University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Pitt SCI

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 6: this Friday @ 11:59 pm
  - Lab 5: next Monday @ 11:59 pm
  - Programming Assignment 1: Late Deadline: Wednesday Oct. 12th
    - Autograder feedback
- Debugging hints
- If you think you lost points in a lab assignment because of the autograder or because of a simple mistake
  - please reach out to Grader TA over Piazza
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous Lecture …

- ADT Stack
  - Application: Building a simple parser of Algebraic expressions
  - Application: Runtime stack
- Recursion
  - Definition
  - Basic examples

Today …

- Recursion
  - More examples
  - Runtime analysis using proof by induction
- Using recursion to solve hard problems
  - Towers of Hanoi

Recursively Processing a Linked Chain

- Display data in first node
- Then, (recursively) display data in rest of chain

```
public void display()
{
   displayChain(firstNode);
} // end display
private void displayChain(Node nodeOne)
{
```

Recursively Processing a Linked Chain

- Display data in first node
- Then, (recursively) display data in rest of chain

```
public void display()
{
   displayChain(firstNode);
} // end display
private void displayChain(Node nodeOne)
{
   if (nodeOne != null)
   {
      System.out.println(nodeOne.getData()); // Display first node
```

Recursively Processing a Linked Chain

- Display data in first node
- Then, (recursively) display data in rest of chain

```
public void display()
{
   displayChain(firstNode);
} // end display
private void displayChain(Node nodeOne)
{
   if (nodeOne != null)
   {
      System.out.println(nodeOne.getData()); // Display first node
      displayChain(nodeOne.getNextNode());   // Display rest of chain
   } // end if
} // end displayChain
```

Traversing a linked chain *backwards*

Traversing chain of linked nodes in reverse order easier when done recursively.

```
public void displayBackward()
{
   displayChainBackward(firstNode);
} // end displayBackward

private void displayChainBackward(Node nodeOne)
{
```

Traversing a linked chain *backwards*

- (recursively) display data in rest of chain

```
public void displayBackward()
{
   displayChainBackward(firstNode);
} // end displayBackward

private void displayChainBackward(Node nodeOne)
{
   if (nodeOne != null)
   {
      displayChainBackward(nodeOne.getNextNode());
```

Traversing a linked chain *backwards*

- (recursively) display data in rest of chain
- Then, display data in first node

```
public void displayBackward()
{
   displayChainBackward(firstNode);
} // end displayBackward

private void displayChainBackward(Node nodeOne)
{
   if (nodeOne != null)
   {
      displayChainBackward(nodeOne.getNextNode());
      System.out.println(nodeOne.getData());
   } // end if
} // end displayChainBackward
```

Running Time Analysis of Recursive Algorithms

- Technique #1: Using **proof by induction**
- Assume running time of countDown is a function of *n*: $T(n)$
- $T(n) = 1 + 1 + ??$
  - What is the running time of countdown(n-1)?
  - Can we use the function $T(n)$?
  - Yes! The running time of *countdown(n-1)* is $T(n-1)$
- $T(n) = 2 + T(n-1)$
- $\quad = T(n-1) + O(1)$
- $T(1) = O(1)$

```java
public static void countDown(int n)
{
    System.out.println(n);
    if (n > 1)
        countDown(n - 1);
} // end countDown
```

Running Time Analysis of Recursive Algorithms

- $T(1) = 1$ … (1)
- $T(n) = T(n-1) + 1$ for $n>1$ … (2)
  - The above equation is called a <u>*Recurrence Relation*</u>
- T(2) = T(1) + 1 = 2
- T(3) = T(2) + 1 = 2 + 1 = 3
- T(4) = T(3) + 1 = 3 + 1 = 4
- …
- We have an intuition that the running time is linear
  - $T(n) = n$ … (3)
  - Let’s prove (3) by induction
- <u>Base Case:</u>
  - From (1): T(1) = 1
  - From (3): T(1) = 1
  - (3) applies to the base case

Running Time Analysis of Recursive Algorithms

- <u>Inductive Step:</u>
  - Assume that (3) is true for *all values* < *k* and prove that it is true for *k*
  - Inductive hypothesis: $T(n) = n$ for all $n<k$ … (4)
  - We want to prove that $T(k) = k$
  - From (2), $T(k) = T(k-1) + 1$
  - From (4), T(k) = (k-1) + 1 = k
  - **End of Proof that $T(n) = n$**
  - So, running time of countdown is $O(n)$

$$T(1) = 1 \quad \ldots (1)$$

$$T(n) = T(n-1) + 1 \text{ for } n>1 \quad \ldots (2)$$

```
public static void countDown(int n)
{
    System.out.println(n);
    if (n > 1)
        countDown(n - 1);
} // end countDown
```

Computing x^n

- Using iteration

```java
public static int powerIterative(int x, int n){
    assert(n >= 0);
    int result = 1;
    for(int i=0; i<n; i++){
        result *= x;
    }
    return result;
}
```

Computing x^n

- What is the running time of powerIterative?
  - *O(n)*

```java
public static int powerIterative(int x, int n){
  assert(n >= 0);
  int result = 1;
  for(int i=0; i<n; i++){
    result *= x;
  }
  return result;
}
```

