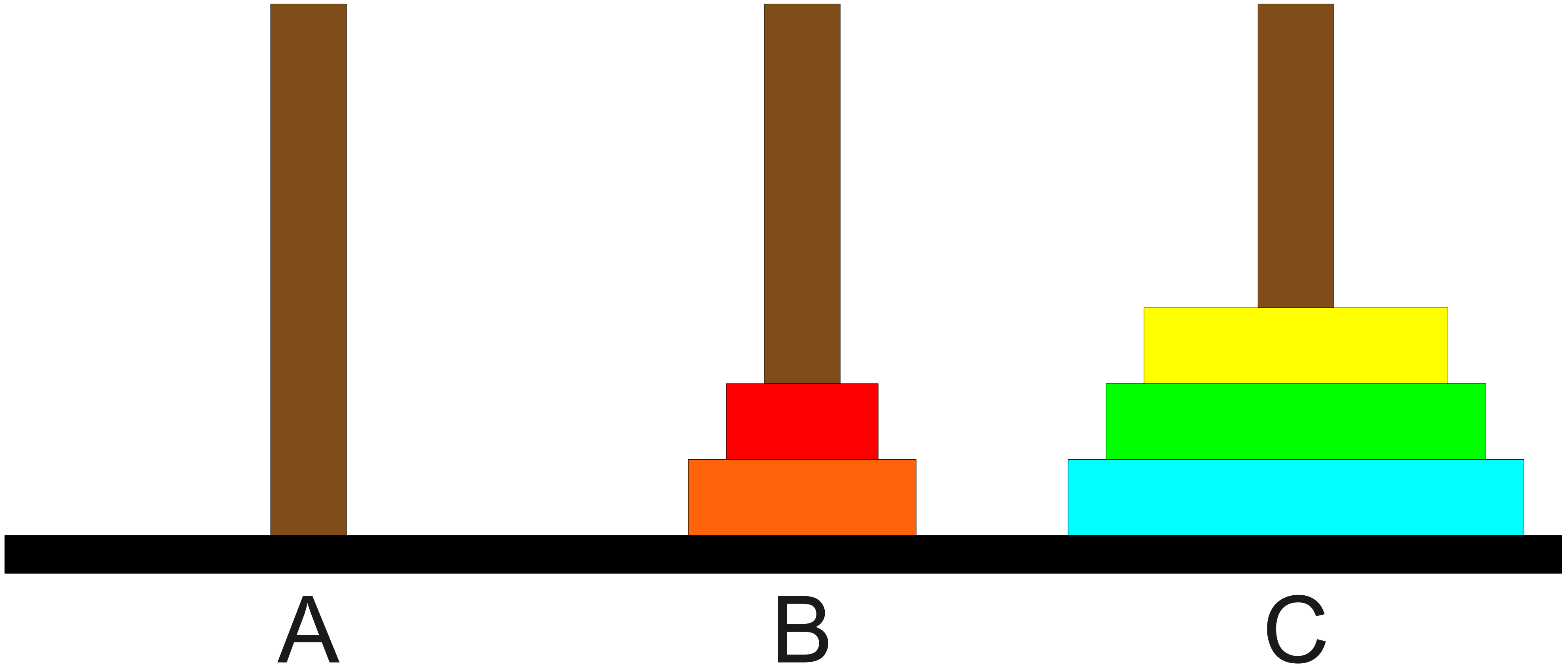
Towers of Hanoi



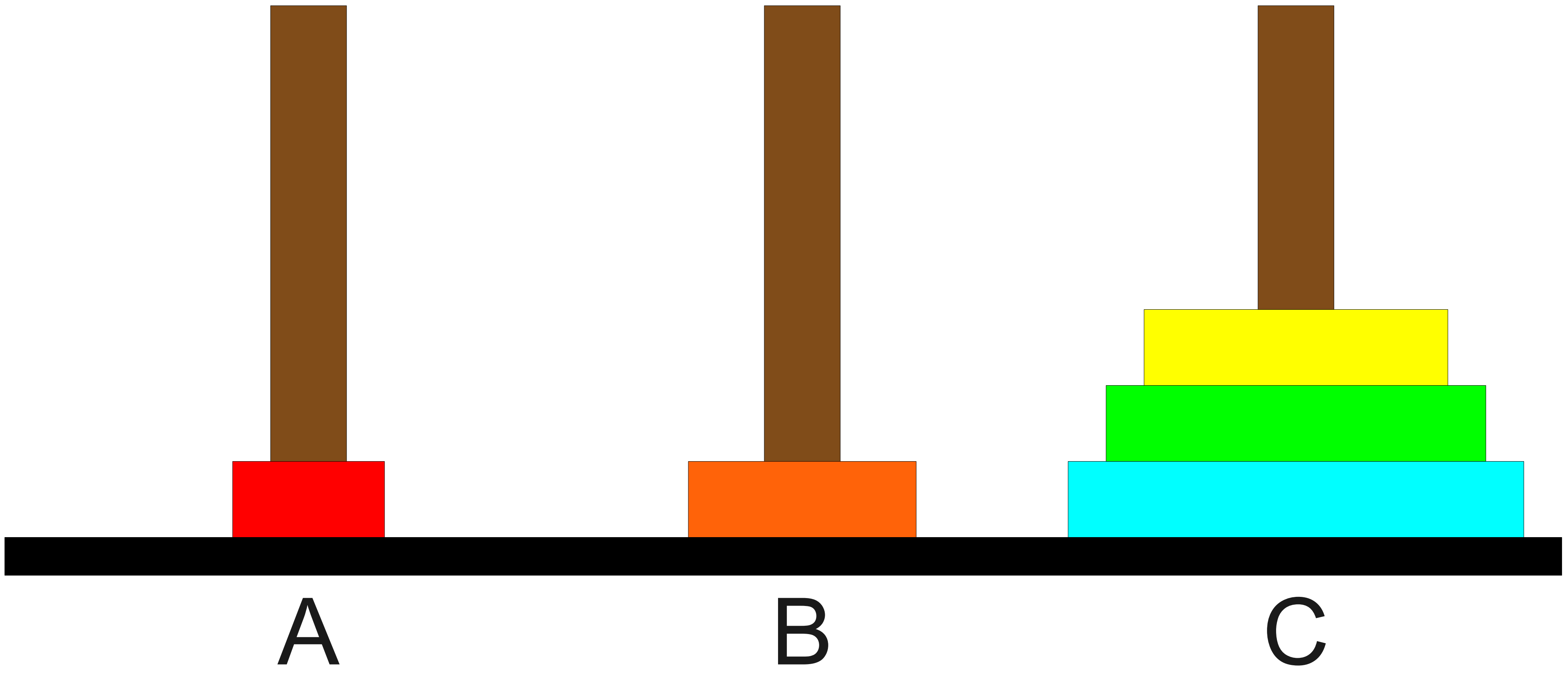
Towers of Hanoi



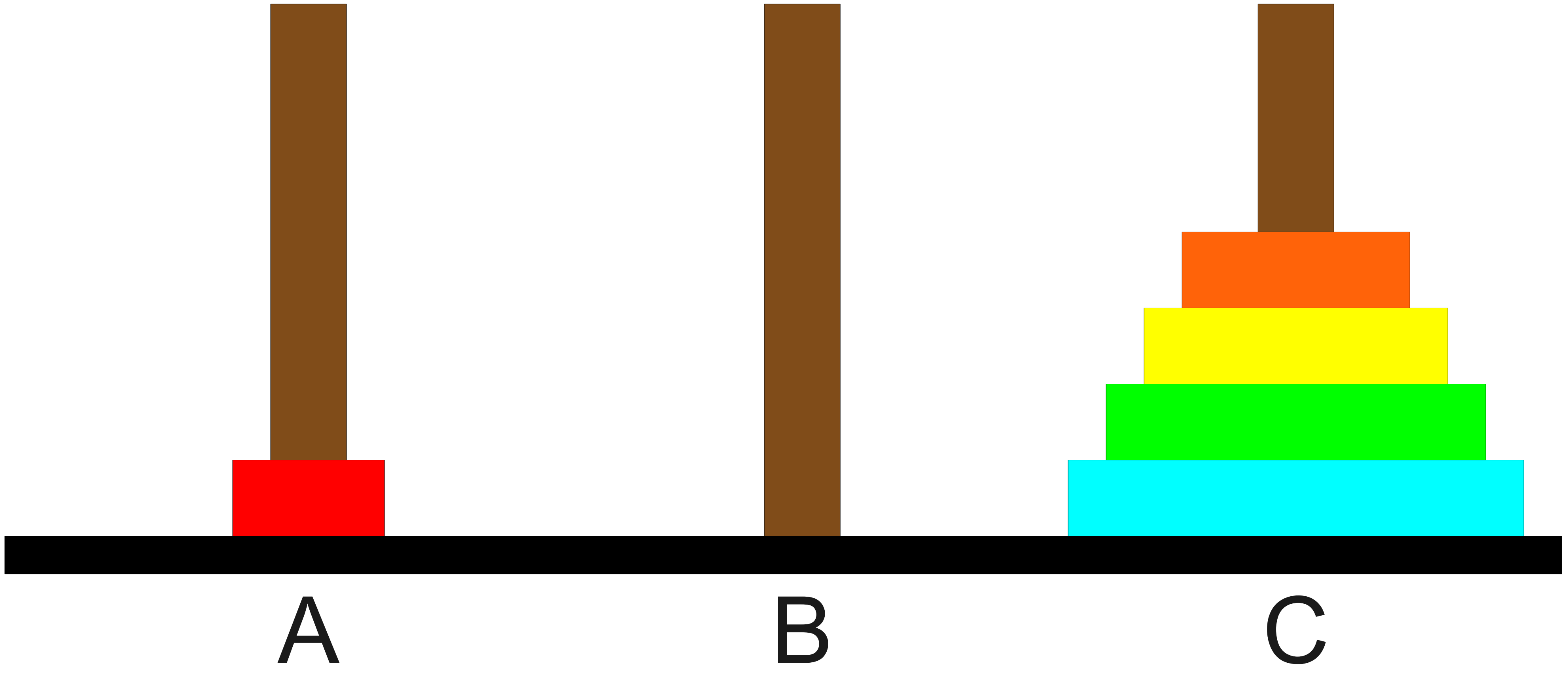
Towers of Hanoi



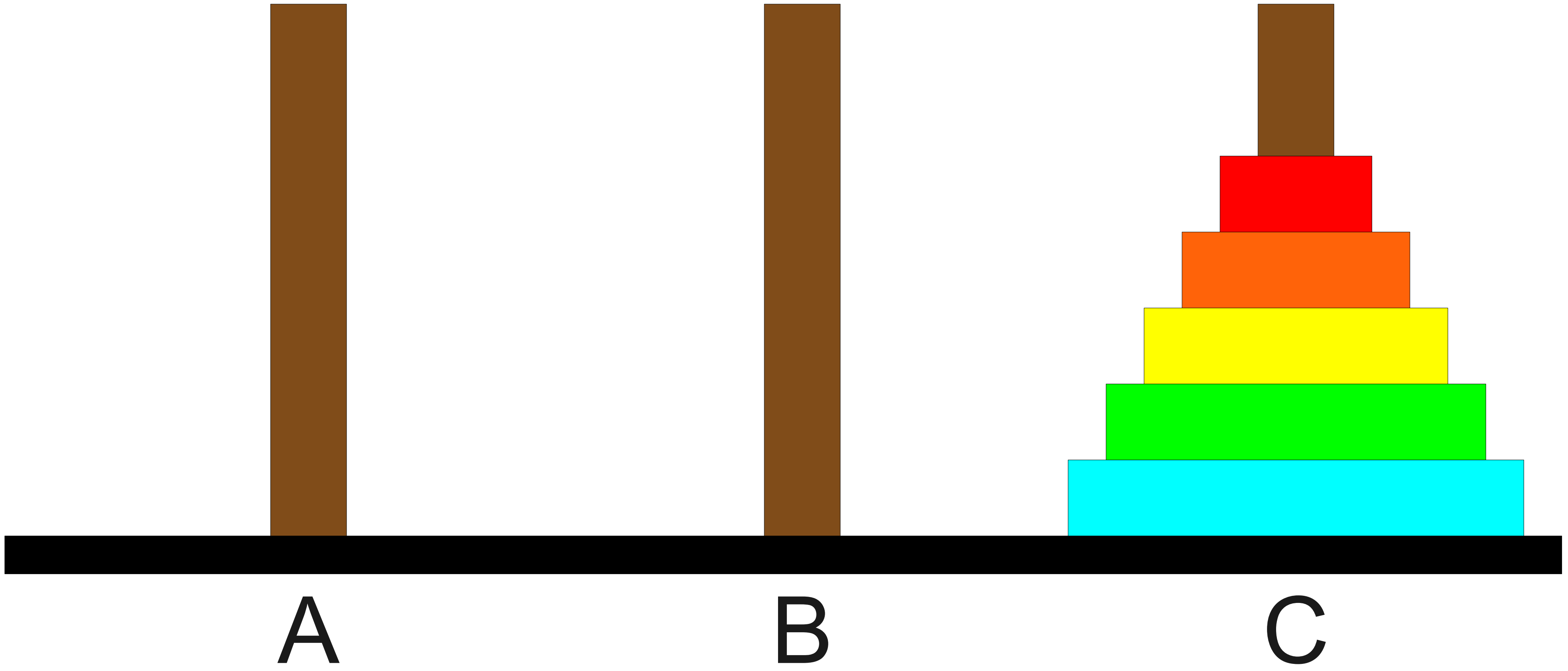
Towers of Hanoi



Towers of Hanoi

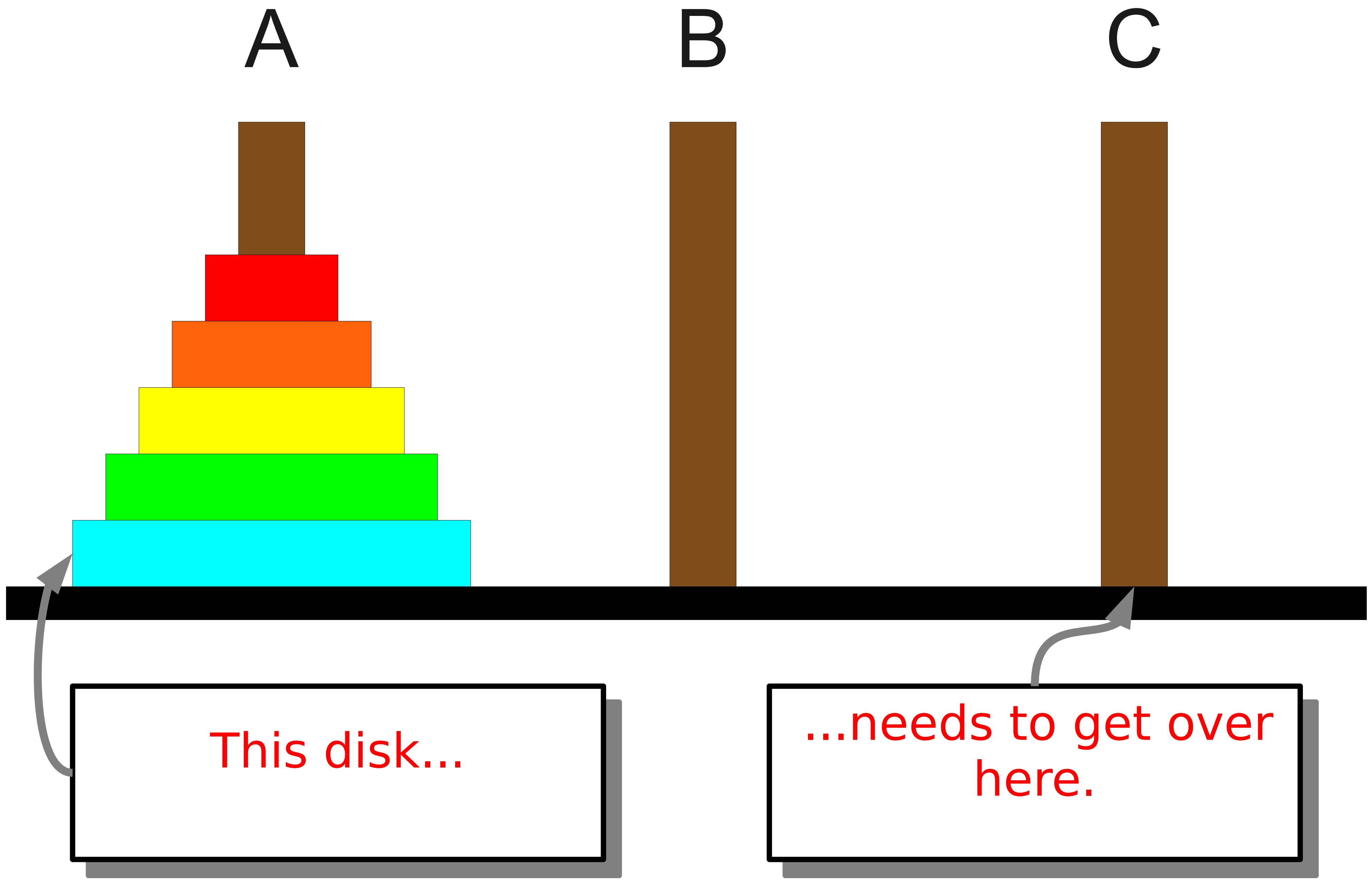


Towers of Hanoi

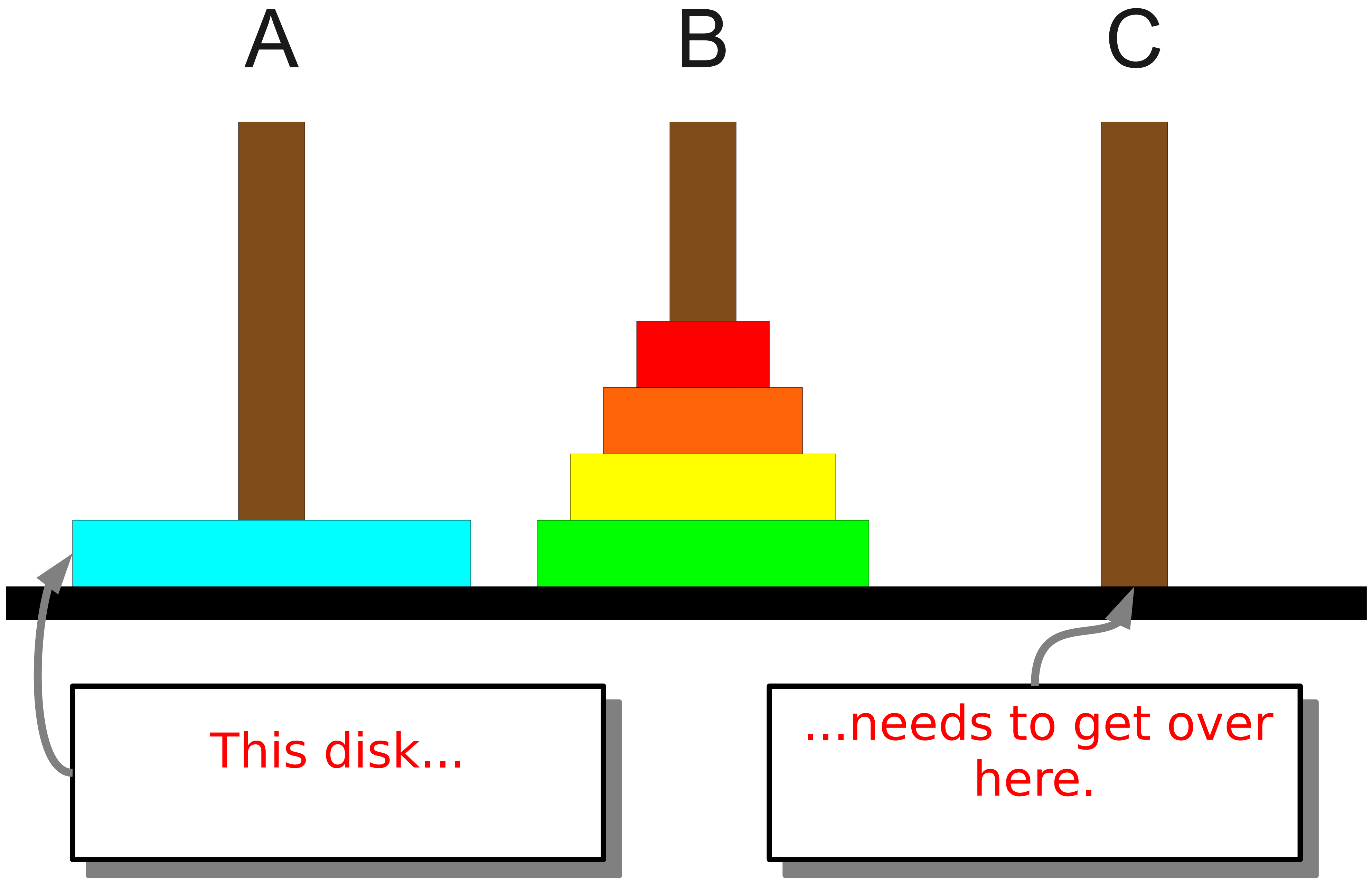


Solving the Towers of Hanoi

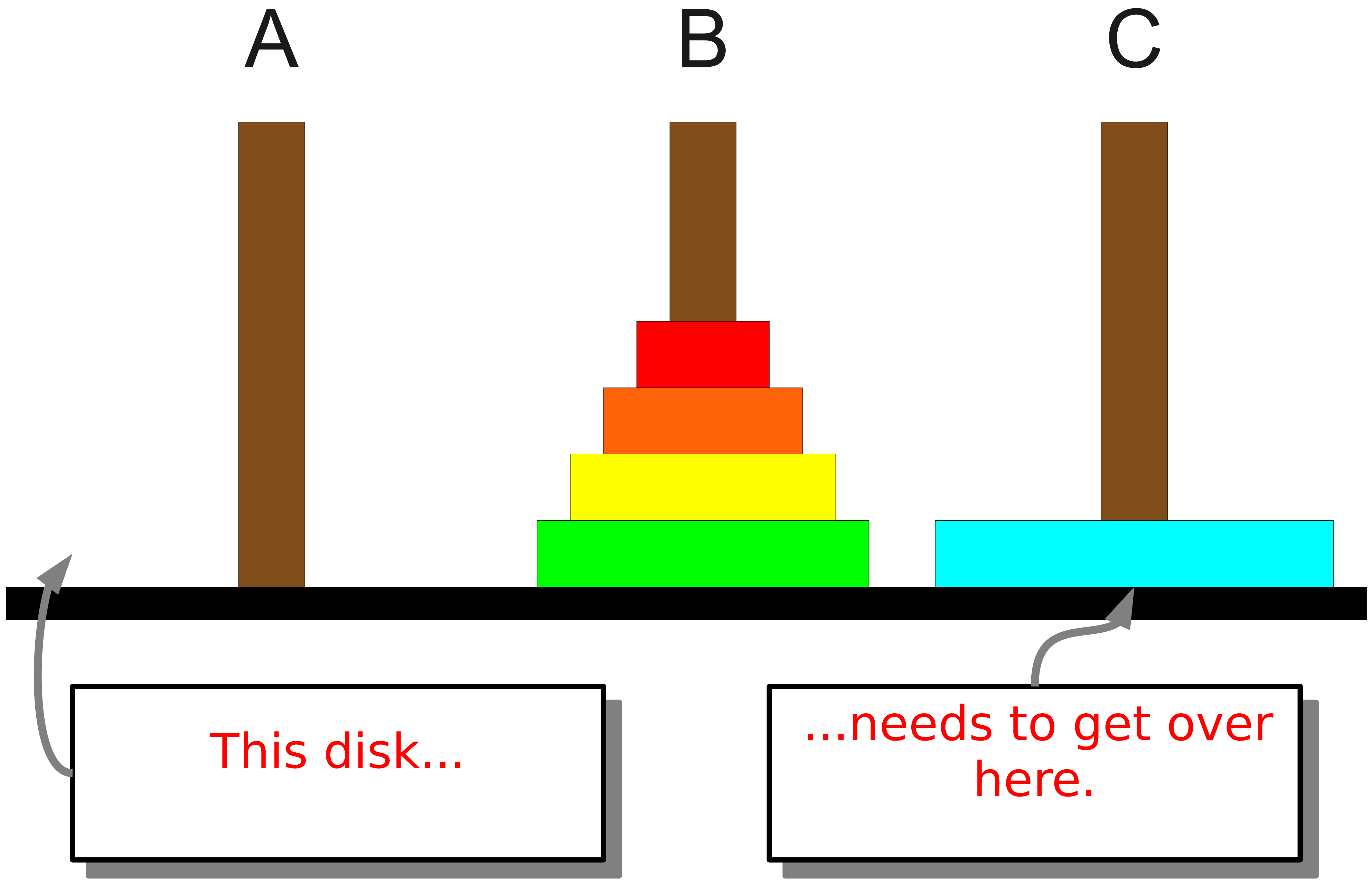
Solving the Towers of Hanoi



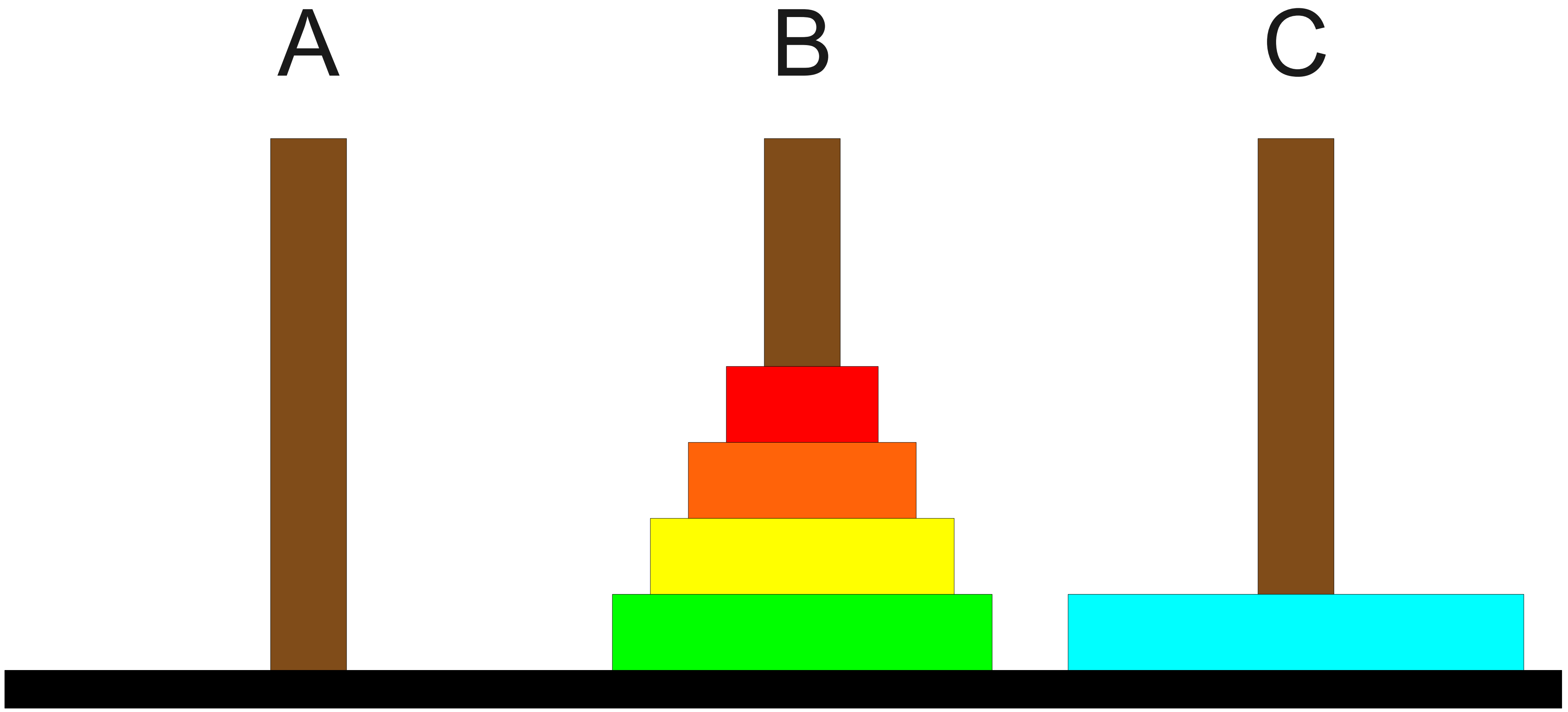
Solving the Towers of Hanoi



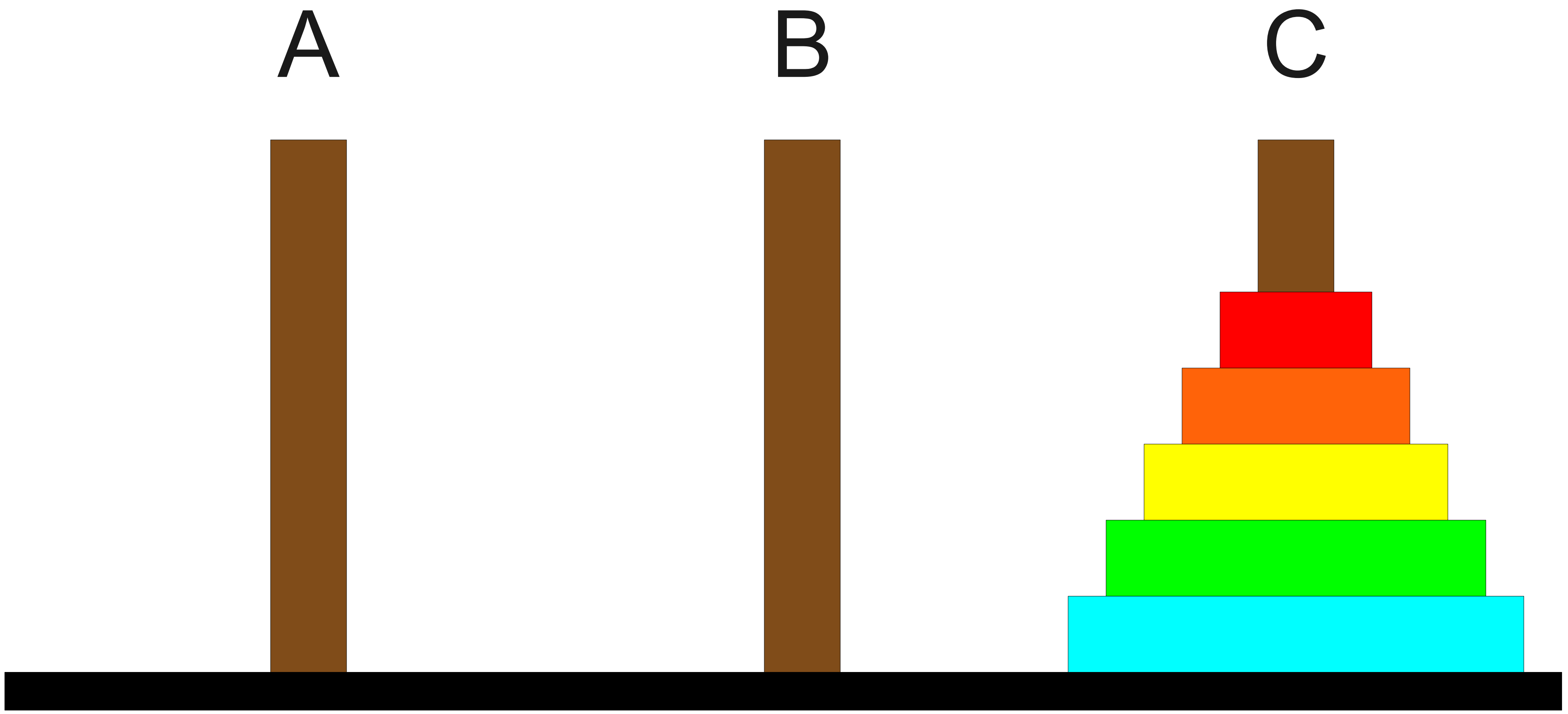
Solving the Towers of Hanoi



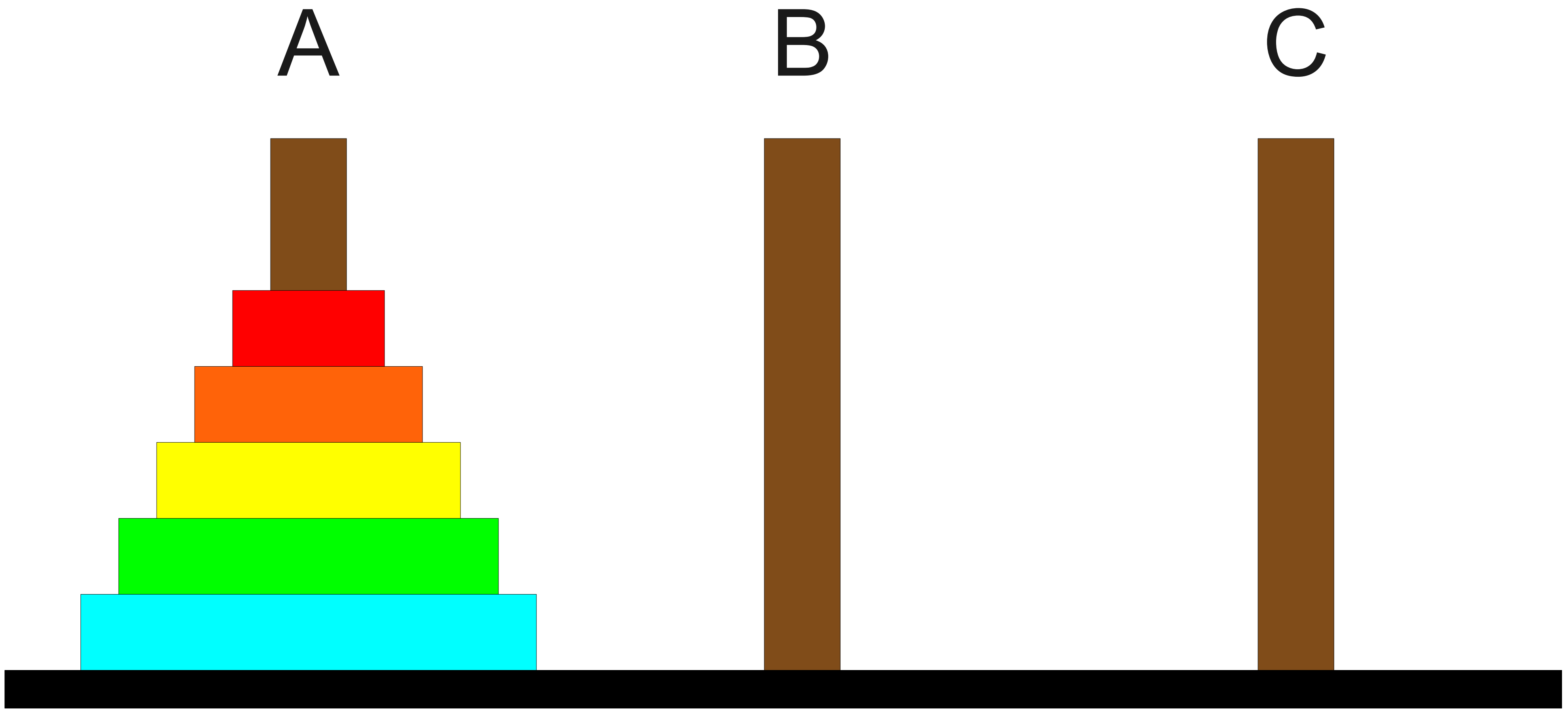
Solving the Towers of Hanoi



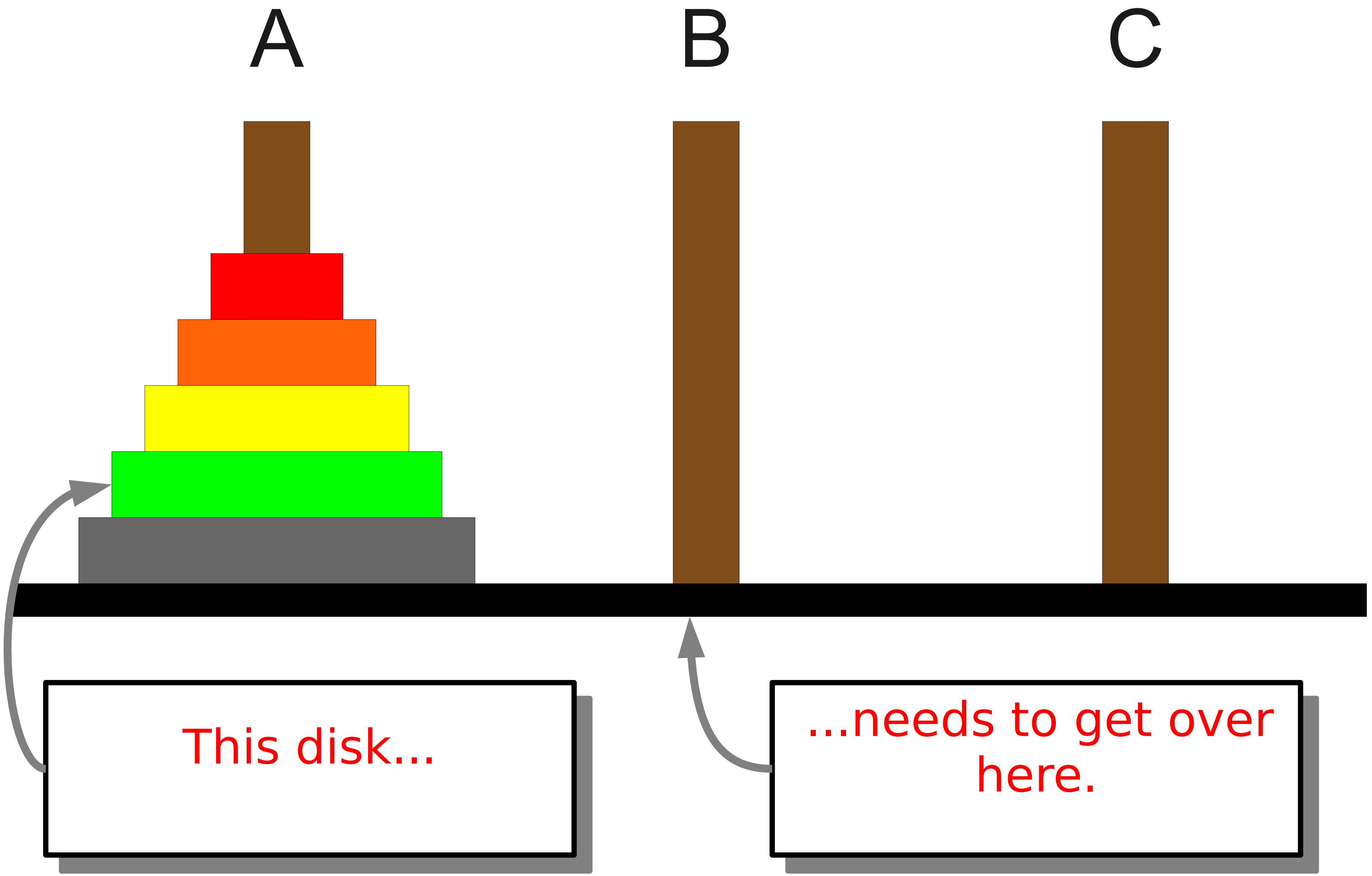
