Chapter 6

Recursion

Introduction

- An object is said to be *recursive* if it is defined in terms of a smaller version of itself
- Recursion is encountered not only in mathematics, but also in daily life

Introduction

Recursion in Mathematics

- Recursion is a particularly powerful means in mathematics
- The power of recursion lies in *the possibility of defining an infinite set of objects by a finite statement*

Recursion in Mathematics – Examples

- Natural numbers
  - 0 is a natural number
  - The successor of a natural number is a natural number
- The factorial of a number
  - $0! = 1$
  - If $n > 0$: $n! = (n-1)! \times n$
- The n^{th} power of a number
  - $x^0 = 1$
  - $x^n = x^{n-1} \times x$

Recursion in Mathematics

- Every recursive definition must have one (or more) base case(s)
- The general case must eventually be reduced to a base case
- The base case stops the recursion

Recursion in Computer Programming

- A function that calls itself is called a *recursive function*

```
void message() {
  cout << "Hello" << endl;
  message();    // recursive function call
}

void message(int n) {
  if (n == 0)   return;
  cout << "Hello" << endl;
  message(n - 1); // recursive function call
}
message(5);    // function call
```

Solving Problems with Recursion

A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem

Solving Problems with Recursion – Example 1

Calculating $n!$

- Definition 1:
  - If $n = 0$: $n! = 1$
  - If $n > 0$: $n! = 1 \times 2 \times \ldots \times n$
- Definition 2:
  - If $n = 0$: $n! = 1$
  - If $n > 0$: $n! = (n-1)! \times n$

Solving Problems with Recursion – Example 1

```
int fact(int n) {
  if (n == 0)
    return 1;
  return  fact(n - 1) * n;
}
…
cout << fact(3) << endl;
```

Direct and Indirect Recursion

- A function is called *directly recursive* if it calls itself
- A function that calls another function and eventually results in the original function call is said to be *indirectly recursive*
  - Indirect recursion can be several layers deep
- A recursive function in which the last statement executed is a recursive call is called a *tail recursive function*

Solving Problems with Recursion – Example 2

Calculating the n^{th} power of x: x^n

- Definition 1:
  - If $n = 0$: $x^n = 1$
  - If $n > 0$: $x^n = x \times x \times \ldots \times x$
- Definition 2:
  - If $n = 0$: $x^n = 1$
  - If $n > 0$: $x^n = x^{n-1} \times x$

Solving Problems with Recursion – Example 2

```
int power(int x, int n) {
  if (n == 0)
    return 1;
  return power(x, n - 1) * x;
}
…
x = 2;
n = 3;
cout << power(x, n) << endl;
```

Solving Problems with Recursion – Example 3

Calculating the sum of all elements in an array

$$a_0 + a_1 + \cdots + a_{n-1}$$

- Let S_n be the sum of the first n elements in array a:

$$S_n = a_0 + a_1 + \cdots + a_{n-1}$$
$$S_n = S_{n-1} + a_{n-1}$$

where S_{n-1} is the notation that indicates the sum of the first $n-1$ elements in array a

- Of course, $S_1 = a_0$

Solving Problems with Recursion – Example 3

```
int sum(int a[], int n) {
  if (n == 1)
    return a[0];
  return sum(a, n - 1) + a[n - 1];
}
…
cout << sum(a, n) << endl;
```

Solving Problems with Recursion – Example 3

- Let's consider the subarray $a_l, a_{l+1}, \ldots a_r$ where $0 \le l \le r \le n-1$
- Let $S_{l,r}$ be the sum of the array being considered

$$S_{l,r} = a_l + a_{l+1} + \cdots + a_m + a_{m+1} + \cdots + a_r$$
$$= (a_l + \cdots + a_m) + (a_{m+1} + \cdots + a_r) = S_{l,m} + S_{m+1,r}$$

where $m = \left\lfloor {}^{(l+r)}/_{2} \right\rfloor$

- If $l = r$: $S_{l,r} = S_{l,l} = a_l$

➡️ The value of $S_{0,n-1}$ is what we want to find out

Solving Problems with Recursion – Example 3

```
int sum(int a[], int l, int r) {
  if (l == r)   // l ≠ 1
    return a[l];
  int m = (l + r) / 2;
  return sum(a, l, m) + sum(a, m + 1, r);
}
…
cout << sum(a, 0, n - 1) << endl;
```

