More Recursion

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Announcements

- Quiz 1 due tonight by 11:55 PM (9/15/10)
  - It is available starting at 11:00 AM
- Program 2 due tonight by 11:55 PM
- Program 3 is now assigned
  - Uses Recursion
  - MUST use recursion or a SIZEABLE deduction will come off the grade
- Good News:
  - Assignments WILL slow down (not as often)
    - 1st three assigned and end up being done in first 6 weeks
    - Remaining three are spread out over last 9 weeks

Recursion

- What is Recursion? *(reminder from last time)*
  - From the programming perspective:
  - Recursion solves large problems by **reducing** them to **smaller** problems of the **same form**
  - Recursion is a function that invokes itself
    - Basically **splits** a problem into one or more SIMPLER versions of itself
    - And we must have a way of stopping the recursion
    - So the function must have some sort of calls or conditional statements that can actually terminate the function

Recursion - Factorial

- Example: Compute Factorial of a Number
  - What is a factorial?
    - 4! = 4 * 3 * 2 * 1 = 24
    - In general, we can say:
    - n! = n * (n-1) * (n-2) * … * 2 * 1
    - Also, 0! = 1
      - (just accept it!)

Recursion - Factorial

- Example: Compute Factorial of a Number
  - Recursive Solution
    - Mathematically, factorial is already defined recursively
      - **Note that each factorial is related to a factorial of the next smaller integer**
    - 4! = 4*3*2*1 = 4 * (4-1)! = 4 * (3!)
    - Right?
    - Another example:
    - 10! = 10*9*8*7*6*5*4*3*2*1
    - 10! = 10*(9!)

This is clear right?
Since 9! clearly is equal to
9*8*7*6*5*4*3*2*1

Recursion - Factorial

- Example: Compute Factorial of a Number
  - Recursive Solution
    - Mathematically, factorial is already defined recursively
      - **Note that each factorial is related to a factorial of the next smaller integer**
    - Now we can say, in general, that:
    - n! = n * (n-1)!
    - But we need something else
      - We need a stopping case, or this will just go on and on and on
      - NOT good!
    - We let 0! = 1
    - So in “math terms”, we say
      - n! = 1 if n = 0
      - n! = n * (n-1)! if n > 0

Recursion - Factorial

- How do we do this recursively?
  - We need a function that we will call
    - And this function will then call itself (recursively)
      - until the stopping case (n = 0)

```
#include <stdio.h>

void   Fact(int n);
int main(void) {
     int factorial = Fact(10);
     printf("%d\n", factorial);
     return 0;
}
```

<u>Here's the Fact Function</u>

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

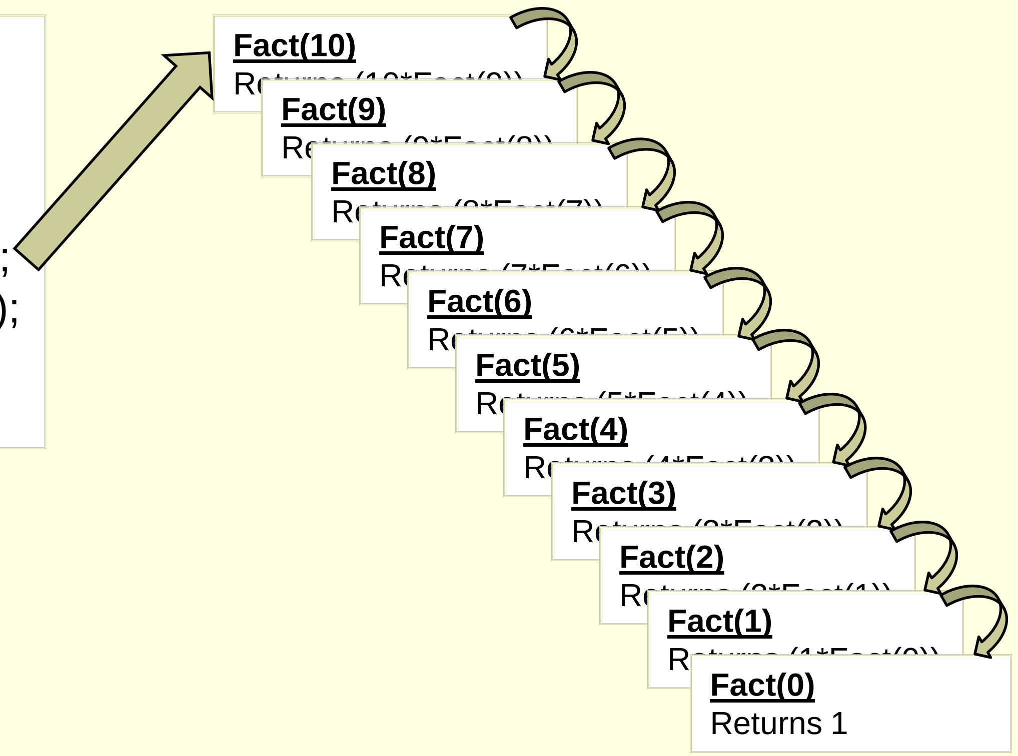
- This program prints the result of 10*9*8*7*6*5*4*3*2*1:
  - 3628800