Solving the Towers of Hanoi



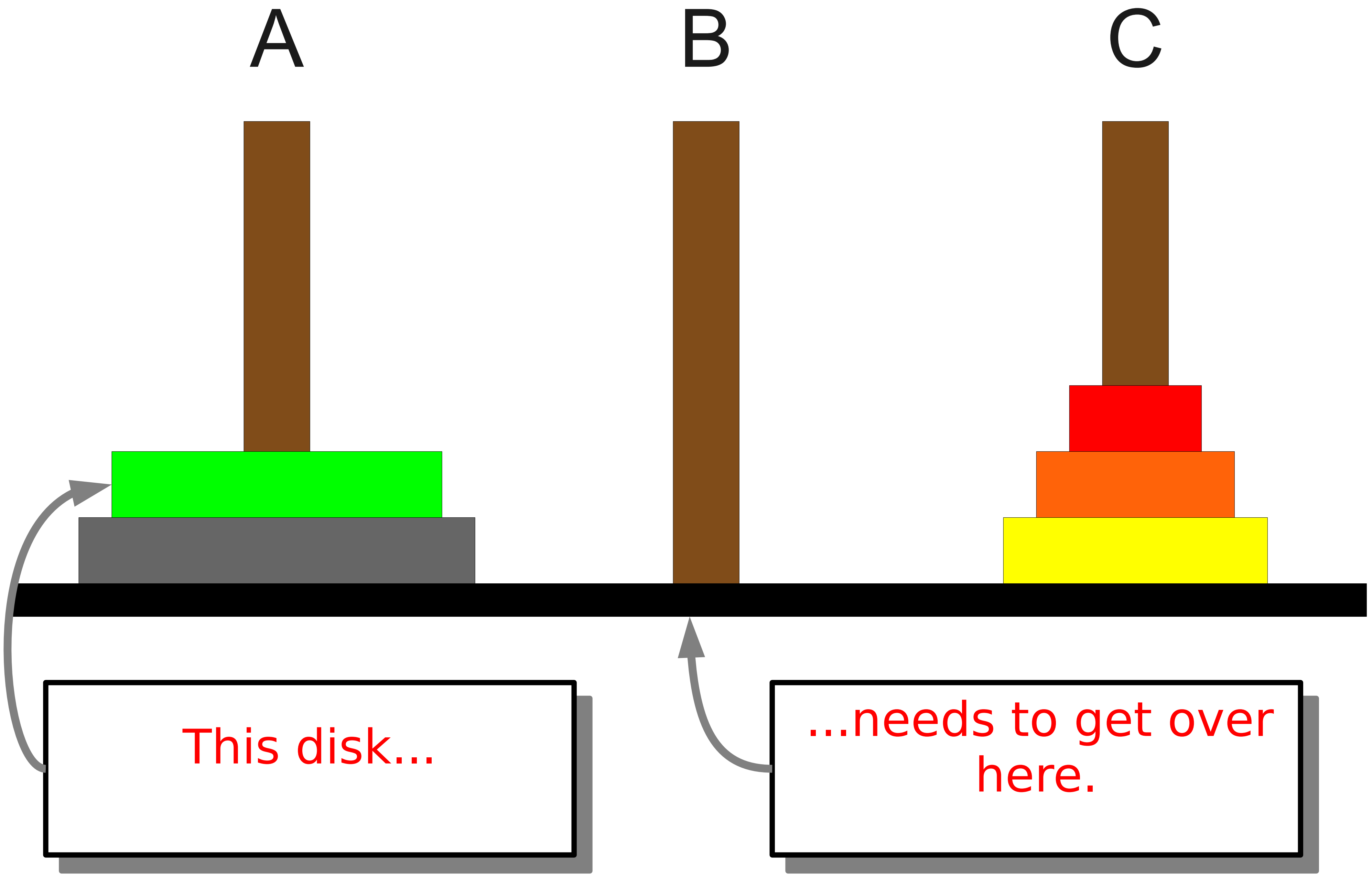
Solving the Towers of Hanoi



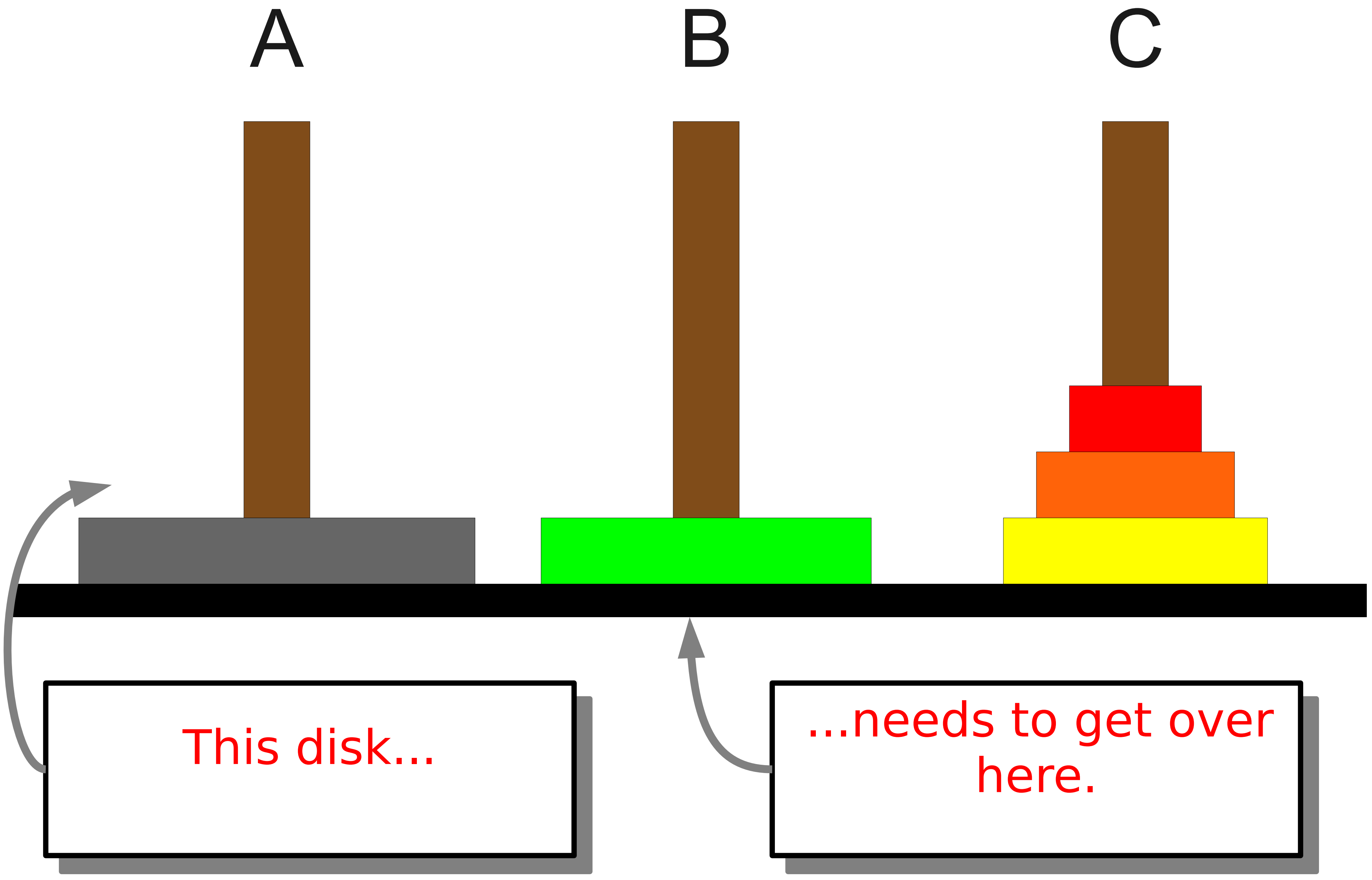
Solving the Towers of Hanoi



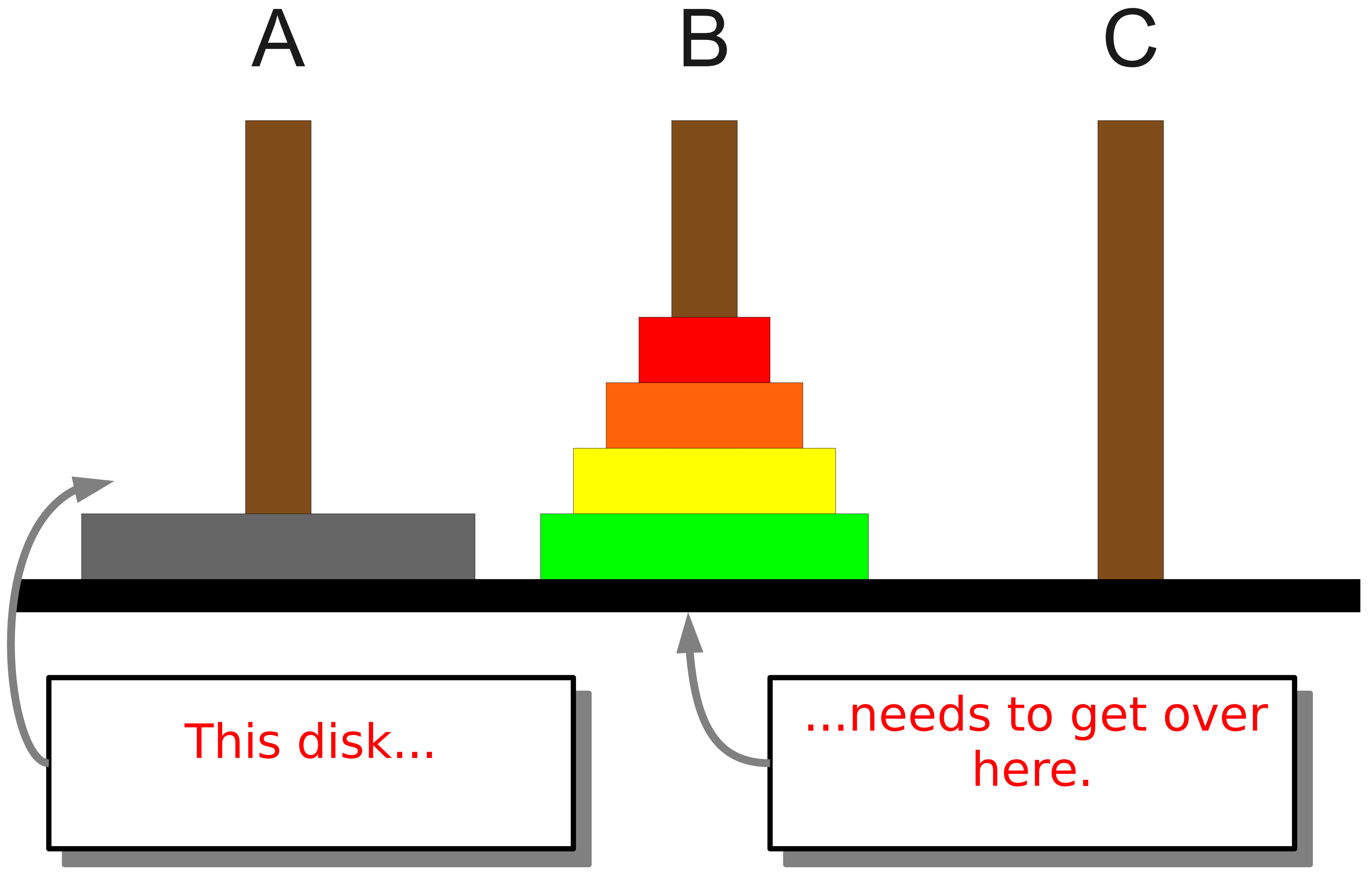
Solving the Towers of Hanoi



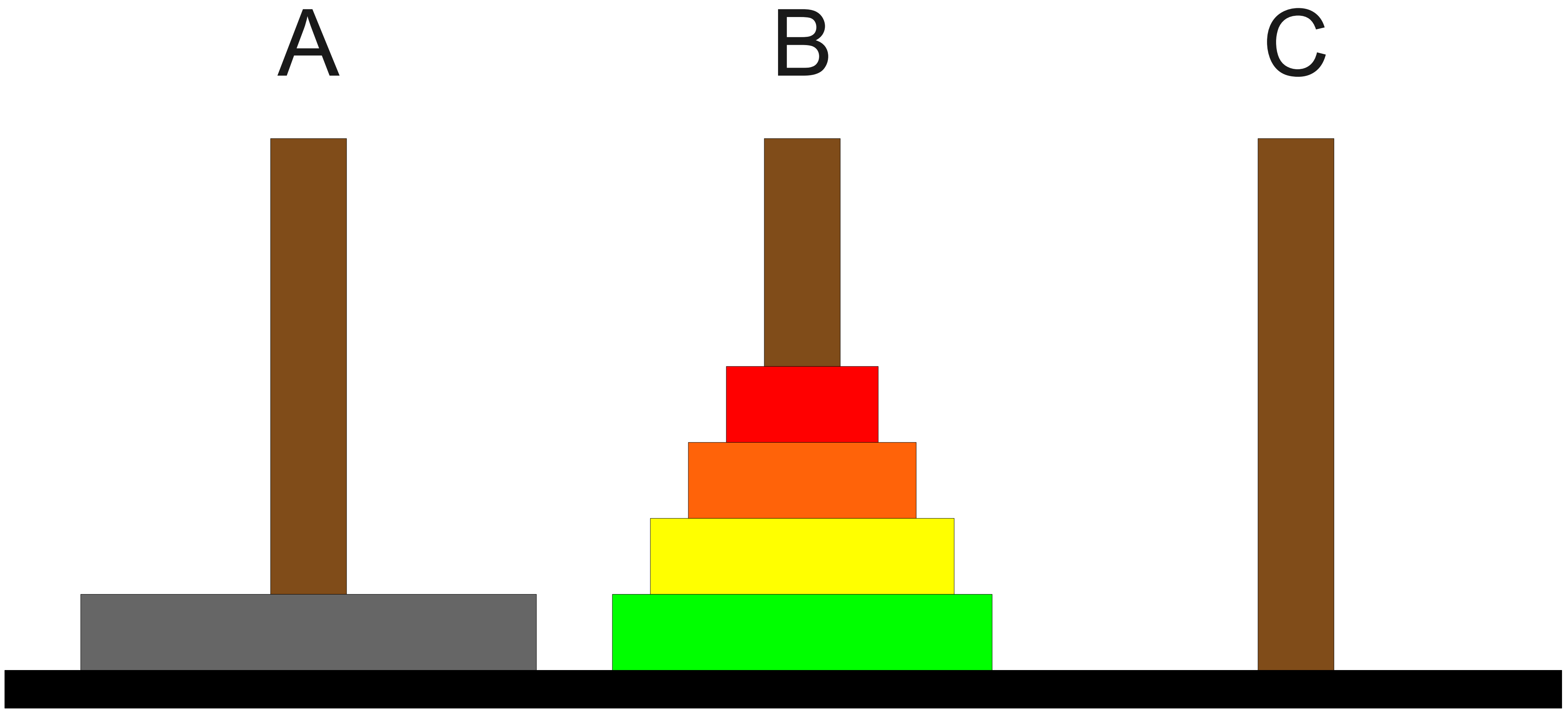
Solving the Towers of Hanoi



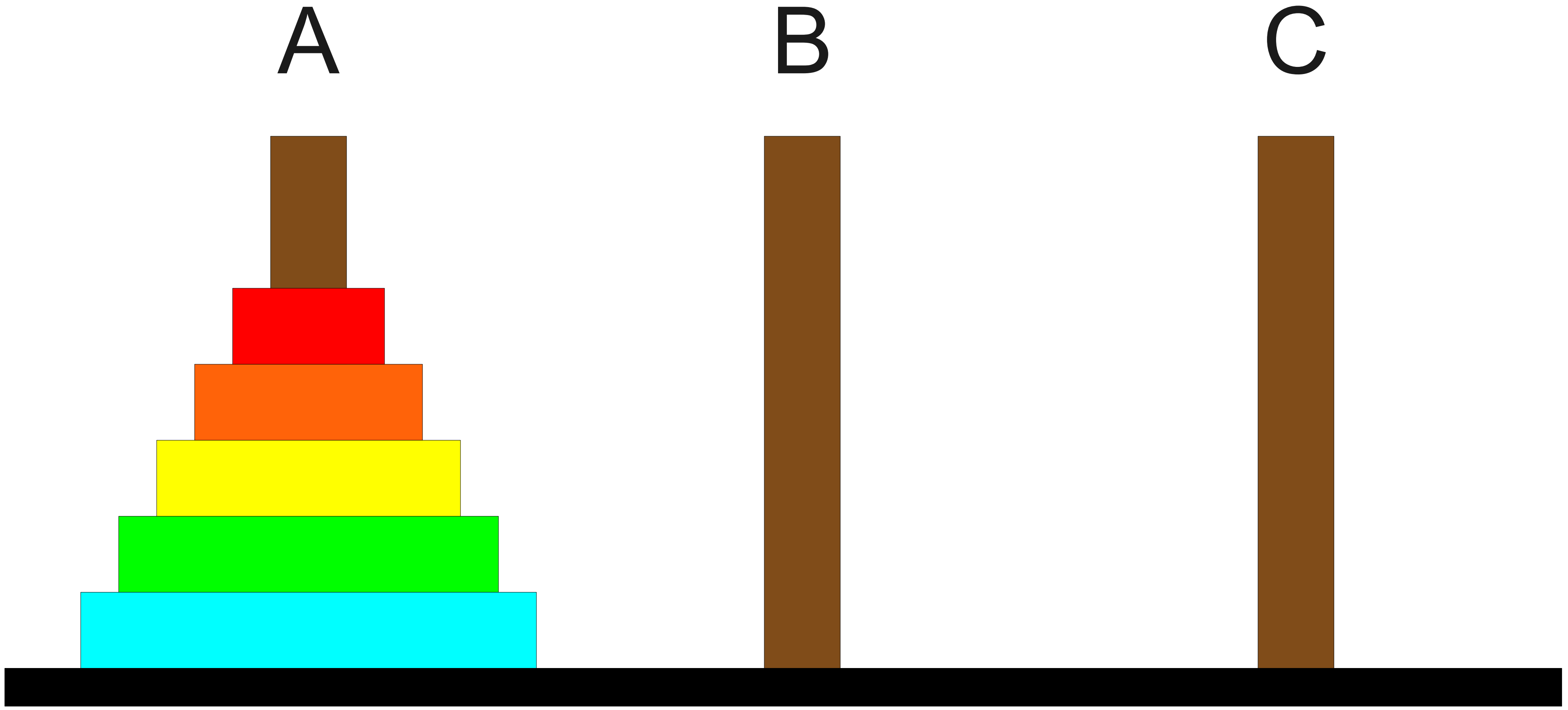
Solving the Towers of Hanoi



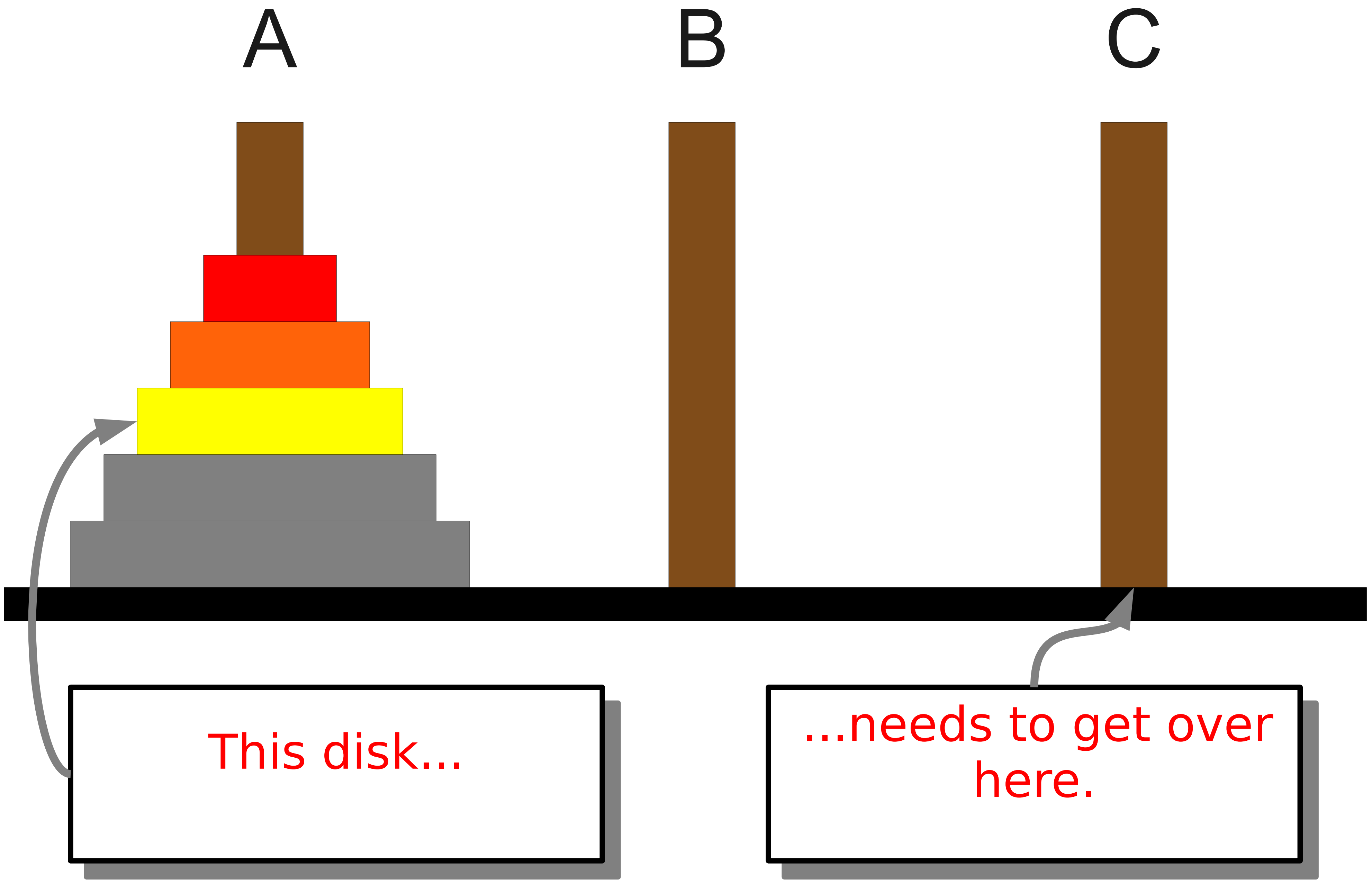
Solving the Towers of Hanoi



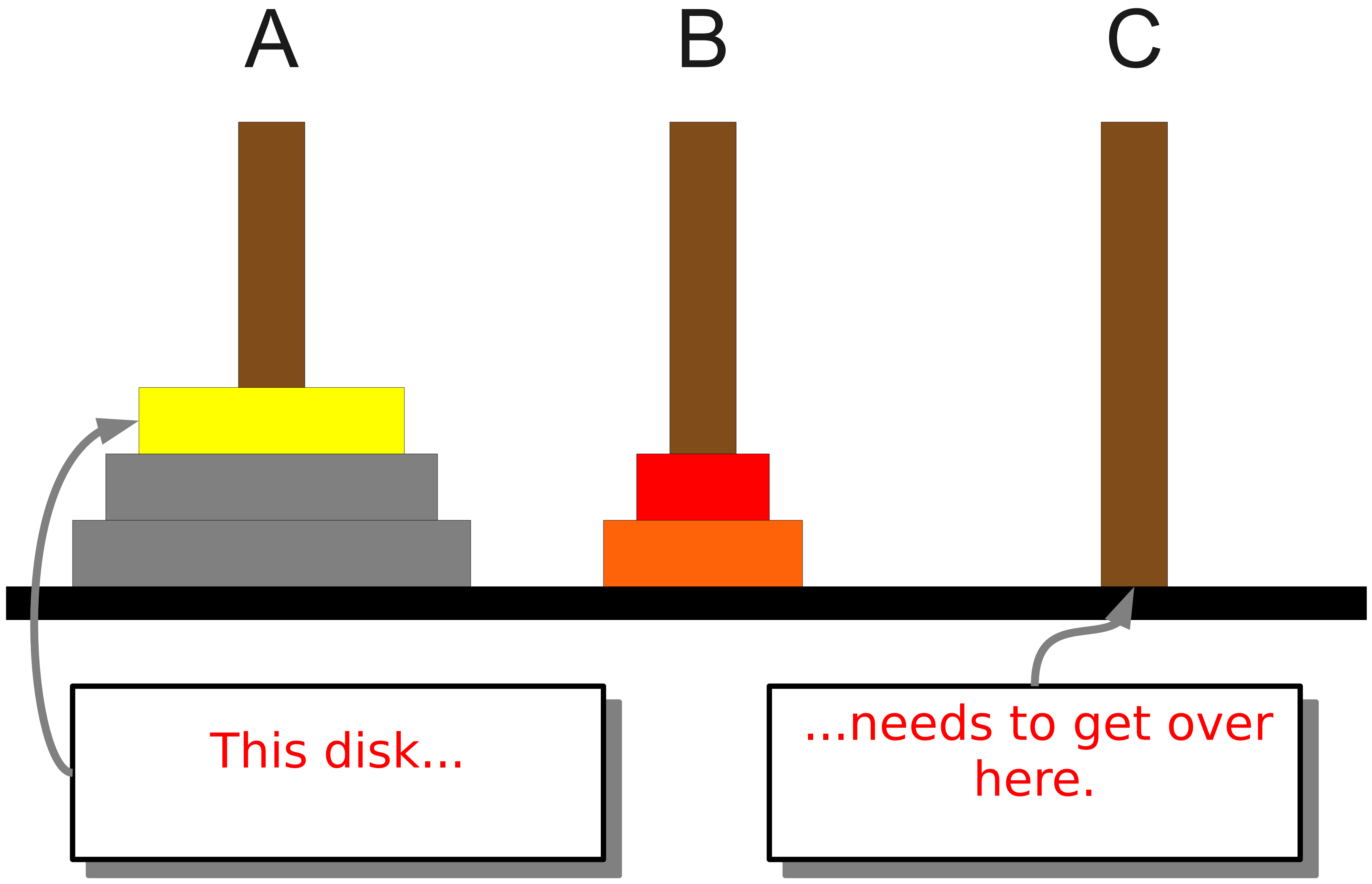
Solving the Towers of Hanoi



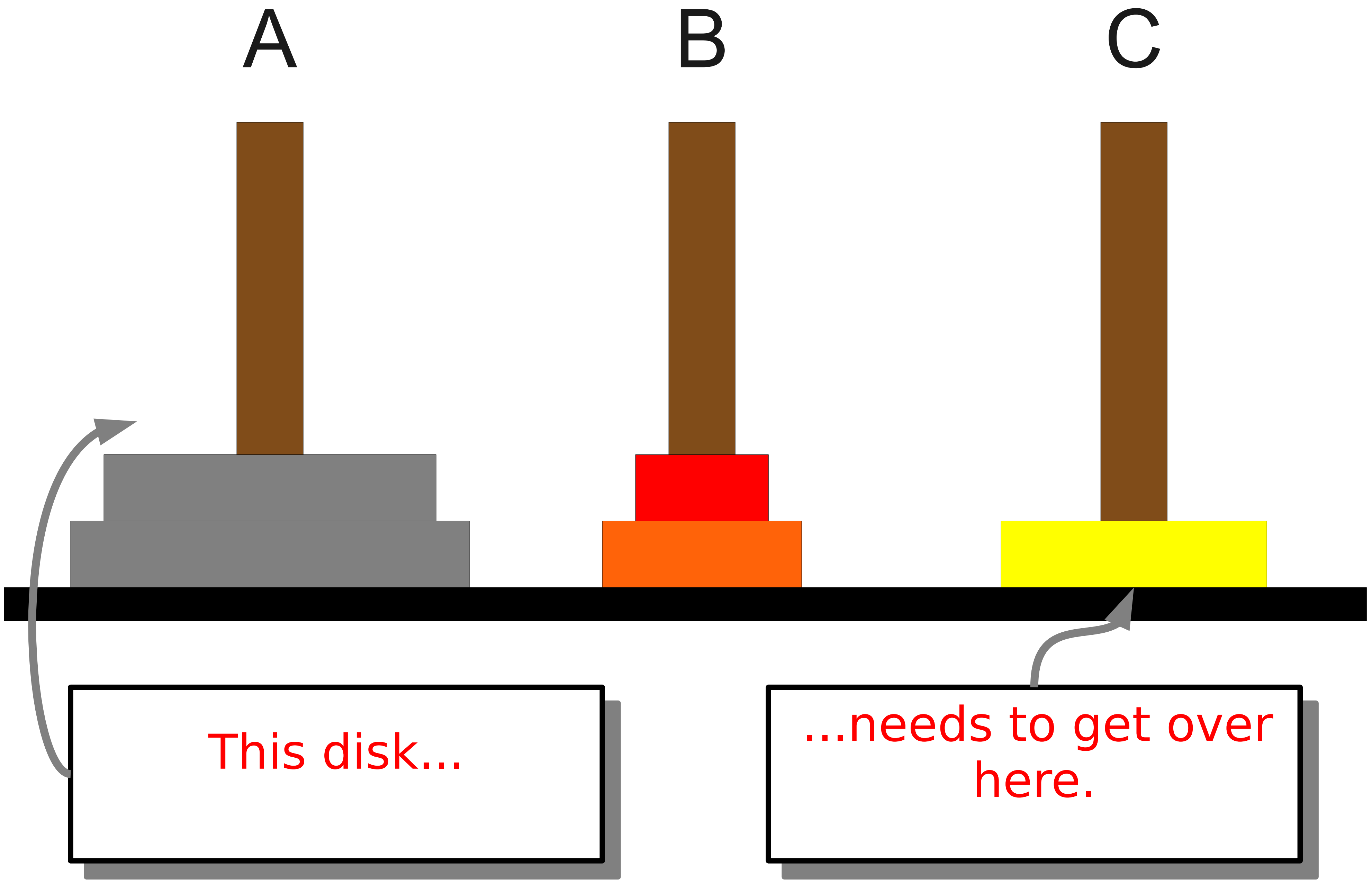
Solving the Towers of Hanoi



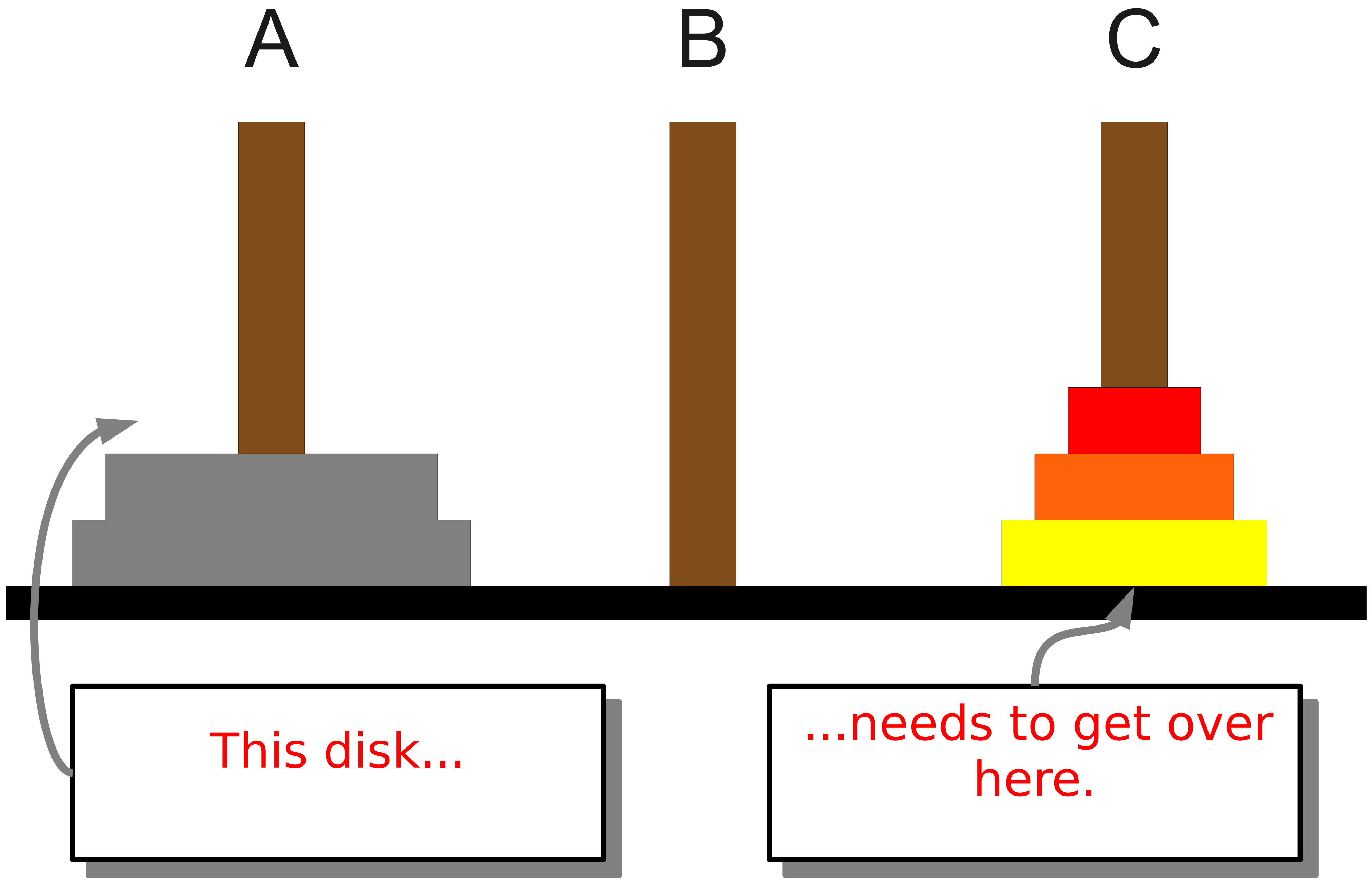
Solving the Towers of Hanoi



Solving the Towers of Hanoi



Solving the Towers of Hanoi



