MIDDLE TENNESSEE STATE UNIVERSITY

CSCI 3110

Recursion

What's recursion?

What's recursion

- A powerful problem solving technique
- A special case of divide-and-conquer technique
  - Sub-problems have the exact type of the original type, with smaller size
- A function of accomplishing a task by invoking itself

How to construct recursive solution?

Four questions for constructing recursive solutions

- How can you define the problem in terms of a smaller problem of the same type?
- How does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?

MIDDLE TENNESSEE STATE UNIVERSITY

Algorithm Analysis, Recursion, and Sorting

CSCI 3110

1 Algorithm Efficiency
- Measuring the Efficiency of Algorithms
- Algorithm Growth Rates
- Keeping Your Perspective

2 Recursion?
- Examples
- Iteration vs. Recursion
- One Real Example: The Towers of Hanoi

3 Summary

4 Sorting Algorithms
Selection Sort
Bubble Sort
Insertion Sort
Mergesort Quick Sort Radix Sort
Summary

STL Sorting Algorithms

The Factorial of n

- Problem
  - Compute the factorial of an integer n
- An iterative definition of factorial(n)

  factorial(n) = n * (n-1) * (n-2) * ... * 1
  for any integer n > 0

  factorial(0) = 1
- A recursive definition of factorial(n)

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

The Factorial of n: two versions

- version 1:

```
int factorial( int n )
{
    if ( n == 0 )
        return 1;
    else
        return n * factorial( n-1 );
}
```

- version 2:

```
void factorial( int n, int& m ) {
    int     n1, m1;
    if ( n == 0 )
        m = 1;
    else
    {
        n1 = n - 1;
        factorial( n1, m1 );
        m = n * m1;
    }
    return;
}
```

MIDDLE TENNESSEE STATE UNIVERSITY

Box trace: the way to understand recursion

CSCI 3110

- Box trace
  - A systematic way to trace the actions of a recursive method
  - Each box roughly corresponds to an activation record
  - Contains a function's local environment at the time of and as a result of the call to the method

Box trace: Procedure

- Box trace
  - Label each recursive call in the function body
  - Represent each call to the function by a new box containing local environment of the function.
    - parameters
    - local variables
    - placeholder for return value
    - the value of the function itself
  - Draw an arrow from the box that makes the call to the newly created box, and label each arrow
  - Start executing the function body corresponding to new created box
  - On exiting the function, cross off the current box and follow its arrow back to the box that called the function

Box trace factorial function

The initial call is made, and method `fact` begins execution:

```
n = 3
A: fact(n-1)=?
return ?
```

At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

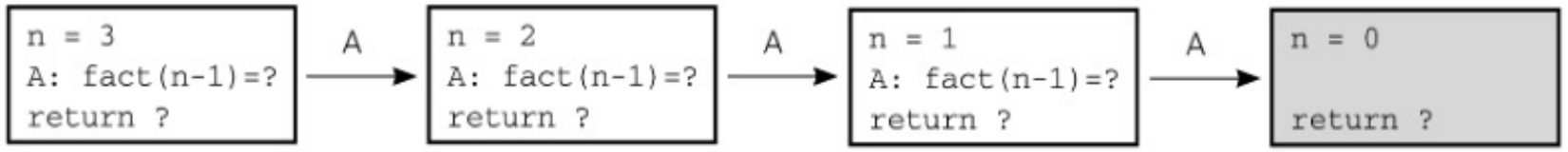
```
n = 3                A      n = 2
A: fact(n-1)=?   ------->   A: fact(n-1)=?
return ?                    return ?
```

