Algorithms and Data Structures 1
CS 0445

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
  - Problem solving techniques that use recursion
    - Divide and Conquer
    - Backtracking

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 $T(n-1)$
- $T(n) = 2 + T(n-1)$
- $\quad = T(n-1) + O(1)$
- $T(1) = O(1)$

```
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)
- $T(n) = T(n\text{-}1) + 1$ for $n>1$ … (2)
  - The above equation is called a *Recurrence Relation*
- 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
- Base Case:
  - From (1): T(1) = 1
  - From (3): T(1) = 1
  - (3) applies to the base case

Running Time Analysis of Recursive Algorithms

- Inductive Step:
  - 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 $O(n)$

$$\boxed{\begin{array}{ll} T(1) = 1 & \dots (1) \\ T(n) = T(n-1) + 1 \text{ for } n>1 & \dots (2) \end{array}}$$

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

Computing x^n

- Using iteration

```
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)*

```
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)*
- $T(n) = T(n/2) + O(1)$
- $T(1) = O(1)$

```
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 *Recurrence Relation*
- 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
- Base Case:
  - 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

- Inductive Step:
  - 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 $O(log\ n)$

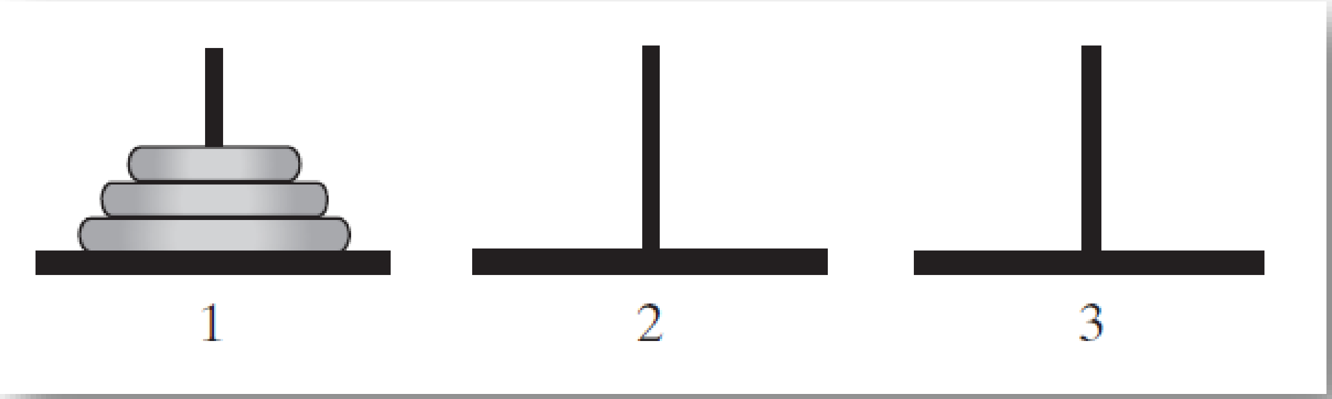
$$\boxed{\begin{array}{ll} T(1) = 1 & \ldots (1) \\ T(n) = T(n/2) + 1 \text{ for } n>1 & \ldots (2) \end{array}}$$

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

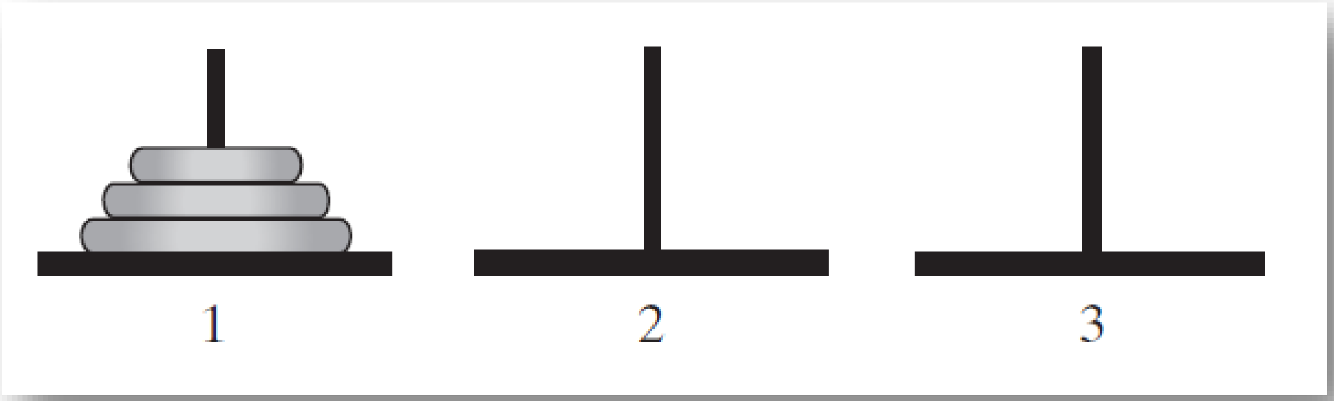
The initial configuration of the
Towers of Hanoi for three disks.



Towers of Hanoi Problem

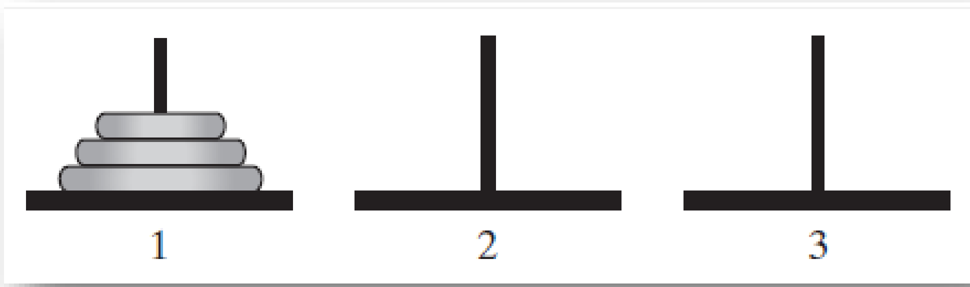
Rules:

