CSCE 121
Introduction to Program Design & Concepts

Recursion

Dr. Tim McGuire

Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

First, a Digression

- Mathematicians and Telephones

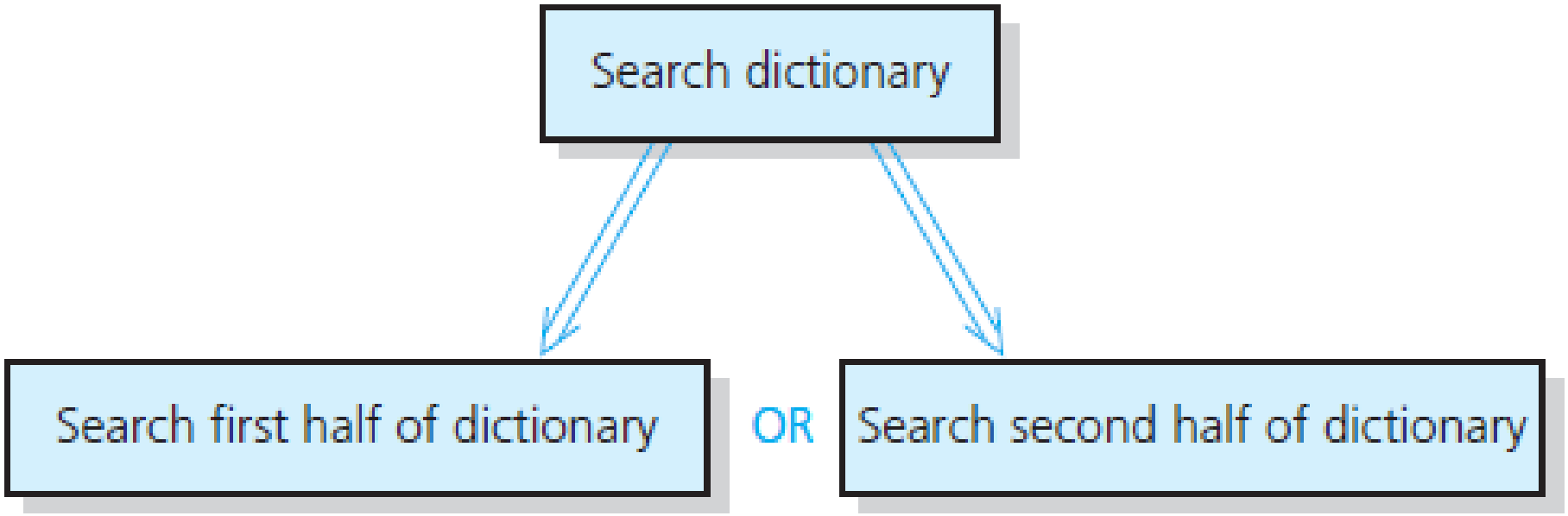
Image from stock.adove.com

$n! = 1$ if $n = 0$

$n! = n*(n\text{-}1!)$ if $n > 0$

Recursive Solutions (1 of 3)

- Recursion breaks problem into smaller identical problems
  - An alternative to iteration



Recursive Solutions (2 of 3)

- A recursive function calls itself
- Each recursive call solves an identical, but smaller, problem
- Test for base case enables recursive calls to stop
- Eventually, one of smaller problems must be the base case

Recursive Solutions (3 of 3)

Questions for constructing recursive solutions:

1. How to define the problem in terms of a smaller problem of same type?
2. How does each recursive call diminish the size of the problem?
3. What instance of problem can serve as base case?
4. As problem size diminishes, will you reach base case?

The Recursion Pattern

- Classic example--the factorial function:
  - n! = 1· 2· 3· ⋯ · (n-1)· n
- Recursive definition:

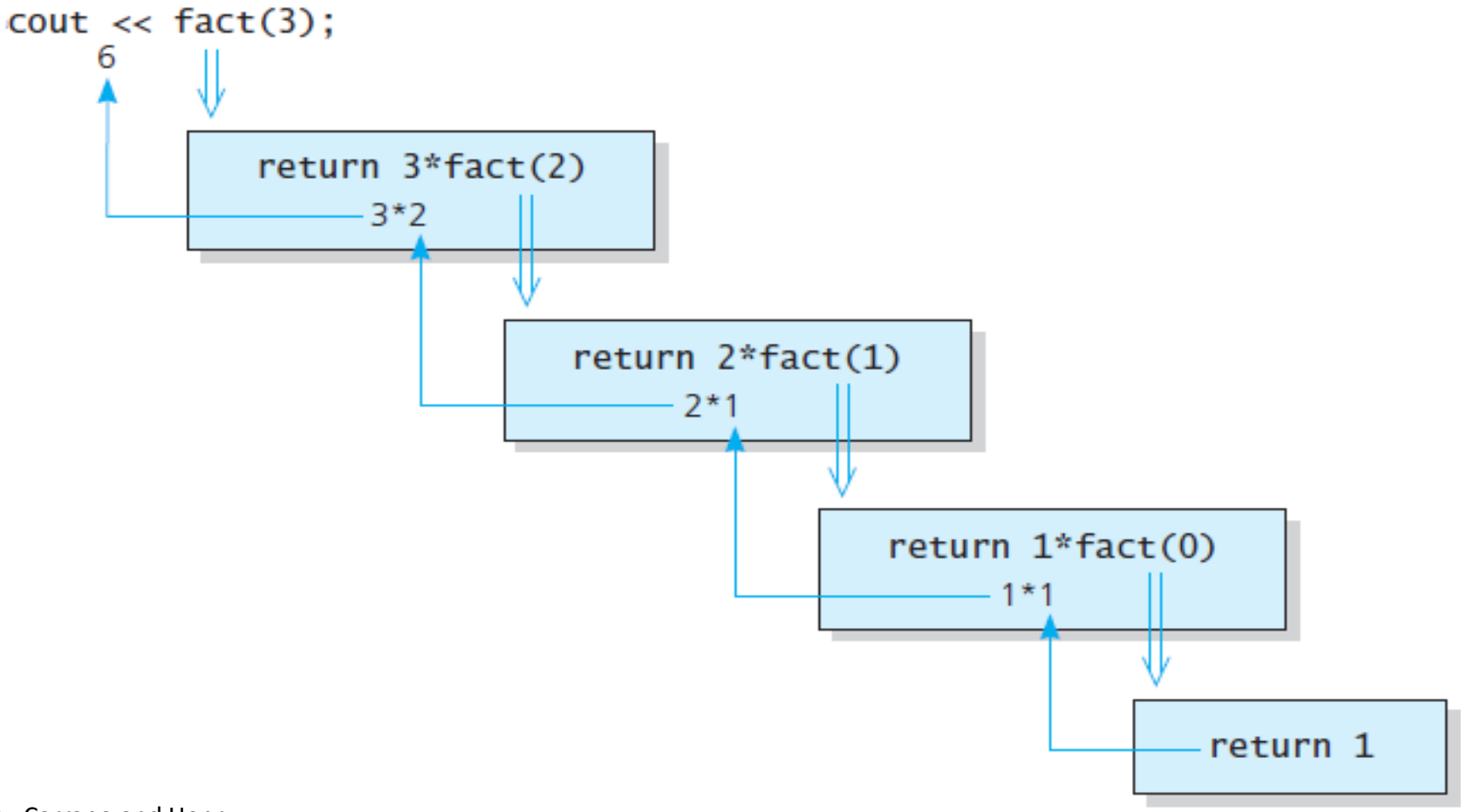
$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- As a C++ function:

```
// recursive factorial function
int  fact(int n)
{
    if  (n  ==  0)
        return  1;                  // base case
    else
        return  n  *  fact(n - 1); // recursive case
}
```

The Factorial of *n*

fact(3)



Source: *Walls & Mirrors*, by Carrano and Henry

Recursive Functions - Purpose

- 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

- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call
- In the factorial function, the test is:

```
if (n == 0)
```

Stopping the Recursion

- Recursion uses a process of breaking a problem down into smaller problems until the problem can be solved
- In the **`factorial`** function, a different value is passed to the function each time it is called
- Eventually, the parameter reaches the value in the test, and the recursion stops

Challenges

- What if you do not include a base case or get to the base case?
- Infinite recursion (think of an infinite loop)
- Each time the recursive call occurs, a new stack frame is created…
  - Eventually, you will run out of memory!
  - Stack Overflow

Rules for Good Recursive Function

- One or more base cases
  - Always returns without further recursion
- One or more recursive function calls
  - Must get closer to base case
  - Don‘t forget to use the result of your recursive call!