Solving Problems with Recursion – Example 4

Verifying if the elements in an array are in ascending order

$$a_0 \leq a_1 \leq \cdots \leq a_{n-1}?$$

- The above array is in ascending order if it satisfies two conditions:
  1. The first $n - 1$ elements are in ascending order, and
  2. $a_{n-2} \leq a_{n-1}$
- If the array contains only one element (a_0), it must be in ascending order

Solving Problems with Recursion – Example 4

```
bool isSorted(int a[], int n) {
  if (n == 1)
    return true;
  if (a[n - 2] ≤ a[n - 1])
    return isSorted(a, n - 1);
  return false;
}
```

Solving Problems with Recursion – Example 5

Verifying if a string is a palindrome

- A *palindrome* is a string that reads the same from left to right as it does from right to left: radar, rotor, ABBA, …
- Formally, a palindrome can be defined as follows:

If a string is a palindrome, it must begin and end with the same letter. Further, when the first and last letters are removed, the resulting string must also be a palindrome

  - A string of length 1 is a palindrome
  - The *empty string* is a palindrome

Solving Problems with Recursion – Example 5

```
bool isPal(string s) {
  if (s.size() ≤ 1)
    return true;
  else
    if (s[0] != s[s.size() - 1])
      return false;
    else
      return isPal(s.substr(1, s.size() - 2));
}
```

Solving Problems with Recursion – Example 6

Finding the n^{th} element of the Fibonacci series

- $0,1,1,2,3,5,8,13,21, \ldots$
- Let $F(n)$ be the n^{th} element of the Fibonacci series