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



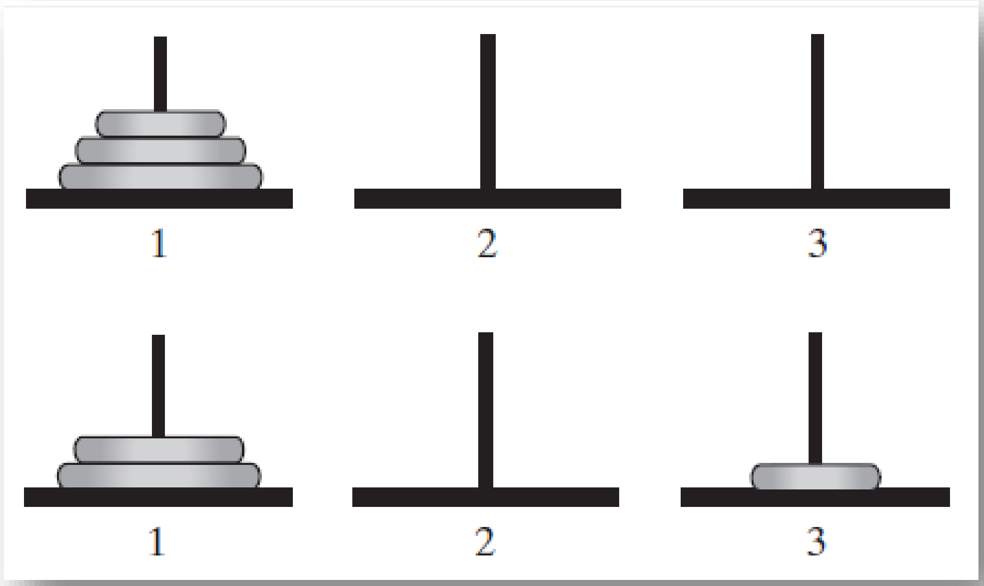
Solutions

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



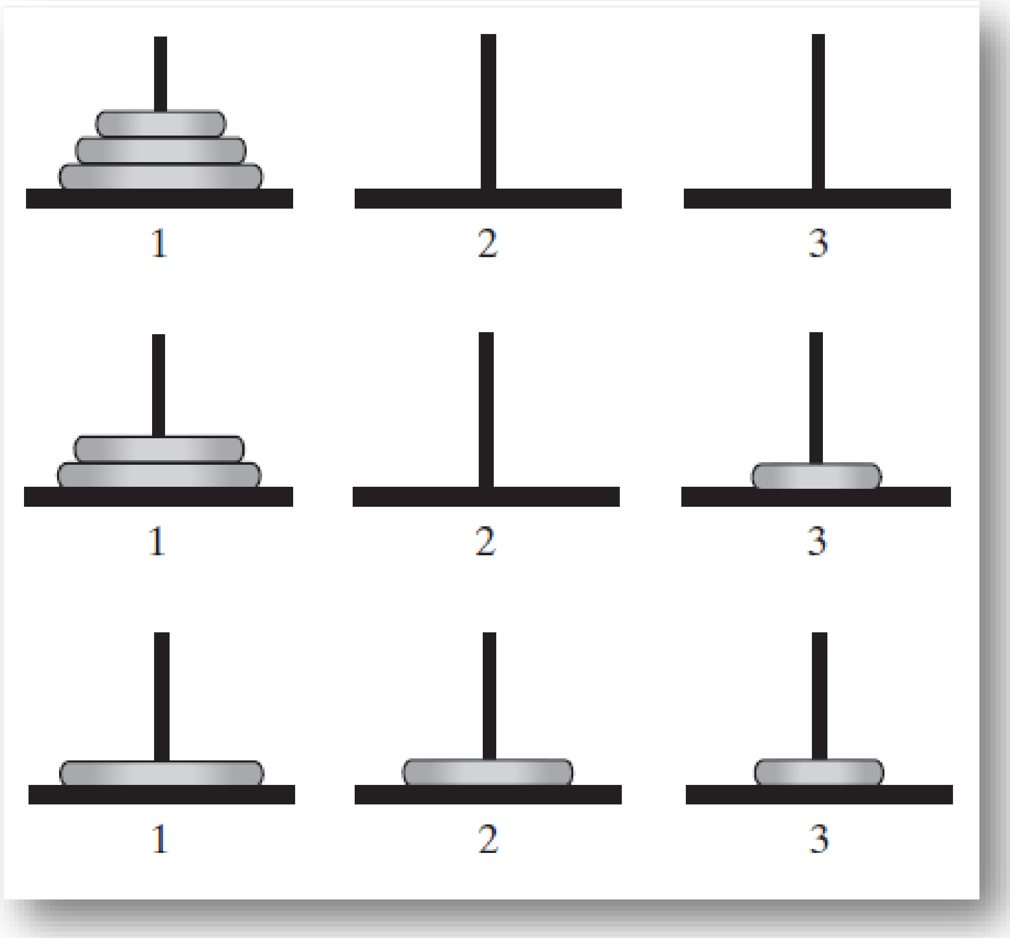
Solutions

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



