CS256

Chapter 19: Recursion

Hanoi University of Science and Technology

Slide contents are mainly based on the following documents:

- Tony Gaddis, Starting out with C++, Eighth Edition, Pearson, 2015
- Tony Gaddis, Power Points Slides:
  `https://www.umsl.edu/~antognolij/gaddis.html`

Introduction to Recursion

- A recursive function contains a call to itself:

```
void countDown(int num)
{
    if (num == 0)
        cout << "Blastoff!";
    else
    {
        cout << num << "...\n";
        countDown(num-1); // recursive call
    }
}
```

Solving Problems with Recursion

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
- The simpler-to-solve problem is known as the base case
- Recursive calls stop when the base case is reached

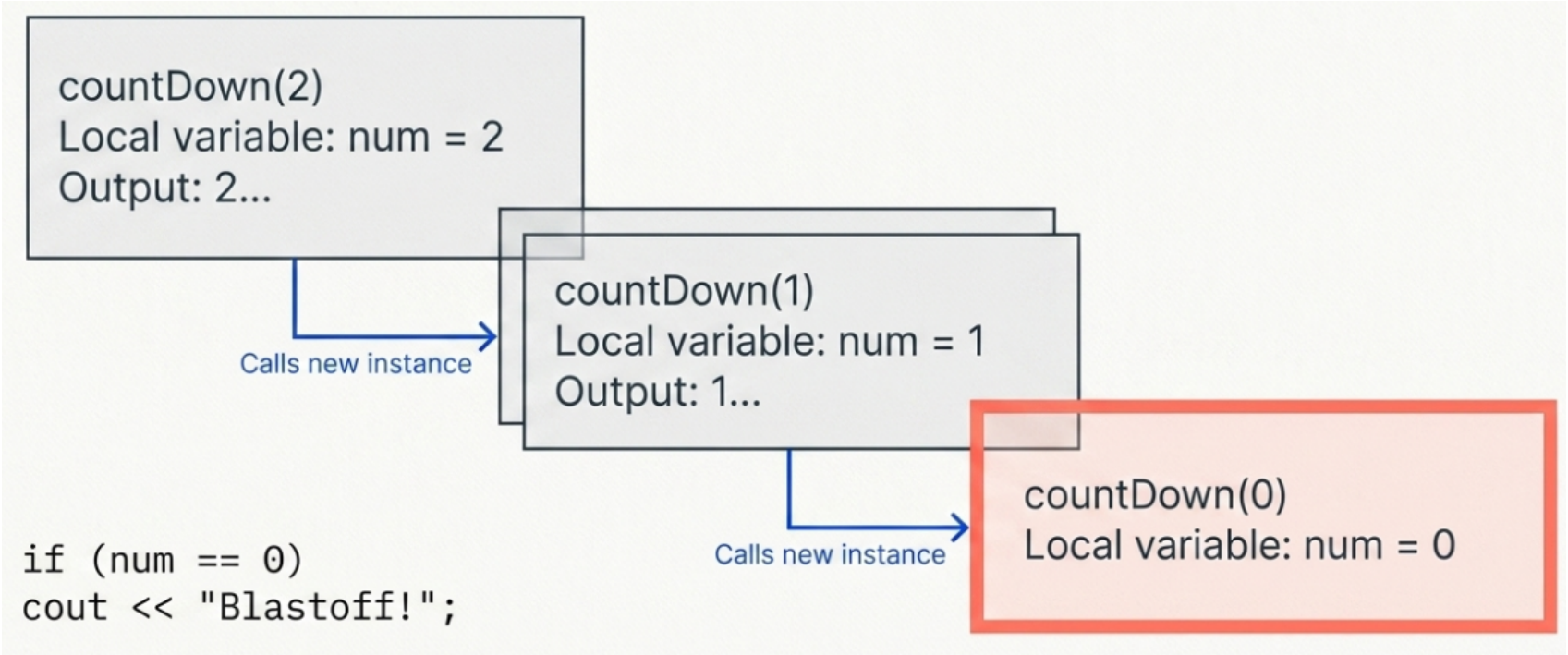
Stopping the Recursion

```
void countDown(int num)
{
     if (num == 0)
          cout << "Blastoff!";
     else
     {
          cout << num << "...\n";
          countDown(num-1);     // note that the value
     }                          // passed to recursive
                                // calls decreases by
                                // one for each call
}
```