Computing x^n

- Using recursion
- How?
  - Let's start with recursive mathematical definition
- $x^n = (x^{n/2})^2$
  - when *n* is even and positive
- $x^n = x(x^{n/2})^2$
  - when *n* is odd and positive
  - *n/2* is integer division
- base case or non-recursive case
  - $x^0 = 1$

Computing x^n

- Using recursion

```
public static int power(int x, int n){
  int result = 1;
  if(n > 0){
    int temp = power(x, n/2);
```

Computing x^n

- Using recursion

```
public static int power(int x, int n){
    int result = 1;
    if(n > 0){
        int temp = power(x, n/2);
        result = temp * temp;
```

Computing x^n

- Using recursion

```
public static int power(int x, int n){
  int result = 1;
  if(n > 0){
    int temp = power(x, n/2);
    result = temp * temp;
    if(n%2 == 1){ //is n odd?
      result = x * result;
    }
```

Computing x^n

- Using recursion

```
public static int power(int x, int n){
    int result = 1;
    if(n > 0){
        int temp = power(x, n/2);
        result = temp * temp;
        if(n%2 == 1){ //is n odd?
            result = x * result;
        }
    }
    return result;
}
```

Running Time Analysis of Recursive Algorithms

- Technique #1: Using **proof by induction**
- Assume running time of recursive power is a function of *n*: $T(n)$
- $T(n) = O(1) + ??$
  - What is the running time of power(x, n/2)?
  - Can we use the function $T(n)$?
  - Yes! The running time of *power(x, n/2)* is $T(n/2)$
- $T(n) = T(n/2) + O(1)$
- $T(1) = O(1)$

```java
public static int power(int x, int n){
  int result = 1;
  if(n > 0){
    int temp = power(x, n/2);
    result = temp * temp;
    if(n%2 == 1){ //is n odd?
      result = x * result;
    }
  }
  return result;
}
```

Running Time Analysis of Recursive Algorithms

- $T(1) = 1$ … (1)
- $T(n) = T(n/2) + 1$ for $n>1$ … (2)
  - The above equation is called a ***<u>Recurrence Relation</u>***
- T(2) = T(1) + 1 = 2
- T(4) = T(2) + 1 = 2 + 1 = 3
- T(8) = T(4) + 1 = 3 + 1 = 4
- T(16) = T(8) + 1 = 4 + 1 = 5
- When n doubles → T(n) increases by 1
- We have an intuition that the running time is logarithmic
  - $T(n) = log(n) + 1$ … (3)
  - Let's prove (3) by induction
- <u>Base Case:</u>
  - From (1): T(1) = 1
  - From (3): T(1) = log(1) + 1 = 0 + 1 = 1
  - (3) applies to the base case

Running Time Analysis of Recursive Algorithms

- <u>Inductive Step:</u>
  - Assume that (3) is true for all n < *k* and prove that it is true for *k*
  - Inductive hypothesis: $T(n) = log(n) + 1$ for all $n<k$ … (4)
  - We want to prove that $T(k) = log(k) + 1$
  - From (2), $T(k) = T(k/2) + 1$
  - From (4), T(k/2) = $log(k/2) + 1$
  - Then, T(k) = $log(k/2) + 1 + 1$
  - $= log(k/2) + 2$
  - $= log(k/2) + log\ 4$
  - $= log(4k/2) = log(2k)$
  - $= log\ k + log\ 2$
  - $= log\ k + 1$
  - **End of Proof that $T(n) = log(n) + 1$**
  - So, running time of recursive power is $O(log\ n)$

$T(1) = 1$ … (1)

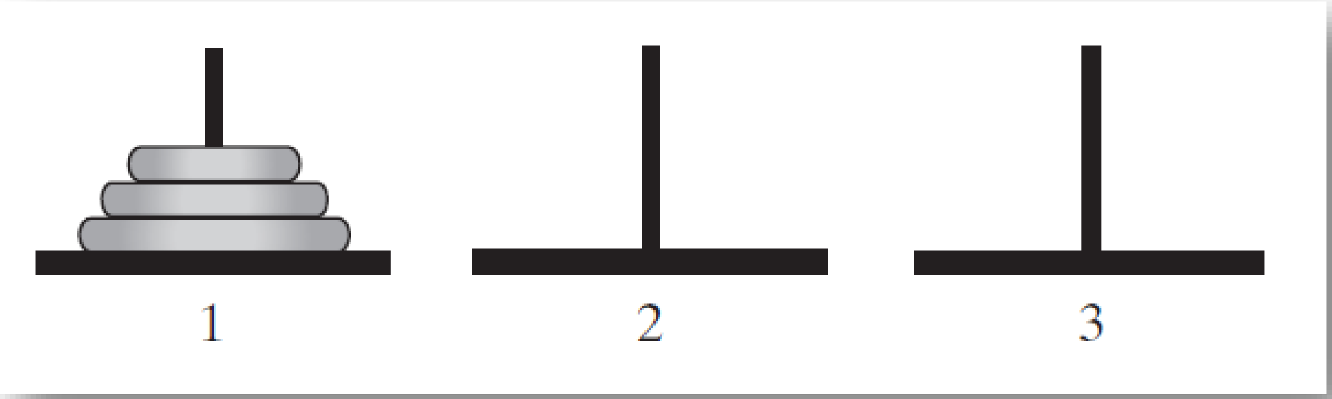
$T(n) = T(n/2) + 1$ for $n>1$ … (2)