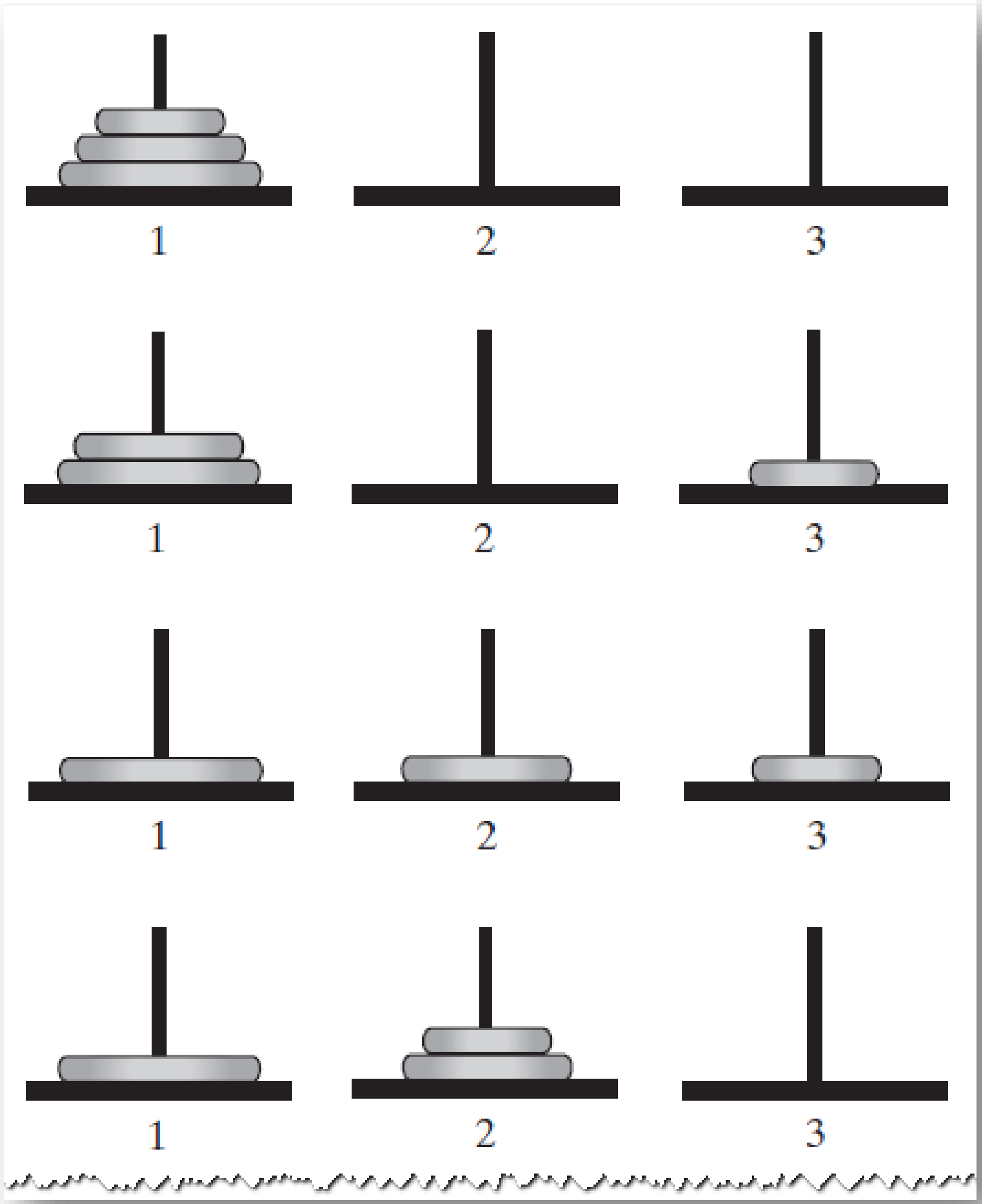
Solutions

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



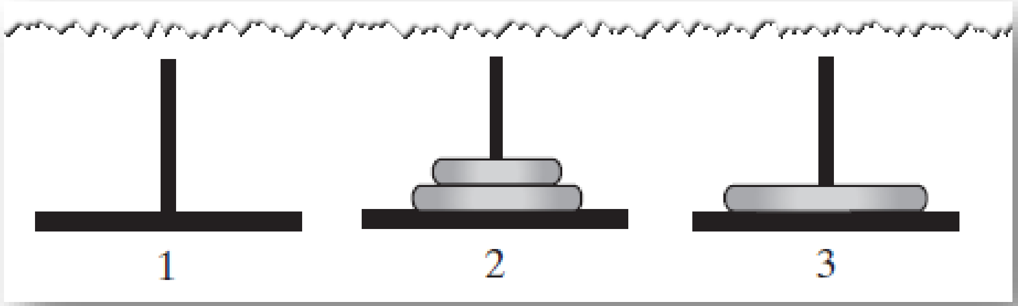
Solutions

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



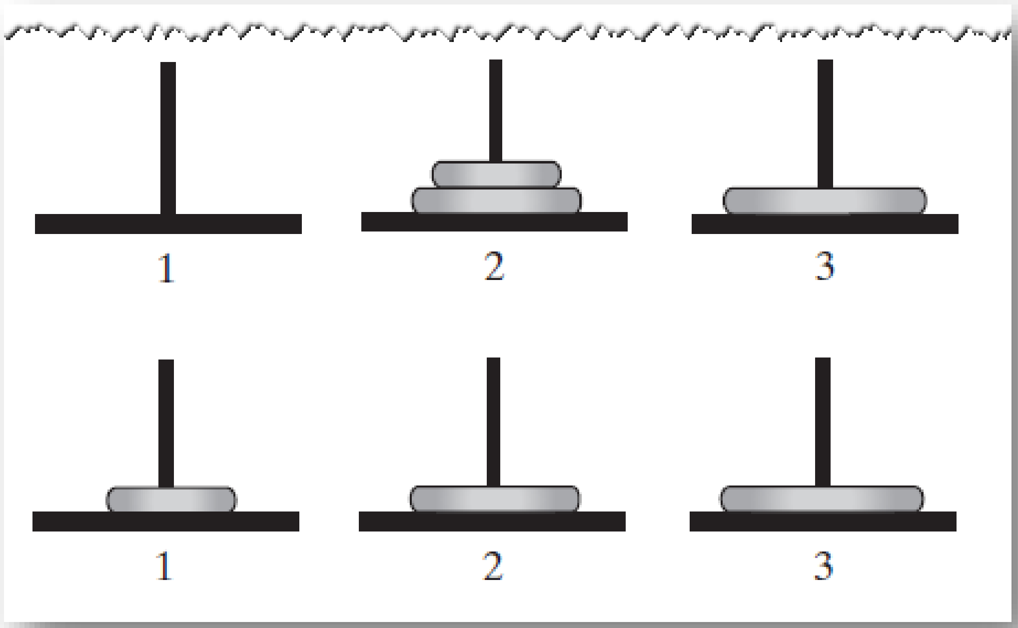
Solutions