At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

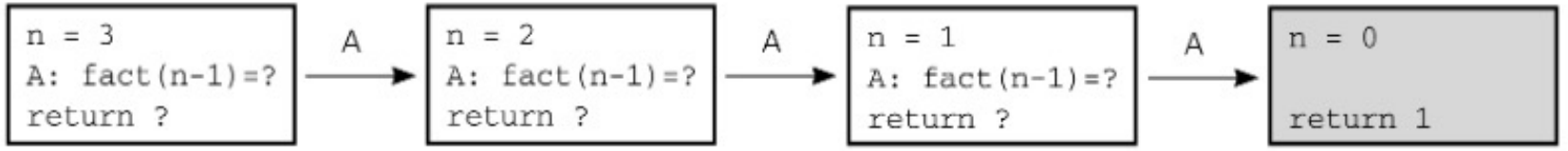
```
n = 3                A      n = 2                A      n = 1
A: fact(n-1)=?   ------->   A: fact(n-1)=?   ------->   A: fact(n-1)=?
return ?                    return ?                    return ?
```

Box trace factorial function

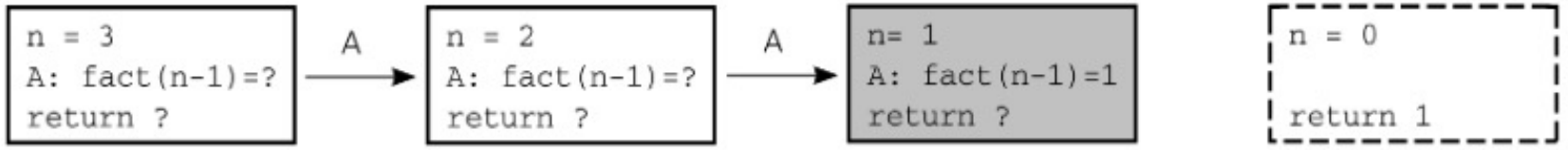
At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



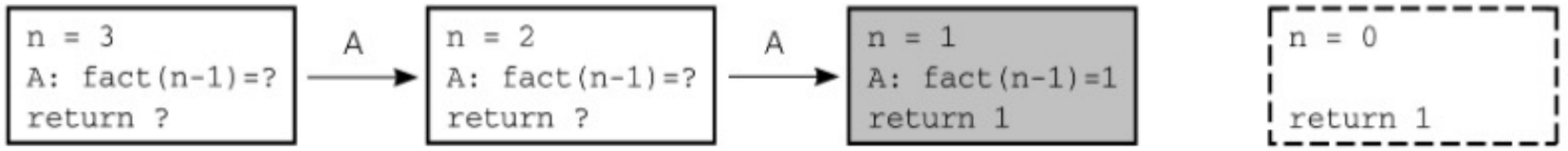
This is the base case, so this invocation of `fact` completes:



The method value is returned to the calling box, which continues execution:

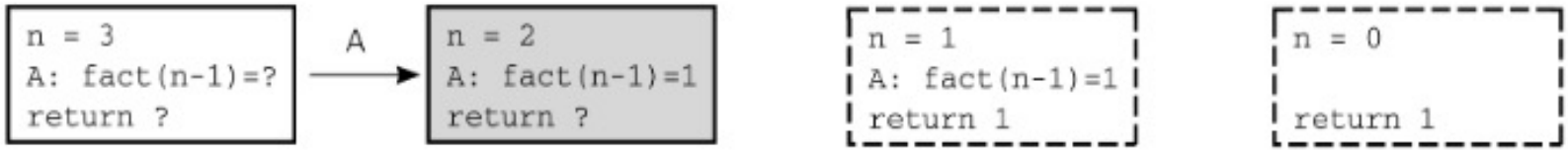


The current invocation of `fact` completes:

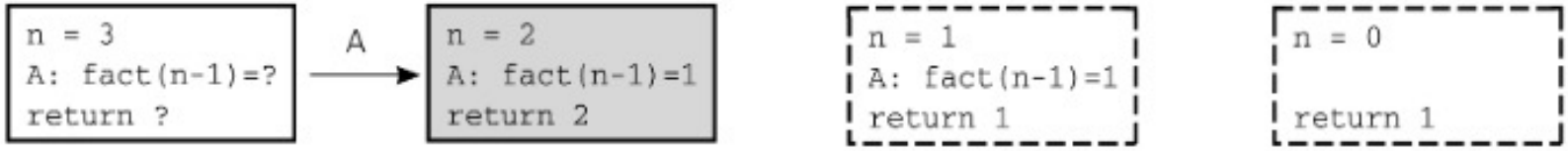


Box trace factorial function

The method value is returned to the calling box, which continues execution:



The current invocation of `fact` completes:



The method value is returned to the calling box, which continues execution:

The current invocation `fact` completes:

The value 6 is returned to the initial call.