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

Solving Problems with Recursion – Example 6

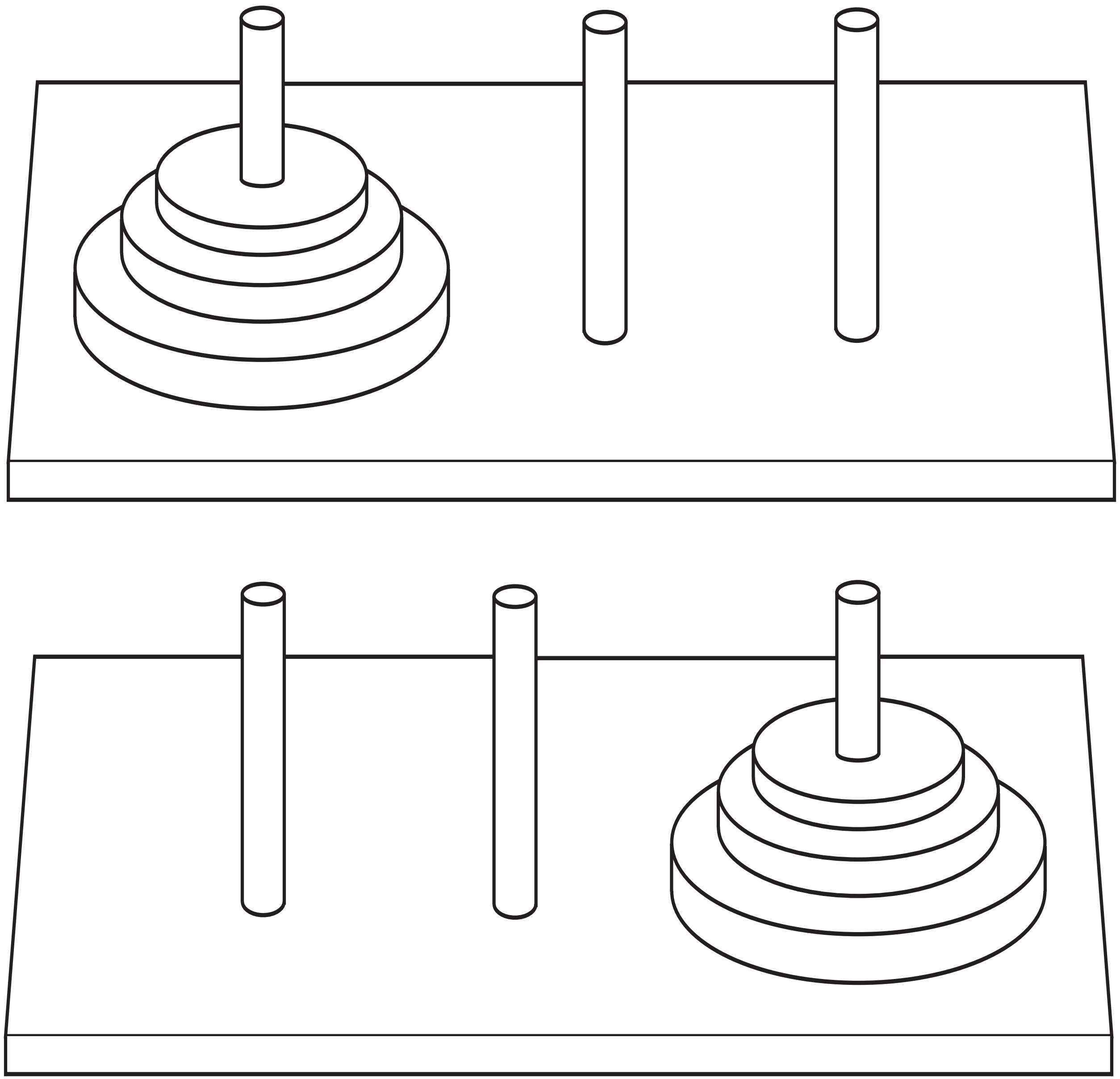
```
int Fib(int n) {      // n ≥ 0
  if (n <= 1)
    return n;
  return Fib(n - 1) + Fib(n - 2);
}
```

Solving Problems with Recursion – Binary Search

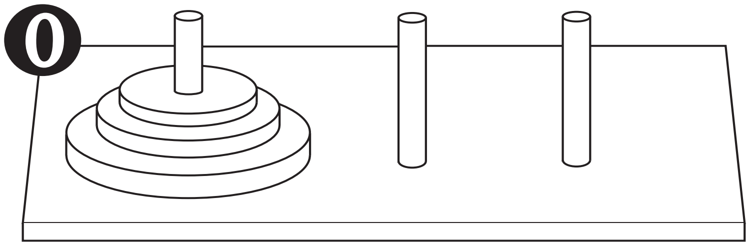
The binary search algorithm can also be implemented recursively

```
bool binSearch(int a[], int l, int r, int k) {
  if (l > r)    return false;
  int m = (l + r) / 2;
  if (a[m] == k)     return true;
  else
    if (a[m] < k)
      return binSearch(a, m + 1, r, k);
    else
      return binSearch(a, l, m - 1, k);
}
```