The Recursive Decomposition

- To get a tower of N+1 disks from spindle X to spindle Y, using Z as a temporary:
  - **Recursively** move the top N disks from spindle X to spindle Z, using Y as a temporary.
  - Move the N+1st disk from X to Y.
  - **Recursively** move the N disks from spindle Z to spindle Y, using X as a temporary.

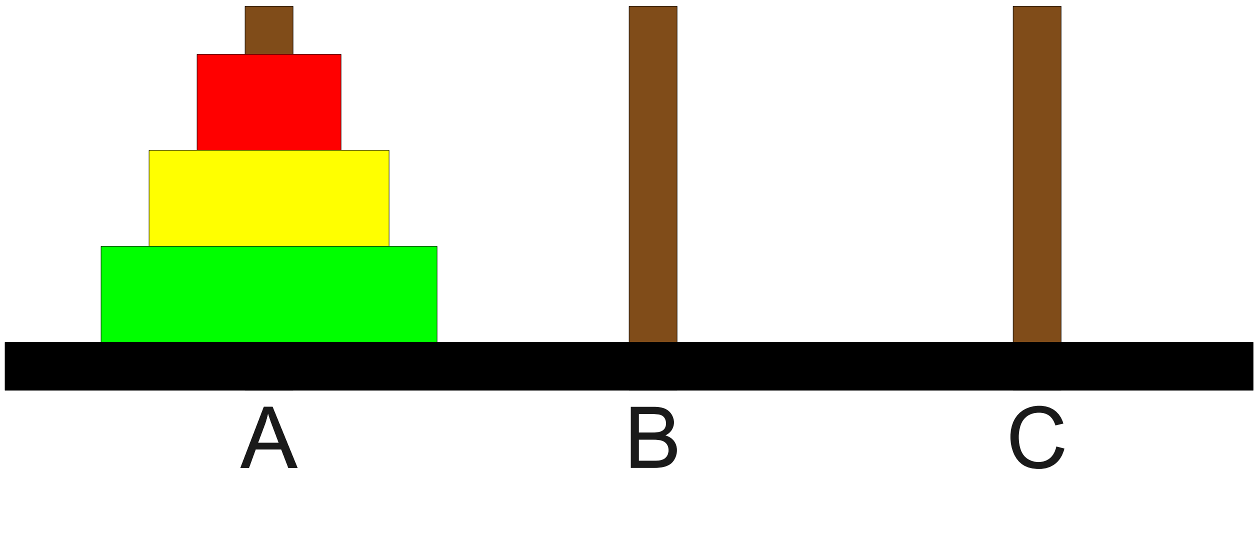
The Base Case

- We need to find a very simple case that we can solve directly in order for the recursion to work.
- If the tower has size one, we can just move that single disk from the source to the destination.

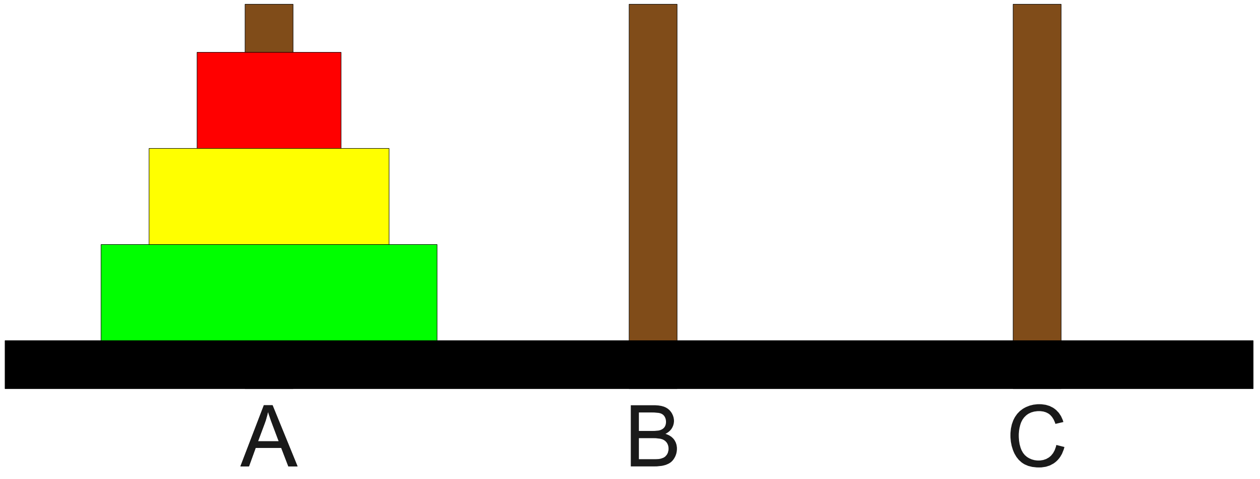
And now, the solution...

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

```c
int main() {