String Backward

- Problem
  - Given a string of characters, write it in reverse order
- Recursive solution
  - Each recursive step of the solution diminishes by 1 the length of the string to be written backward
  - Base case: write the empty string backward

Recursive Solutions

- Using StrBackward(a[0...n]) to indicate the solution to write string a[0...n] backward
- Solution 1
  StrBackward(a[0...n]) = output a[n] + StrBackward(a[0...n-1])
- Solution 2
  StrBackward(a[0...n]) = StrBackward(a[1...n]) + output a[0]

String Backward


```
void writeBackward1( string s )
{
  if ( s.size() > 0 )
  {
    string  subs = s.substr(0, s.size()-1);
    cout << s[s.size()-1];
    writeBackward1( subs );
  }
}

void writeBackward2( string s )
{
  if ( s.size() > 0 )
  {
    string  subs = s.substr(1, s.size()-1);
    writeBackward2( subs );
    cout << s[0];
  }
}
```

Choosing K out of n things

- Problem
  - How many different choices are there for exploring k planets out of n planets in a solar system?
- Pay attention to the problem size and base cases

Choosing K out of n things

- Let c(n, k) be the number of groups of k planets chosen from n
- In terms of Planet X:

  c(n, k) = the number of groups of k planets
  that include Planet X
  +
  the number of groups of k planets
  that do not include Planet X

Choosing K out of n things

- The number of ways to choose k out of n things is the sum of
  - the number of ways to choose k-1 out of n-1 things
  - the number of ways to choose k out of n-1 things
- $c(n, k) = c(n - 1, k - 1) + c(n - 1, k)$

Choosing K out of n things

- Base Cases
  - There is one group of everything
    $c(k, k) = 1$
  - There is one group of nothing
    $c(n, 0) = 1$
  - $c(n, k) = 0$ if $k > n$

Choosing K out of n things

- Recursive solution

$$c(n, k) = \begin{cases} 1 & \text{if } k = 0 \\ 1 & \text{if } k = n \\ 0 & \text{if } k > n \\ c(n-1, k-1) + c(n-1, k) & \text{if } 0 < k < n \end{cases}$$