Introduction to Recursion

```
void countDown(int num)
{
    if (num == 0);
        cout << "Blastoff!";
    else
    {
        cout << num << "...\n";
        countDown(num-1); // recursive call
    }
}
```

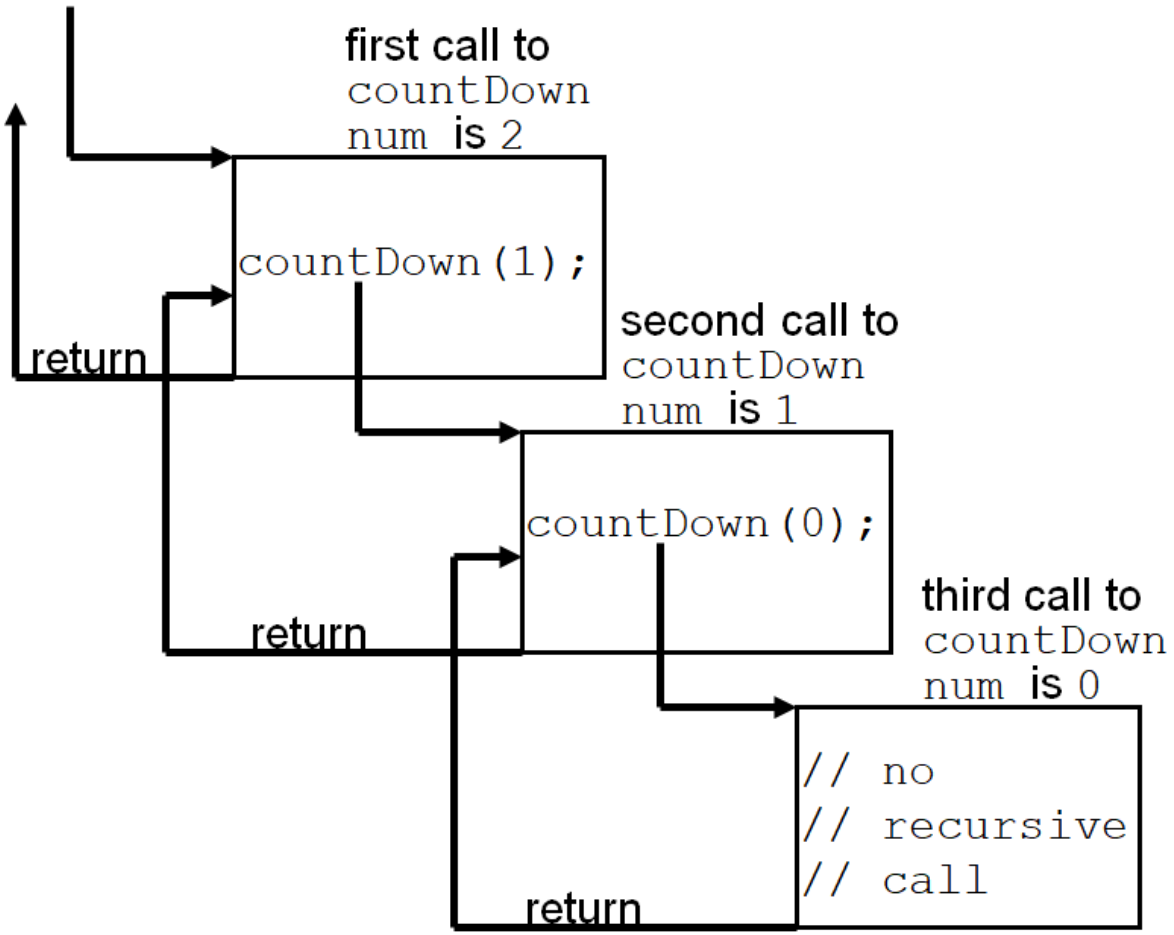
Introduction to Recursion



Stopping the Recursion

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

Stopping the Recursion



output:

```
2...

1...

Blastoff!
```

Types of Recursion

- Direct: a function calls itself
- Indirect:
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls, ..., which calls function A