    moveTower(3, 'a', 'c', 'b');
}
```



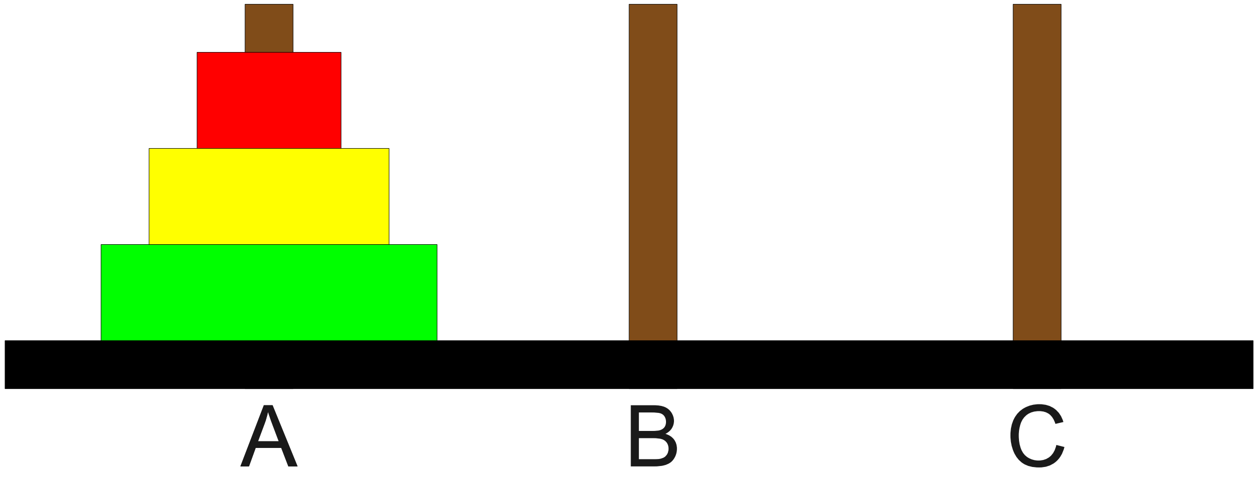
```
int main() {
    moveTower(3, 'a', 'c', 'b');
}
```



```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

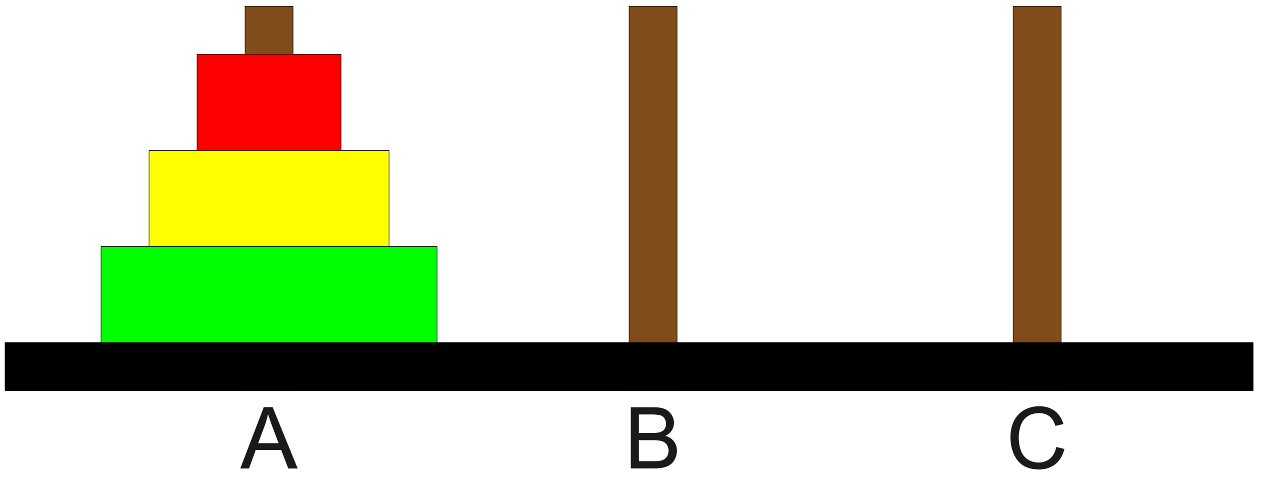
n	3	from	a	to	c	temp	b



```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

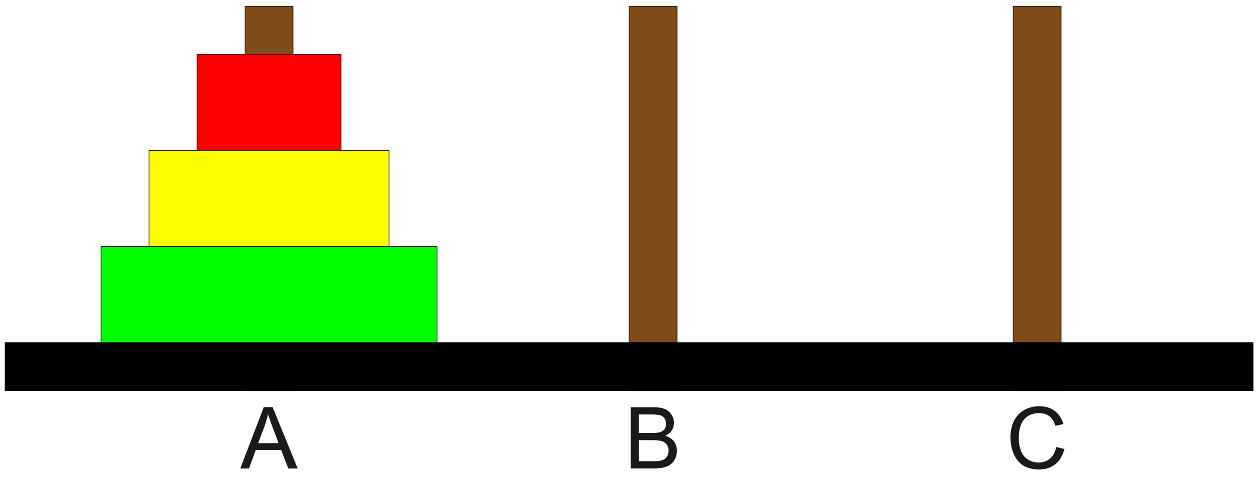
n		from		to		temp	
	3		a		c		b



```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

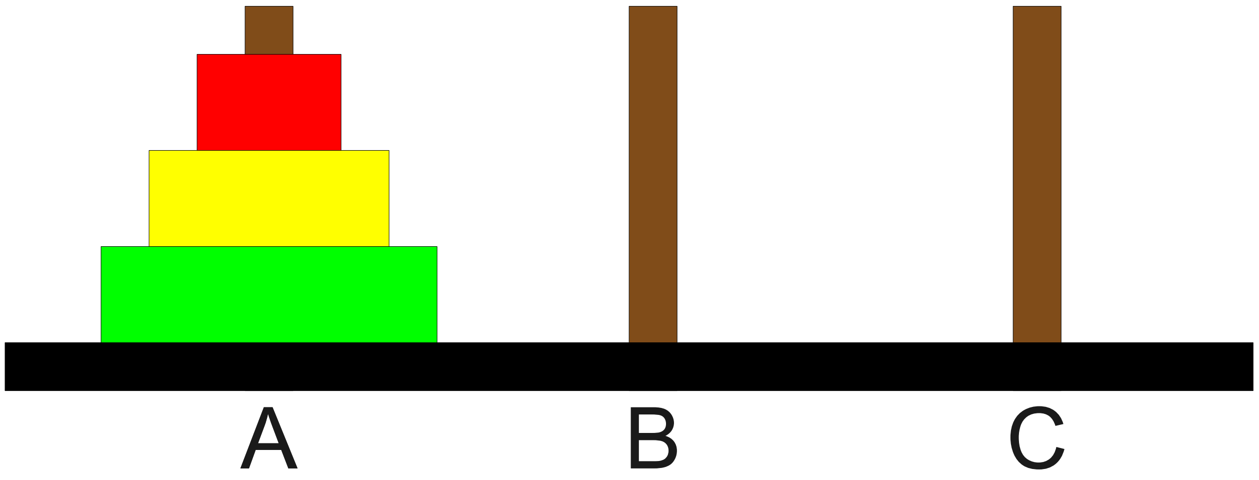
n	from	to	temp
3	a	c	b



```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	3	from	a	to	c	temp	b

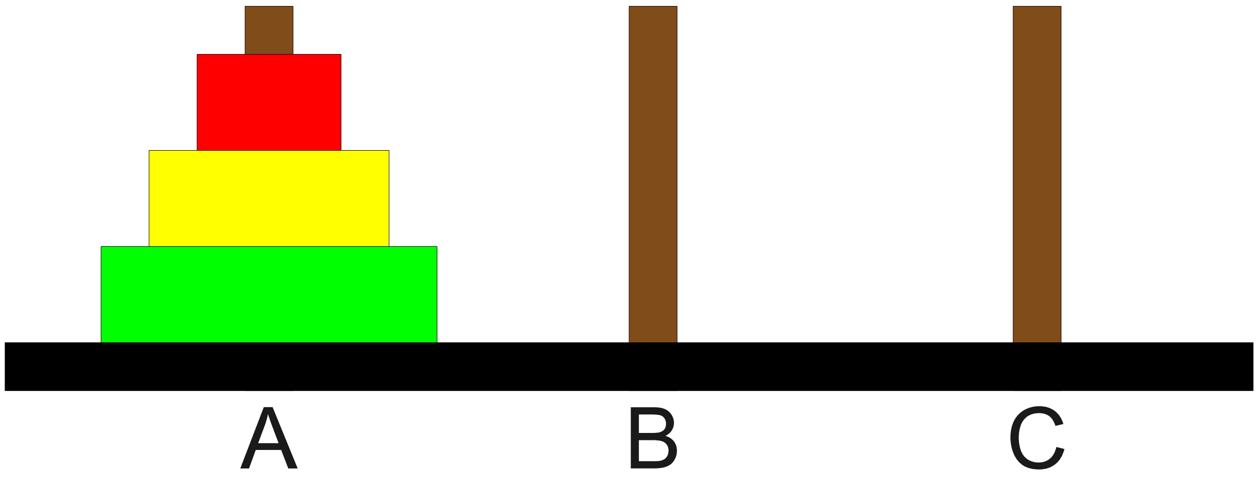


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n `2`   from `a`   to `b`   temp `c`