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

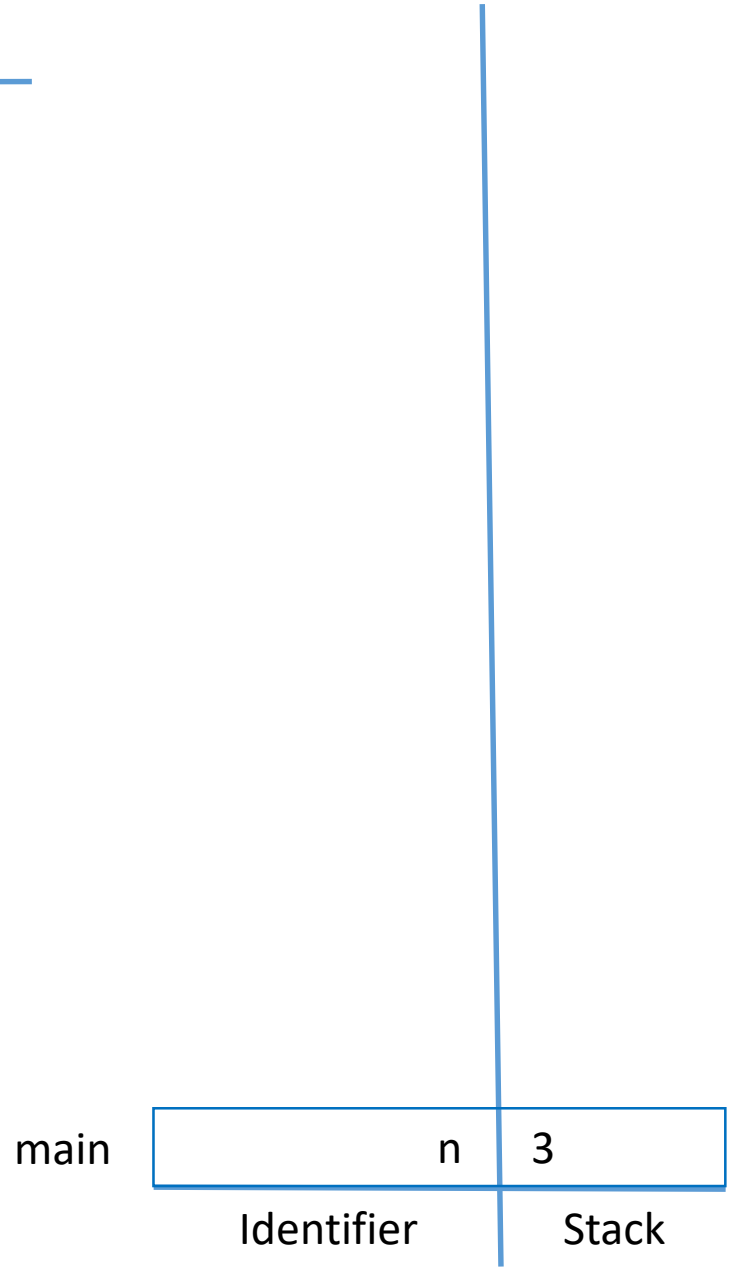
Recursion Memory Diagram

CSCE 121

```
void rPrint(int n) {
   if (n<=0) {
      return;
   }

   cout << n << endl;
   rPrint(n-1);
}

int main() {
   int n = 1;
   do {
      cout << "Enter Number: ";
      cin >> n;
      cout << "Start Printing" << endl;
      rPrint(n);
      cout << "End Printing" << endl;
   } while (n>0);
}
```

Output

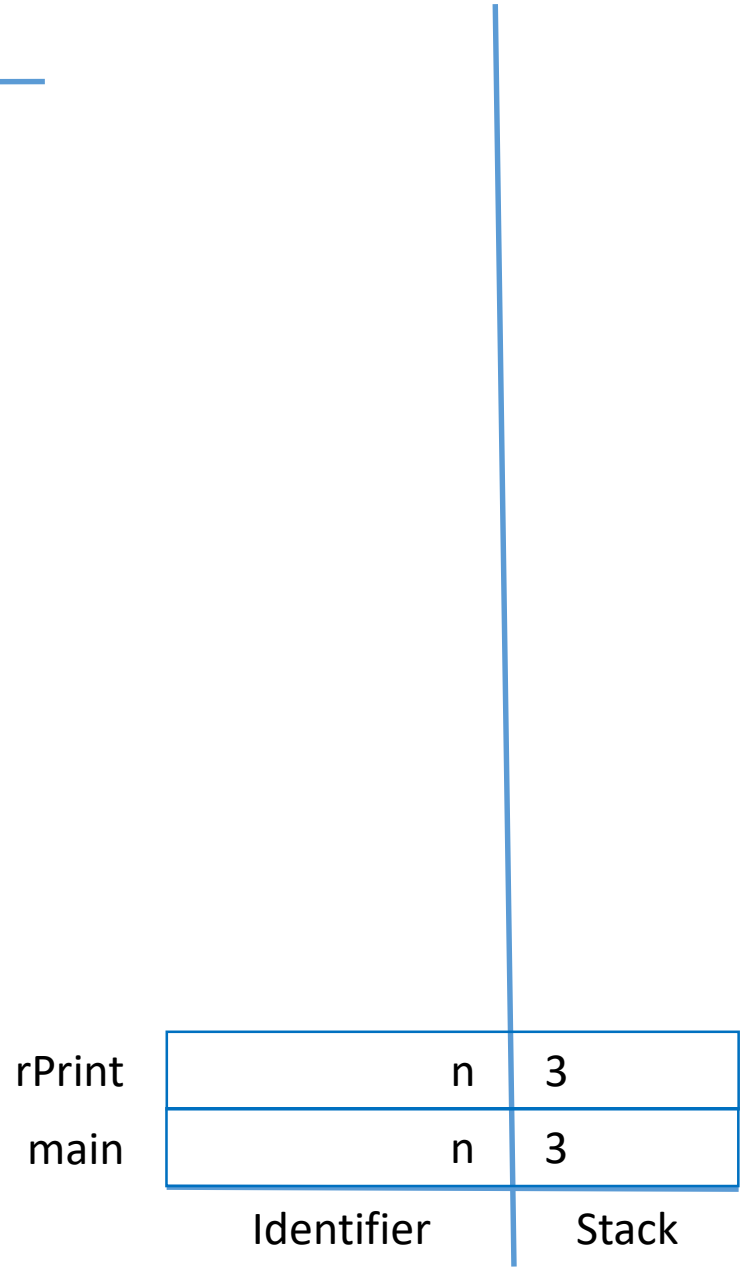
```
Enter Number: 3
Start Printing
```



```
void rPrint(int n) {
   if (n<=0) {
      return;
   }

   cout << n << endl;
   rPrint(n-1);
}

int main() {
   int n = 1;
   do {
      cout << "Enter Number: ";
      cin >> n;
      cout << "Start Printing" << endl;
      rPrint(n);
      cout << "End Printing" << endl;
   } while (n>0);
}
```