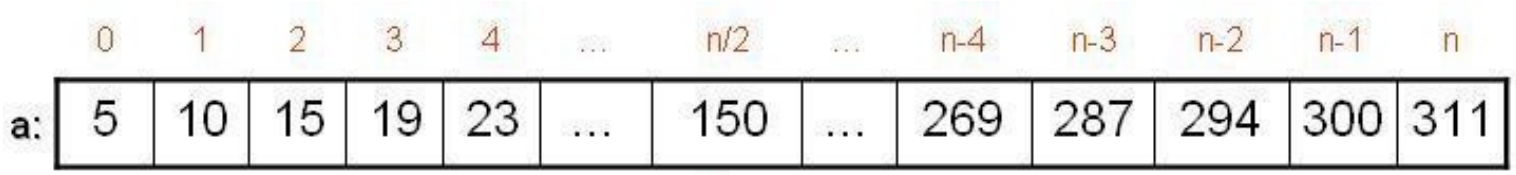
MIDDLE TENNESSEE STATE UNIVERSITY

Binary search

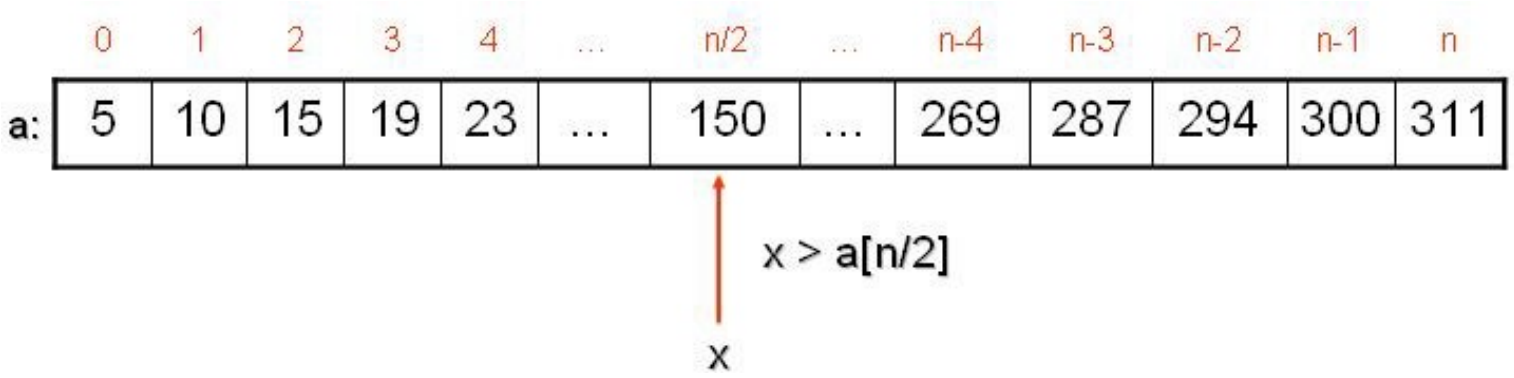
CSCI 3110

- Problem
  - Given a number x, and an array of ascent sorted integers with size n, give an efficient algorithm to find if x appears in the array.