Recursion - Factorial

- Here's what's going on…in pictures

```
#include <stdio.h>

void   Fact(int n);
int main(void) {
     int factorial = Fact(10);
     printf("%d\n", factorial);
     return 0;
}
```

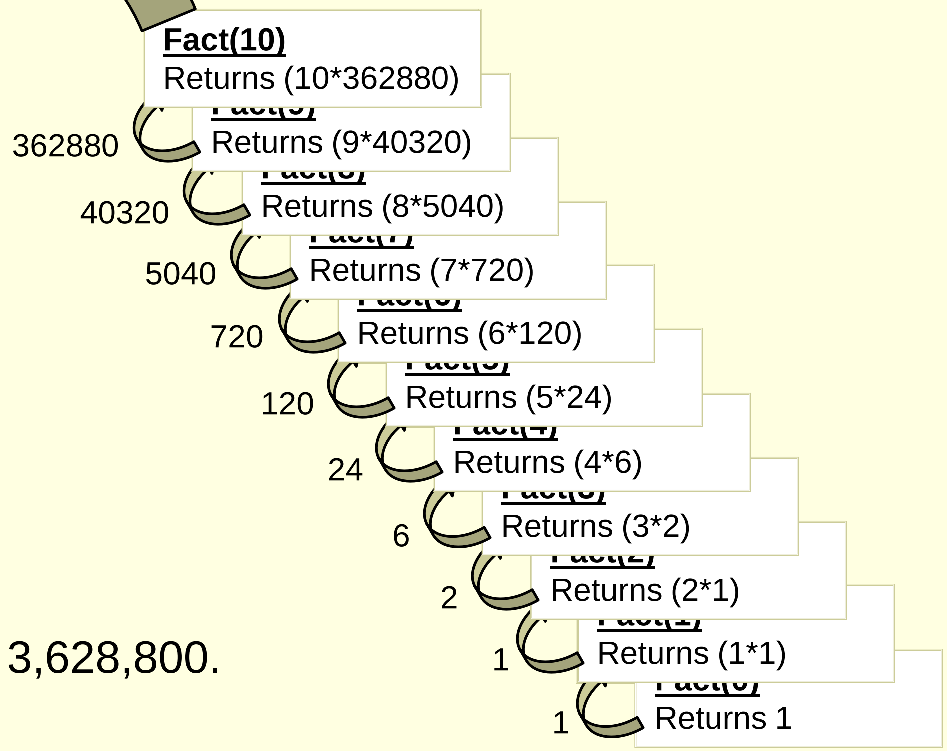


Recursion - Factorial

- Here's what's going on…in pictures

```
#include <stdio.h>

void   Fact(int n);
int main(void) {
     int factorial = Fact(10);
     printf("%d\n", factorial);
     return 0;
}
```



- Now factorial has the value 3,628,800.

Recursion: General Structure

- General Structure of Recursive Functions:
  - What we can determine from previous examples:
    - When we have a problem, we want to break it into chunks
    - Where one of the chunks is a smaller version of the same problem
  - Factorial Example:
    - We utilized the fact that n! = n*(n-1)!
    - And we realized that (n-1)! is, in essence, an easier version of the original problem
    - Right?
    - We all should agree that 9! is a bit easier than 10!

Recursion: General Structure

- General Structure of Recursive Functions:
  - What we can determine from previous examples:
    - Eventually, we break down our original problem to such an extent that the small sub-problem becomes quite easy to solve
    - At this point, we don't make more recursive calls
    - Rather, we directly return the answer
    - Or complete whatever task we are doing
  - This allows us to think about a general structure of a recursive function

Recursion: General Structure

- General Structure of Recursive Functions:
  - Basic structure has 2 main options:
  1) Break down the problem further
     - Into a smaller sub-problem
  2) OR, the problem is small enough on its own
     - Solve it
  - In programming, when we have two options, we us an if statement
  - So here are our two constructs of recursive functions

Recursion: General Structure

- General Structure of Recursive Functions:
  - 2 general constructs:
  - **Construct 1:**

```
if (terminating condition) {
      DO FINAL ACTION
}
else {
      Take one step closer to terminating condition
      Call function RECURSIVELY on smaller subproblem
}
```

  - Functions that return values take on this construct

Recursion: General Structure

- General Structure of Recursive Functions:
  - 2 general constructs:
  - **Construct 2:**

```
if (!(terminating condition) ) {
      Take a step closer to terminating condition
      Call function RECURSIVELY on smaller subproblem
}
```

  - void recursive functions use this construct

Recursion: General Structure

- **Example using Construct 1**
  - Our function (Sum Integers):
    - Takes in one positive integer parameter, n
    - Returns the sum 1+2+…+n
    - So our recursive function must sum all the integers up until (and including) n
  - How do we do this recursively?
    - We need to solve this in such a way that part of the solution is a sub-problem of the EXACT same nature of the original problem.