Output

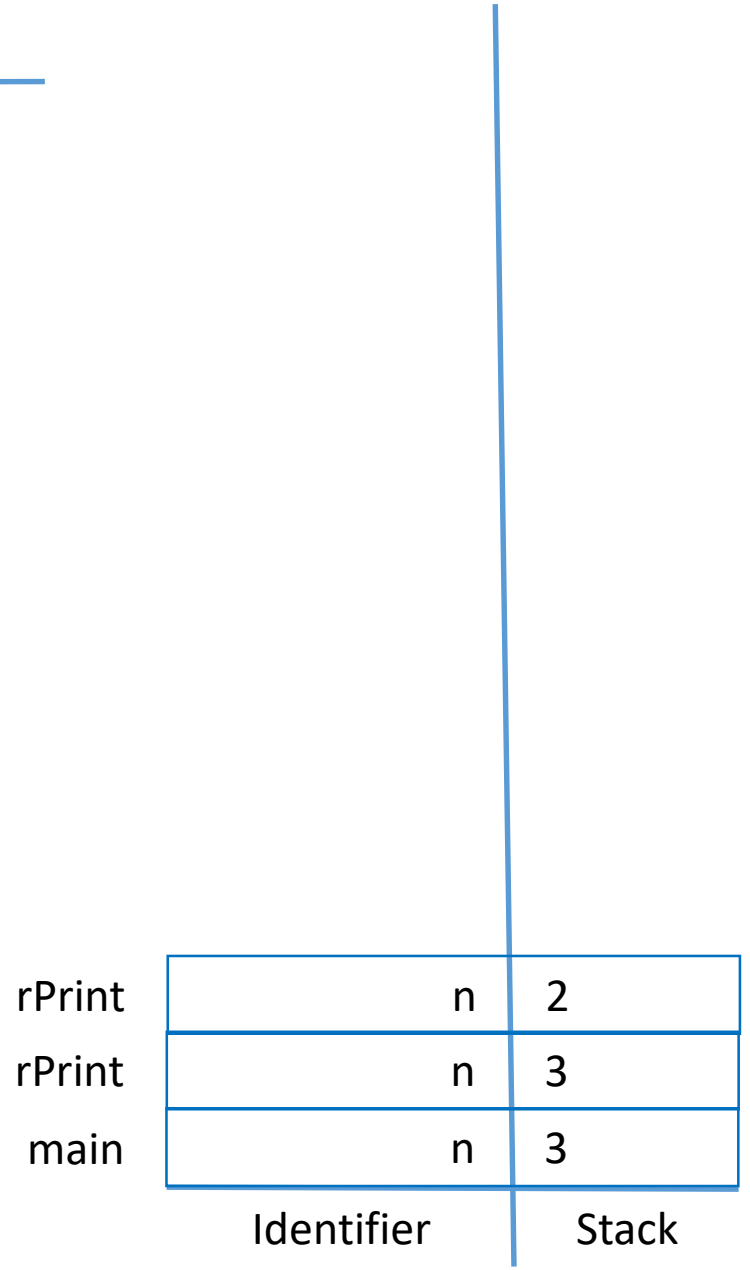
```
Enter Number: 3
Start Printing
3
```



```
void rPrint(int n) {
    if (n<=0) {
        return;
    }

    cout << n << endl;
    rPrint(n-1);
}

int main() {
    int n = 1;
    do {
        cout << "Enter Number: ";
        cin >> n;
        cout << "Start Printing" << endl;
        rPrint(n);
        cout << "End Printing" << endl;
    } while (n>0);
}
```

Output

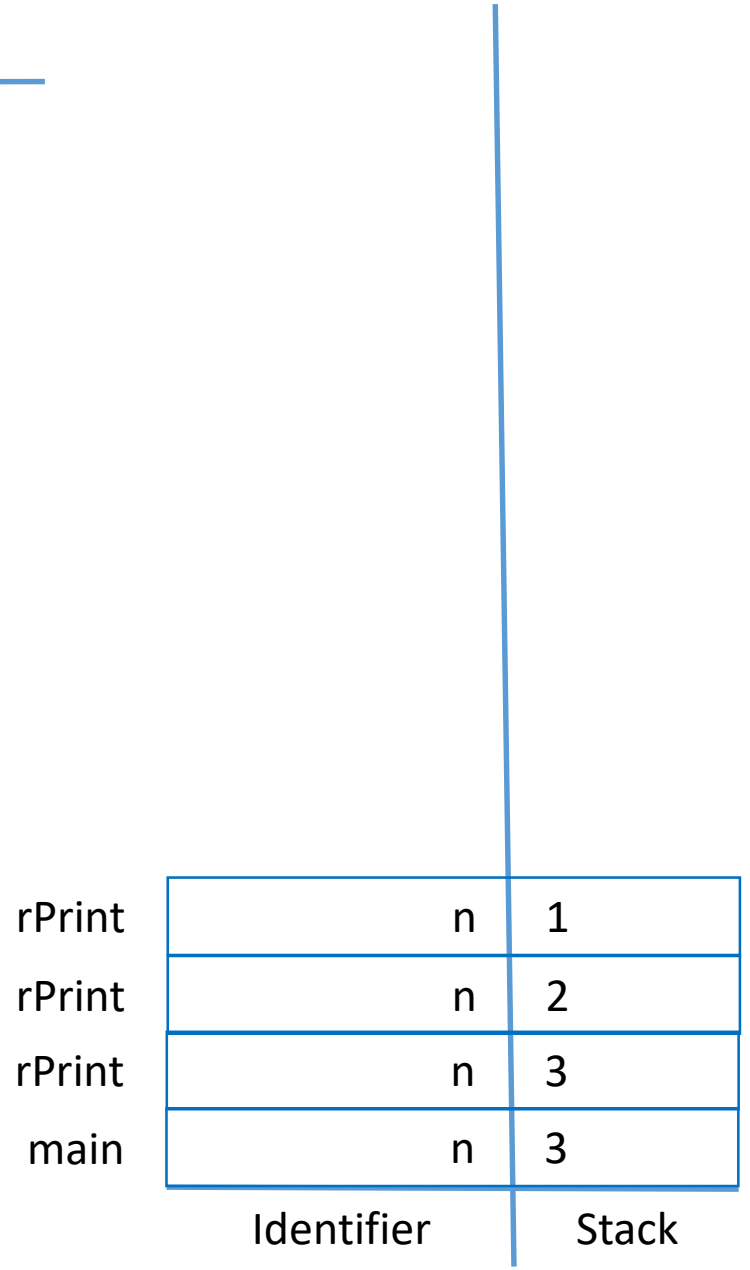
```
Enter Number: 3
Start Printing
3
2
```



```
void rPrint(int n) {
   if (n<=0) {
      return;
   }

   cout << n << endl;
   rPrint(n-1);
}

int main() {
   int n = 1;
   do {
      cout << "Enter Number: ";
      cin >> n;
      cout << "Start Printing" << endl;
      rPrint(n);
      cout << "End Printing" << endl;
   } while (n>0);
}
```

Output

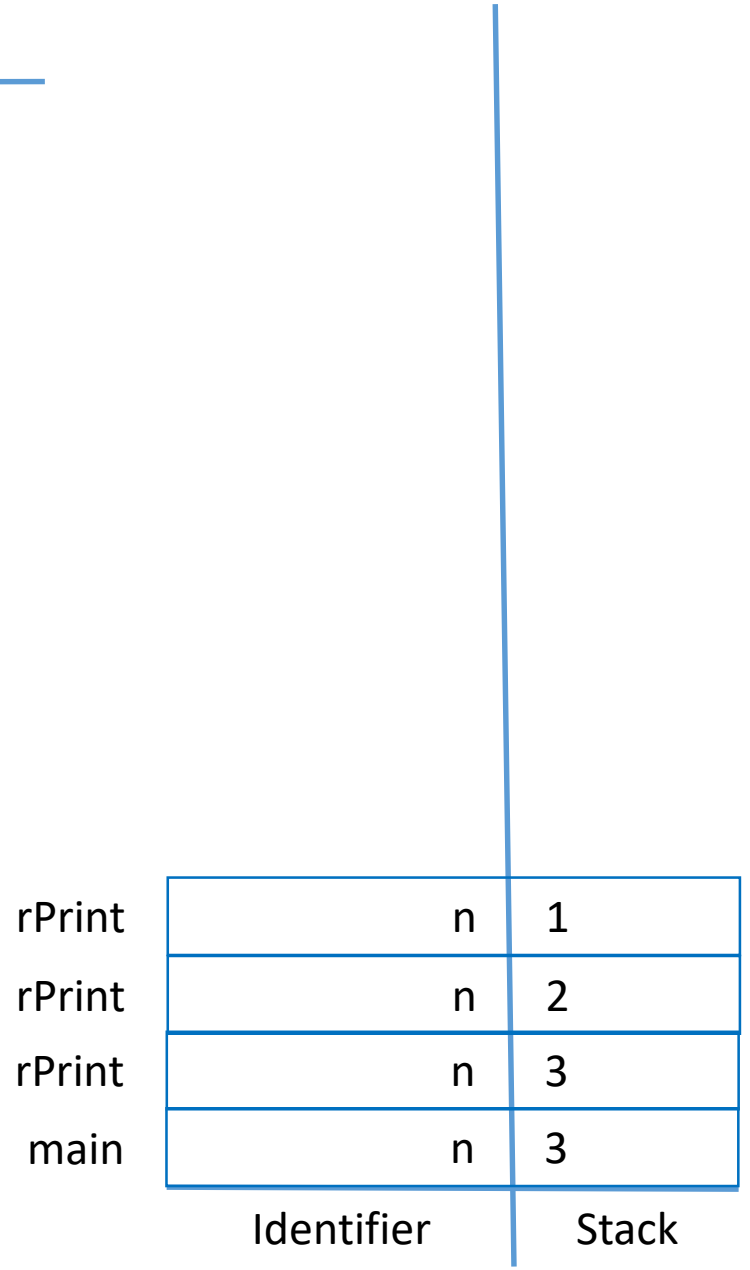
```
Enter Number: 3
Start Printing
3
2
1
```



```
void rPrint(int n) {
    if (n<=0) {
        return;
    }

    cout << n << endl;
    rPrint(n-1);
}

int main() {
    int n = 1;
    do {
        cout << "Enter Number: ";
        cin >> n;
        cout << "Start Printing" << endl;
        rPrint(n);
        cout << "End Printing" << endl;
    } while (n>0);
}
```