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



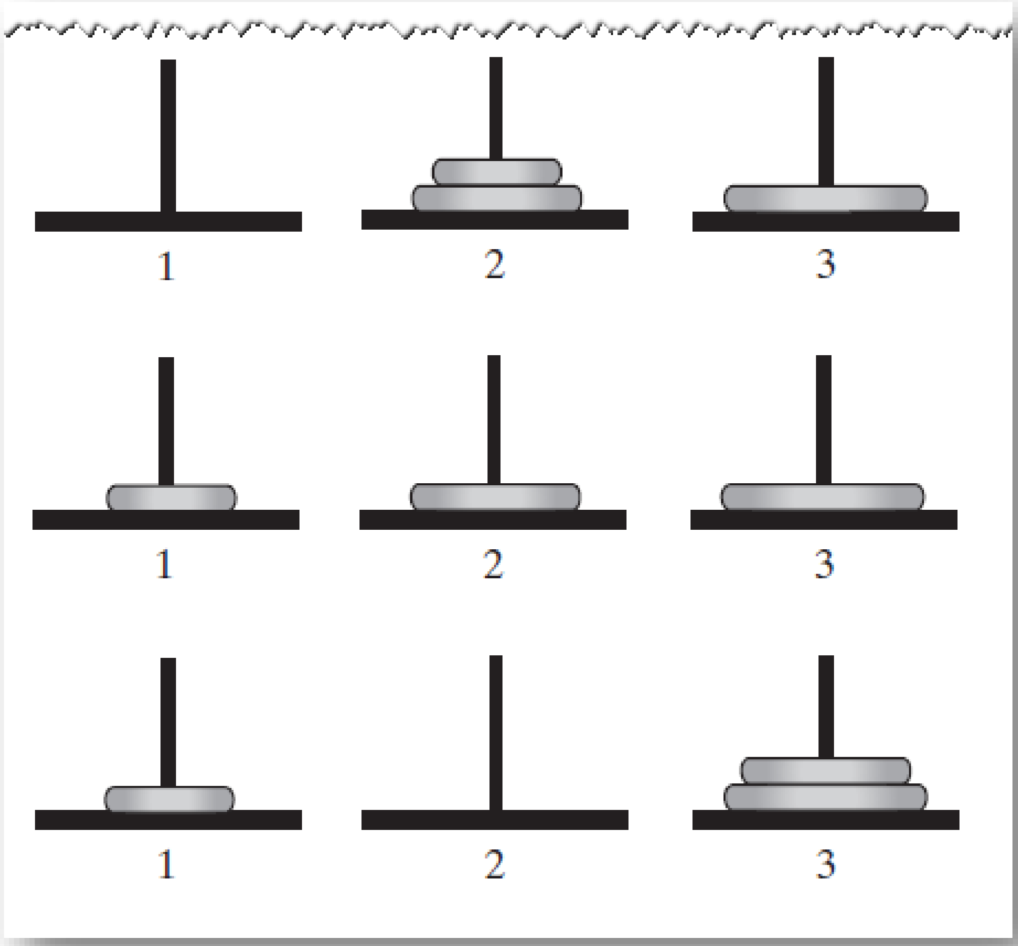
Solutions

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



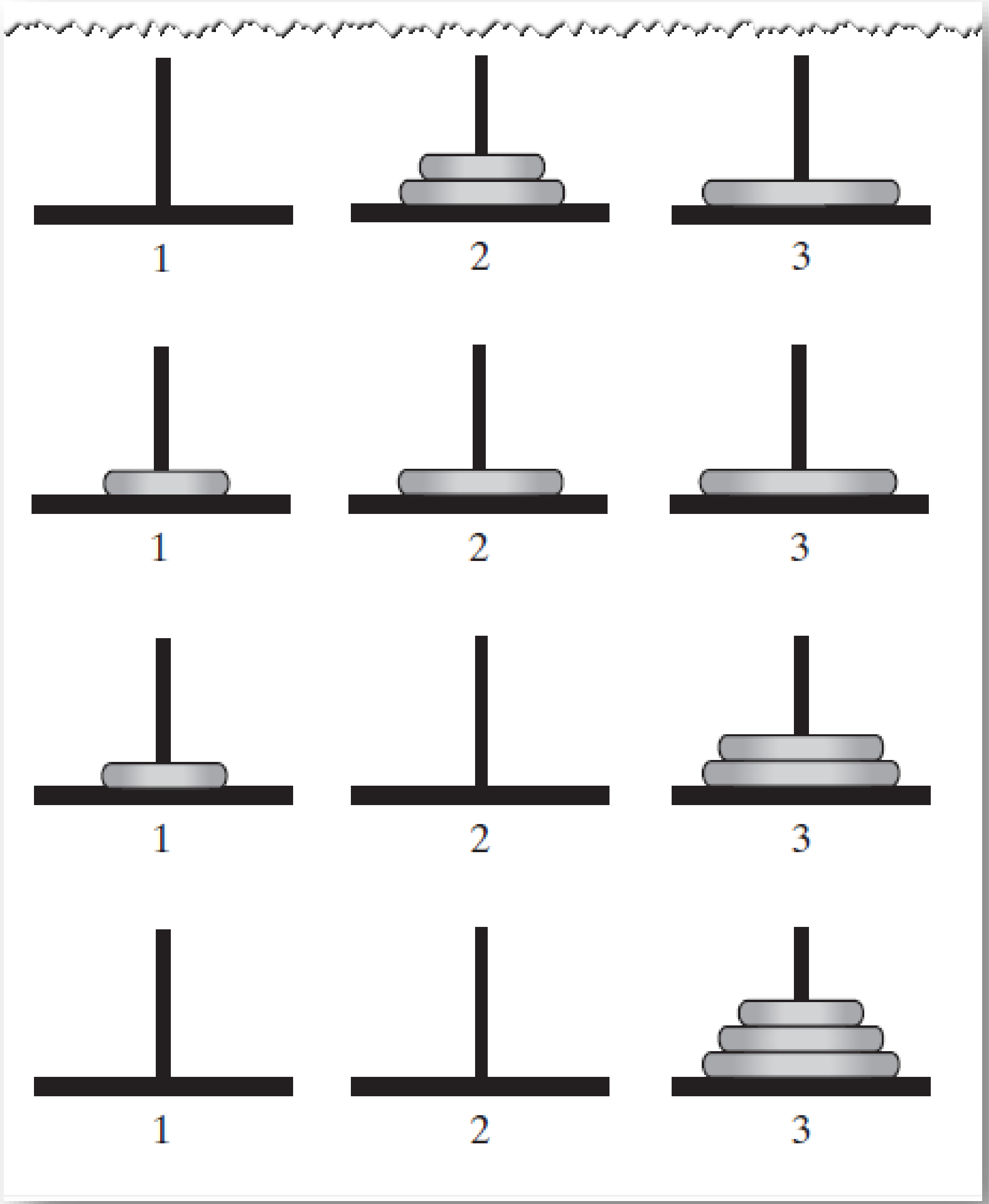
Solutions

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



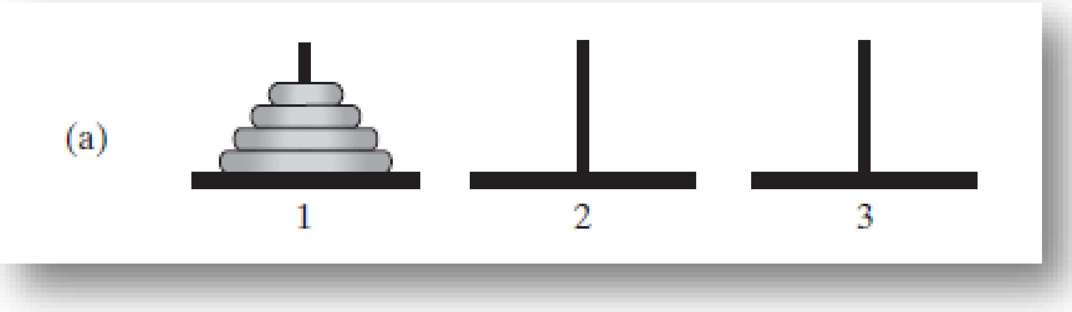