Note on input size

- Our goal is to model running time in terms of input size
- The input size is the number of bits needed to represent the input
- For the power function, the exponent n is represented using how many bits?
  - *log n* bits
  - So, the input size of the exponentiation problem is not *n*, the exponent value
  - The input size is *log n*
- So, the recursive power function has linear running time
  - *O(log n)* is linear in *log n*, the input size

Simple Solution to a Difficult Problem

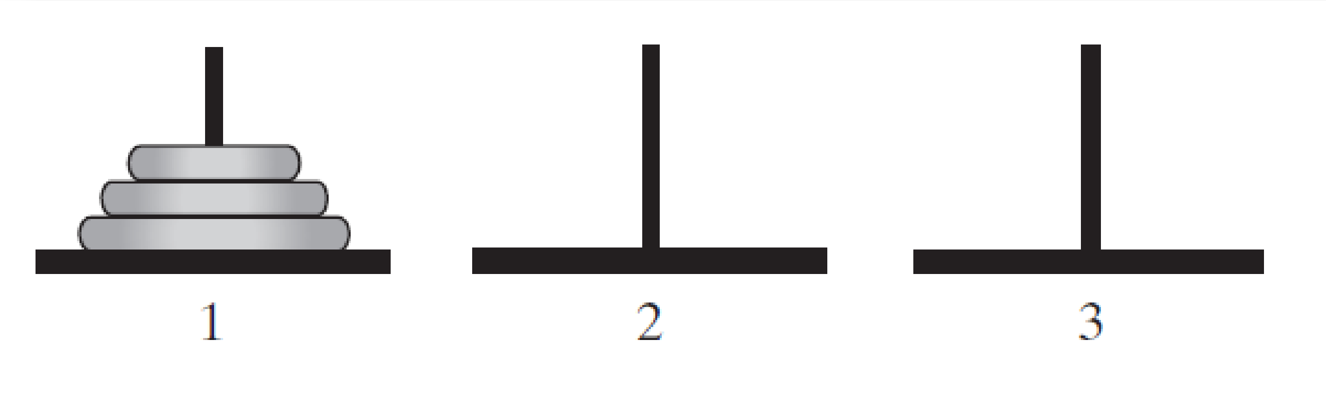
The initial configuration of the
Towers of Hanoi for three disks.



Towers of Hanoi Problem

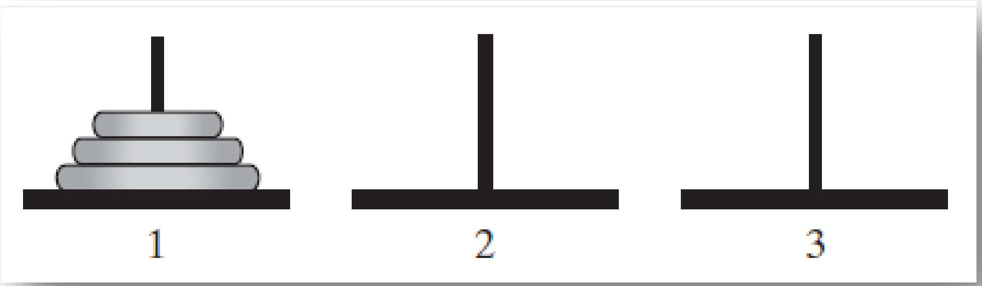
Rules:

1. Move one disk at a time.
2. Disk moved must be topmost disk in its pole
3. No disk may rest on top of a disk smaller than itself
4. You can store disks on the second pole temporarily



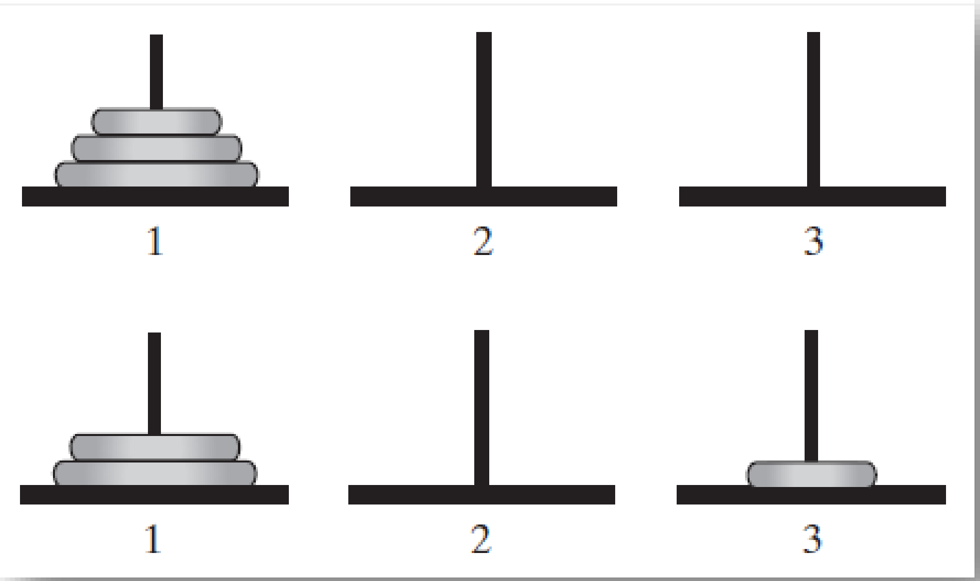
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