Output

```
Enter Number: 3
Start Printing
3
2
1
```

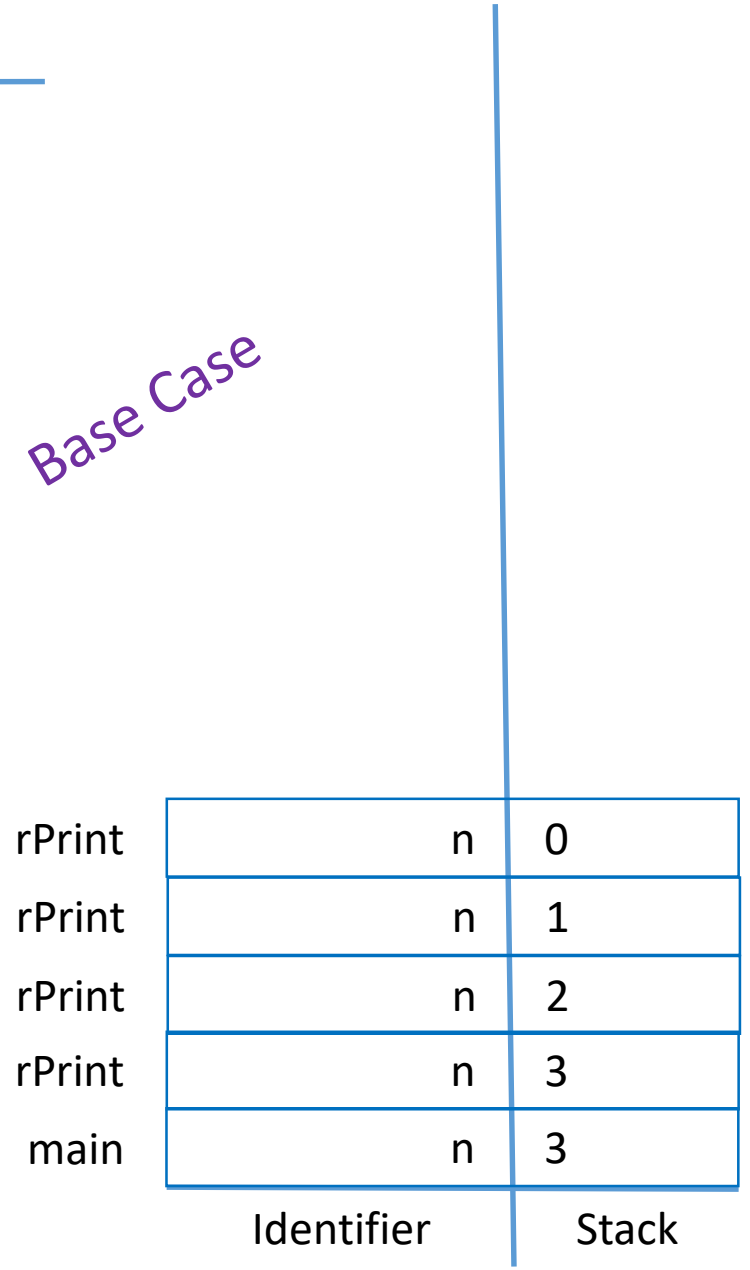


```
void rPrint(int n) {
    if (n<=0) {
        return;
    }

    cout << n << endl;
    rPrint(n-1);
}

int main() {
    int n = 1;
    do {
        cout << "Enter Number: ";
        cin >> n;
        cout << "Start Printing" << endl;
        rPrint(n);
        cout << "End Printing" << endl;
    } while (n>0);
}
```

Output

```
Enter Number: 3
Start Printing
3
2
1
```

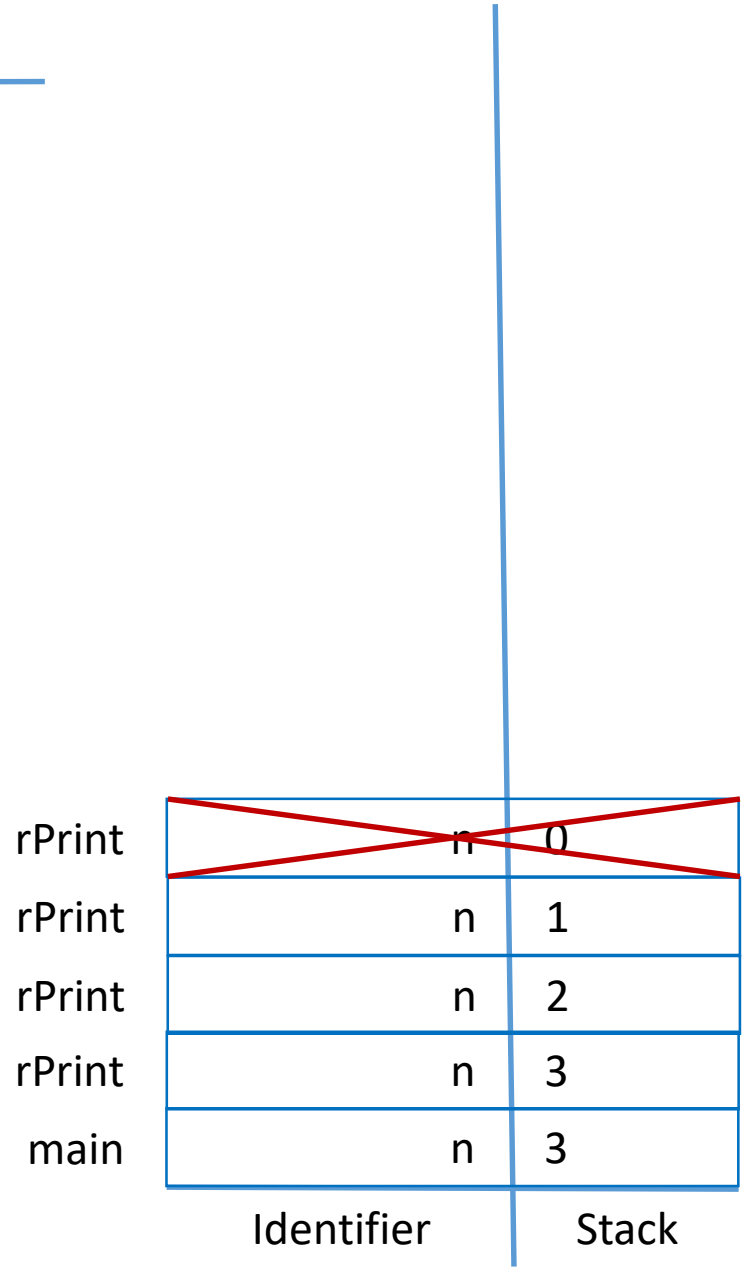
Base Case



```
void rPrint(int n) {
    if (n<=0) {
        return;
    }

    cout << n << endl;
    rPrint(n-1);
}

int main() {
    int n = 1;
    do {
        cout << "Enter Number: ";
        cin >> n;
        cout << "Start Printing" << endl;
        rPrint(n);
        cout << "End Printing" << endl;
    } while (n>0);
}
```