Recursion: General Structure

- Example using Construct 1
  - Our function:
    - Using n as the input, we define the following function
      - $f(n) = 1 + 2 + 3 + \ldots + n$
        - Hopefully it is clear that this is our desired function
    - So to make this recursive, can we say:
      - $f(n) = 1 + (2 + 3 + \ldots + n)$

?

    - Does that "look" recursive?
    - Is there a sub-problem that is the EXACT same form as the original problem?
      - NO!
    - $2+3+\ldots+n$ IS NOT a sub-problem of the form $1+2+\ldots+n$

Recursion: General Structure

- Example using Construct 1
  - Our function:
    - Using n as the input, we get the following function
      - $f(n) = 1 + 2 + 3 + \ldots + n$
    - Let's now try this:
      - $f(n) = 1 + 2 + \ldots + n = n + (1 + 2 + \ldots + (n-1))$
    - AAAHHH.
    - Here we have an expression
      - $1 + 2 + \ldots + (n-1)$
    - which IS indeed a sub-problem of the same form

Recursion: General Structure

- Example using Construct 1
  - Our function:
    - Using n as the input, we get the following function
      - f(n) = 1 + 2 + 3 + … + n
    - So now we have:
      - f(n) = 1 + 2 + … + n = n + (1 + 2 + … + (n-1))
    - Now, realize the following:
      - f(n) = n + f(n-1)
      - Right?
      - We've defined f(n) to be a function that sums the first n integers

Recursion: General Structure

- Example using Construct 1
  - Our function:
    - Using n as the input, we get the following function
      - $f(n) = 1 + 2 + 3 + \ldots + n$
    - So now we have:
      - $f(n) = 1 + 2 + \ldots + n = n + (1 + 2 + \ldots + (n-1))$
    - Now, realize the following:
      - Example:
        - $f(10) = 1 + 2 + \ldots + 10 = 10 + (1 + 2 + \ldots + 9)$
        - And what is $(1 + 2 + \ldots + 9)$? It is $f(9)$!
        - Thus, we say $f(10) = 10 + f(9)$
        - In general, **$f(n) = n + f(n-1)$**

Recursion: General Structure

- Example using Construct 1
  - Our function:
    - Using n as the input, we get the following function
      - $f(n) = 1 + 2 + 3 + \ldots + n$
    - So now we have:
      - $f(n) = 1 + 2 + \ldots + n = n + (1 + 2 + \ldots + (n-1))$
    - Now, realize the following:
      - So here is our function, defined recursively
      - $f(n) = n + f(n-1)$

Recursion: General Structure

- Example using Construct 1
  - Our function (now recursive):
    - f(n) = n + f(n-1)
    - Reminder of construct 1:

```
if (terminating condition) {
      DO FINAL ACTION
}
else {
      Take one step closer to terminating condition
      Call function RECURSIVELY on smaller subproblem
}
```

Recursion: General Structure

- Example using Construct 1
  - Our function:
    - f(n) = n + f(n-1)
    - Reminder of construct 1:
    - So we need to determine the terminating condition!
    - We know that f(0) = 0
      - So our terminating condition can be n = 0
    - Additionally, our recursive calls need to return an expression for f(n) in terms of f(k)
      - for some k < n
    - We just found that f(n) = n + f(n-1)
    - So now we can write our actual function…

Recursion: General Structure

- Example using Construct 1
  - Our function:

```
// Pre-condition: n is a positive integer.
// Post-condition: Function returns the sum
// 1 + 2 + ... + n
int sumNumbers(int n) {

      if ( n == 0 )
            return 0;
      else
            return (n + sumNumbers(n-1));
}
```

Recursion: General Structure

- Another example using Construct 1
  - Our function:
    - Calculates b^e
      - Some base raised to a power, e
      - The input is the base, b, and the exponent, e
      - So if the input was 2 for the base and 4 for the exponent
        - The answer would be $2^4 = 16$
  - How do we do this recursively?
    - We need to solve this in such a way that part of the solution is a sub-problem of the EXACT same nature of the original problem.

Recursion: General Structure

- Another example using Construct 1
  - Our function:
    - Using b and e as input, here is our function
      - $f(b,e) = b^e$
    - So to make this recursive, can we say:
      - $f(b,e) = b^e = b*b^{(e-1)}$
    - Does that “look” recursive?
    - YES it does!
    - Why?
    - Cuz the right side is indeed a sub-problem of the original
    - We want to evaluate b^e
    - And our right side evaluates $b^{(e-1)}$

Recursion: General Structure

- Another example using Construct 1
  - Our function:
    - $f(b,e) = b*b^{(e-1)}$
    - Reminder of construct 1:

```
if (terminating condition) {
      DO FINAL ACTION
}
else {
      Take one step closer to terminating condition
      Call function RECURSIVELY on smaller subproblem
}
```

Recursion: General Structure

- Another example using Construct 1
  - Our function:
    - $f(b,e) = b*b^{(e-1)}$
    - Reminder of construct 1:
    - So we need to determine the terminating condition!
    - We know that $f(b,0) = b^0 = 1$
      - So our terminating condition can be when e = 1
    - Additionally, our recursive calls need to return an expression for f(b,e) in terms of f(b,k)
      - for some k < e
    - We just found that $f(b,e) = b*b^{(e-1)}$
    - So now we can write our actual function…