The Recursive Factorial Function

```
int factorial (int num)
{
    if (num > 0)
        return num * factorial(num - 1);
    else
            return 1;
}
```

The Recursive gcd Function

- Greatest common divisor (gcd) is the largest factor that two integers have in common
- Computed using Euclid's algorithm:
  `gcd(x, y) = y` if $x\%y = 0$
  `gcd(x, y) = gcd(y, x % y)` otherwise

The Recursive gcd Function

```
int gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    else
        return gcd(y, x % y);
}
```

Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:
  0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- After the starting 0, 1, each number is the sum of the two preceding numbers
- Recursive solution:
  $$\text{fib(n)} = \text{fib(n} - 1) + \text{fib(n} - 2);$$
- Base cases: $n = 0$, $n = 1$

Solving Recursively Defined Problems

```
int fib(int n)
{
  if (n <= 0)
    return 0;
  else if (n == 1)
    return 1;
  else
      return fib(n - 1) + fib(n - 2);
}
```

Recursive Linked List Operations

- Recursive functions can be members of a linked list class
- Some applications:
  - Compute the size of (number of nodes in) a list
  - Traverse the list in reverse order

Counting the Nodes in a Linked List

- Uses a pointer to visit each node
- Algorithm:
  - Pointer starts at head of list
  - If pointer is NULL, return 0 (base case)
    else, return 1 + number of nodes in the list pointed to by current node

Counting the Nodes in a Linked List

```
int NumberList::countNodes(ListNode *nodePtr) const
{
     if (nodePtr != nullptr)
          return 1 + countNodes(nodePtr->next);
     else
          return 0;
}
```

Contents of a List in Reverse Order

- Pointer starts at head of list
- If the pointer is NULL, return (base case)
- If the pointer is not NULL, advance to next node
- Upon returning from recursive call, display contents of current node

Contents of a List in Reverse Order

```
void NumberList::showReverse(ListNode *nodePtr) const
{
    if (nodePtr != nullptr)
    {
        showReverse(nodePtr->next);
        cout << nodePtr->value << " ";
    }
}
```

A Recursive Binary Search Function

- Base cases: desired value is found, or no more array elements to search
- Algorithm (array in ascending order):
  - If middle element of array segment is desired value, then done
  - Else, if the middle element is too large, repeat binary search in first half of array segment
  - Else, if the middle element is too small, repeat binary search on the second half of array segment