Output

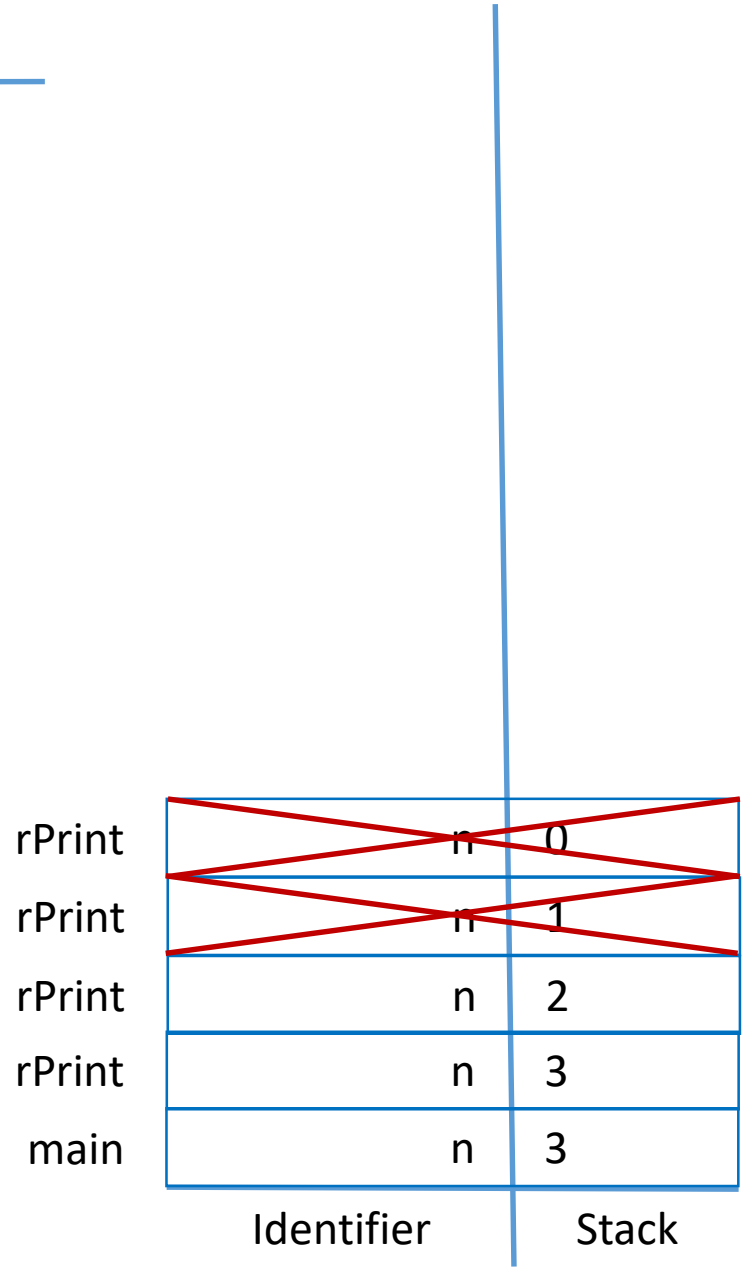
```
Enter Number: 3
Start Printing
3
2
1
```



```
void rPrint(int n) {
    if (n<=0) {
        return;
    }

    cout << n << endl;
    rPrint(n-1);
}

int main() {
    int n = 1;
    do {
        cout << "Enter Number: ";
        cin >> n;
        cout << "Start Printing" << endl;
        rPrint(n);
        cout << "End Printing" << endl;
    } while (n>0);
}
```

Output

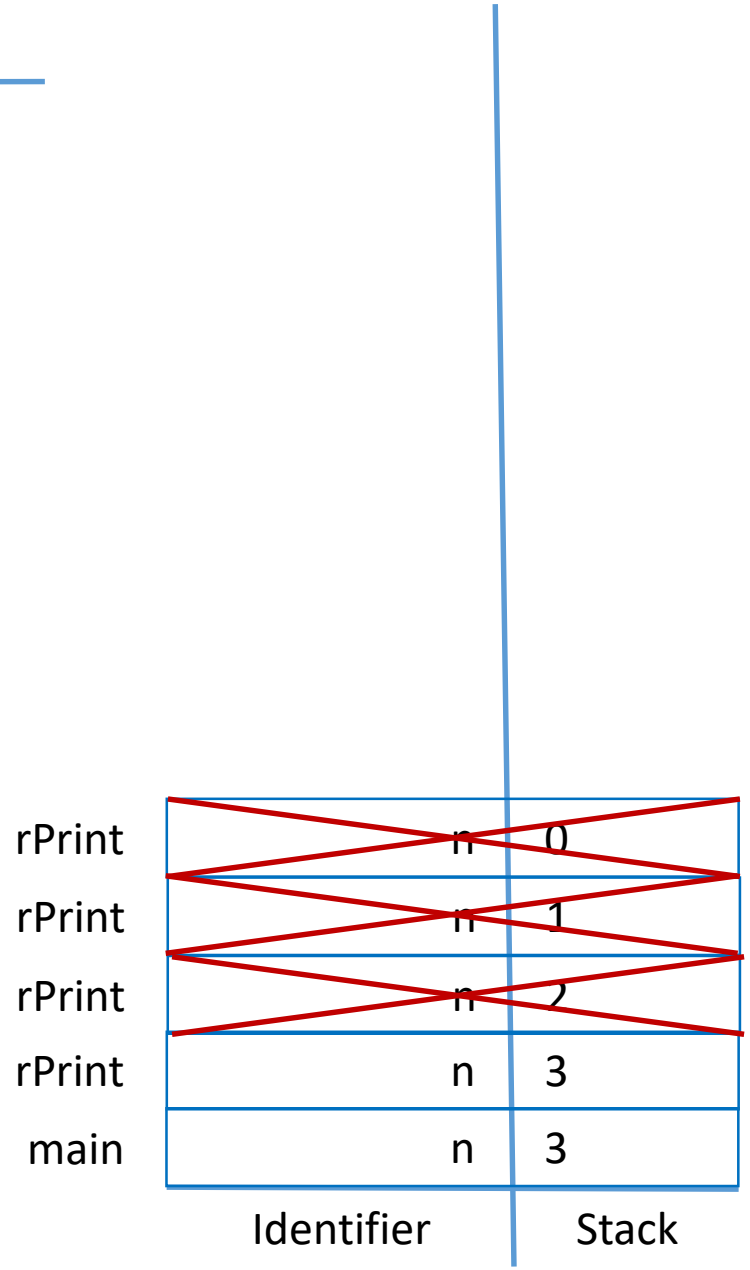
```
Enter Number: 3
Start Printing
3
2
1
```



```
void rPrint(int n) {
    if (n<=0) {
        return;
    }

    cout << n << endl;
    rPrint(n-1);
}

int main() {
    int n = 1;
    do {
        cout << "Enter Number: ";
        cin >> n;
        cout << "Start Printing" << endl;
        rPrint(n);
        cout << "End Printing" << endl;
    } while (n>0);
}
```