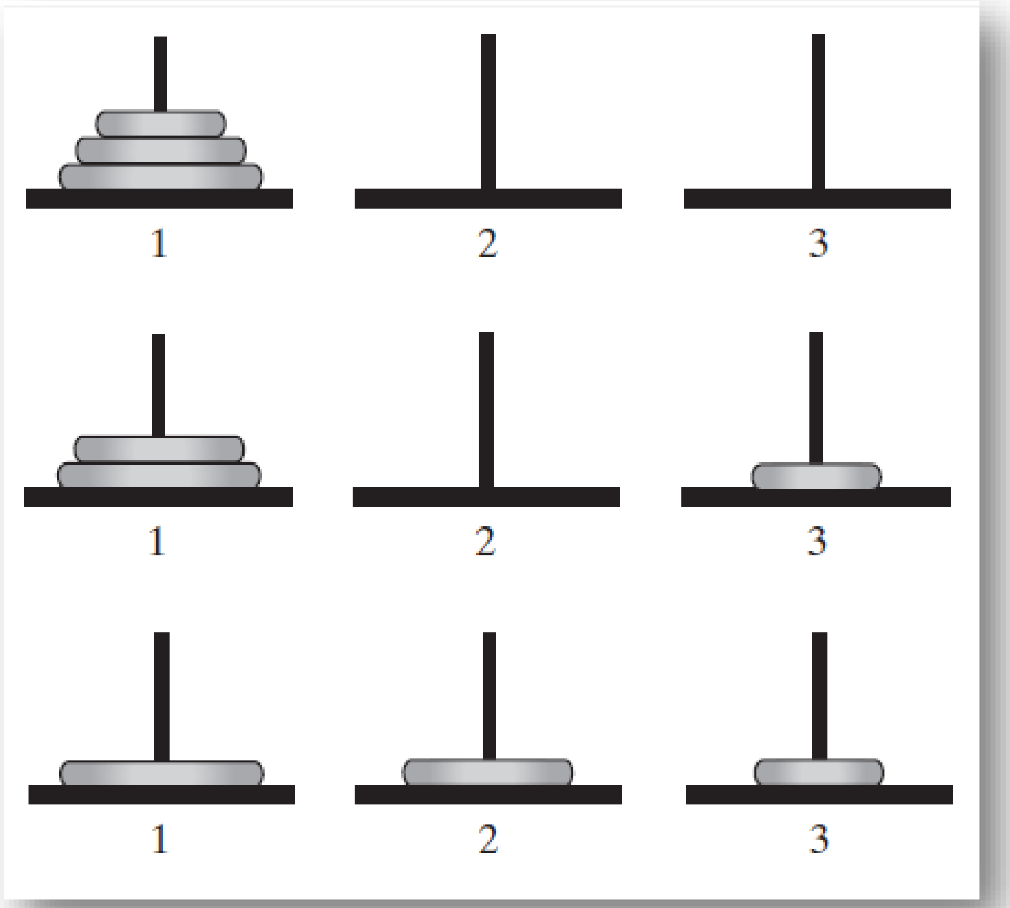
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



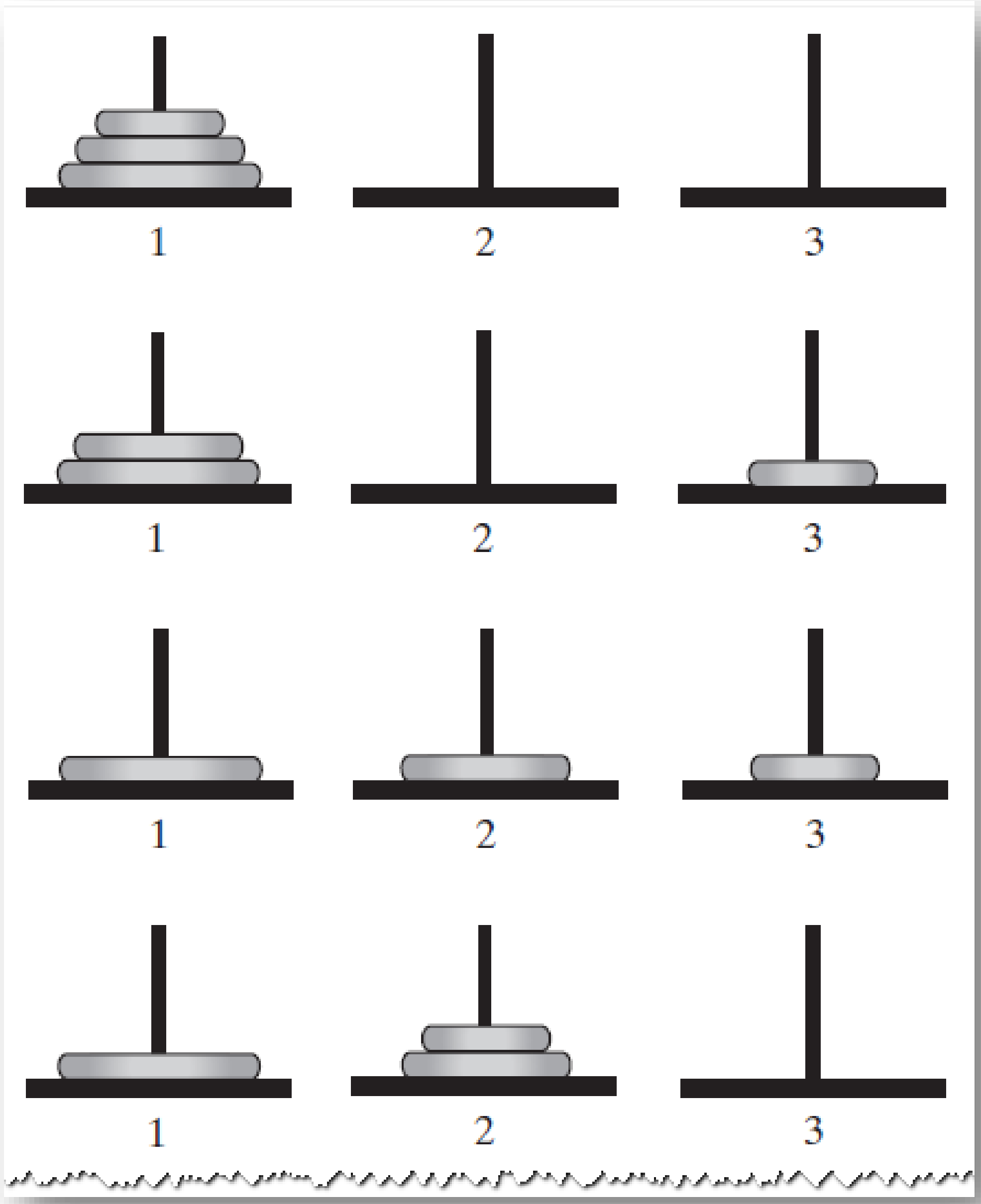
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