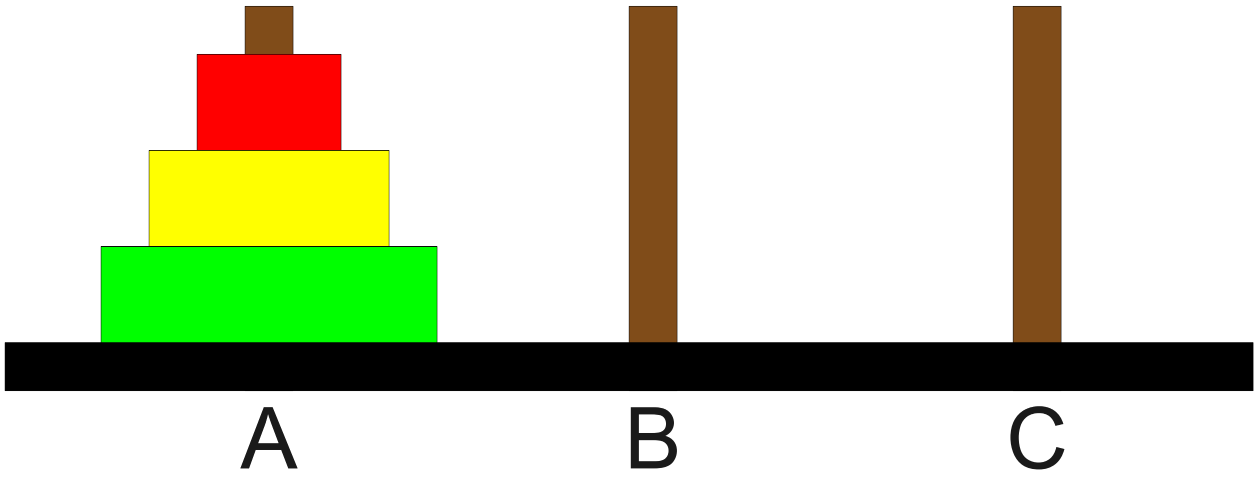


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n 2 from a to b temp c

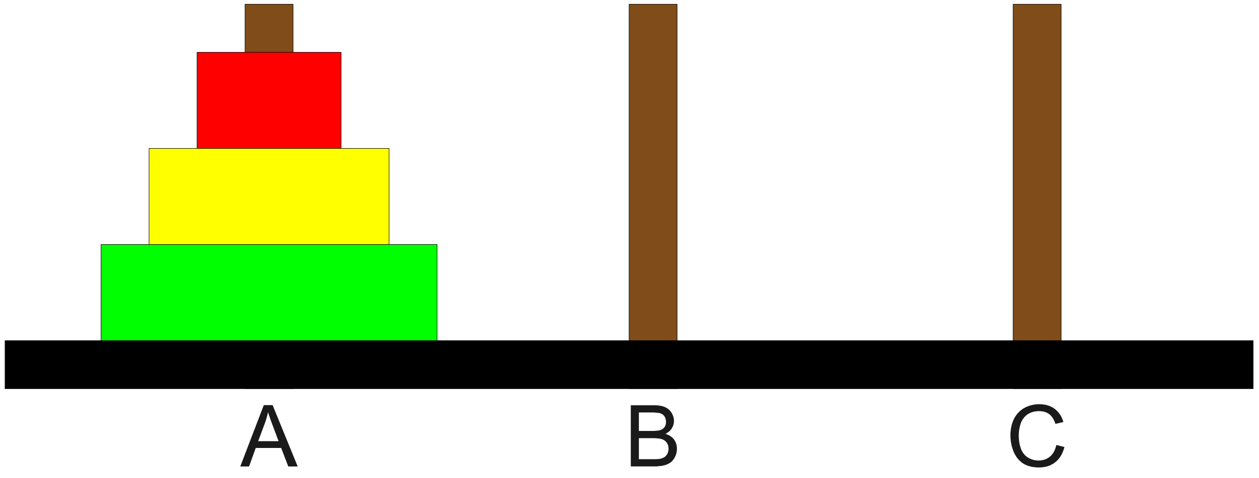


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	2	from	a	to	b	temp	c

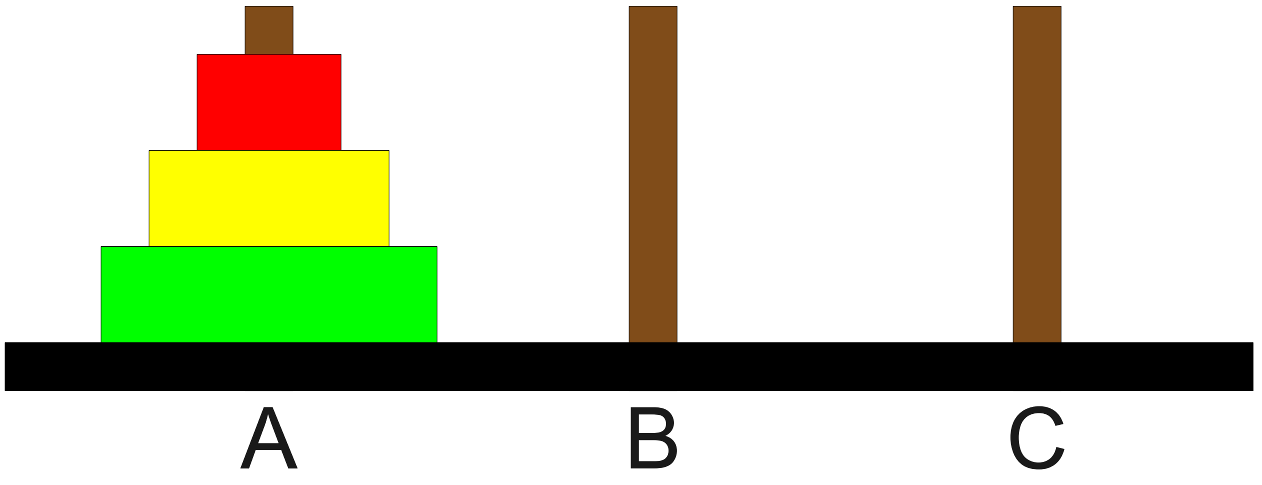


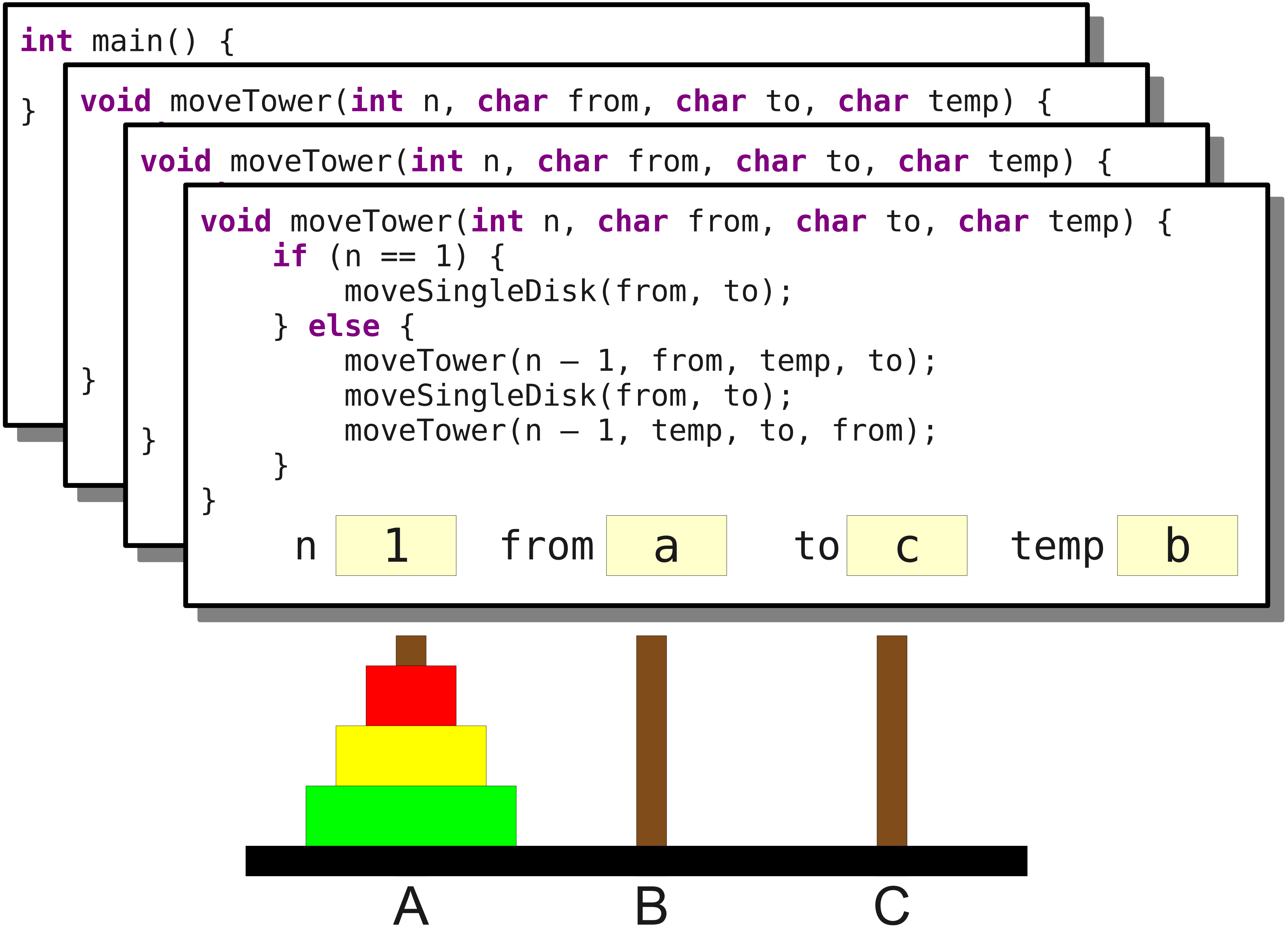
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n 2 from a to b temp c





```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

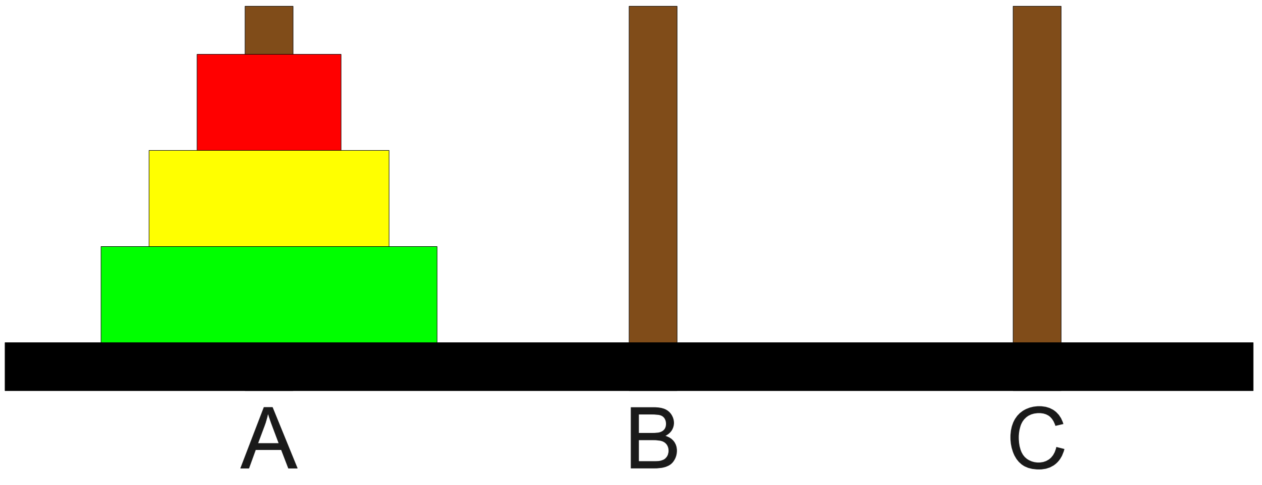
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
1	a	c	b



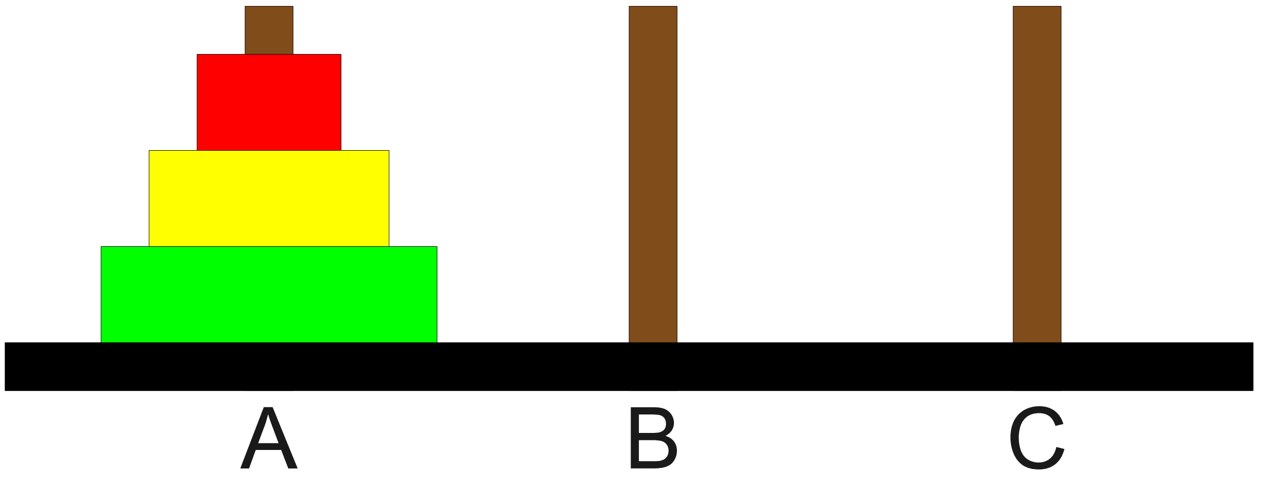
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
1	a	c	b



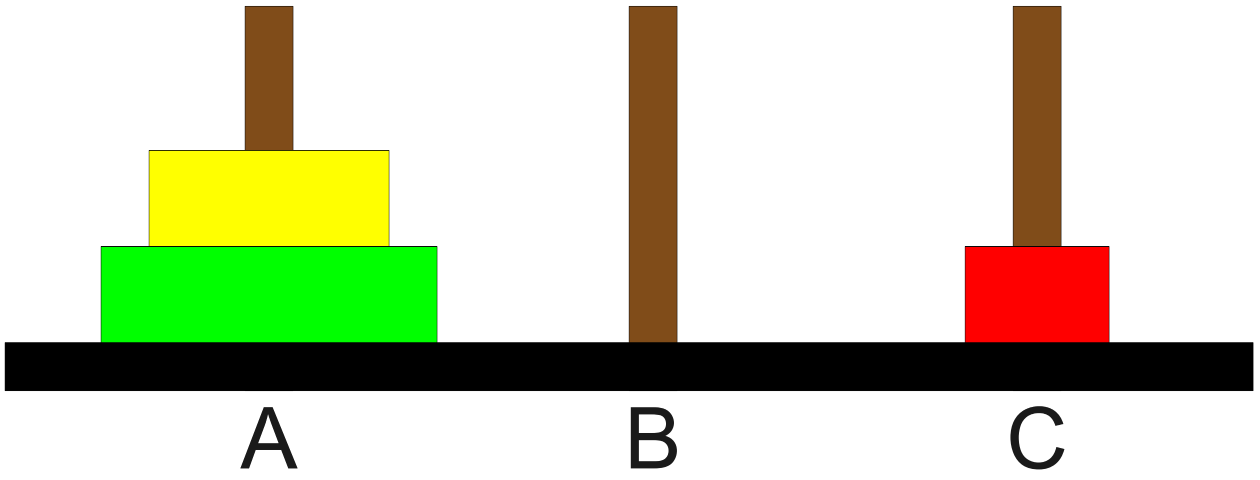
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
1	a	c	b

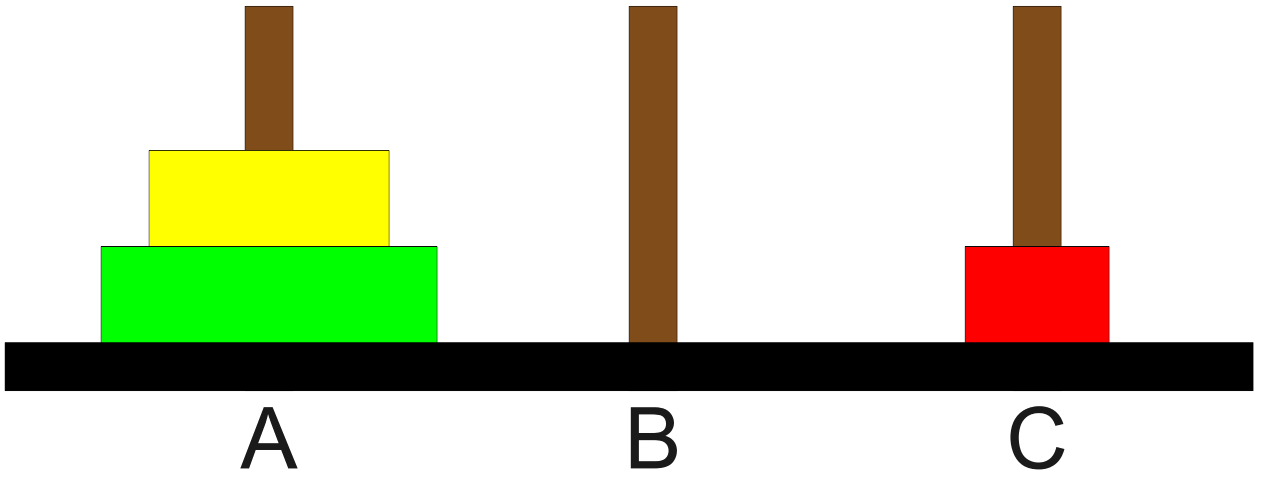


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	2	from	a	to	b	temp	c

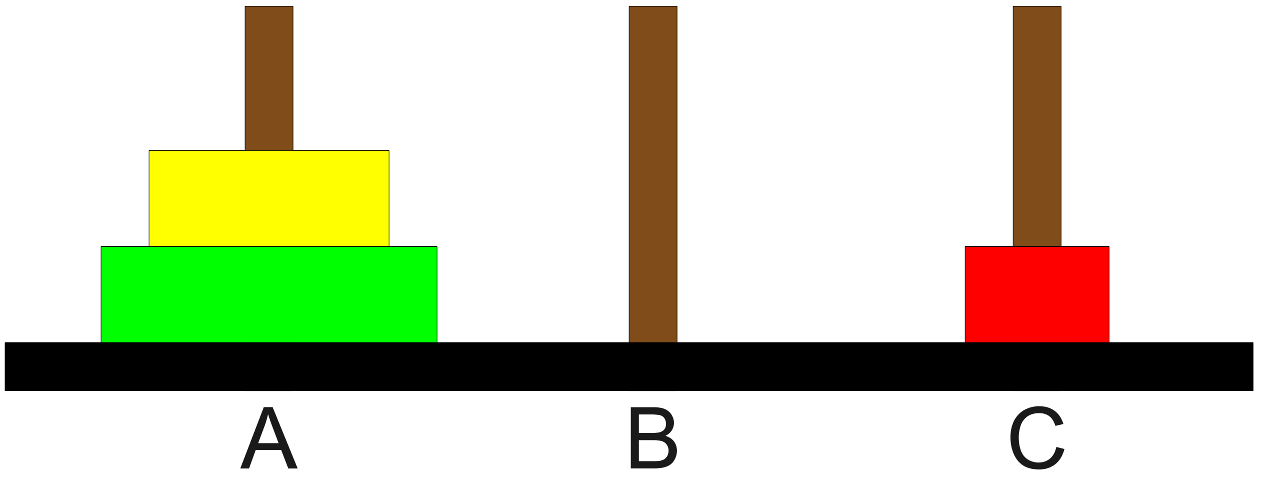


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	2	from	a	to	b	temp	c

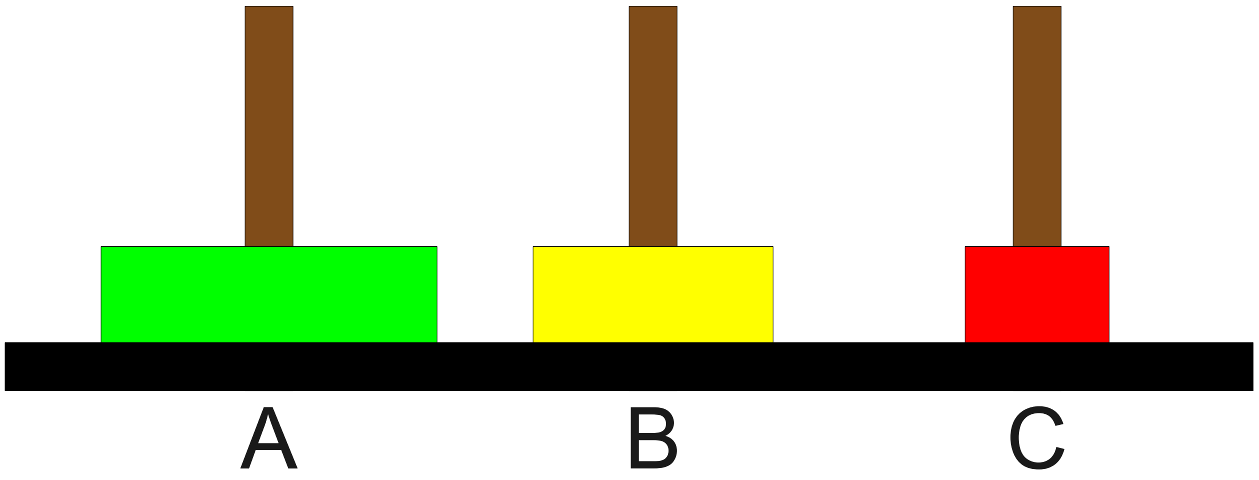


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
2	a	b	c

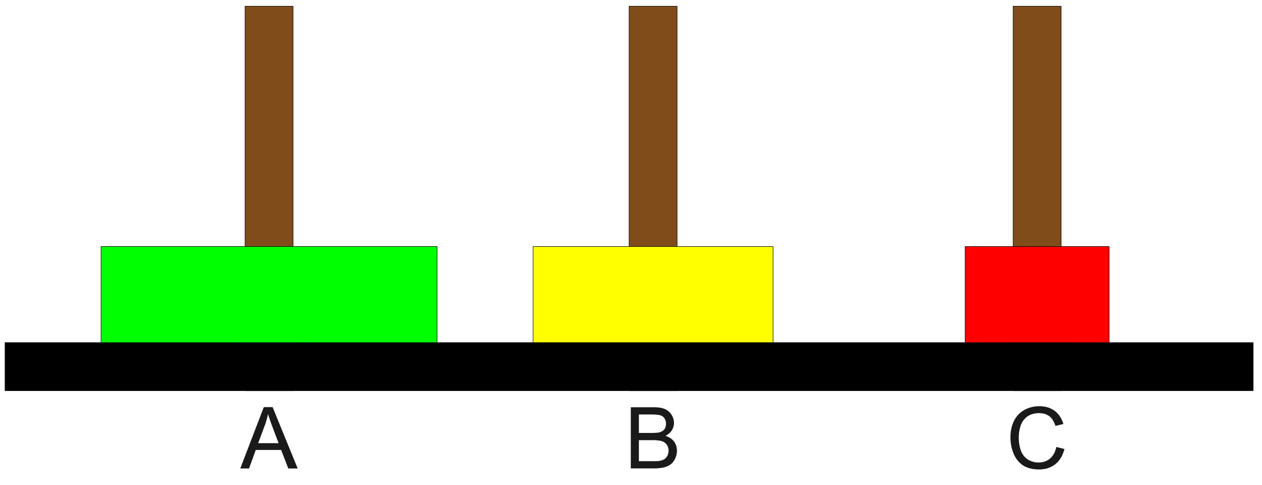


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n 2 from a to b temp c



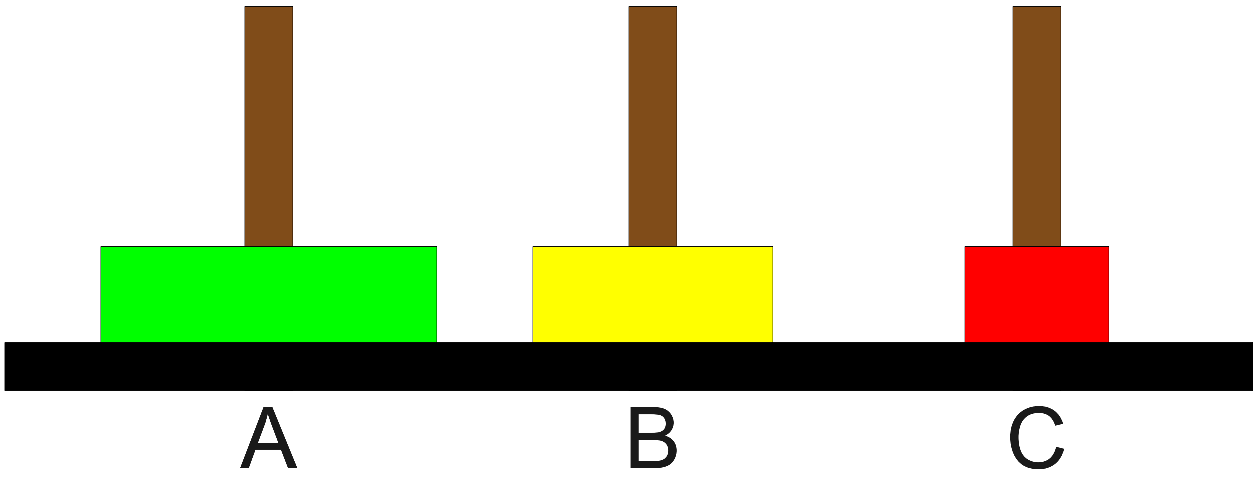
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
1	c	b	a



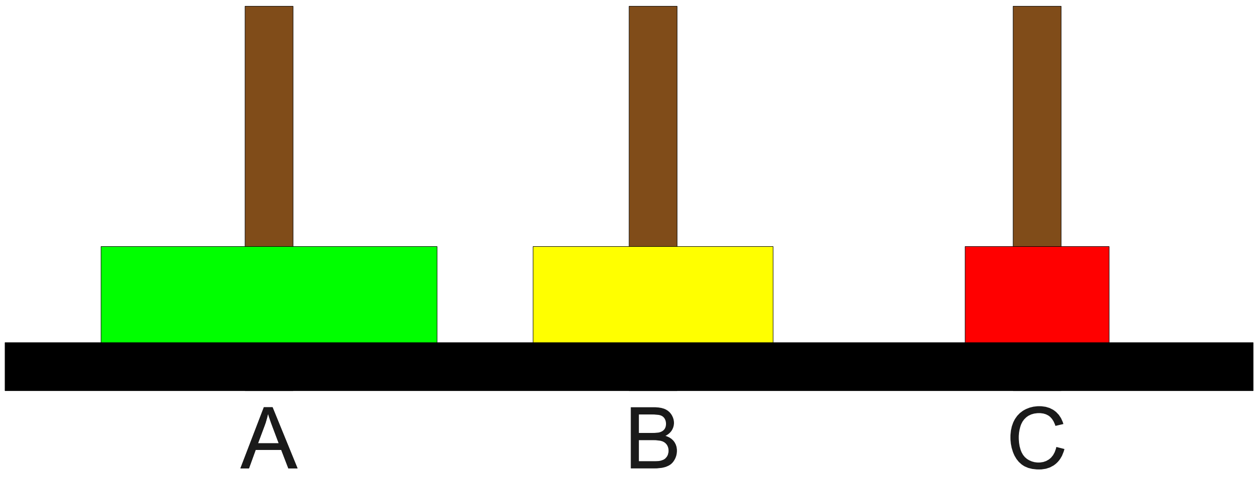
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n 1 from c to b temp a



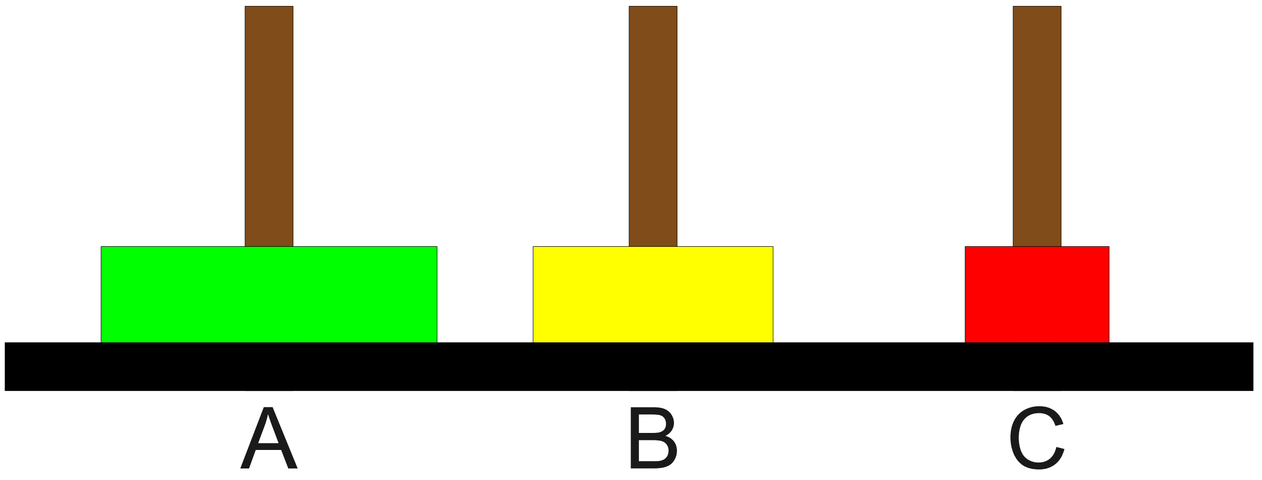
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
1	c	b	a



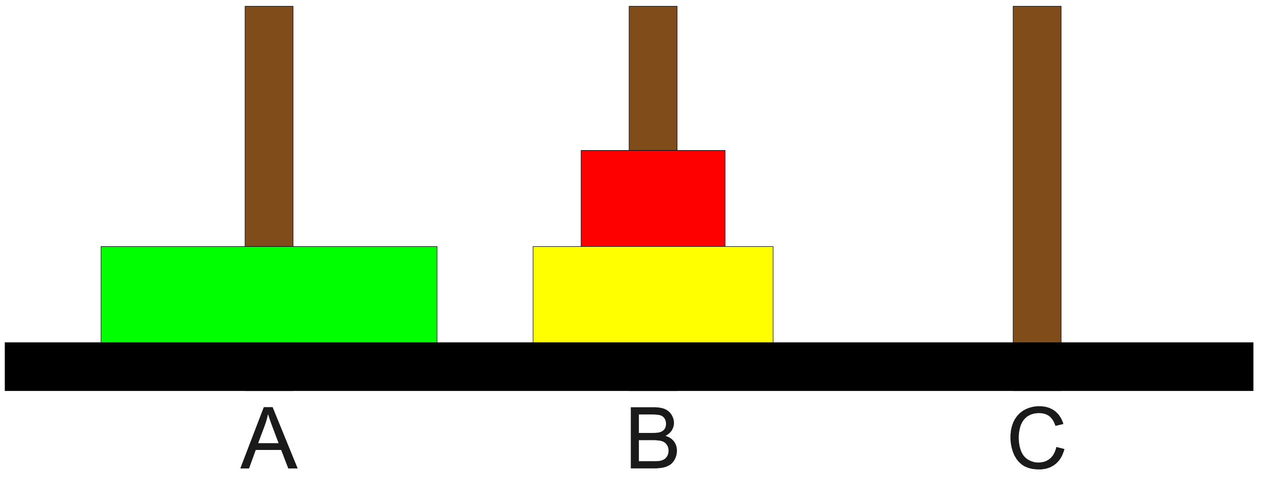
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
1	c	b	a

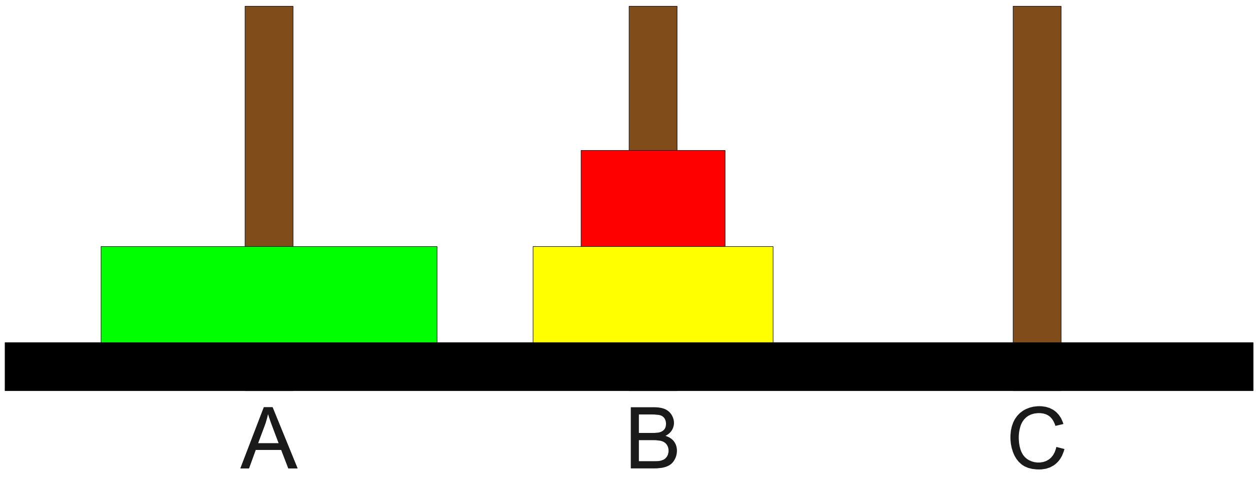


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

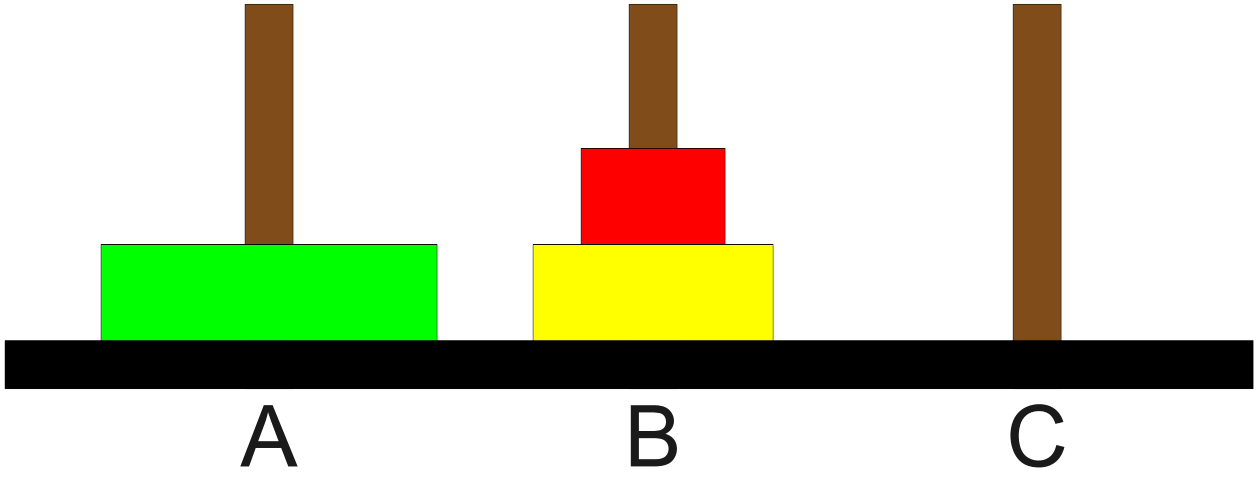
n	2	from	a	to	b	temp	c