Output

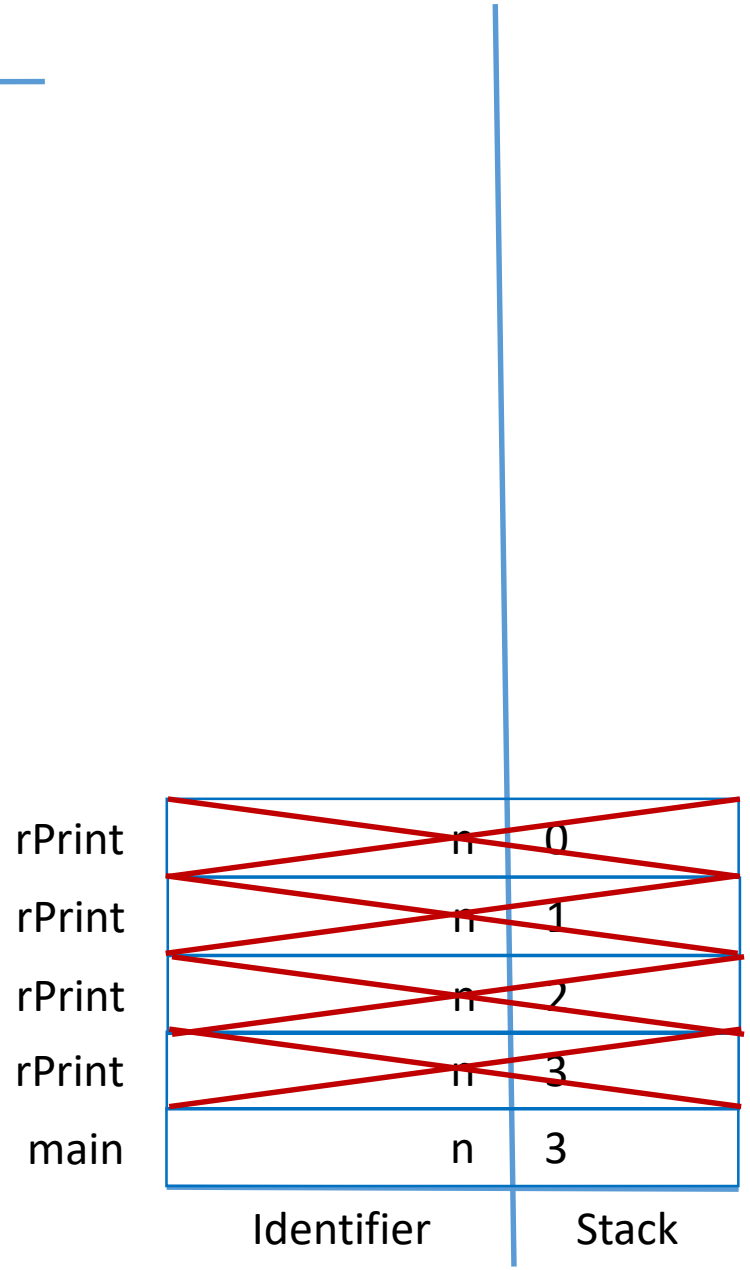
```
Enter Number: 3
Start Printing
3
2
1
```



```
void rPrint(int n) {
   if (n<=0) {
      return;
   }

   cout << n << endl;
   rPrint(n-1);
}

int main() {
   int n = 1;
   do {
      cout << "Enter Number: ";
      cin >> n;
      cout << "Start Printing" <<
endl;
      rPrint(n);
      cout << "End Printing" << endl;
   } while (n>0);
}
```

Output

```
Enter Number: 3
Start Printing
3
2
1
End Printing
```



Solving Recursively Defined Problems

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:

```
fib(n) = fib(n – 1) + fib(n – 2);
```

- Base cases: `n <= 0, n == 1`

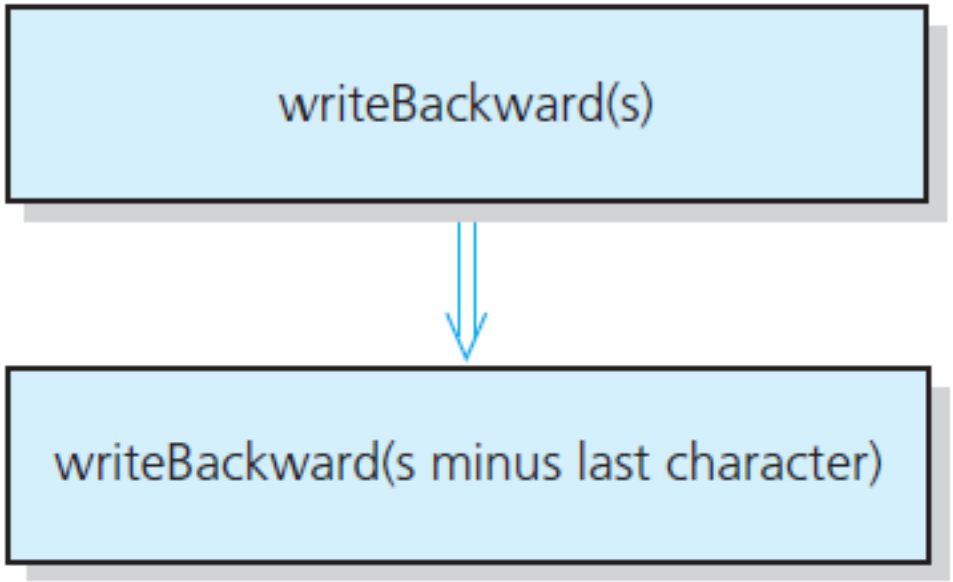
Solving Recursively Defined Problems

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

A Recursive Void Function: Writing a String Backward

- Likely candidate for minor task is writing a single character.
  - Possible solution: strip away the last character

A recursive solution



A Recursive Void Function: Writing a String Backward

- Tracing: Write BAT backwards
- The initial call is made: s = “BAT”, length of s = 3
  print T and write BA backwards
- The next call: s = “BA”, length of s = 2
  print A and write B backwards
- The next call: s = “B”, length of s = 1
  print B and write “” backwards
- The next call: s = “”, length of s = 0
  do nothing, since the string has no characters

- Output

  T A B

WriteBackwards.cpp

Search

- Common problem in computing

- Large data sets
- Want to find specific information.

Searching Arrays

- A sequential search is one way to search an array for a given value
  - Look at each element from first to last to see if the target value is equal to any of the array elements
  - The index of the target value can be returned to indicate where the value was found in the array
  - A value of -1 can be returned if the value was not found

Linear Search - Example

- Array `arr` contains:

17.1	23.2	5.0	11.3	2.5	29.4	3.8

- Searching for the the value `11.3`, linear search examines `17.1, 23.2, 5.0,` and `11.3`
- Searching for the the value `7`, linear search examines `17.1, 23.2, 5.0, 11.3, 2.5, 29.4,` and `3.8`

The search Function

- The search function
  - Uses a while loop to compare array elements to the target value
  - Sets a variable of type bool to true if the target value is found, ending the loop
  - Checks the boolean variable when the loop ends to see if the target value was found
  - Returns the index of the target value if found, otherwise returns -1

The search Function

```
int search(const double a[], int size, double target)
{
    int index = 0;
    bool found = false;
    while ((!found) && (index < size)) {
        if (target == a[index])
            found = true;
        else
            index++;
    }
    if (found)
        return index;
    else
        return -1;
}
```

Linear Search - Tradeoffs

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order

- Disadvantages:
  - Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

Binary Search

Requires array elements to be in order

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine

Binary Search - Example

- Array `ordered` contains:

2.5	3.8	5.0	11.3	17.1	23.2	29.4

- Searching for the value `11.3`, binary search examines `11.3` and stops
- Searching for the value `7.0`, linear search examines `11.3, 3.8, 5.0,` and stops

Iterative Binary Search

```
Set first index to 0.
Set last index to the last subscript in the array.
Set found to false.
Set position to -1.
While found is not true and first is less than or equal to last
    Set middle to the subscript half-way between array[first] and array[last].
    If array[middle] equals the desired value
        Set found to true.
        Set position to middle.
    Else If array[middle] is greater than the desired value
        Set last to middle - 1.
    Else
        Set first to middle + 1.
    End If.
End While.
Return position.
```

A Binary Search Function

```
int binarySearch(double array[], int size, double target)
{
   int first = 0,              // First array element
       last = size - 1,        // Last array element
       middle,                 // Mid point of search
       position = -1;          // Position of search value
   bool found = false;         // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;      // Calculate mid point
      if (array[middle] == target)      // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > target)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;            // If value is in upper half
   }
   return position;
}
```

Arrays/BinarySearch.cpp

Binary Search - Tradeoffs

- Benefits:
  - Much more efficient than linear search. For example, if an array has 1000 elements, performs at most 10 comparisons, as opposed to 1000 for linear search
  - (For 1,000,000 elements, only 20 comparisons!)