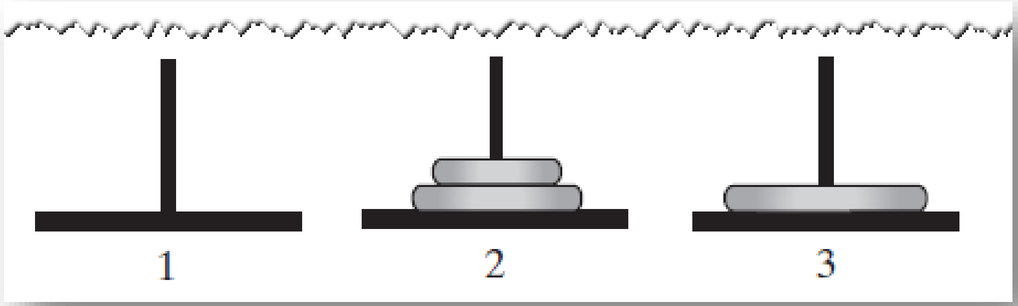
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



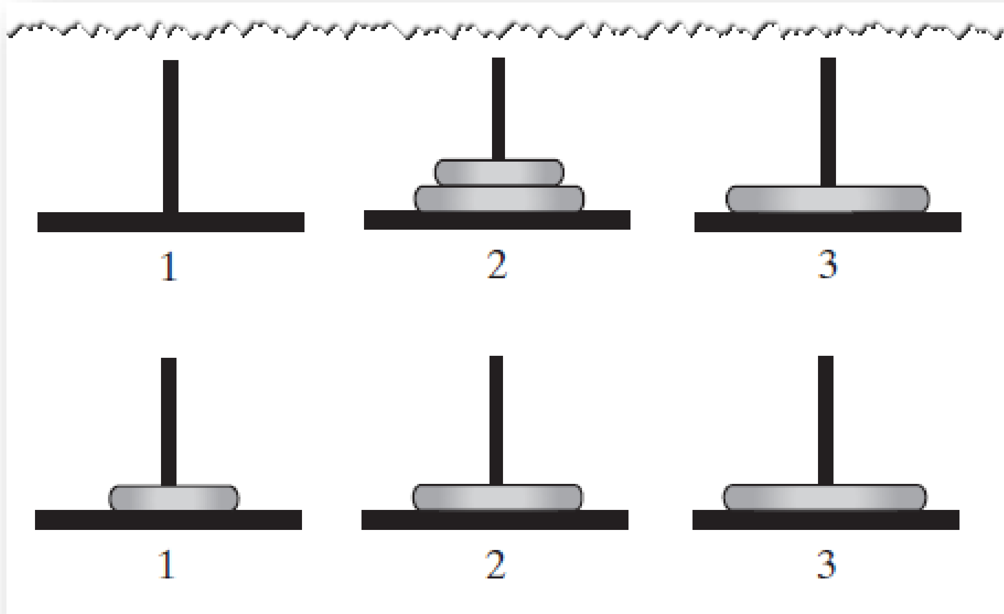
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