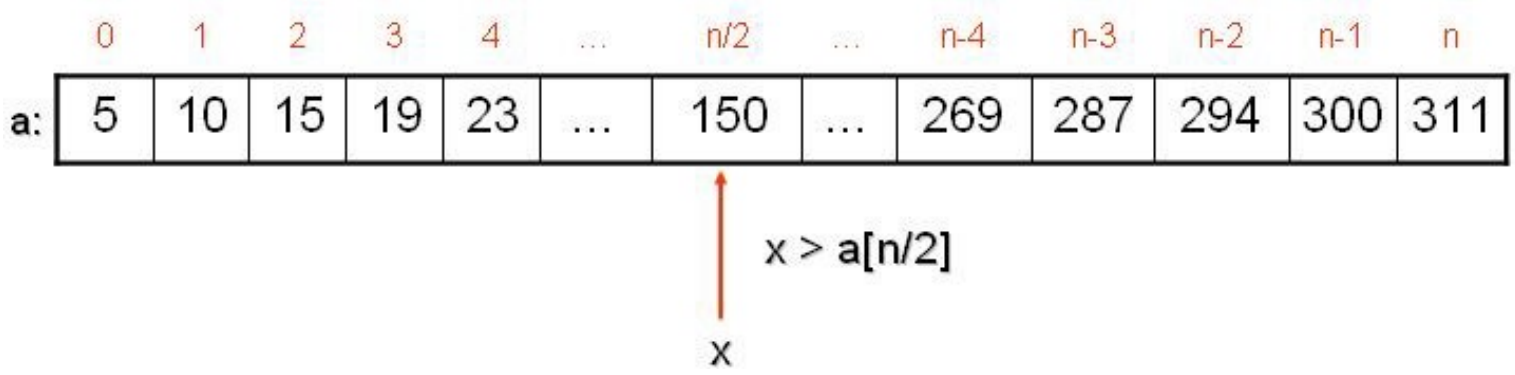
Binary search



Binary search

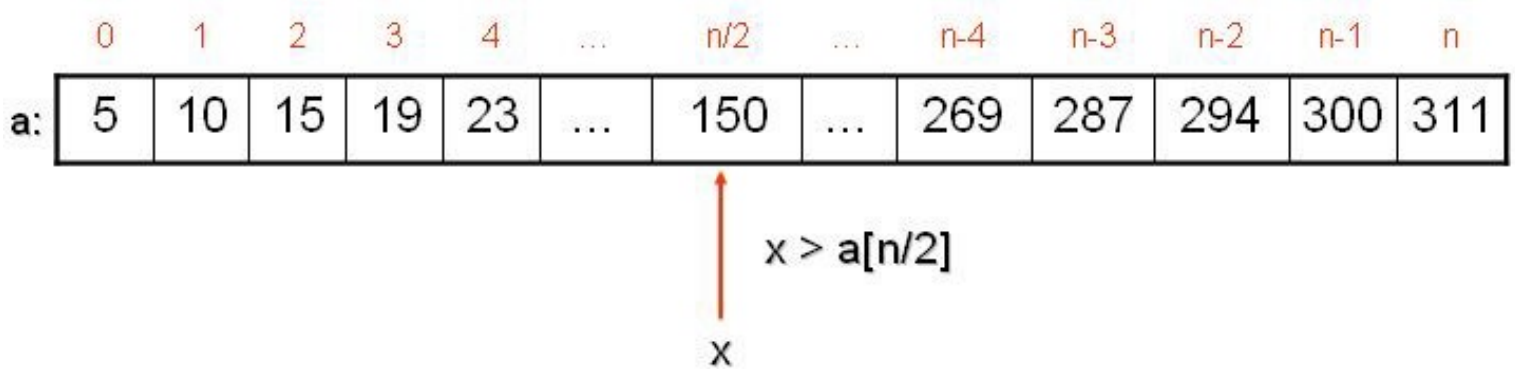


Binary search



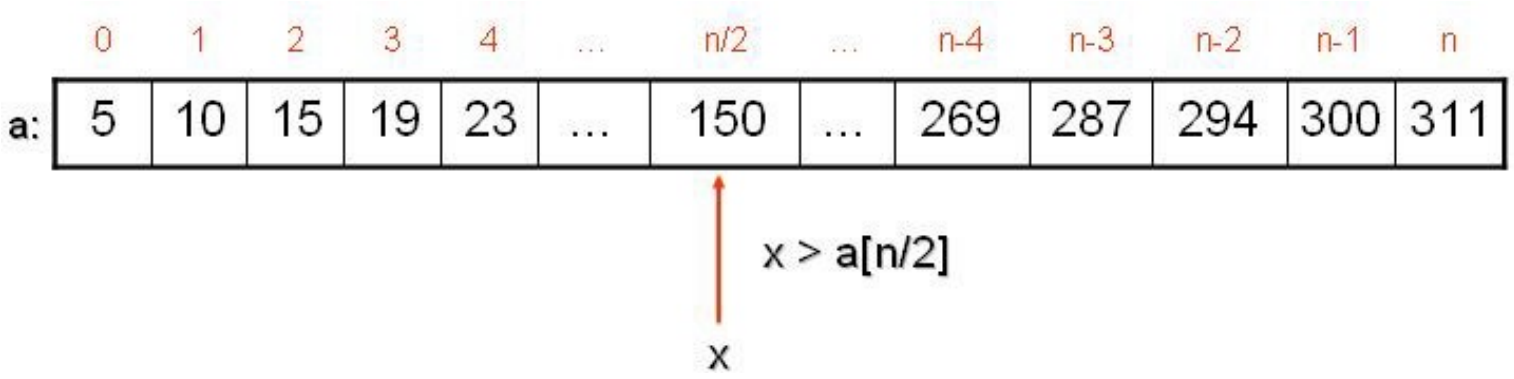
- x > a[n/2]: search x in a[n/2..n]

Binary search



- x > a[n/2]: search x in a[n/2..n]
- x == a[n/2]: return true

Binary search



- x > a[n/2]: search x in a[n/2..n]
- x == a[n/2]: return true
- x < a[n/2]: search x in a[0..n/2]