Solutions

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



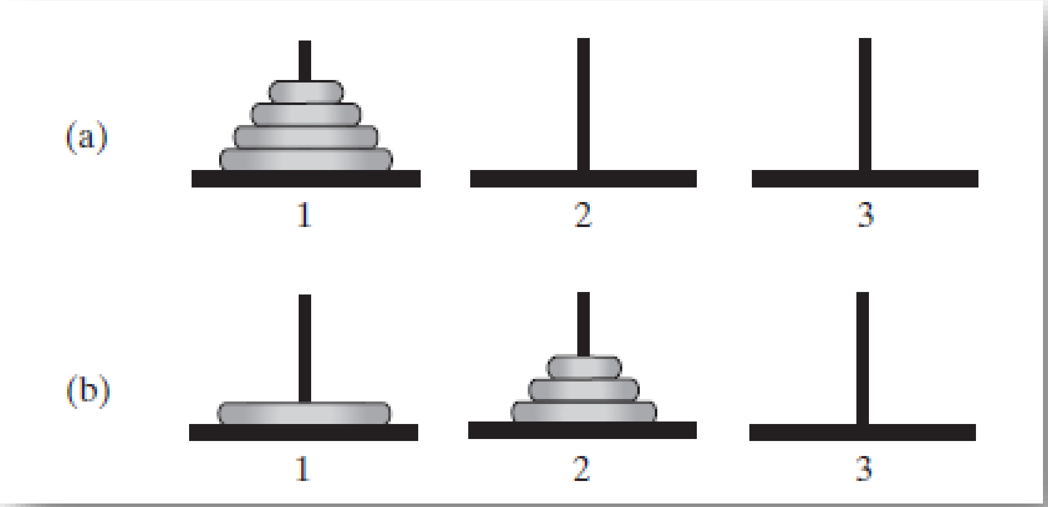
Solutions

The smaller problems in a recursive solution for four disks



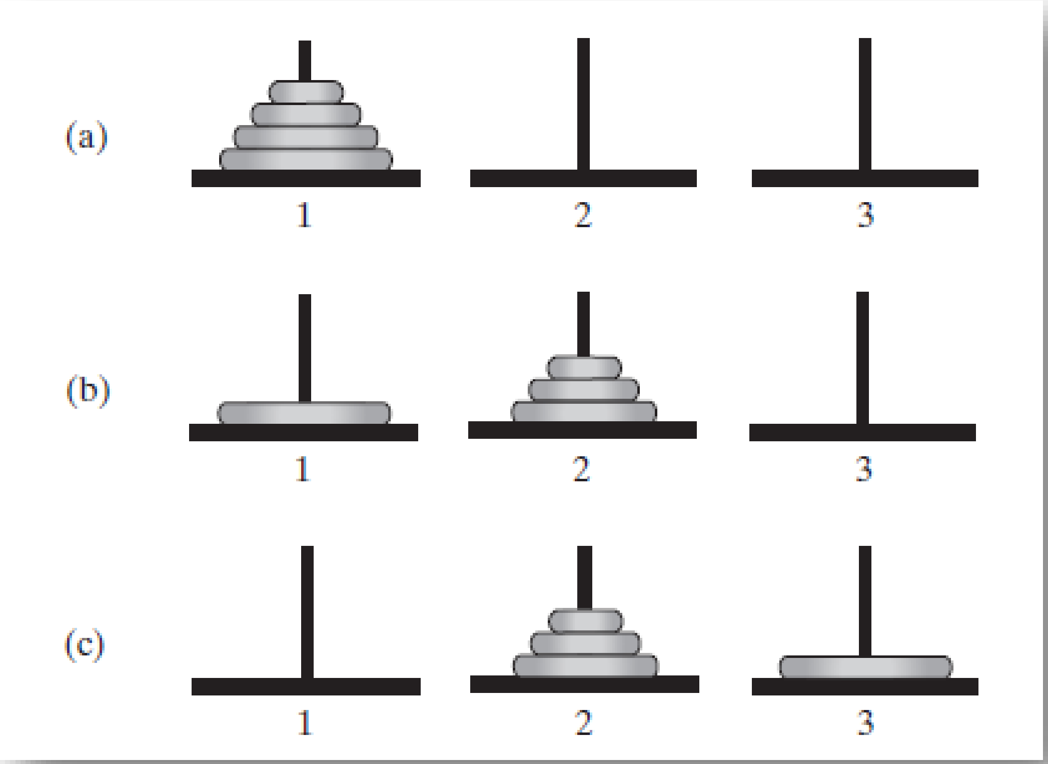
Solutions

The smaller problems in a recursive solution for four disks