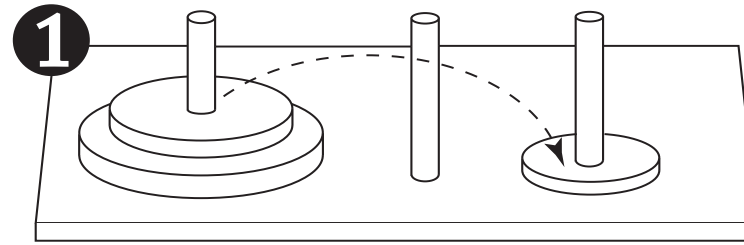
The Towers of Hanoi Puzzle



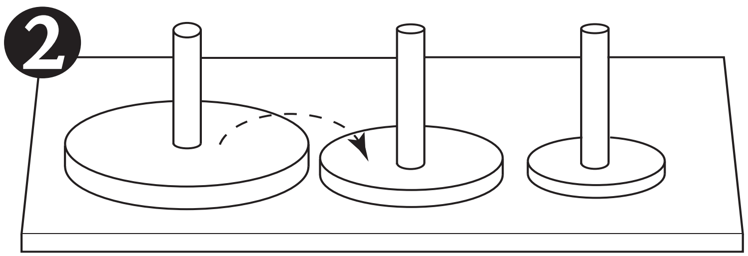
The Towers of Hanoi Puzzle – Example



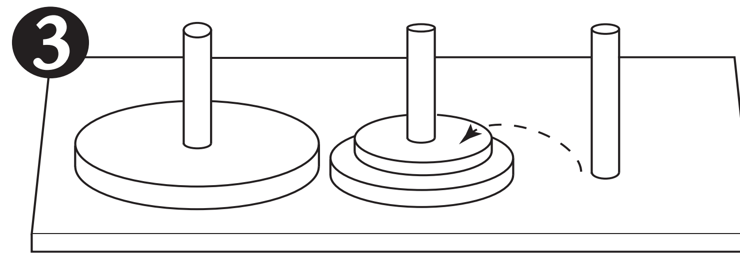
Original setup.



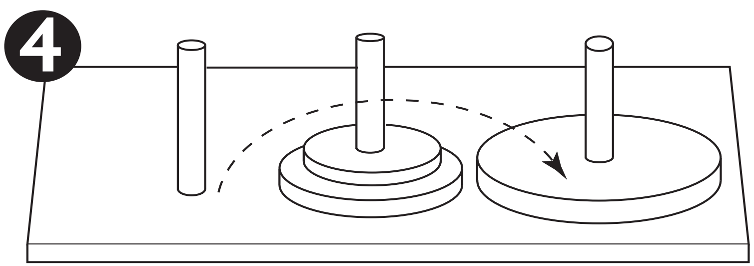
First move: Move disc 1 to peg 3.



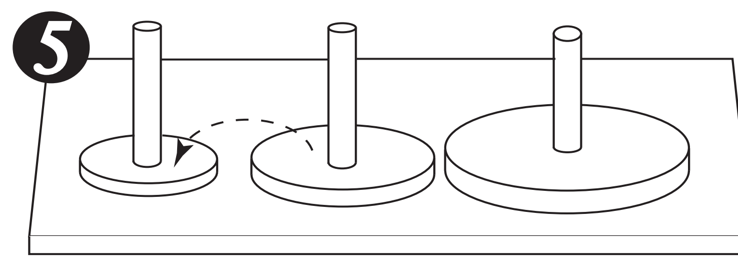
Second move: Move disc 2 to peg 2.



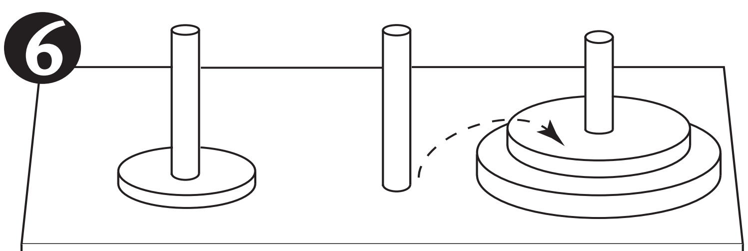
Third move: Move disc 1 to peg 2.



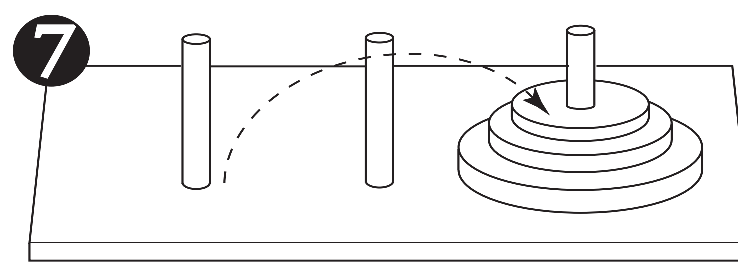
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.