A Recursive Binary Search Function

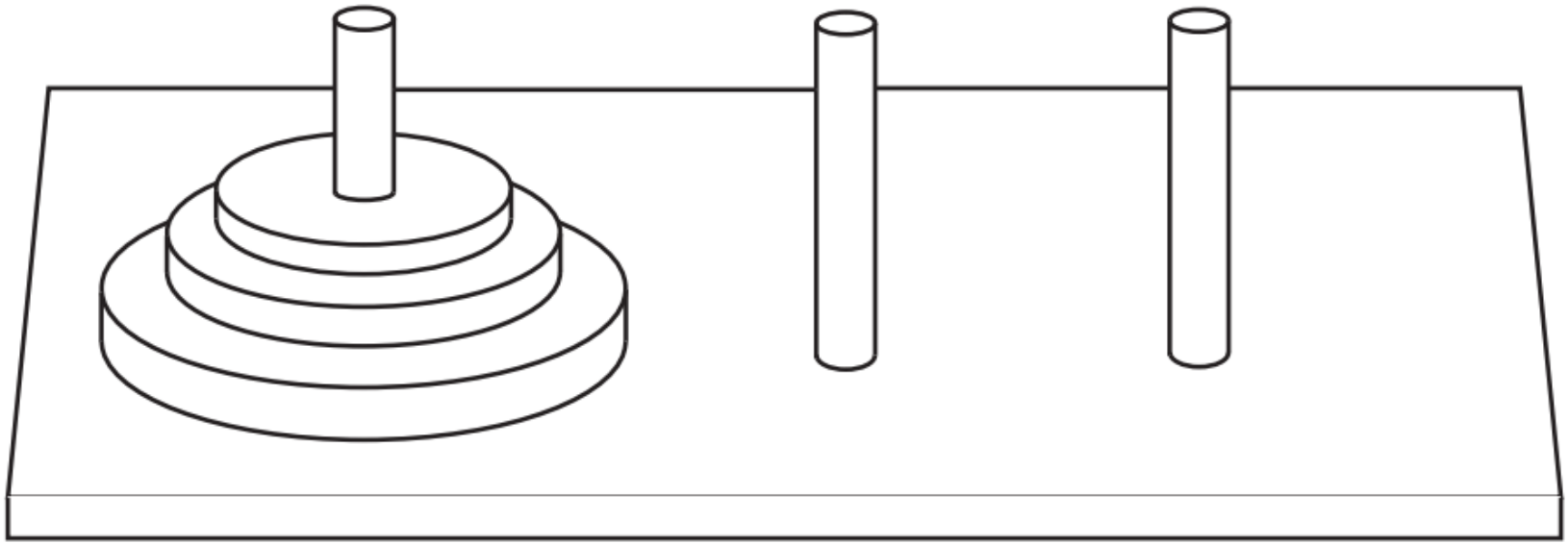
```
int binarySearch(int array[], int first, int last, int value)
{
    int middle; // Midpoint of search

    if (first > last)
        return -1;
    middle = (first + last)/2;

    if (array[middle]==value)
        return middle;
    if (array[middle]<value)
        return binarySearch(array, middle+1,last,value);
    else
        return binarySearch(array, first,middle-1,value);
}
```

The Towers of Hanoi

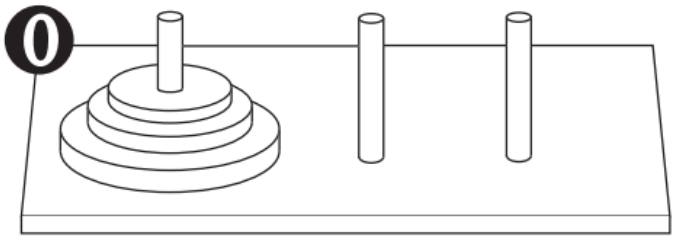
- The Towers of Hanoi is a mathematical game that is often used to demonstrate the power of recursion.
- The game uses three pegs and a set of discs, stacked on one of the pegs.



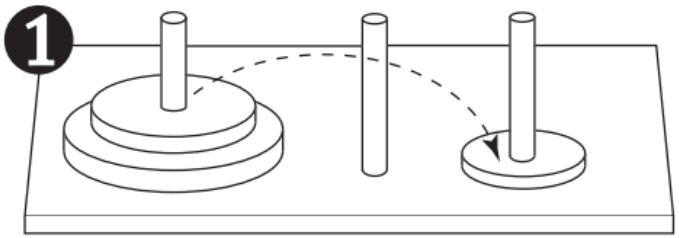
The Towers of Hanoi

- The object of the game is to move the discs from the first peg to the third peg. Here are the rules:
  - Only one disc may be moved at a time.
  - A disc cannot be placed on top of a smaller disc.
  - All discs must be stored on a peg except while being moved.