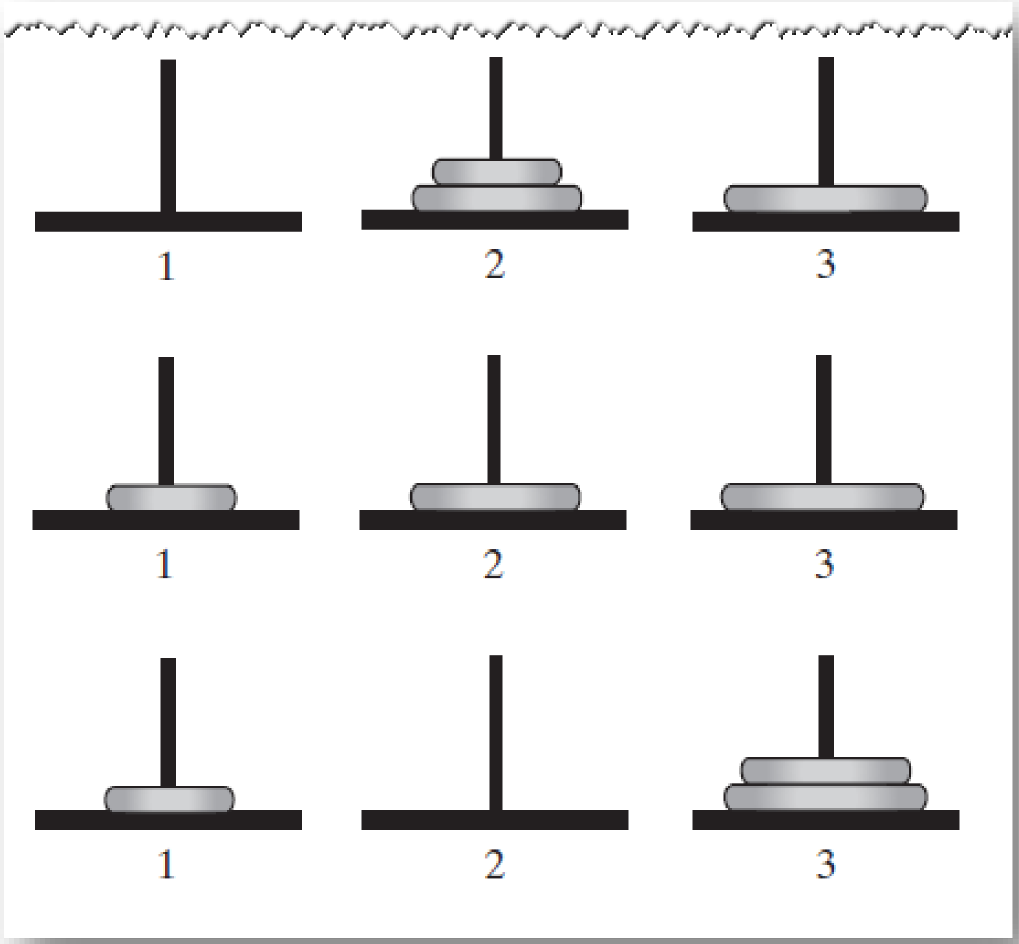
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



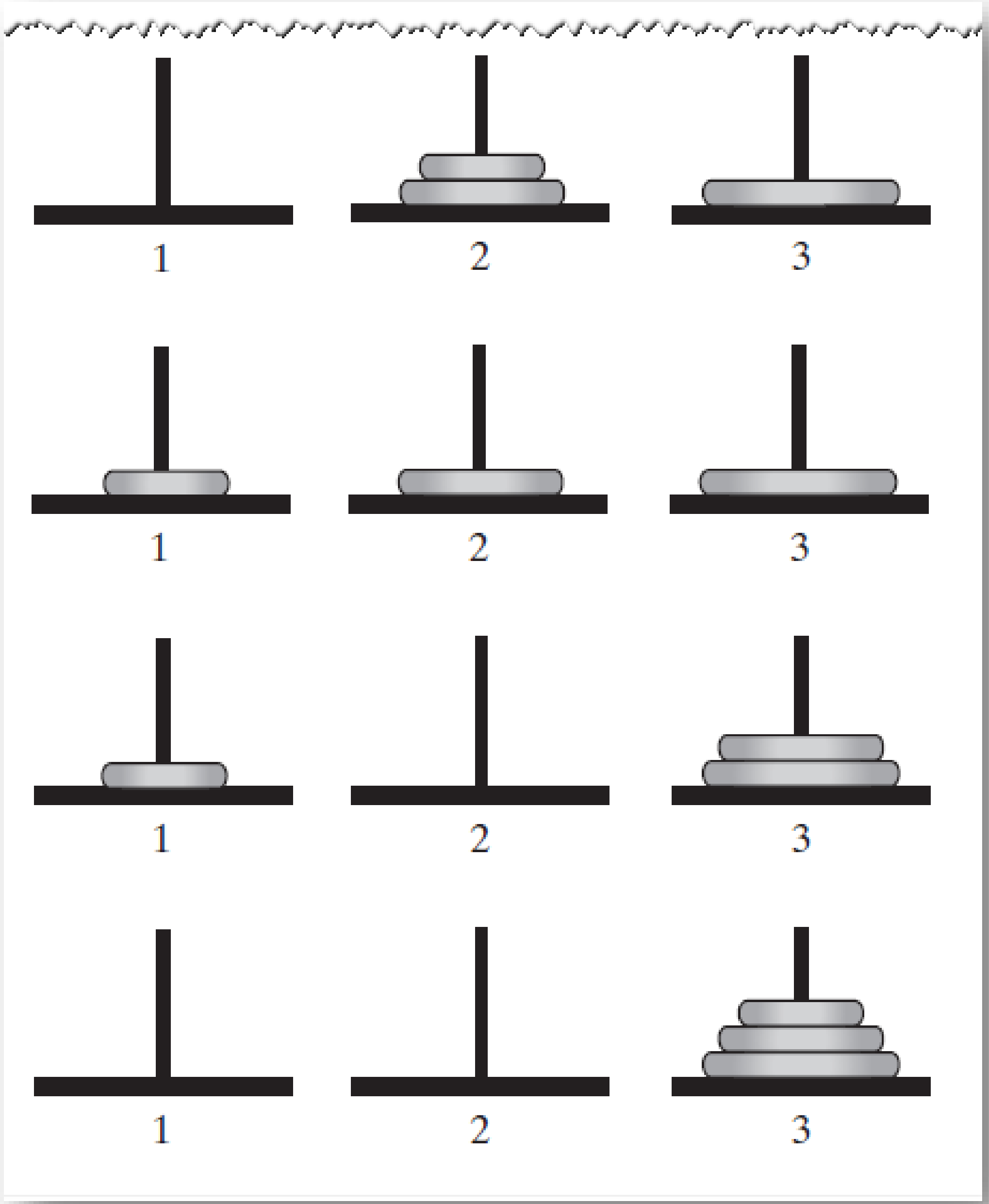
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