```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

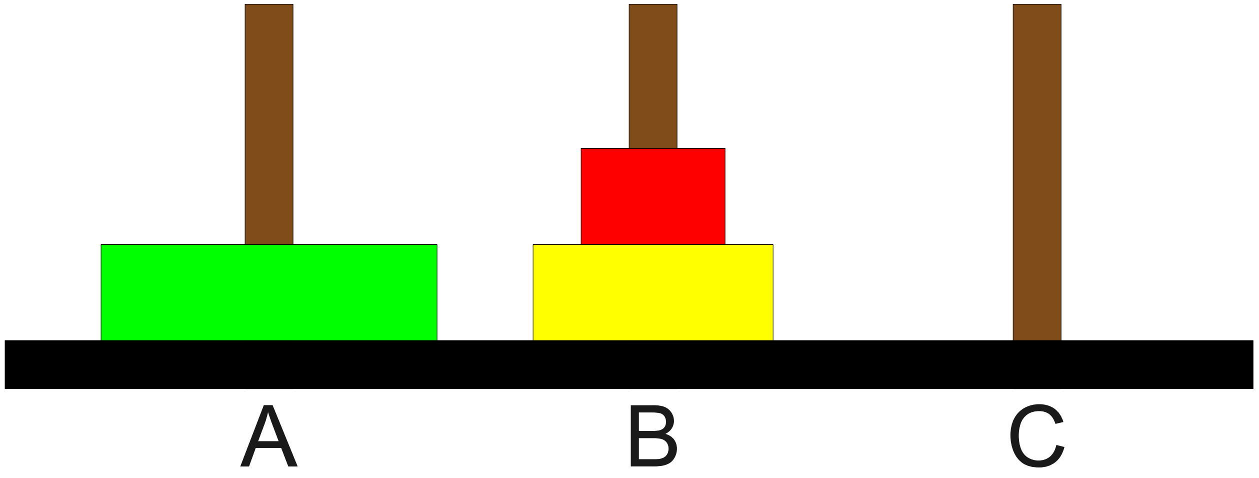
n	from	to	temp
3	a	c	b



```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

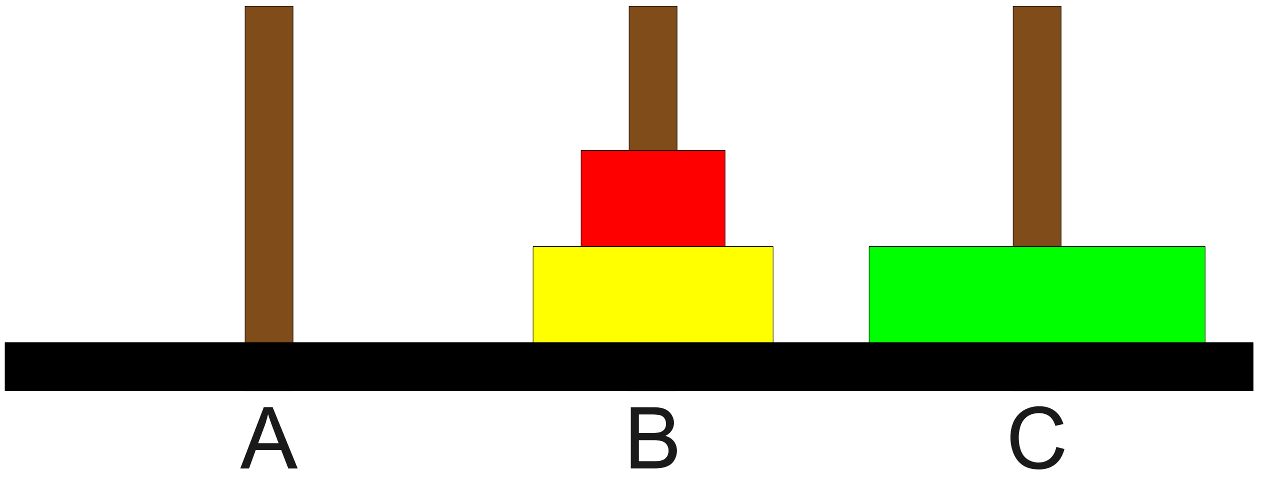
n	3	from	a	to	c	temp	b