Recursion: General Structure

- Another example using Construct 1
  - Our function:

```
// Pre-conditions: e is greater than or equal to 0.
// Post-conditions: returns b^e.
int Power(int base, int exponent) {

      if ( exponent == 0 )
            return 1;
      else
            return (base*Power(base, exponent-1));
}
```

Recursion: General Structure

- **<u>Example using Construct 2</u>**
  - Remember the construct:
    - This is used when the return type is void

```
if (!(terminating condition) ) {
      Take a step closer to terminating condition
      Call function RECURSIVELY on smaller subproblem
}
```

Recursion: General Structure

- Example using Construct 2
  - Our function:
    - Takes in a string (character array)
    - Also takes in an integer, the length of the string
    - The function simply prints the string in REVERSE order
  - So what is the terminating condition?
    - We will print the string, in reverse order, character by character
    - So we terminate when there are no more characters left to print
    - The 2^{nd} argument to the function (length) will be reduced until it is 0 (showing no more characters left to print)

Recursion: General Structure

- Example using Construct 2
  - Our function:

```
void printReverse(char word[], int length) {
       if (length > 0) {
              printf("%c", word[length-1]);
              printReverse(word, length-1);
       }
}
```

  - What's going on:
    - Let's say the word is "computer"
      - 8 characters long
    - So we print word[7]
      - Which would refer to the "r" in computer

Recursion: General Structure

- Example using Construct 2
  - Our function:

```
void printReverse(char word[], int length) {
	if (length > 0) {
		printf("%c", word[length-1]);
		printReverse(word, length-1);
	}
}
```

  - What's going on:
    - We then recursively call the function
    - Sending over two arguments:
      - The string, "computer"
      - And the length, minus 1

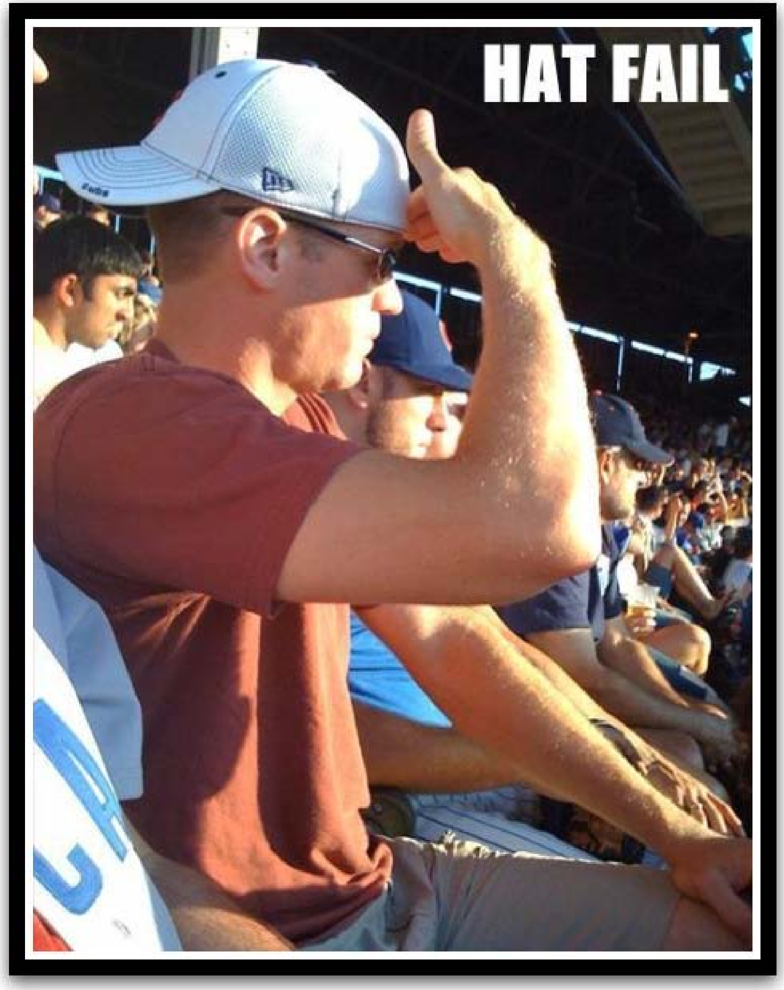
Recursion: General Structure

- Example using Construct 2
  - Our function:

```
void printReverse(char word[], int length) {
      if (length > 0) {
            printf("%c", word[length-1]);
            printReverse(word, length-1);
      }
}
```

  - What's going on:
    - After the first recursive call, length is now 7
    - Therefore, word[6] is printed
      - Referring to the "e" in computer
    - Then we recurse (again and again) and finish once length <= 0

Brief Interlude: Human Stupidity



Recursion – Practice Problem

- Practice Problem:
  - Write a recursive function that:
    - Takes in two non-negative integer parameters
    - Returns the product of these parameters
      - But it does NOT use multiplication to get the answer
    - So if the parameters are 6 and 4
    - The answer would be 24
  - How do we do this not actually using multiplication
  - What another way of saying 6*4?
  - We are adding 6, 4 times!
  - 6*4 = 6 + 6 + 6 + 6
  - So now think of your function…