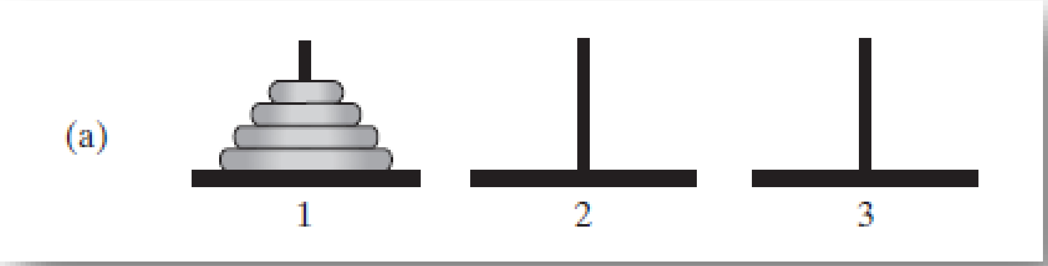
Solutions

The sequence of moves for solving the Towers of Hanoi problem with three disks



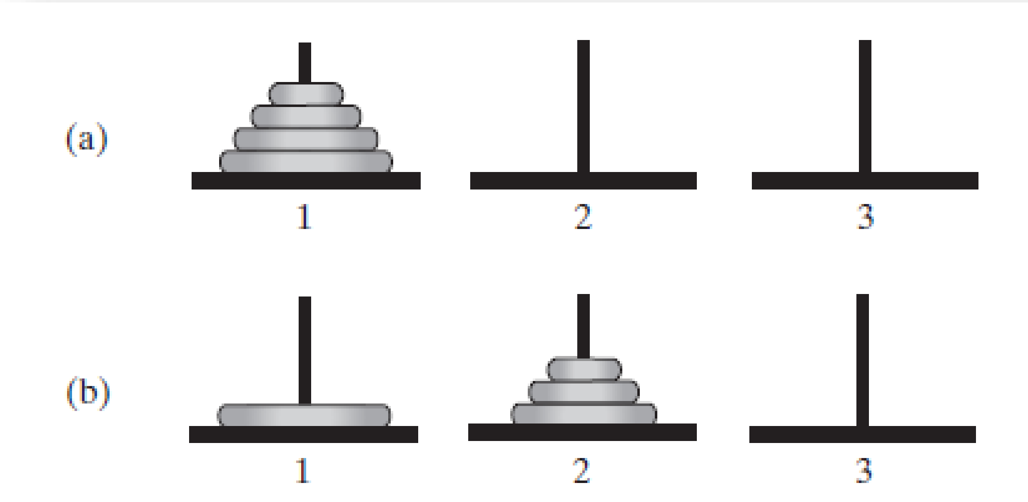
Solutions

The smaller problems in a recursive solution for four disks



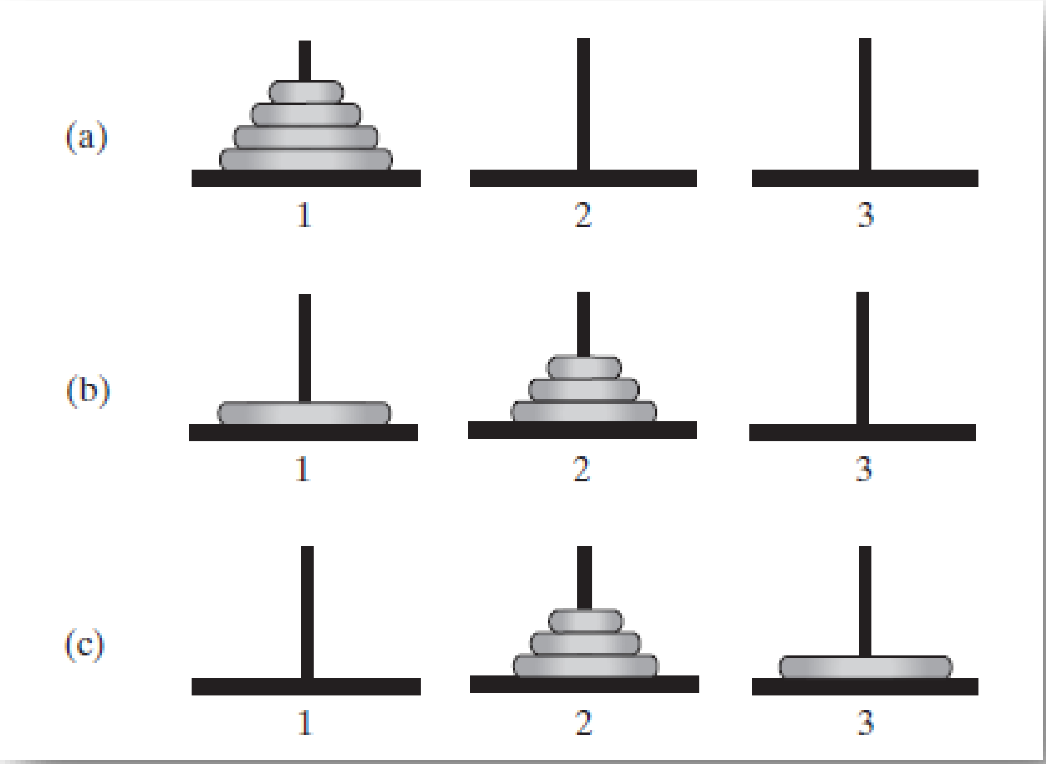
Solutions

The smaller problems in a recursive solution for four disks



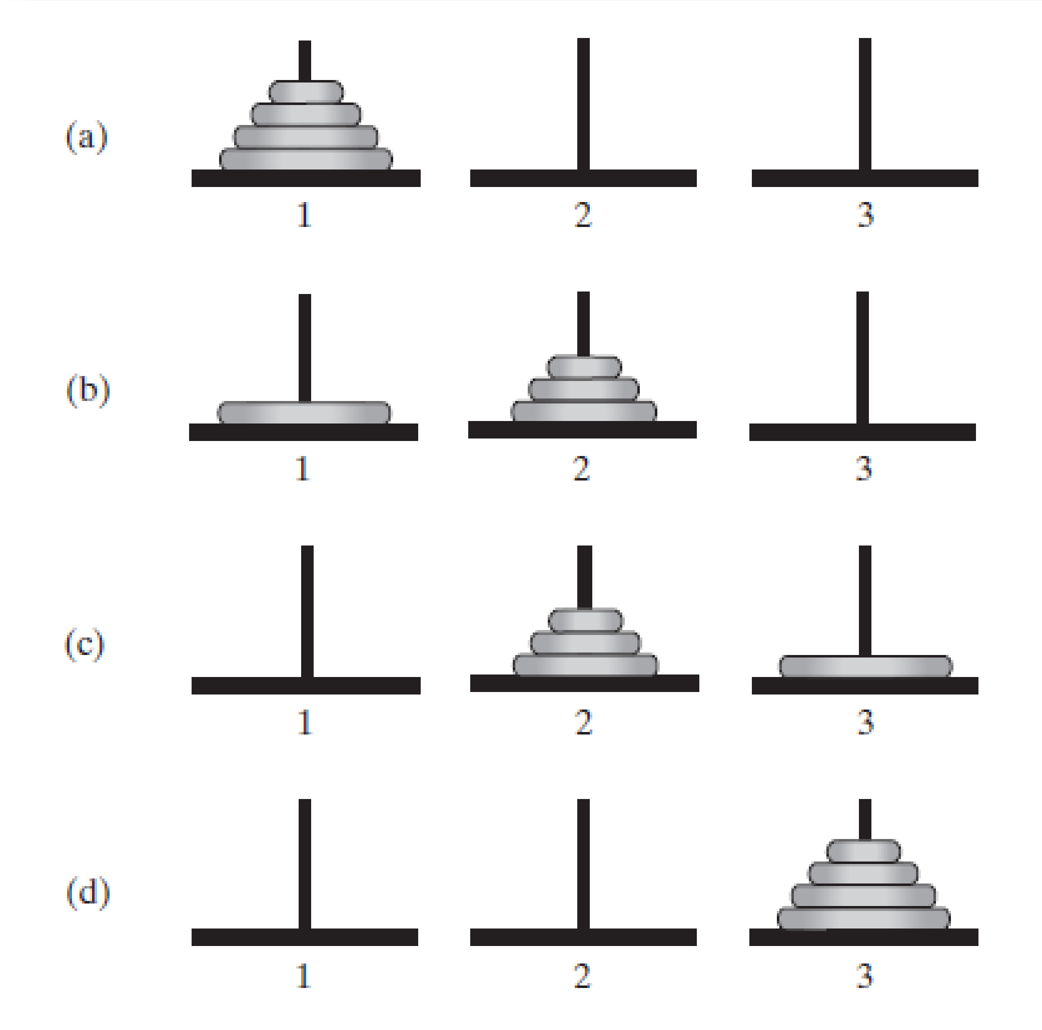
Solutions

The smaller problems in a recursive solution for four disks



Solutions

The smaller problems in a recursive solution for four disks



Solutions

- Recursive algorithm to solve any number of disks.

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
```

Solutions

- Recursive algorithm to solve any number of disks.

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
```

Solutions

- Recursive algorithm to solve any number of disks.

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
{
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
```

Solutions

- Recursive algorithm to solve any number of disks.

Algorithm `solveTowers(numberOfDisks, startPole, tempPole, endPole)`
```
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
{
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
```

Solutions

- Recursive algorithm to solve any number of disks.

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
{
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
    solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

Exercise

- Prove that the running time of solveTowers is $2^n - 1$
- *Hint: use proof by induction*