```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

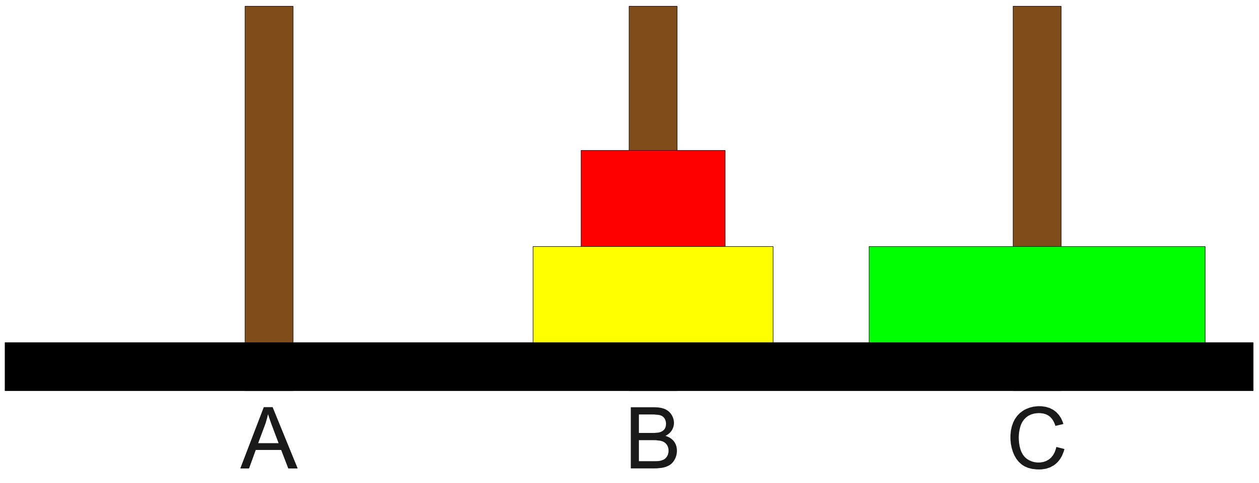
n	from	to	temp
3	a	c	b



```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
3	a	c	b

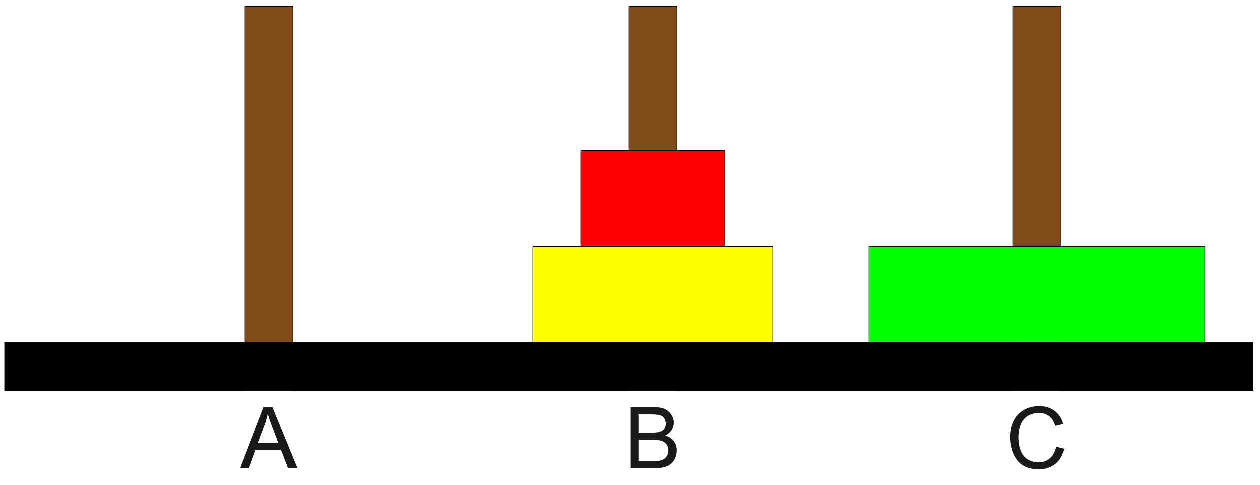


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
2	b	c	a

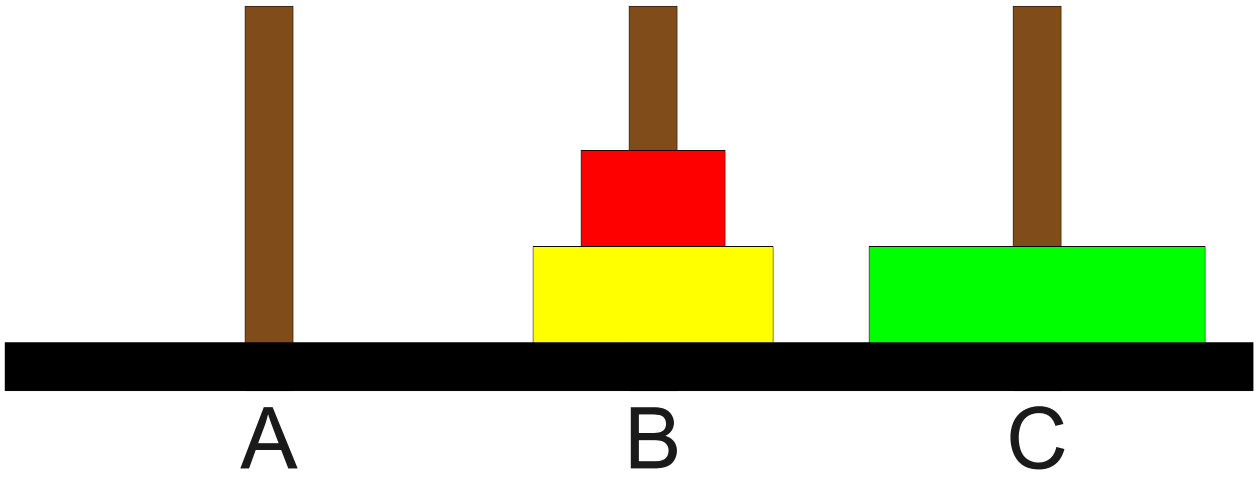


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
2	b	c	a

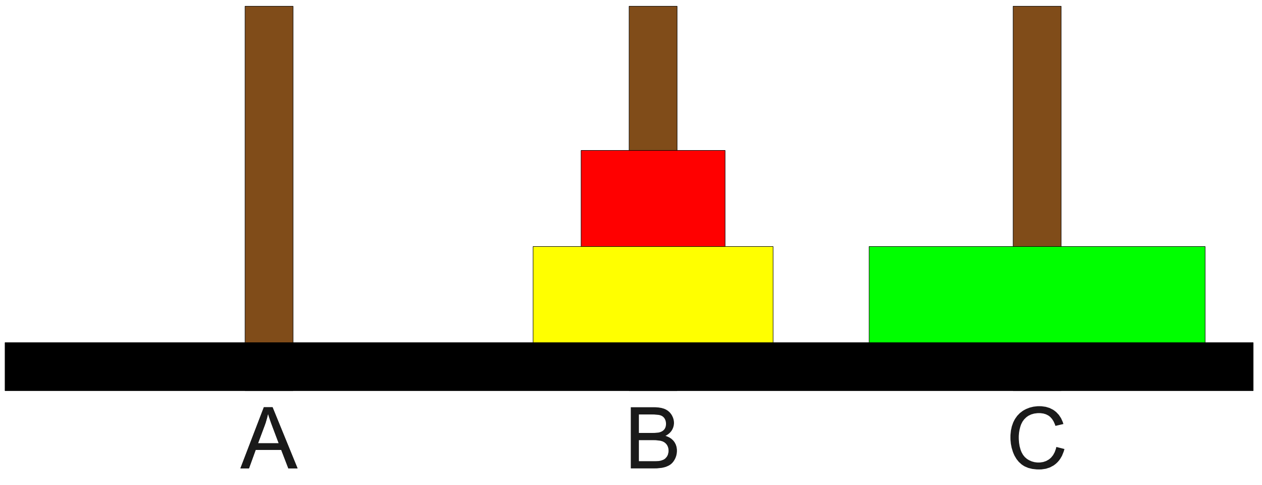


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	2	from	b	to	c	temp	a

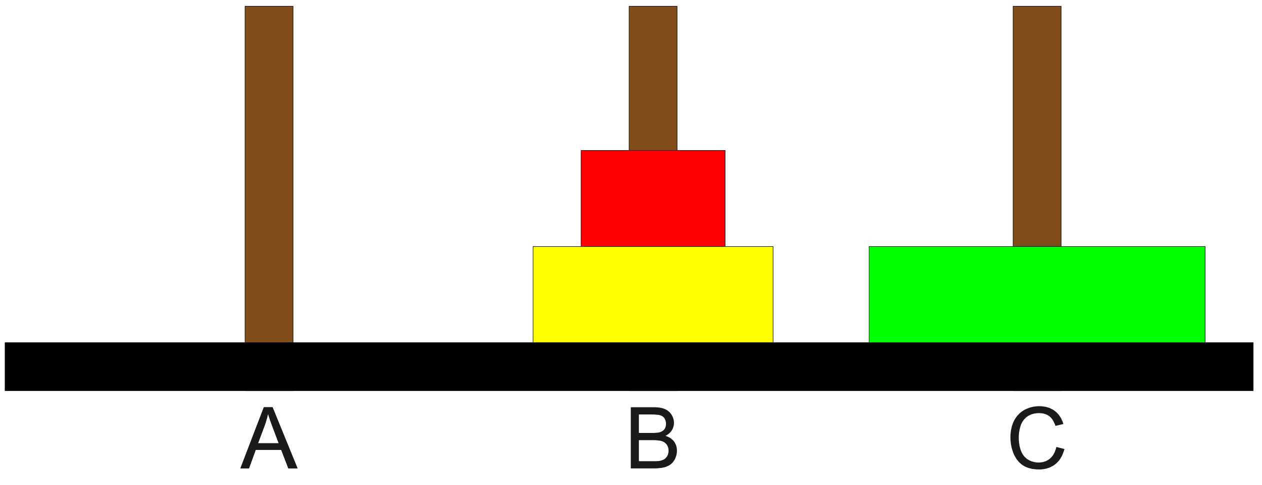


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
2	b	c	a



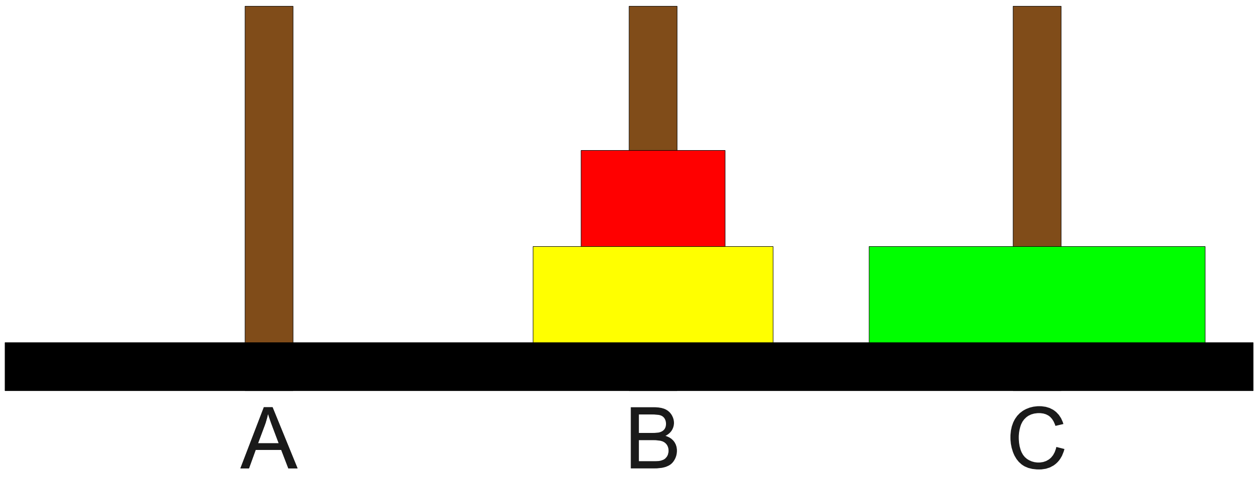
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
1	b	a	c



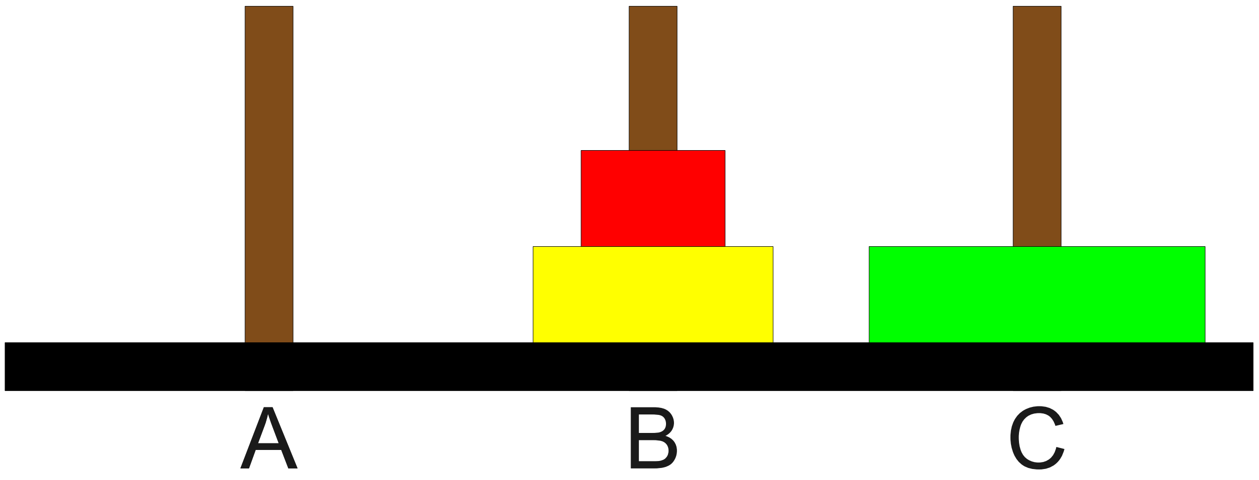
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
1	b	a	c



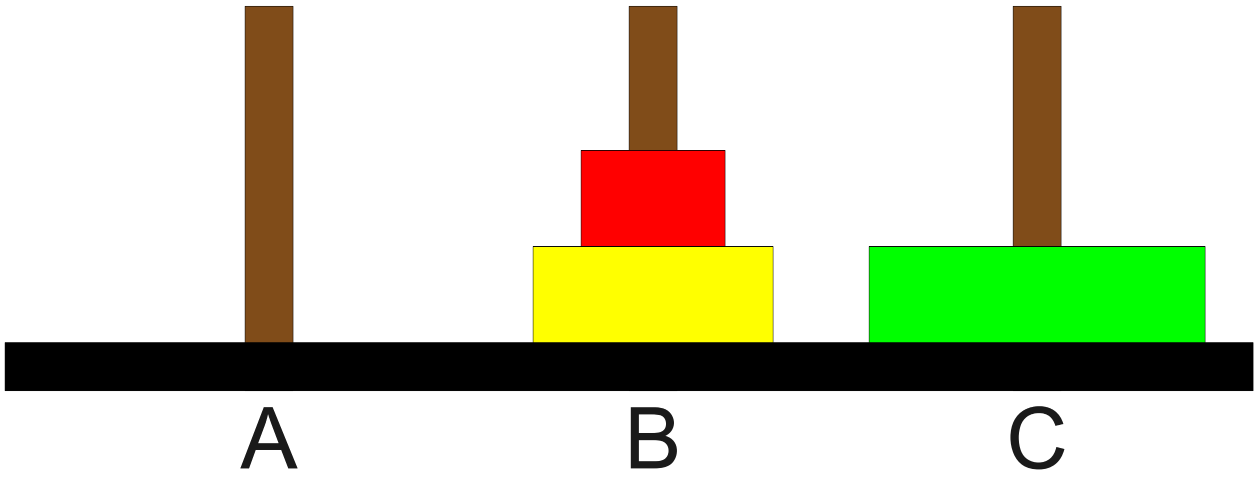
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
1	b	a	c



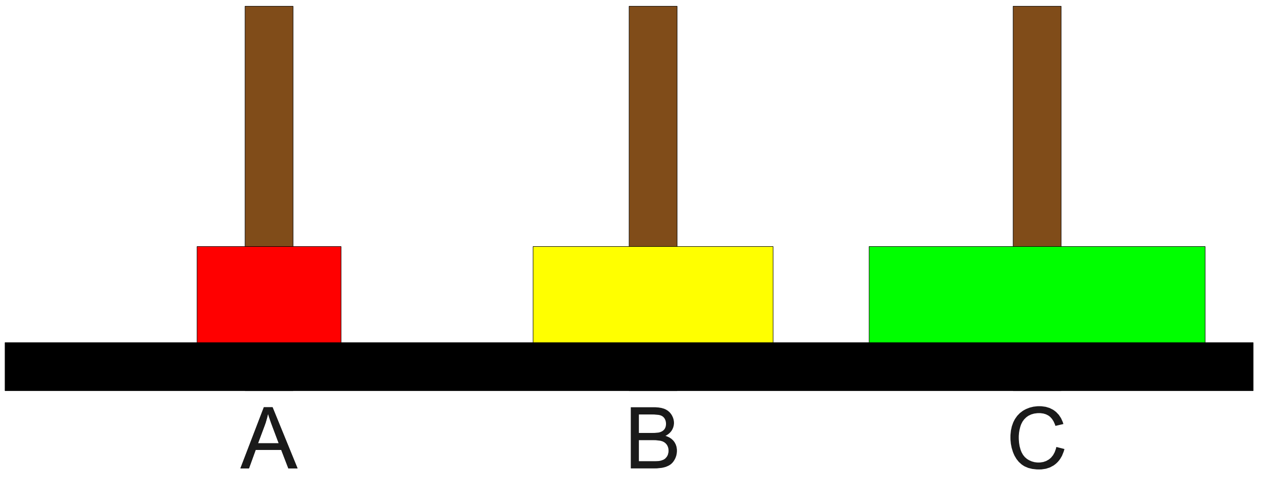
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
1	b	a	c

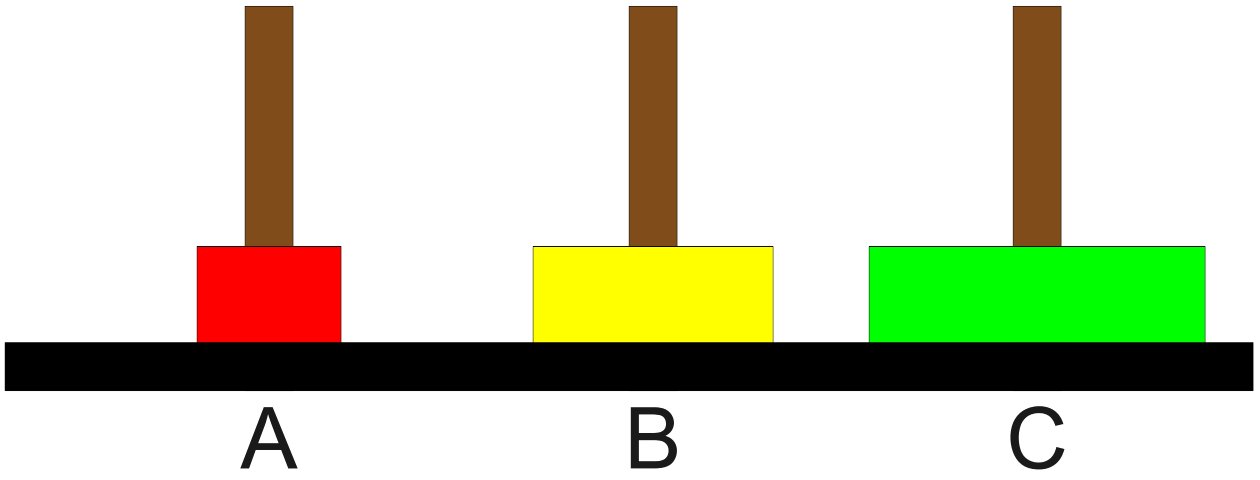


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
2	b	c	a

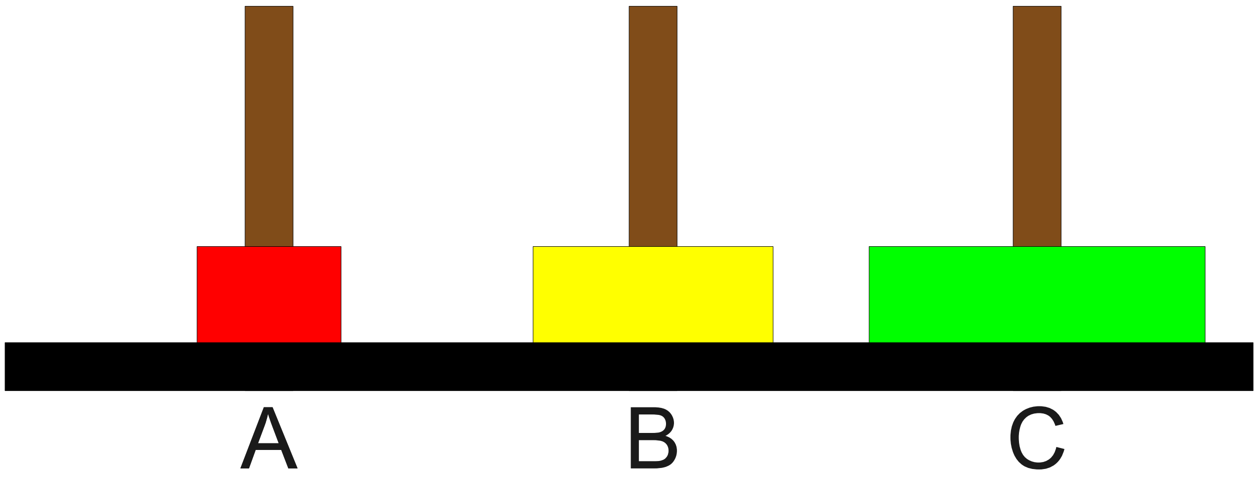


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	2	from	b	to	c	temp	a

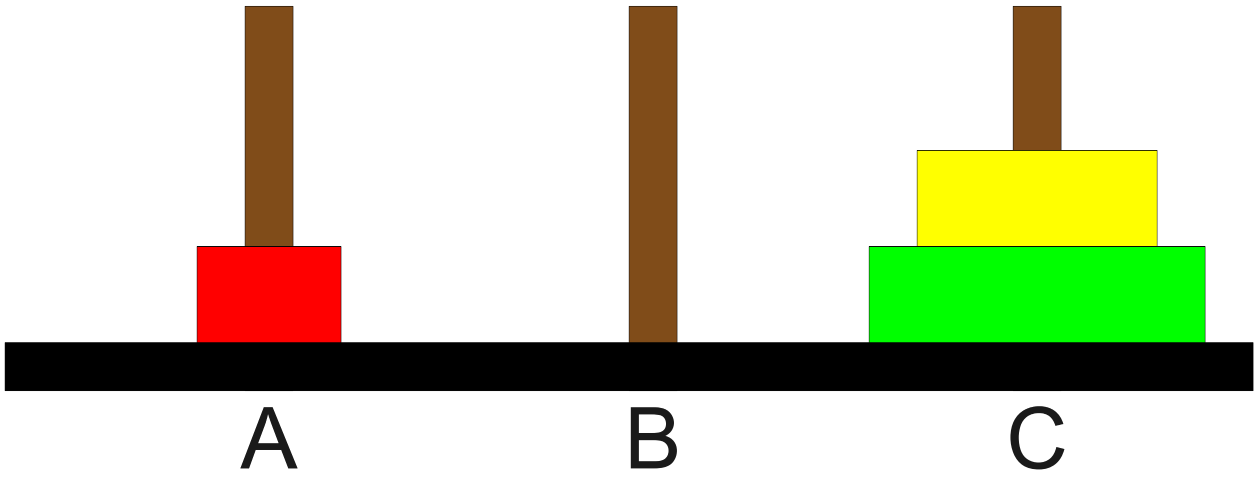


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n 2 from b to c temp a

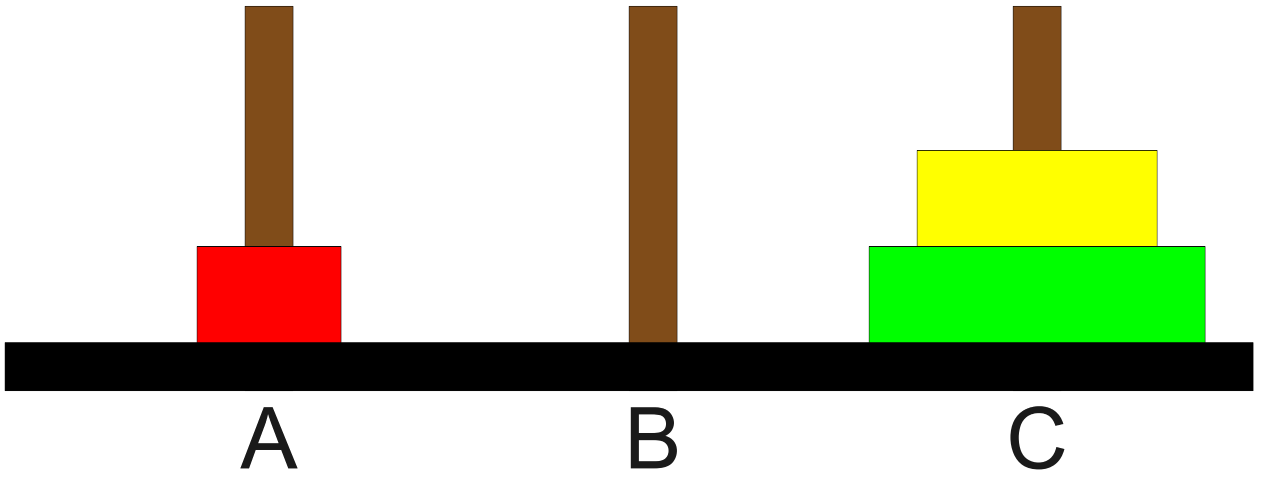


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
2	b	c	a



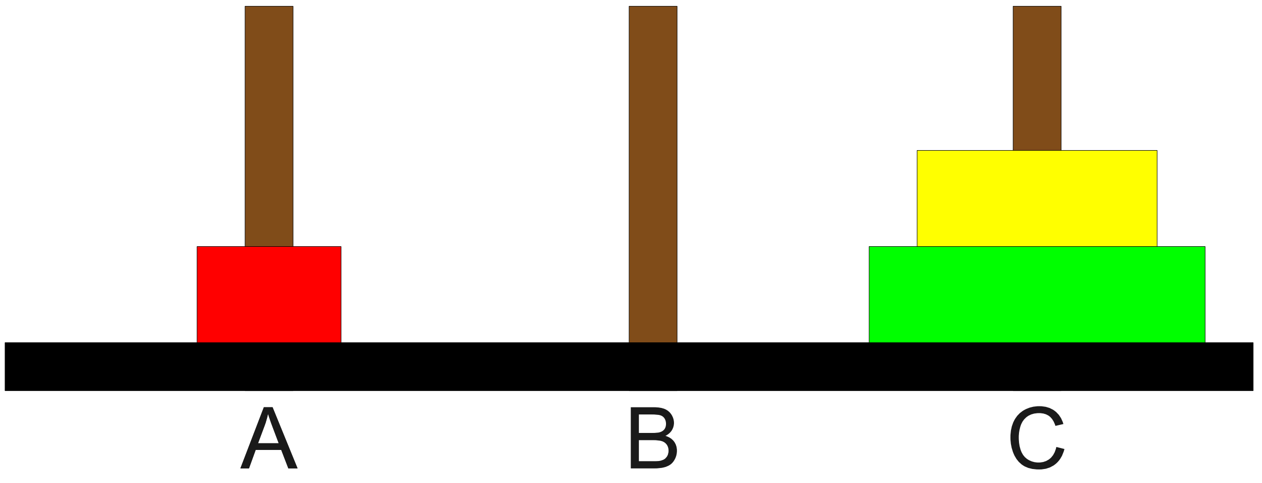
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	1	from	a	to	c	temp	b



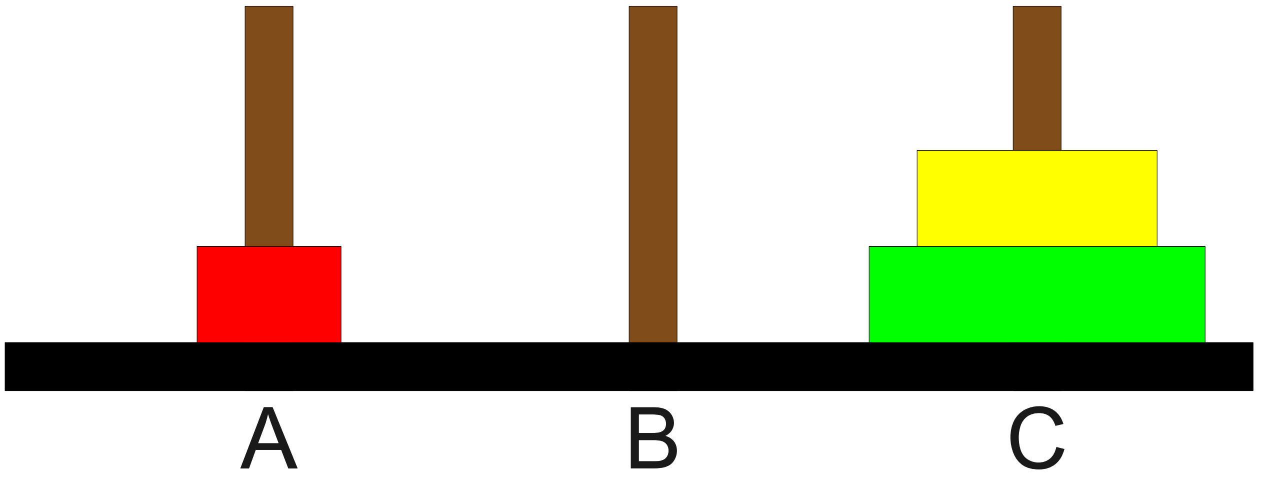
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
1	a	c	b



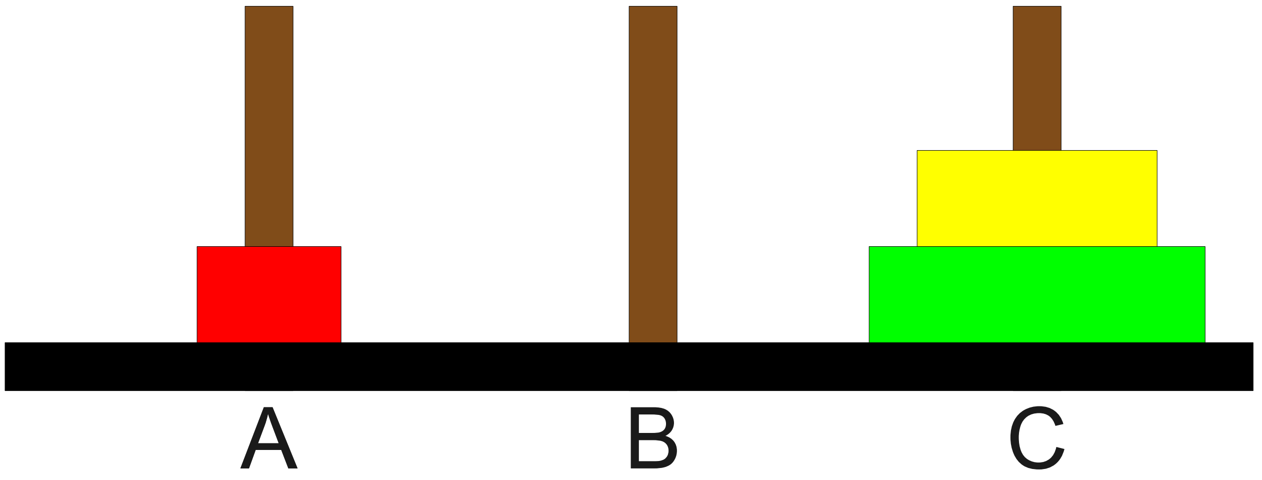
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	from	to	temp
1	a	c	b



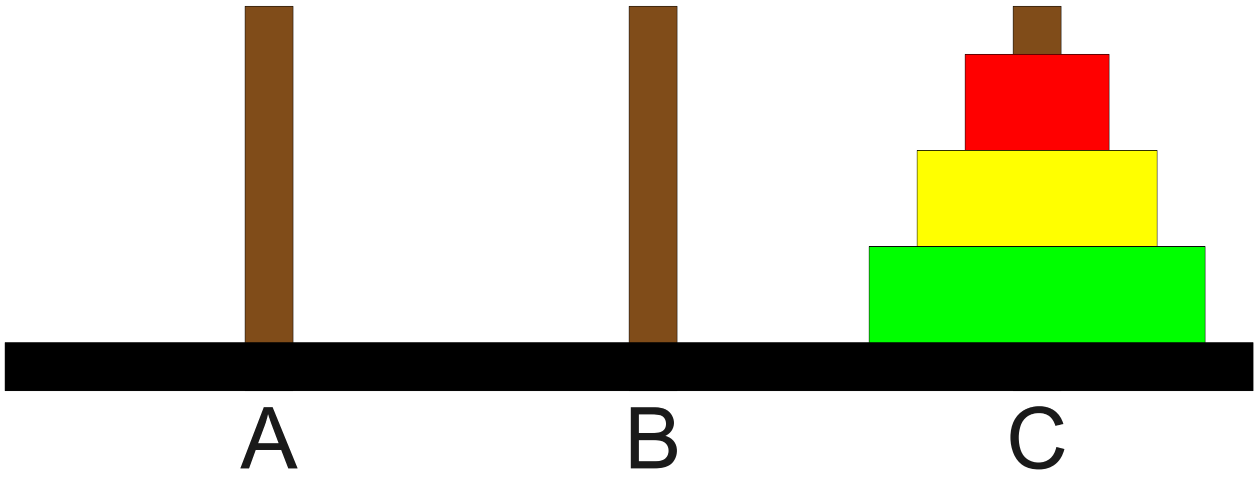
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

n	from	to	temp
1	a	c	b

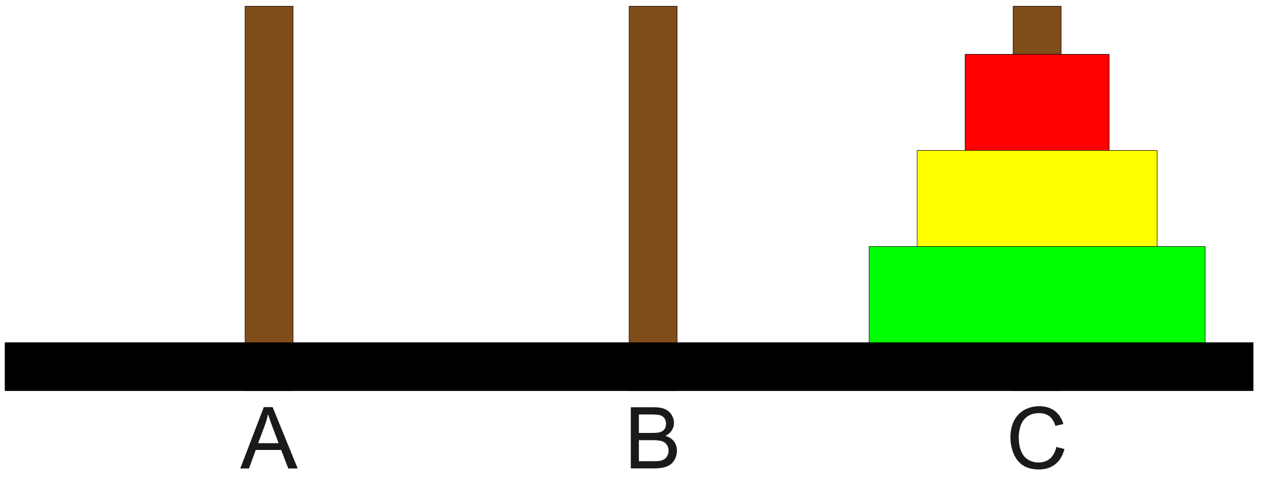