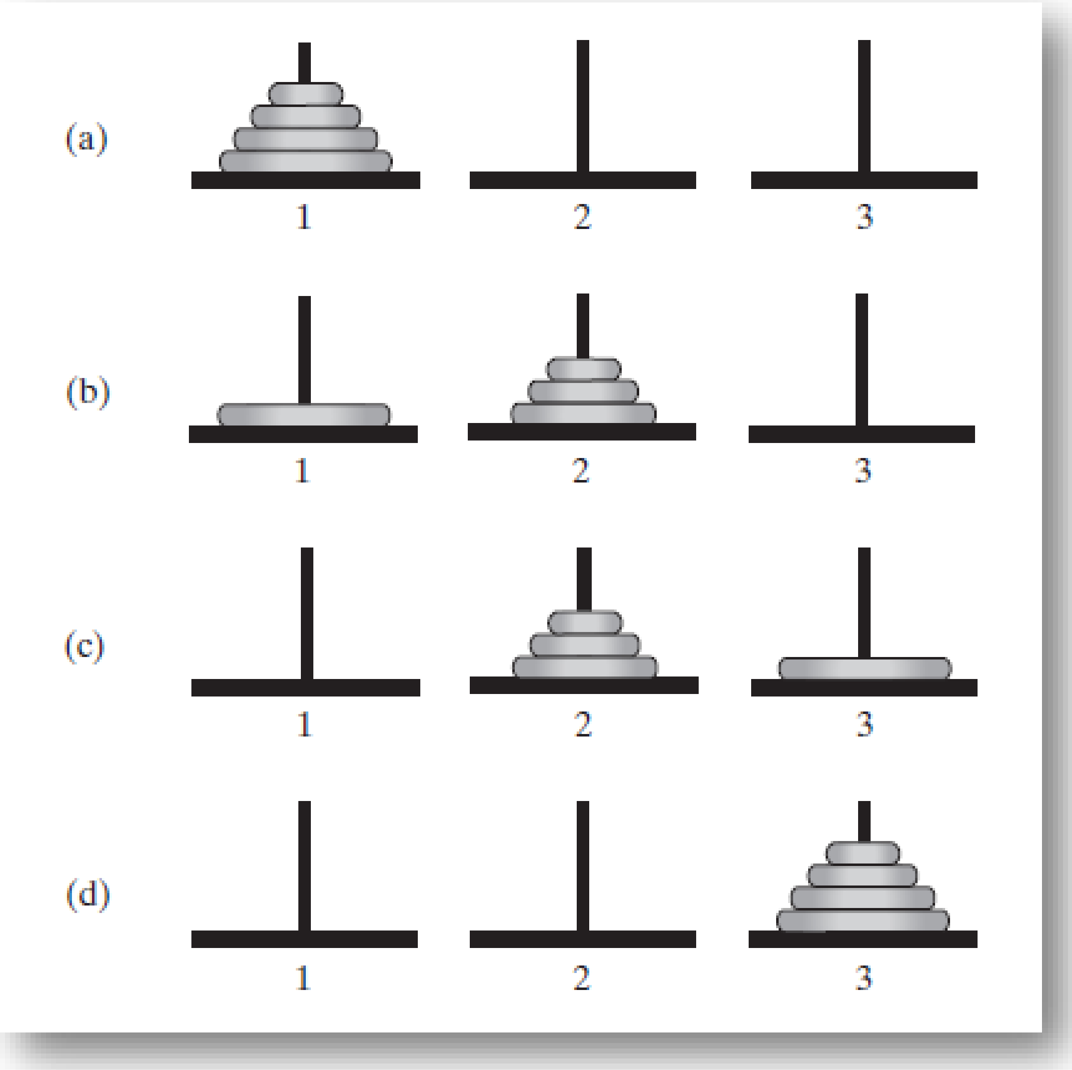
Solutions

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*

Poor Solution to a Simple Problem

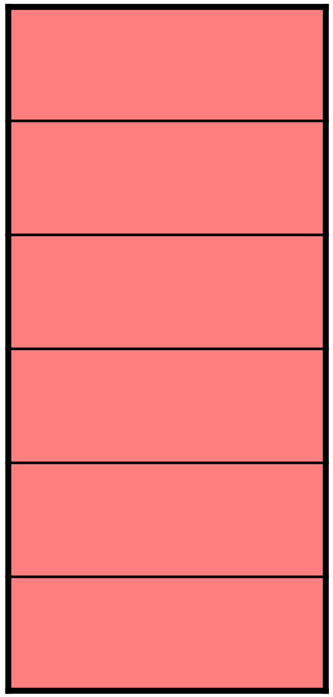
- Algorithm to generate Fibonacci numbers.
- Why is this inefficient?