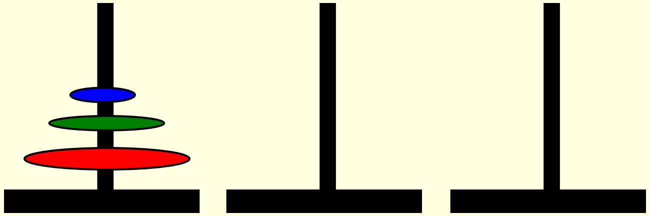
Recursion – Practice Problem

- Practice Problem:
  - Solution:

```
// Precondition: Both parameters are
// non-negative integers.
// Postcondition: The product of the two
// parameters is returned.
function Multiply(int first, int second) {
       if (( second == 0 ) || ( first = 0 ))
              return 0;
       else
              return (first + Multiply(first, second-1));
}
```

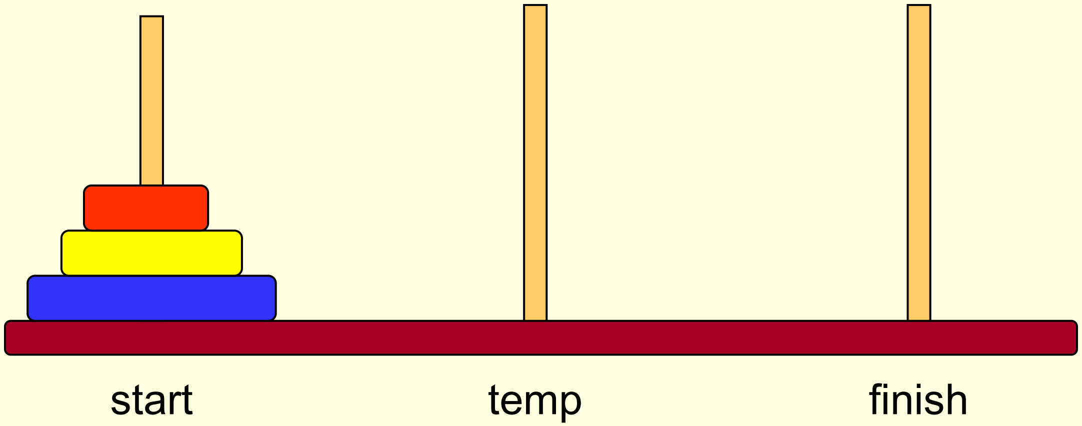
Recursion – Towers of Hanoi

- Towers of Hanoi:
  - Here's the problem:
    - There are three vertical poles
    - There are 64 disks on tower 1 (left most tower)
      - The disks are arranged with the largest diameter disks at the bottom
    - Some monk has the daunting task of moving disks from one tower to another tower
      - Often defined as moving from Tower #1 to Tower #3
        - Tower #2 is just an intermediate pole
      - He can only move ONE disk at a time
      - And he MUST follow the rule of NEVER putting a bigger disk on top of a smaller disk



Recursion – Towers of Hanoi

- Towers of Hanoi:
  - Solution:
    - We must find a recursive strategy
    - Thoughts:
      - Any tower with more than one disk must clearly be moved in pieces
      - If there is just one disk on a pole, then we move it

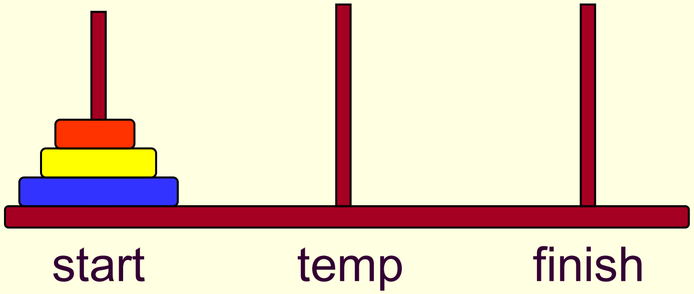


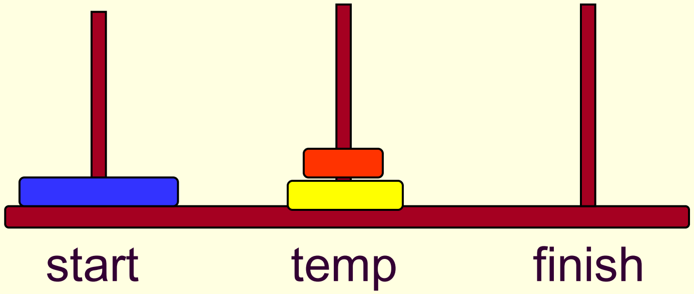
Recursion – Towers of Hanoi

- Towers of Hanoi:
  - Solution:
    - Irrespective of the number of disks, the following steps MUST be carried out:
      - The bottom disk needs to move to the destination tower
      1) So step 1 must be to move all disks above the bottom disk to the intermediate tower
      2) In step 2, the bottom disk can now be moved to the destination tower
      3) In step 3, the disks that were initially above the bottom disk must now be put back on top
         - Of course, at the destination
    - Let's look at the situation with only 3 disks