- Disadvantages:
  - Requires that array elements be sorted

Linear vs. Binary Search

- Value more clear when you have lots of values
- Suppose we have values ordered from smallest to largest.

Number of Integers	Linear Search Worst case number of elements examined	Binary Search Worst case number of elements examined
1,000,000,000	1,000,000,000	30
1,000,000,000,000,000,000,000 ($1X10^{21}$)	$1X10^{21}$	70
$1X10^{50}$	$1X10^{50}$	166

Sort???

- Use built in sort

```
#include <algorithm>
int values[100];
// load with values
```

- Or write your own… (Coming soon)

A Recursive Binary Search Function

A Recursive Binary Search Function

- Binary search algorithm can easily be written to use recursion
- 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 (Continued)

```
int binarySearch(double array[], int first, int last, double target)
{
   int middle; // Mid point of search

   if (first > last)
      return -1;
   middle = (first + last)/2;
   if (array[middle]==target)
      return middle;
   if (array[middle]<target)
      return binarySearch(array, middle+1,last,target);
   else
      return binarySearch(array, first,middle-1,target);
}
```

BinarySearch.cpp

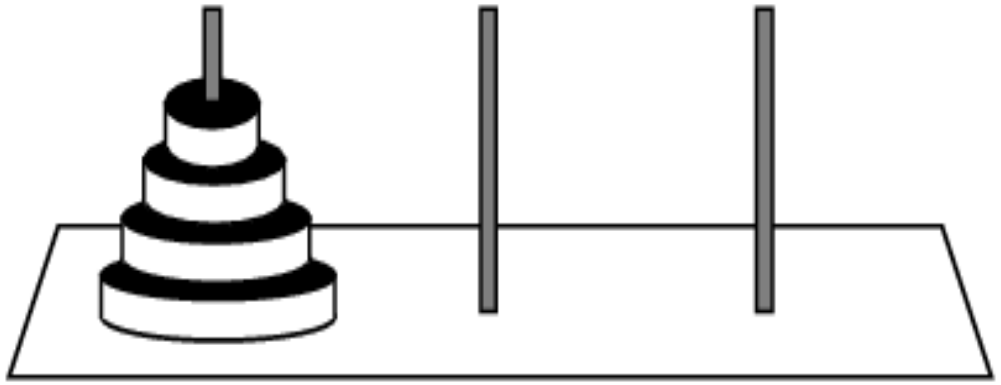
Recursion and Efficiency

- Some recursive solutions are so inefficient that they should not be used
- Factors that contribute to the inefficiency of some recursive solutions
  - Overhead associated with method calls
  - Inherent inefficiency of some recursive algorithms
- Do not use a recursive solution if it is inefficient and there is a clear, efficient iterative solution

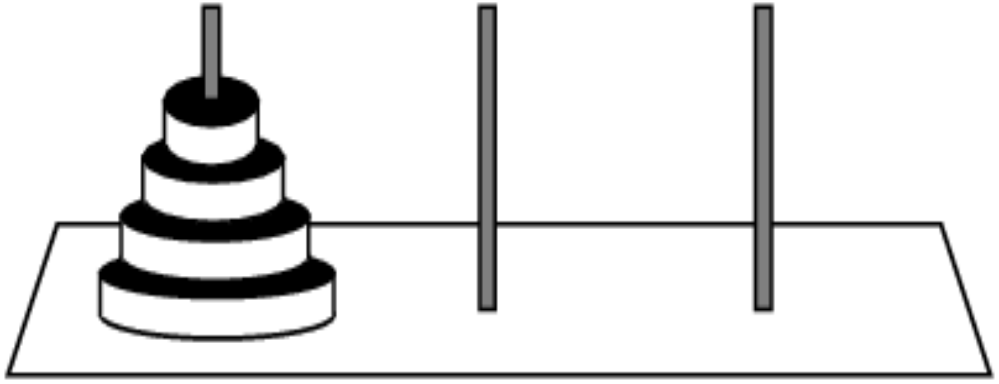
The Towers of Hanoi

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.

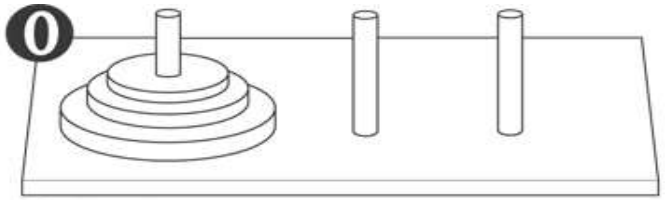


The Towers of Hanoi

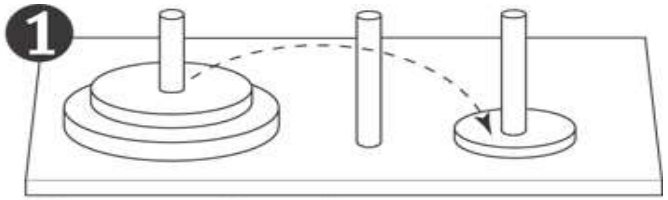


- The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
  - Only one disk can be moved at a time.
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
  - No larger disk may be placed on top of a smaller disk.
  - All discs must be stored on a peg except while being moved.

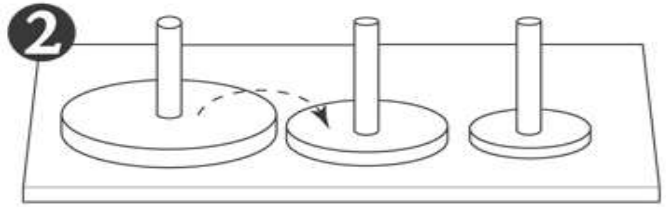
Solving with Three Discs



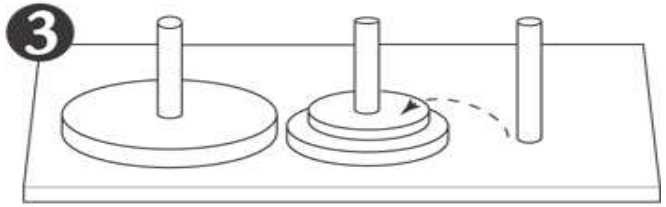
Original setup.



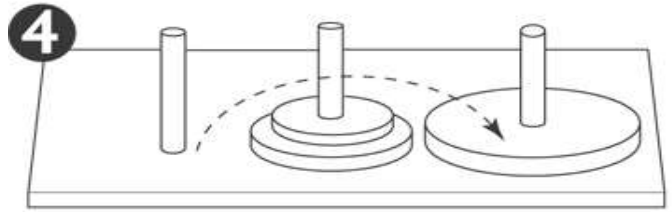
First move: Move disc 1 to peg 3.



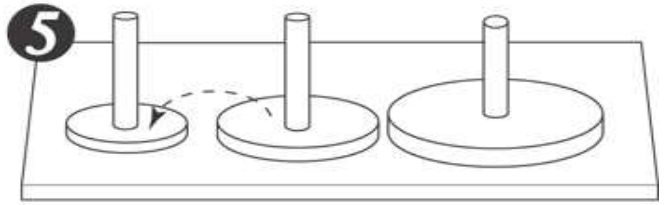
Second move: Move disc 2 to peg 2.



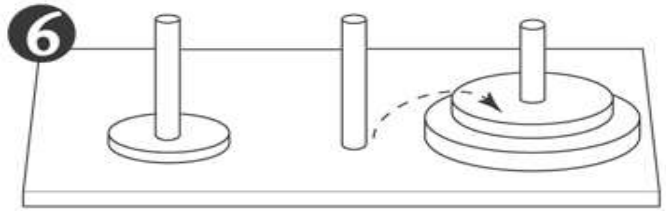
Third move: Move disc 1 to peg 2.



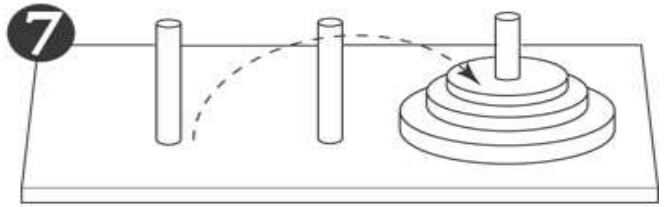
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.