Algorithm Fibonacci(n)

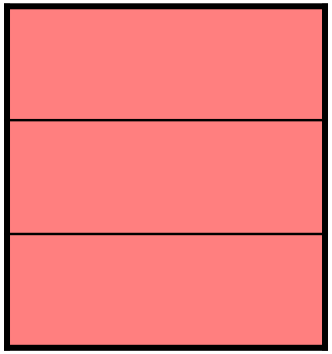
```
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Single recursion

- A recursive algorithm with a single recursive call still provides a linear chain of calls



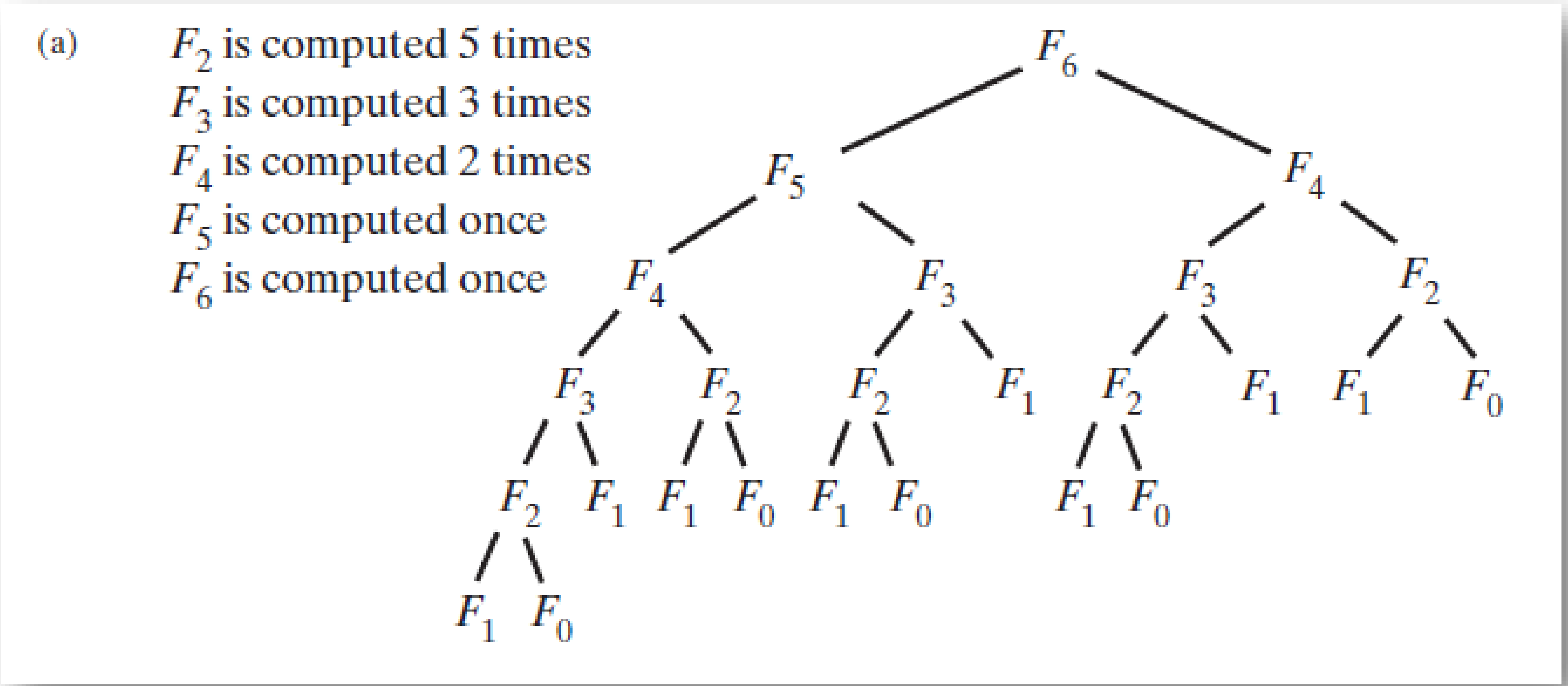
Calls build run-time stack



Stack shrinks as calls finish

Poor Solution to a Simple Problem

- The computation of the Fibonacci number F_6 using recursion



Double recursion

- When a recursive algorithm has 2 calls, the execution trace is now a ***binary tree***, as we saw with the trace on the board of Fibonacci
  - This execution is more difficult to do without recursion
    - To do it, programmer must create and maintain his/her own stack to keep all of the various data values
    - This increases the likelihood of errors / bugs in the code
- Later we will see some other classic recursive algorithms with multiple calls
  - Ex: MergeSort, QuickSort

Converting Recursion into Iteration

- Can we tell if a recursive algorithm can be easily done in an iterative way?
  - Yes – any recursive algorithm that is exclusively tail recursive can be done simply using iteration without recursion
  - Some algorithms we have seen so far are tail recursive

Tail Recursion

- So, what is tail recursion?
  - Recursive algorithm in which the recursive call is the LAST statement of the method
- What are the implications of tail recursion?
  - Any tail recursive algorithm can be converted into an iterative algorithm in a methodical way
    - some compilers do this automatically

Tail Recursion

- When the last action performed by a recursive method is a recursive call.

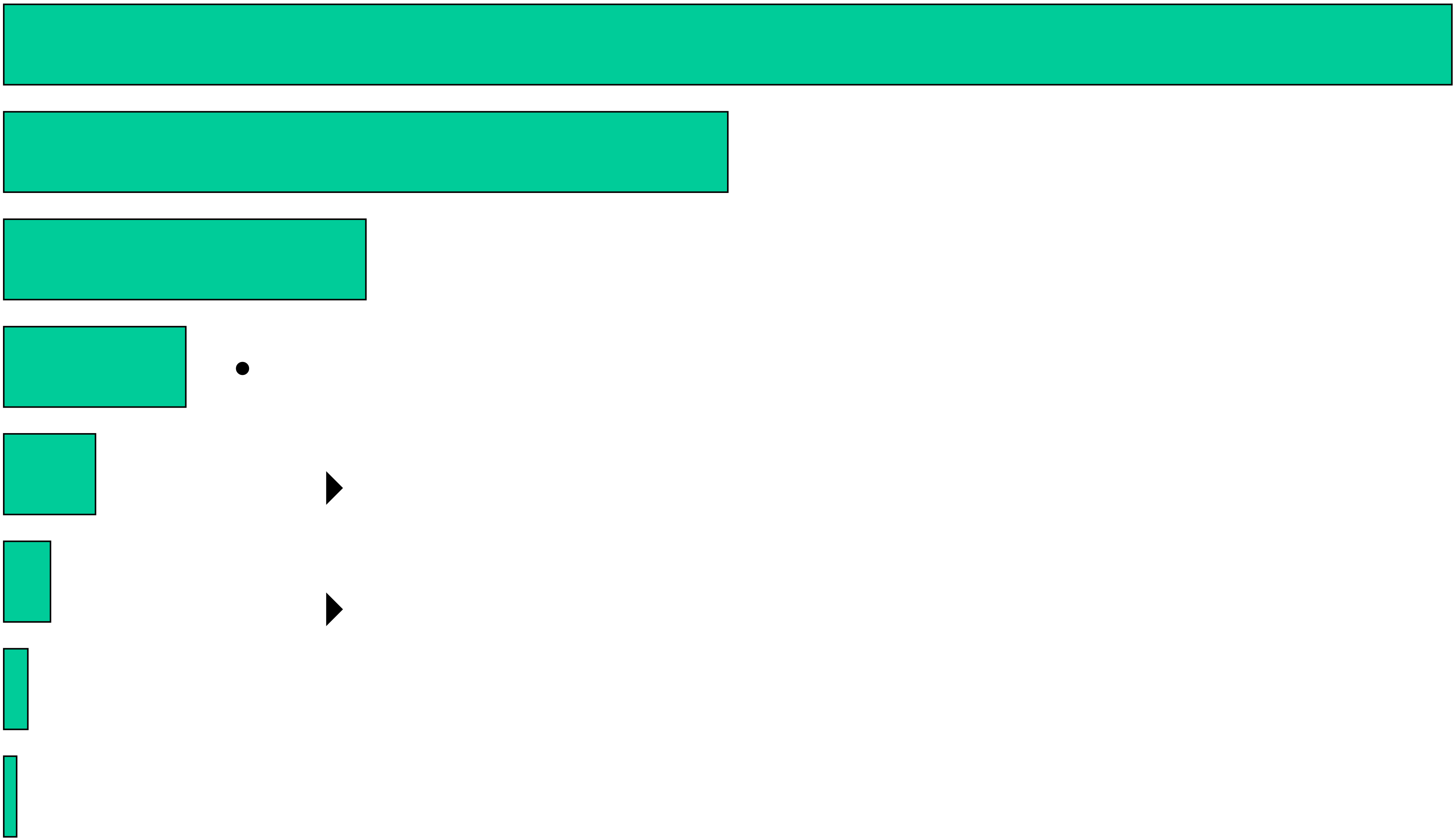
```java
public static void countDown(int integer)
{
    if (integer >= 1)
    {
        System.out.println(integer);
        countDown(integer - 1);
    } // end if
} // end countDown
```

Overhead of Recursion

- Why do we care?
  - Recursive algorithms have overhead associated with them
    - Space: each activation record (AR) takes up memory in the run-time stack (RTS)
      - If too many calls "stack up" memory can be a problem
    - Time: generating ARs and manipulating the RTS takes time
      - A recursive algorithm will always run more slowly than an equivalent iterative version

- Divide and Conquer
  - The idea is that a problem can be solved by breaking it down to one or more "smaller" problems in a systematic way
    - Usually the subproblem(s) are a fraction of the size of the original problem
    - Usually the subproblems(s) are identical in nature to the original problem
    - It is fairly clear why these algorithms can typically be solved quite nicely using recursion

Recursion and Divide and Conquer



Recursion Applications

- So what else is recursion good for?

1) For some problems, a recursive approach is more natural and simpler to understand than an iterative approach