Recursive Solution

- Let **BinarySearch**(a[0...n],x) be the solution to the problem of checking if x is in the array a[0...n]

  BinarySearch(a[0...n], x) =

  BinarySearch(a[0...n/2-1], x)
  2
  BinarySearch(a[n/2+1...n], x)

- Base cases:
  - array has no elements, which indicates x is not in this array
  - x == array[arraySize/2], which indicates x is in the array with index arraySize/2

Binary Search


```
int BinarySearch(int array[], int first, int last, int key)

{
   int mid;

   if (first > last)
      return -1;
   else {
         mid = (first + last) / 2;

         if (key == array[mid])
            return mid;
         else if (value < array[mid])
            return BinarySearch(array, first, mid - 1, key);
         else if (value > array[mid])
            return BinarySearch(array, mid + 1, last, key);
      }
}
```

MIDDLE TENNESSEE STATE UNIVERSITY

Algorithm Analysis, Recursion, and Sorting

CSCI 3110

1 Algorithm Efficiency
- Measuring the Efficiency of Algorithms
- Algorithm Growth Rates
- Keeping Your Perspective

2 Recursion?
- Examples
- Iteration vs. Recursion
- One Real Example: The Towers of Hanoi

3 Summary

4 Sorting Algorithms
Selection Sort
Bubble Sort
Insertion Sort
Mergesort Quick Sort Radix Sort
Summary

STL Sorting Algorithms

Iteration vs. Recursion

- Costs
  - Recursion: calling the same function over and over
  - Iteration: 'jumps back' to the beginning of the loop.
- Iteration is more efficient
  - A function call is often more expensive than a jump.
    - Space: Every function call may require space for parameters and local variables. A recursive algorithm may need space proportional to the number of nested calls.
    - Time: Allocating and later releasing local memory, copying values into the local memory for the parameters, branching to/returning from the function.
  - "Clever" compilers may produce machine code that incurs minimal or no overhead for recursion.
  - Most recursive solution can be rewritten as iteration

Iteration vs. Recursion

- Recursion is easier to maintain
  - clearer, simpler, shorter, and easier to understand than their non-recursive counterparts.
  - reflects the abstract solution strategy (algorithm).
- Recursion provides an elegant solution to some kind of problems, sometimes, the only solution.