Seventh move: Move disc 1 to peg 3.

Image source:
Tony Gaddis, *Starting out with Java*

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.

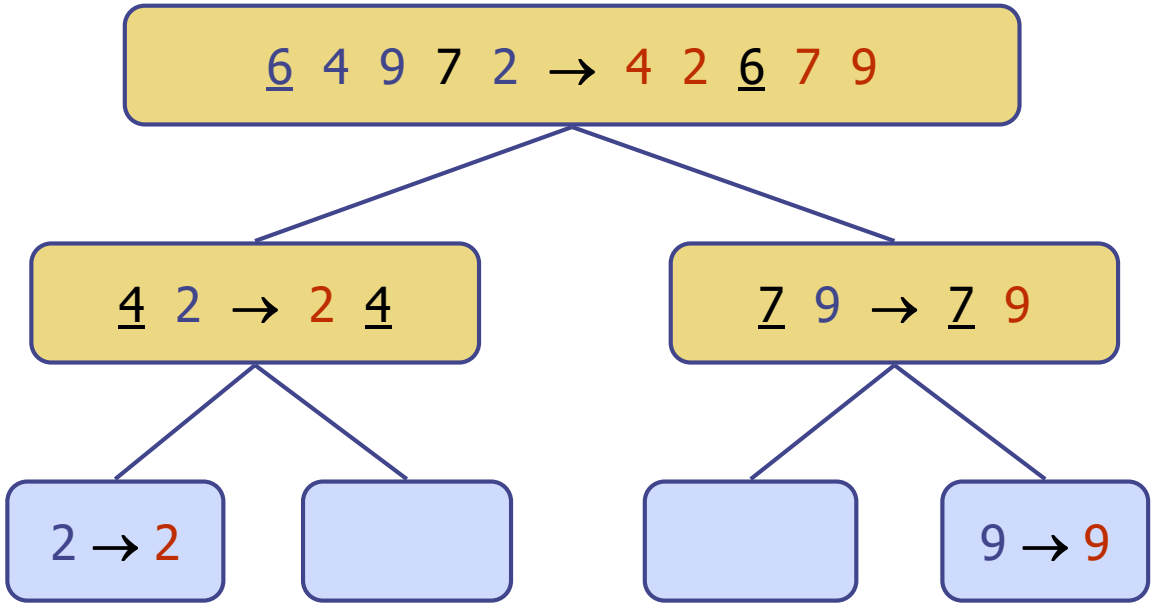
      Move n – 1 discs from peg B to peg C, using peg A as a temporary peg.

    End If

Hanoi.cpp

Hanoi2.cpp

Quick-Sort

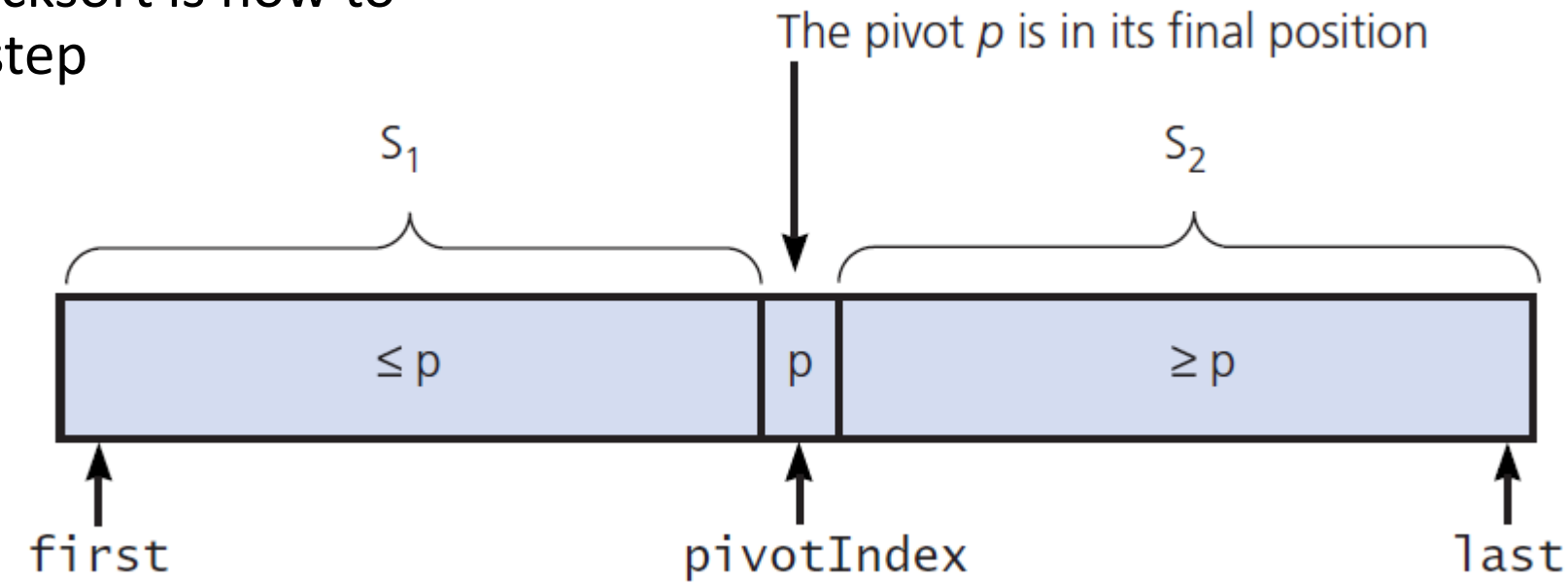


Quick Sort

- AKA “partition-exchange” sort
  - First developed by C.A.R Hoare c. 1960
  - Like binary search, it is a “divide-and-conquer” algorithm
- It partitions an array into items that are
  - Less than or equal to the pivot and
  - Those that are greater than or equal to the pivot
- Partitioning places pivot in its correct position within the array
- Then the two “halves” are sorted (using quicksort)

Quick Sort

The important part of Quicksort is how to perform the partitioning step



A partition around a pivot

First draft of pseudocode for the quick sort algorithm

The algorithm is coded primarily in two functions: `quickSort` and `partition`. `quickSort` is a recursive function.

quickSort:

IF *Starting Index < Ending Index*

Partition the List around a Pivot.

quickSort Sublist 1.

quickSort Sublist 2.

ENDIF

Here is the C++ code for the quickSort function:

```
void quickSort(int set[], int start, int end)
{
   int pivotPoint;
   if (start < end)
   {
      // Get the pivot point.
      pivotPoint = partition(set, start, end);
      // Sort the first sublist.
      quickSort(set, start, pivotPoint – 1);
      // Sort the second sublist.
      quickSort(set, pivotPoint + 1, end);
   }
}
```

Pseudocode for Lomuto's Partitioning algorithm

```
partition(set[], first, last)
    mid = ⌊first + last)/2⌋
    swap set[first] with set[mid]
    pivot = set[lo]
    s = first     // place for swapping
    for j := first +1 to last do
        if set[j] < pivot then
            s = s + 1
            swap set[s] with set[j]
        endif
    endfor
    swap set[s] with set[lo]
    return s
end partition
```

Partitioning Example

- 83 52 24 65 17 35 96 42
- 65 | 52 24 83 17 35 96 42 - swap 1st with midpoint
- 65 52 24 17 | 83 35 96 42 - scan until one is less than pivot, then swap with dividing element
- 65 52 24 17 35 | 83 96 42 - scan until one is less than pivot, then swap with dividing element
- 65 52 24 17 35 42 | 96 83
- 42 52 24 17 35 | 65 | 96 83 - at end, put pivot in proper place

Partitioning

- As mentioned, the hard part of quicksort is the partitioning of the array.
- There are several ways to do this.
  - The Zybook uses a slight variation of Tony Hoare's original partitioning technique.
  - Nico Lomuto's partitioning technique is easier to implement
    - It requires more element moves, but fewer comparisons than Hoare's

Partitioning Example

65 52 24 83 17 35 96 42

Partitioning Example

3 5 7 11 13 17 19 23 29