Recursion – Towers of Hanoi

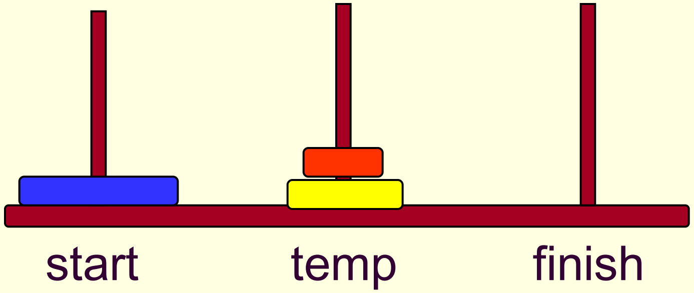
- Towers of Hanoi:
  - Solution:
    - Step 1:
      - Move 2 disks from start to temp using finish Tower.
      - To understand the recursive routine, let us assume that we know how to solve 2 disk problem, and go for the next step.
        - Meaning, we “know” how to move 2 disks appropriately

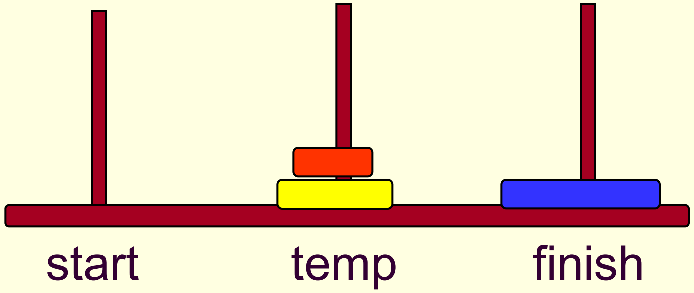




Recursion – Towers of Hanoi

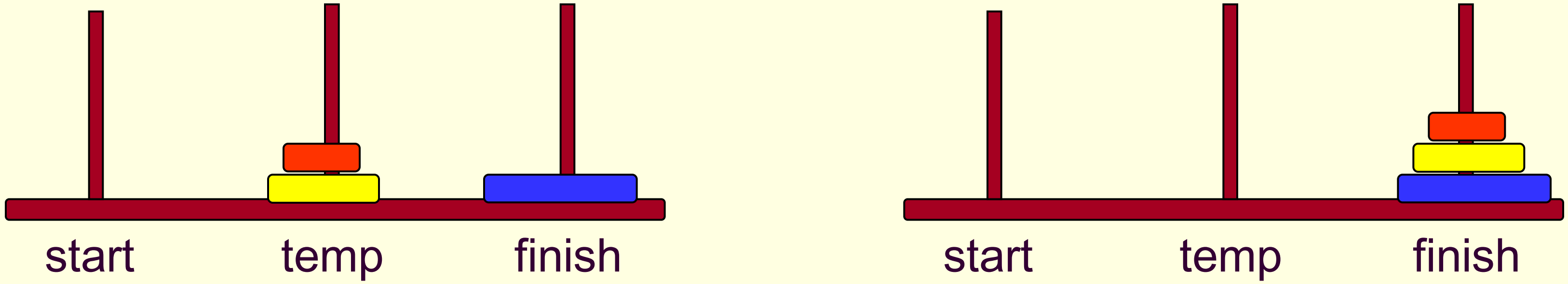
- Towers of Hanoi:
  - Solution:
    - Step 2:
      - Move the (remaining) single disk from start to finish
      - This does not involve recursion
        - and can be carried out without using temp tower.
      - In our program, this is just a print statement
        - Showing what we moved and to where