   - Once the algorithm is developed, if it is tail recursive, we can always convert it into a faster iterative version
2) For some problems, it is very difficult to even conceive an iterative approach, especially if multiple recursive calls are required in the recursive solution

- Example: Backtracking problems

Recursion and Backtracking

- Idea of backtracking:
  - Proceed forward to a solution until it becomes apparent that no solution can be achieved along the current path
    - At that point UNDO the solution (backtrack) to a point where we can again proceed forward
  - Example: 8 Queens Problem
    - How can I place 8 queens on a chessboard such that no queen can take any other in the next move?
      - Recall that queens can move horizontally, vertically or diagonally for multiple spaces

8 Queens Problem

- How can we solve this with recursion and backtracking?
  - We note that all queens must be in different rows and different columns, so each row and each column must have exactly one queen when we are finished
    - Complicating it a bit is the fact that queens can move diagonally
  - So, thinking recursively, we see the following
    - To place 8 queens on the board we need to
      - Place a queen in a legal (row, column)
      - Recursively place 7 queens on the rest of the board
  - Where does backtracking come in?
    - Our initial choices may not lead to a solution – we need a way to undo a choice and try another one

8 Queens Problem

- Using this approach we come up with the solution as shown in 8-Queens handout
  - 8Queens.java
- Idea of solution:
  - Each recursive call attempts to place a queen in a specific column
    - A loop is used, since there are 8 squares in the column
  - For a given call, the state of the board from previous placements is known (i.e. where are the other queens?)
    - This is used to determine if a square is legal or not
  - If a placement within the column does not lead to a solution, the queen is removed and moved "down" the column

8 Queens Problem

  - When all rows in a column have been tried, the call terminates and backtracks to the previous call (in the previous column)
  - If a queen cannot be placed into column i, do not even try to place one onto column i+1 – rather, backtrack to column i-1 and move the queen that had been placed there
  - See handout for code details

- Why is this difficult to do iteratively?
  - We need to store a lot of state information as we try (and un-try) many locations on the board
    - For each column so far, where has a queen been placed?

8 Queens Problem

- The run-time stack does this automatically for us via activation records
  - Without recursion, we would need to store / update this information ourselves
  - This can be done (using our own Stack rather than the run-time stack), but since the mechanism is already built into recursive programming, why not utilize it?
- There are many other famous backtracking problems
  - http://en.wikipedia.org/wiki/Backtracking