```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

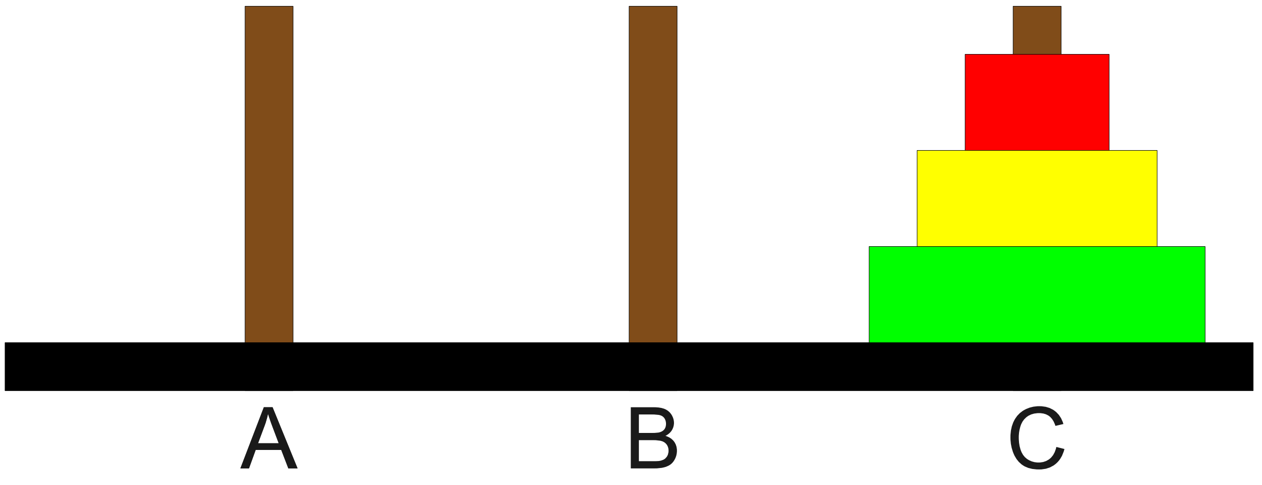
n	2	from	b	to	c	temp	a



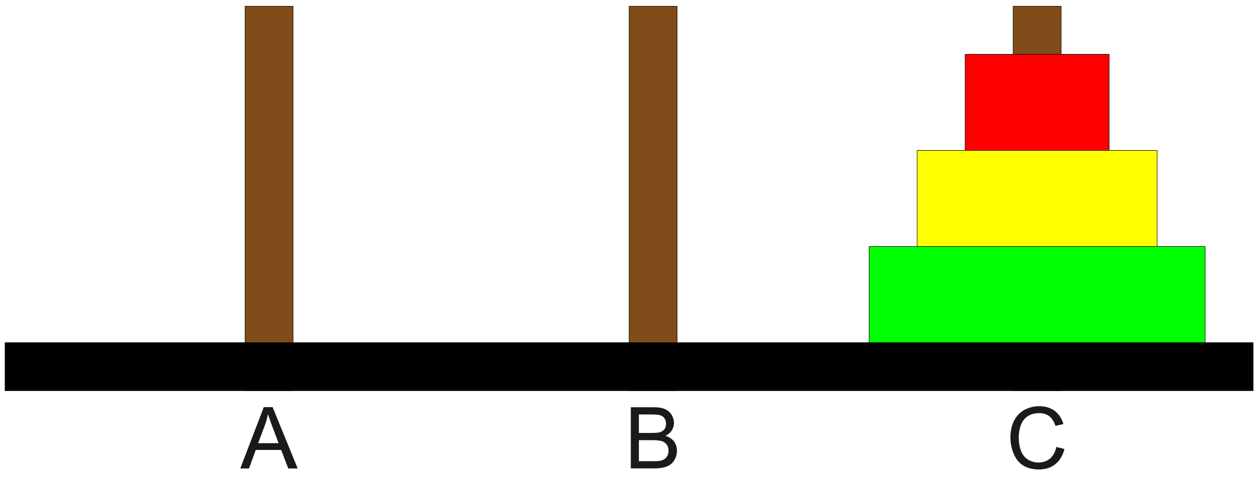
```
int main() {

}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

n	3	from	a	to	b	temp	c



```
int main() {
    moveTower(3, 'a', 'b', 'c');
}
```



Emergent Behavior

- Even though each function call does very little work, the overall behavior of the function is to solve the Towers of Hanoi.
- It's often tricky to think recursively because of this **emergent behavior**:
  - No one function call solves the entire problem.
  - Each function does only a small amount of work on its own and delegates the rest.

Writing Recursive Functions

- Although it is good to be able to trace through a set of recursive calls to understand how they work, you will need to build up an intuition for recursion to use it effectively.
- You will need to learn to trust that your recursive calls – which are to the function that you are currently writing! - will indeed work correctly.
  - Eric Roberts calls this the “Recursive Leap of Faith.”
- Everyone can learn to think recursively. If this seems confusing now, **don't panic**. You'll start picking this up as we continue forward.

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 1) {
        moveSingleDisk(from, to);
    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n == 0) {

    } else {
        moveTower(n - 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n - 1, temp, to, from);
    }
}
```

```
void moveTower(int n, char from, char to, char temp) {
    if (n != 0) {
        moveTower(n – 1, from, temp, to);
        moveSingleDisk(from, to);
        moveTower(n – 1, temp, to, from);
    }
}
```

Exhaustive Recursion

Generating All Possibilities

- Commonly, you will need to generate all objects matching some criteria.
  - Word Ladders: Generate all words that differ by exactly one letter.
- Often, structures can be generated iteratively.
- In many cases, however, it is best to think about generating all options recursively.

Subsets

- Given a set *S*, a **subset** of *S* is a set *T* composed of elements of *S*.
- Examples:
  - {0, 1, 2} ⊆ {0, 1, 2, 3, 4, 5}
  - {dikdik, ibex} ⊆ {dikdik, ibex}
  - { A, G, C, T } ⊆ { A, B, C, D, E, …, Z }
  - { } ⊆ {a, b, c}
  - { } ⊆ { }

Generating Subsets

- Many important problems in computer science can be solved by listing all the subsets of a set S and finding the “best” one out of every option.
- Example:
  - You have a collection of sensors on an autonomous vehicle, each of which has data coming in.
  - Which **subset** of the sensors do you choose to listen to, given that each takes a different amount of time to read?

Generating Subsets

$$\left\{ 0, 1, 2 \right\}$$

$$\left\{ \quad \right\} \qquad \left\{ 0 \right\}$$

$$\left\{ 2 \right\} \qquad \left\{ 0, \quad 2 \right\}$$

$$\left\{ 1 \right\} \qquad \left\{ 0, 1 \right\}$$

$$\left\{ 1, 2 \right\} \qquad \left\{ 0, 1, 2 \right\}$$

Generating Subsets

$$\{0, 1, 2\}$$

$\{\}$	$\{0\}$
$\{2\}$	$\{0, 2\}$
$\{1\}$	$\{0, 1\}$
$\{1, 2\}$	$\{0, 1, 2\}$

Generating Subsets

$$\{0, 1, 2\}$$

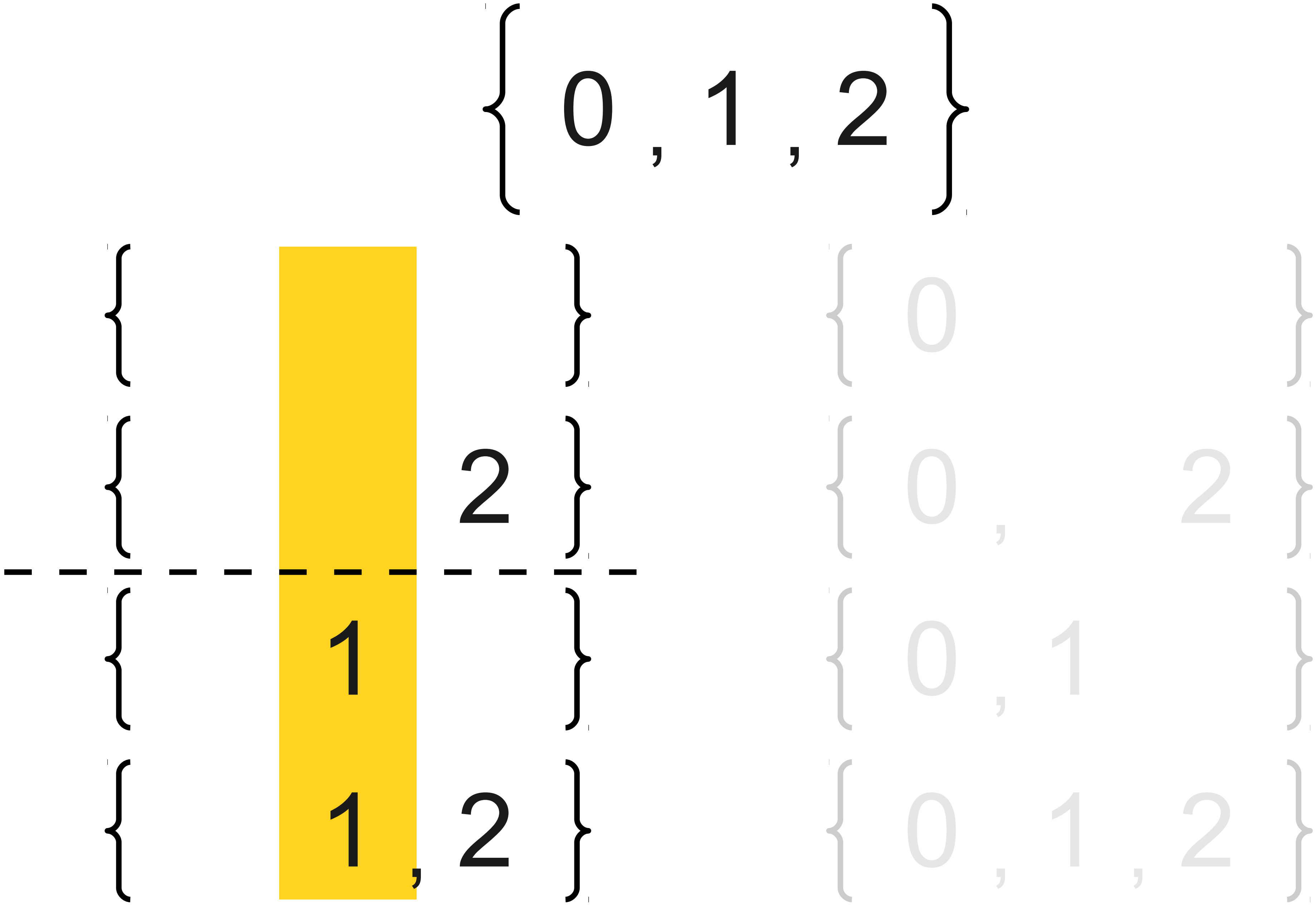
$$\{\ \} \qquad \{0\}$$

$$\{2\} \qquad \{0, 2\}$$

$$\{1\} \qquad \{0, 1\}$$

$$\{1, 2\} \qquad \{0, 1, 2\}$$

Generating Subsets



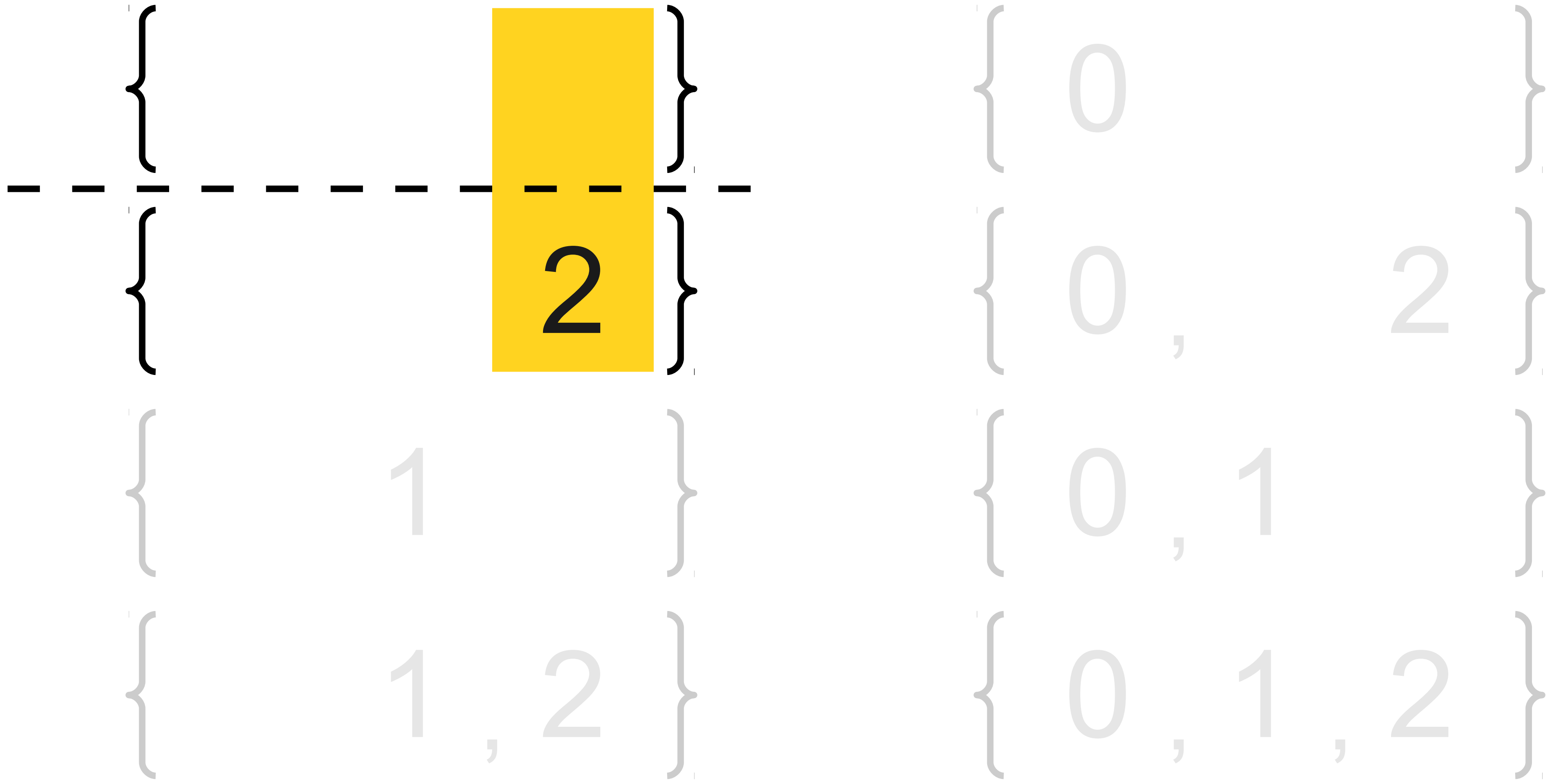
Generating Subsets

$$\{0, 1, 2\}$$

$\{\}$	$\{0\}$
$\{2\}$	$\{0, 2\}$
$\{1\}$	$\{0, 1\}$
$\{1, 2\}$	$\{0, 1, 2\}$

Generating Subsets

$\{0, 1, 2\}$



Generating Subsets

- The only subset of an empty set is the empty set itself.
- Otherwise:
  - Fix some element *x* of the set.
  - Generate all subsets of the set formed by removing *x* from the main set.
  - These subsets are subsets of the original set.
  - All of the sets formed by adding *x* into those subsets are subsets of the original set.

Tracing the Recursion

Tracing the Recursion

{ A, H, I }

Tracing the Recursion

{ A, H, I }

{ H, I }

Tracing the Recursion

{ A, H, I }

{ H, I }

{ I }

Tracing the Recursion

{ A, H, I }

{ H, I }

{ I }

{ }

Tracing the Recursion

{ A, H, I }

{ H, I }

{ I }

{ }

{ }

Tracing the Recursion

{ A, H, I }

{ H, I }

{ I }

{I}, { }

{ }

{ }

Tracing the Recursion

{ A, H, I }	
{ H, I }	`{H, I}, {H}, {I}, { }`
{ I }	`{I}, { }`
{ }	`{ }`

Tracing the Recursion

{ A, H, I }	`{A, H, I}, {A, H}, {A, I}, {A} {H, I}, {H}, {I}, { }`
{ H, I }	`{H, I}, {H}, {I}, { }`
{ I }	`{I}, { }`
{ }	`{ }`