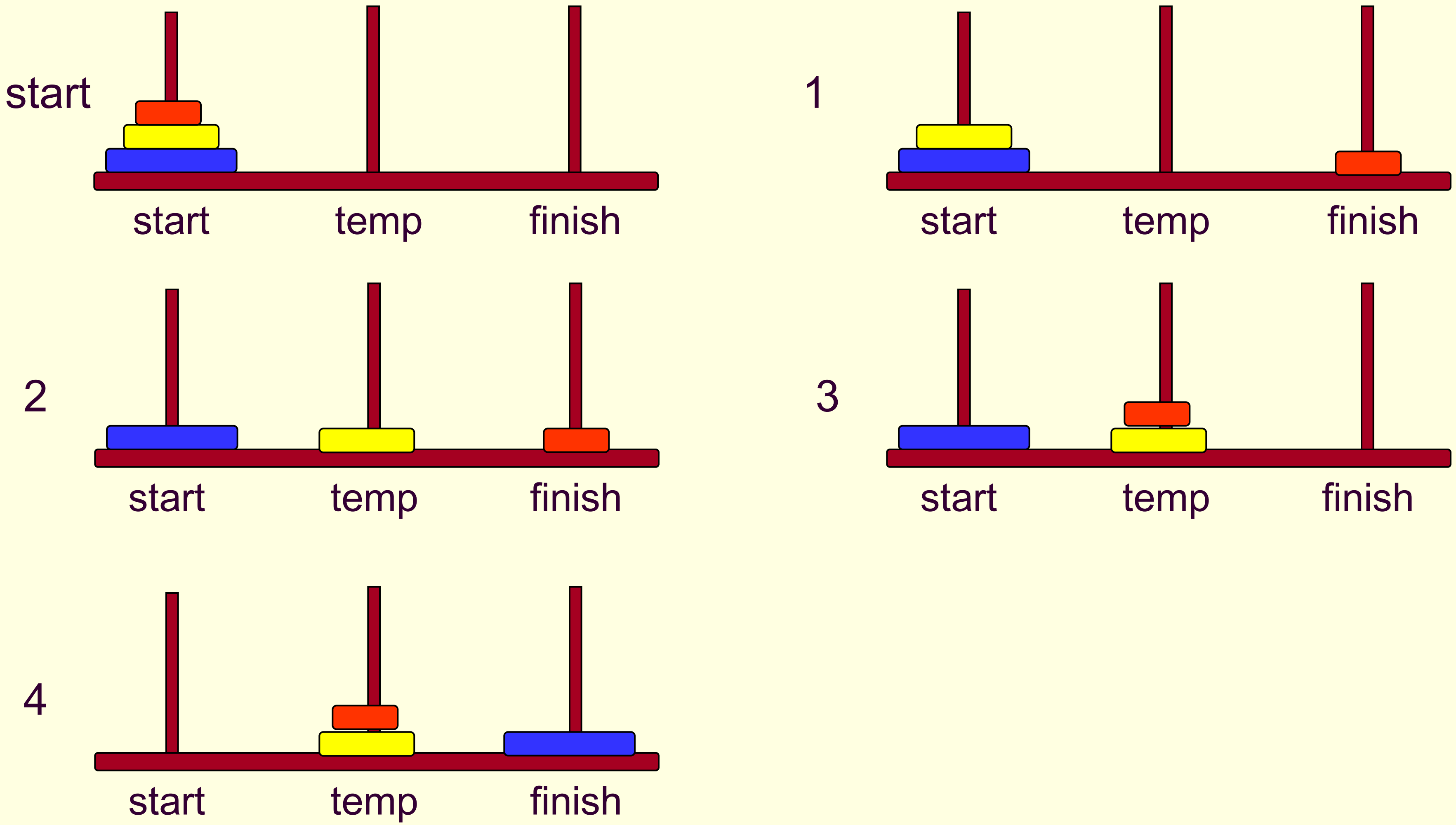


Recursion – Towers of Hanoi

- Towers of Hanoi:
  - Solution:
    - Step 3:
      - Now we are at the last step of the routine.
      - Move the 2 disks from temp tower to finish tower using the start tower
        - This is done recursively

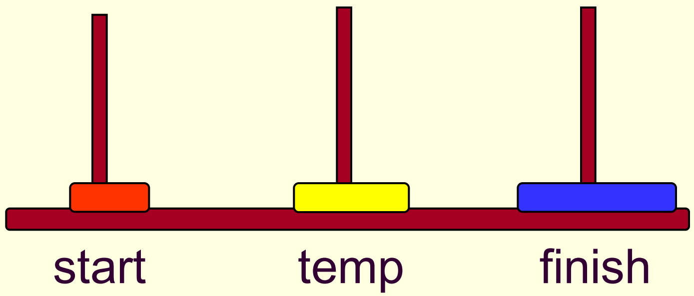


Recursion – Towers of Hanoi

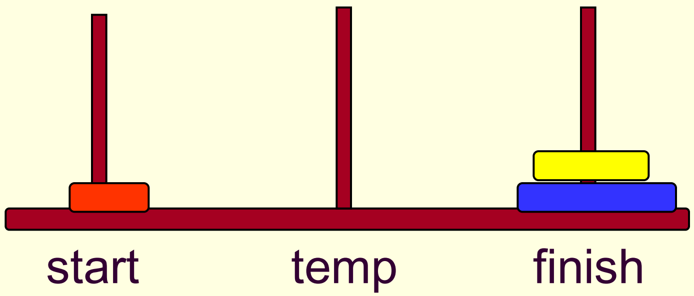


Recursion – Towers of Hanoi

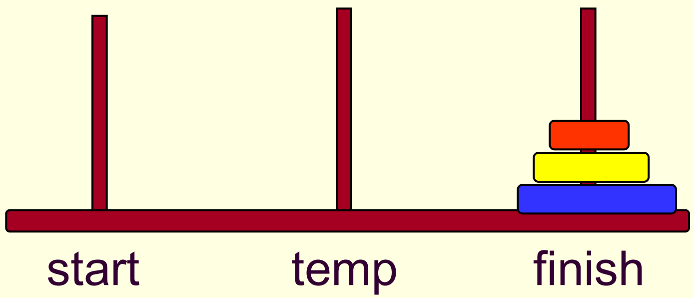
5



6



7



- # of steps needed:
  - We had 3 disks requiring seven steps
  - 4 disks would require 15 steps
  - n disks would require 2^n -1 steps
    - HUGE number