MIDDLE TENNESSEE STATE UNIVERSITY

Algorithm Analysis, Recursion, and Sorting

CSCI 3110

1 Algorithm Efficiency
- Measuring the Efficiency of Algorithms
- Algorithm Growth Rates
- Keeping Your Perspective

2 Recursion?
- Examples
- Iteration vs. Recursion
- One Real Example: The Towers of Hanoi

3 Summary

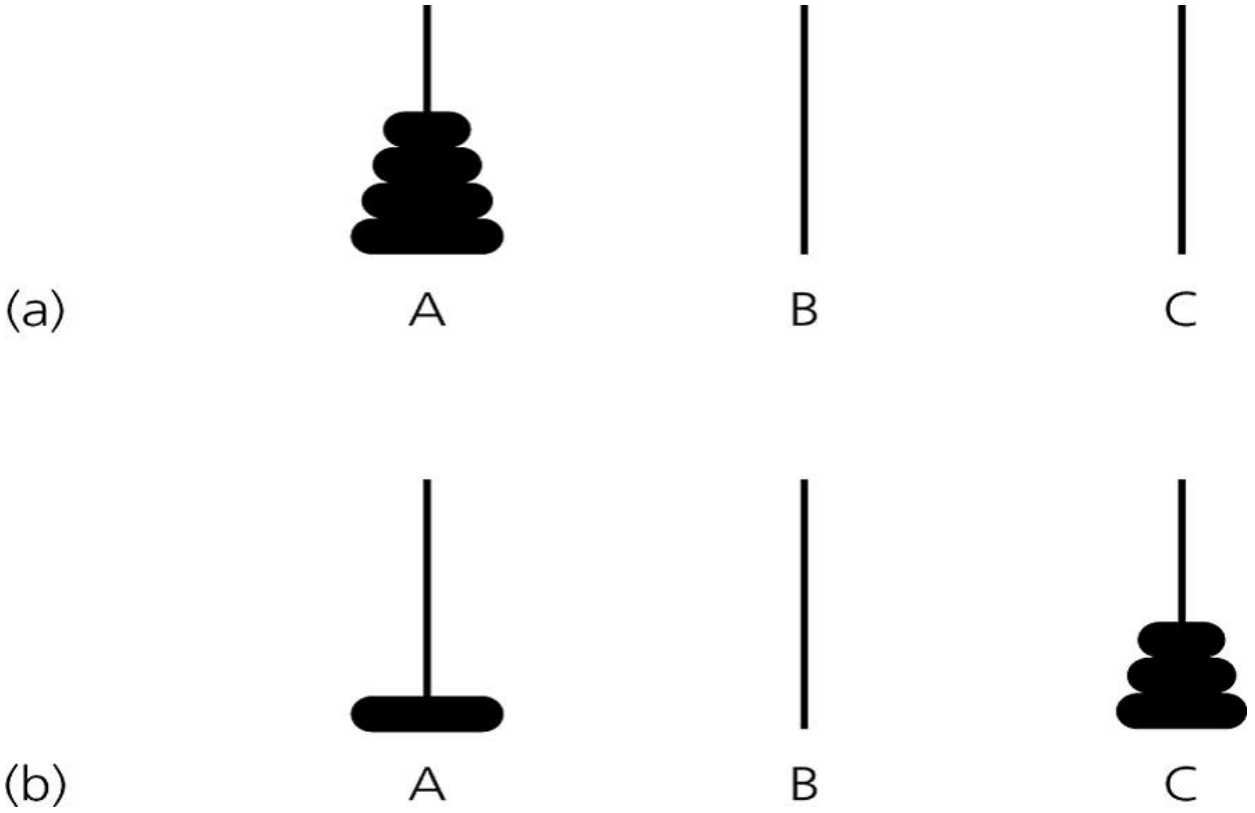
4 Sorting Algorithms
Selection Sort
Bubble Sort
Insertion Sort
Mergesort Quick Sort Radix Sort
Summary

STL Sorting Algorithms

What's the Towers of Hanoi problem?