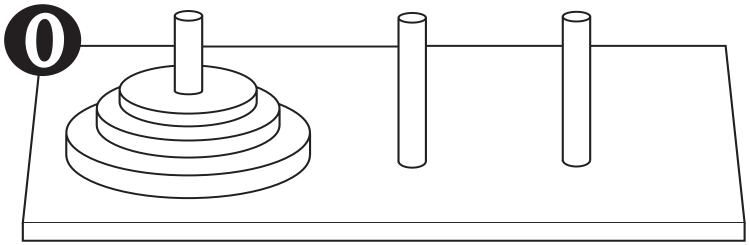


Sixth move: Move disc 2 to peg 3.

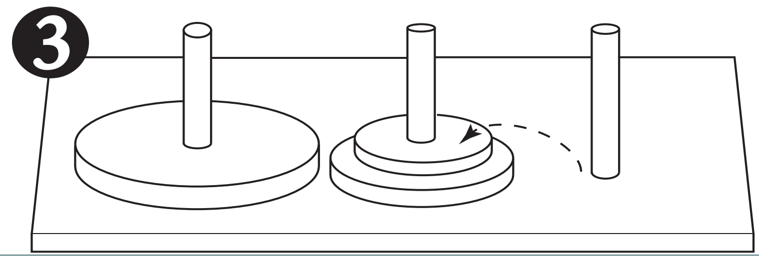


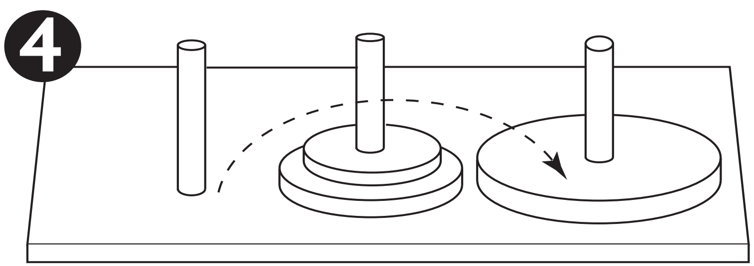
Seventh move: Move disc 1 to peg 3.

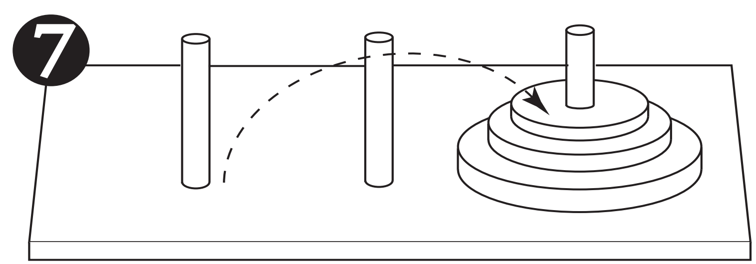
The Towers of Hanoi Puzzle – Example



Original setup.







The Towers of Hanoi Puzzle – Algorithm

// Move n discs from peg s to peg d using peg a as an auxiliary peg

If n > 0 **then**

Move n – 1 discs from peg s to peg a, using peg d as an auxiliary peg

Move the remaining disc from the peg s to peg d

Move n – 1 discs from peg a to peg d, using peg s as an auxiliary peg

End If

The Towers of Hanoi Puzzle – Source Code

```
void TowersOfHanoi(int n, int s, int a, int d)
{
  if (n) {
    TowersOfHanoi(n - 1, s, d, a);
    cout << "Move one disc from peg " << s <<
" to peg " << d << endl;
    TowersOfHanoi(n - 1, a, s, d);
  }
}
```

Recursion vs. Iteration

- Any algorithm that can be coded with recursion can also be coded with an iterative control structure
- Recursive algorithms are certainly less efficient than iterative algorithms
  - Each time a function is called, the system incurs overhead that is not necessary with a loop
- Some problems have a recursive essence, such as Towers of Hanoi puzzle, Quicksort, … then recursion approach should be your choice
  - It usually results in a better design