Recursion – Towers of Hanoi

- Towers of Hanoi:
  - Solution:

```
// Function Prototype
void moveDisks(int n, char start, char finish, char temp);

void main() {
       int disk;
       int moves;
       printf("Enter the # of disks you want to play with:");
       scanf("%d",&disk);
       // Print out the # of moves required
       moves = pow(2,disk)-1;
       printf("\nThe No of moves required is=%d \n",moves);
       // Initiate the recursion
       moveDisks(disk,'A','C','B');
}
```

Recursion – Towers of Hanoi

- Towers of Hanoi:
  - Solution:

```
void moveDisks(int n, char start, char finish, char temp) {
       if (n == 1) {
              printf("Move Disk from %c to %c\n", start, finish);
       }
       else  {
              moveDisks(n-1, start, temp, finish);
              printf("Move Disk from %c to %c\n", start, finish);
              moveDisks(n-1, temp,  finish, start);
       }
}
```

Recursion

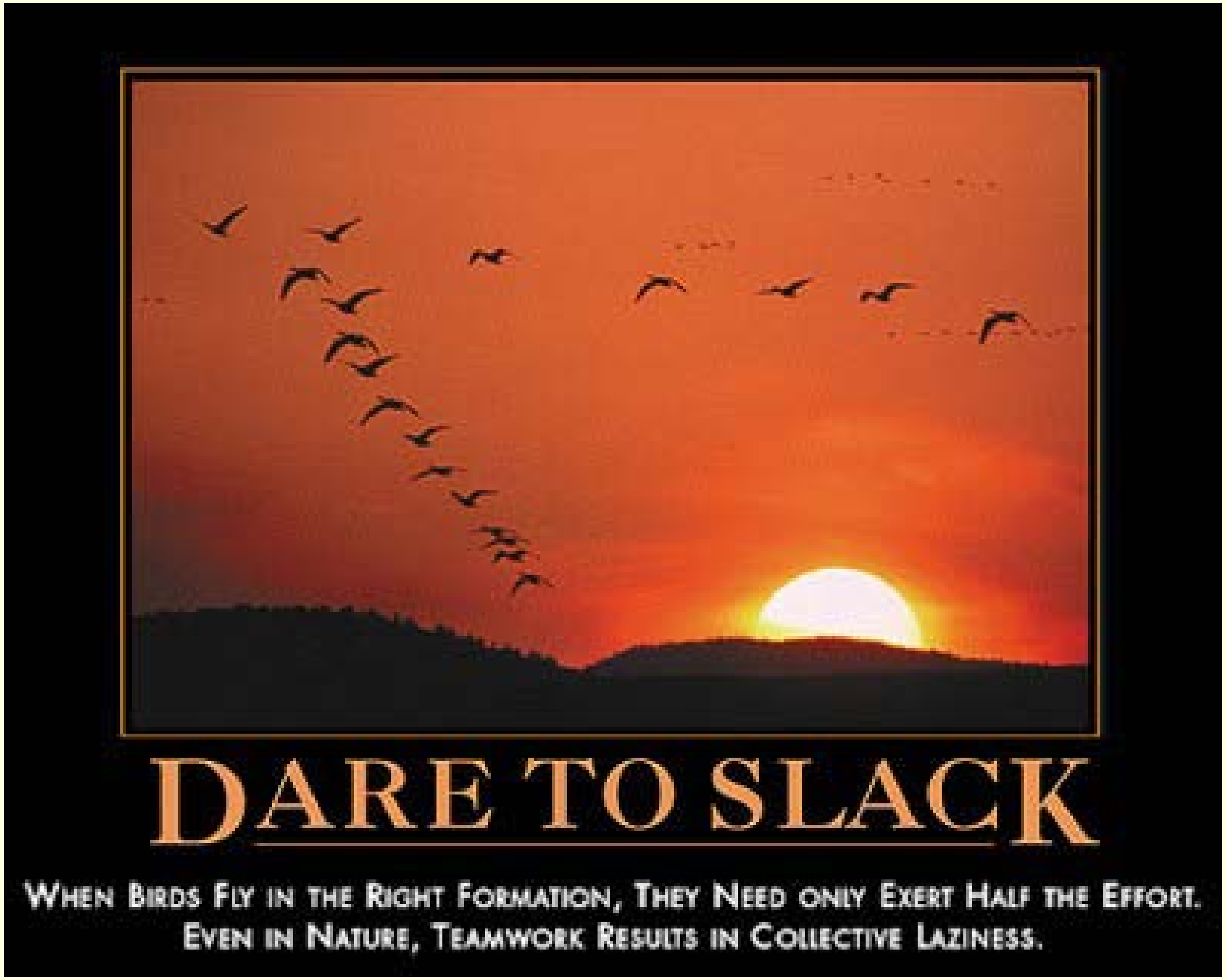
WASN'T

THAT

ENCHANTING!

(Sorry, wanted a "word of the day", and this is what I got from the wife!)

Daily Demotivator



More Recursion

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I