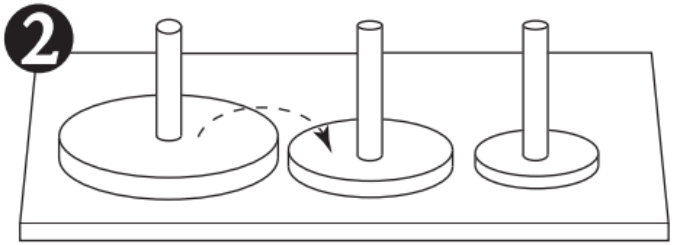
The Towers of Hanoi



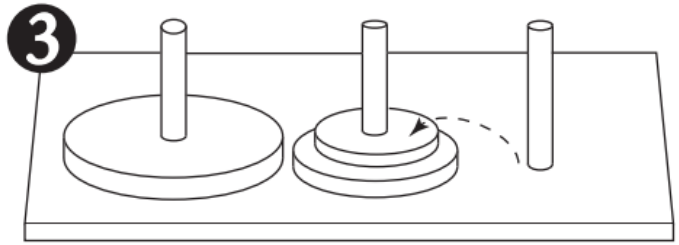
Original setup.



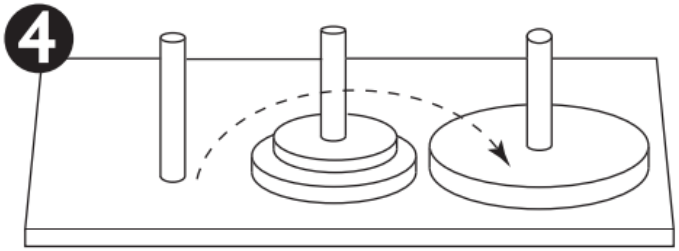
First move: Move disc 1 to peg 3.



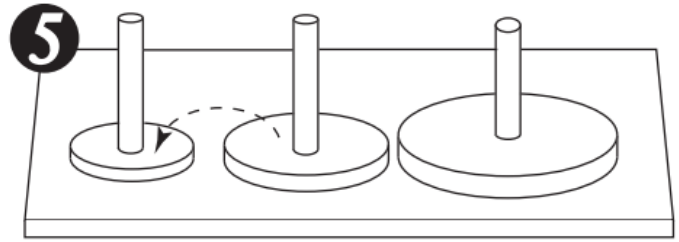
Second move: Move disc 2 to peg 2.



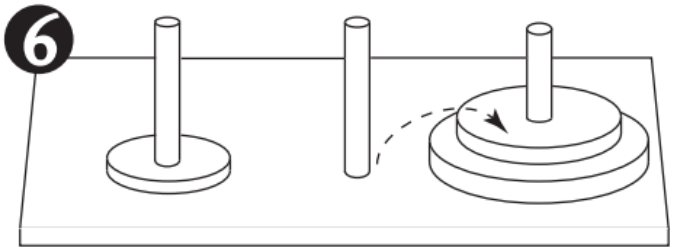
Third move: Move disc 1 to peg 2.



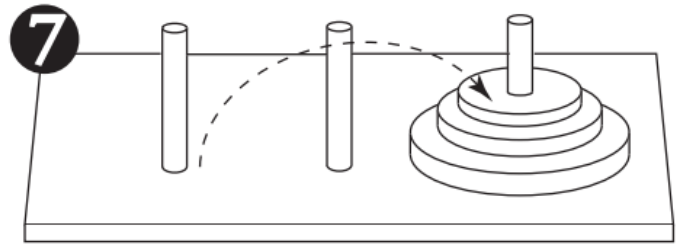
Fourth move: Move disc 3 to peg 3.



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

- The following statement describes the overall solution to the problem:
  - Move n discs from peg 1 to peg 3 using peg 2 as a temporary peg.

The Towers of Hanoi

Algorithm:

- To move n discs from peg A to peg C, using peg B as a temporary peg:
  If $n > 0$ Then
    Move n – 1 discs from peg A to peg B, using peg C as a temporary peg.
    Move the remaining disc from the peg A to peg C.
    Move n – 1 discs from peg B to peg C, using peg A as a temporary peg.
  End If

The Towers of Hanoi

```
    void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg
{
    if (num > 0)
    {
        moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
        cout << "Move a disc from peg " << fromPeg
             << " to peg " << toPeg << endl;
        moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
    }
}
```