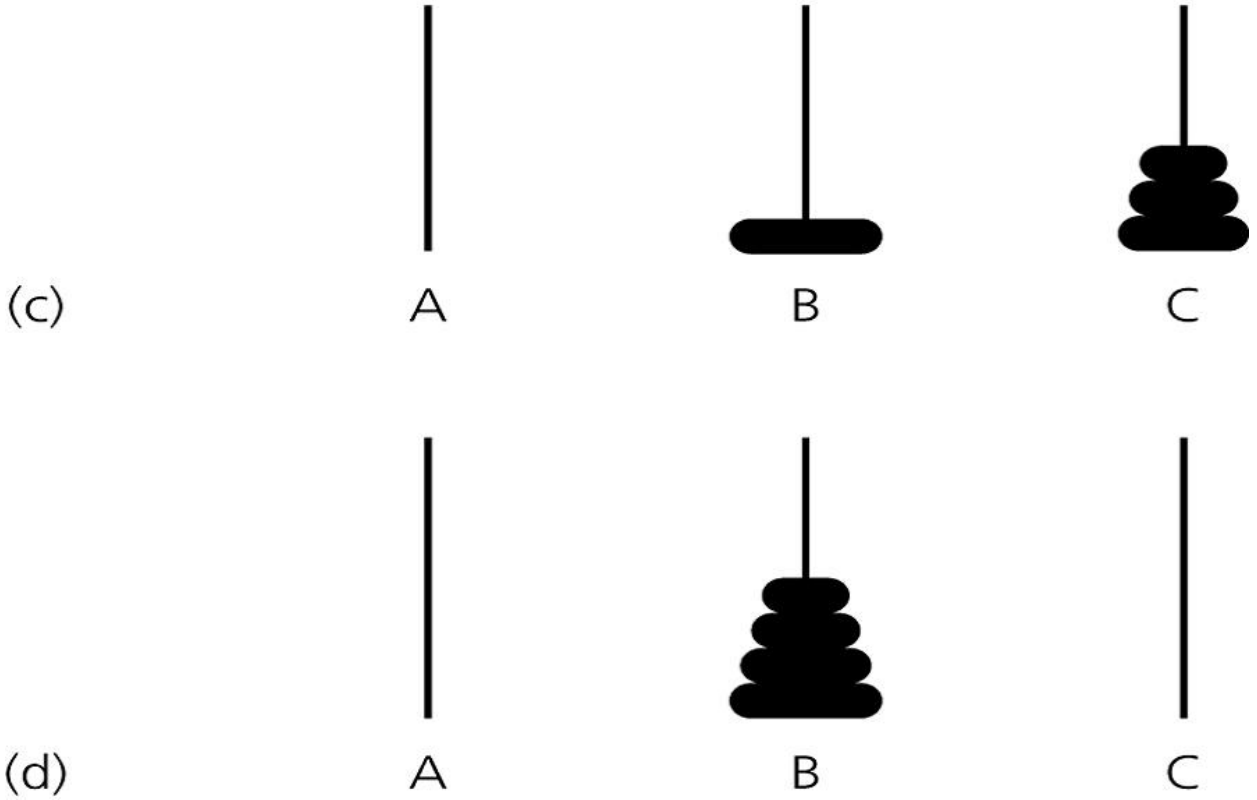
Towers of Hanoi Puzzle

The Towers of Hanoi



a) The initial state;
b) move n - 1 disks from A to C

The Towers of Hanoi



c) move one disk from A to B;

d) move n - 1 disks from C to B

The Towers of Hanoi


```
void  SolveTowers(int count, char source, char destination, char spare)
{
    if (count ==1)
          cout << "moving disk from " << source << " to "  << destination << endl;
   else
   {
          SolveTowers(count-1, source, spare, destination);
          SolveTowers(1, source, destination, spare);
          SolveTowers(count-1, spare, destination, source);
   }
}
```

The Towers of Hanoi

When N=3 , i.e., solve SolveTowers(3, A, B, C)
Trace of program execution

SolveTowers(3, A, B, C)

SolveTowers(2, A, C, B) → SolveTowers(1, A, B, C) A→B
SolveTowers(1, A, C, B) A→C
SolveTowers(1, B, C, A) B→ C
SolveTowers(1, A, B, C) → A→ B
SolveTowers(2, C, B, A) → SolveTowers(1, C, A, B) C→ A
→ SolveTowers(1, C, B, A) C→ B
→ SolveTowers(1, A, B, C) A→ B

MIDDLE TENNESSEE STATE UNIVERSITY

Algorithm Analysis, Recursion, and Sorting

CSCI 3110

Summary

- Recursion is a technique that solves a problem by solving smaller problems of the same type
- When constructing a recursive solution, keep the following 4 questions in mind
  - How to define the problem in terms of a smaller problem of the same type?
  - How does each recursive call diminish the size of the problem?
  - What instance of the problem can serve as the base case(s)?
  - As the problem size diminishes, will you reach this base case?
- You can use the box method to trace the actions of a recursive functions. Each box represents a activation record.

Summary

- Recursion allow you to solve problems – such as the Towers of Hanoi – whose iterative solution are difficult to conceptualize. Even the most complex problems often have straightforward recursive solutions. Such solutions can be easier to understand, describe, and implement than iterative solutions.
- Some recursive solutions are much less efficient than a corresponding iterative solution due to their inherently inefficient algorithms and the overhead of function calls.
- If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so.