Programming for Engineers

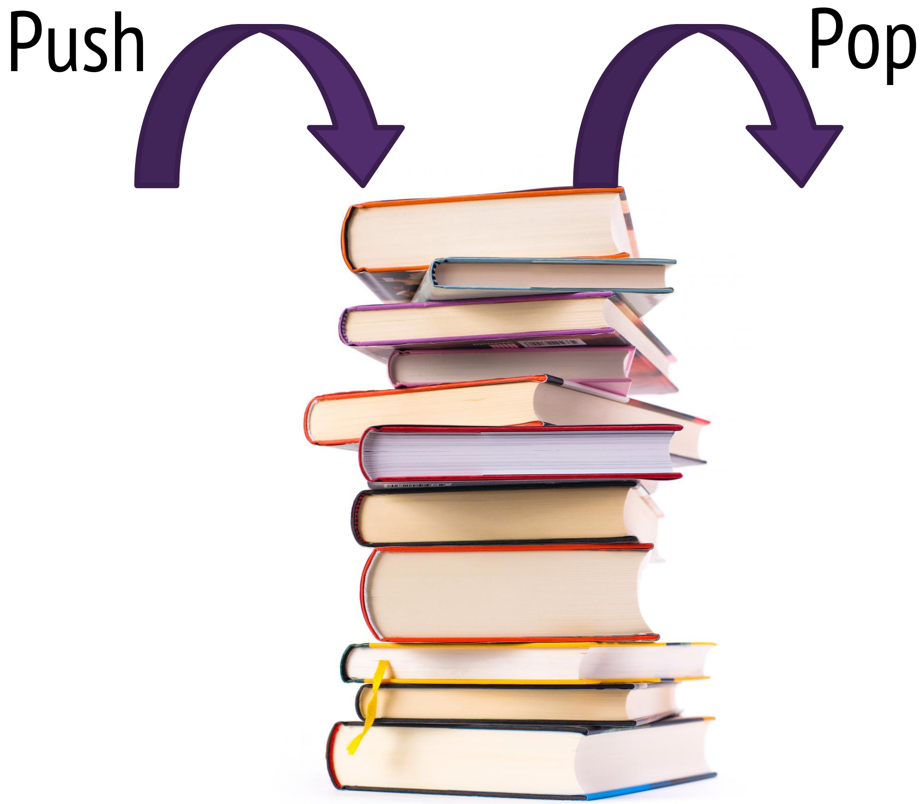
Recursions

ICEN 200– Spring 2018

Prof. Dola Saha

Function call stack and stack frames

- Stack is analogous to a pile of books
- Known as last-in, first-out (LIFO) data structures



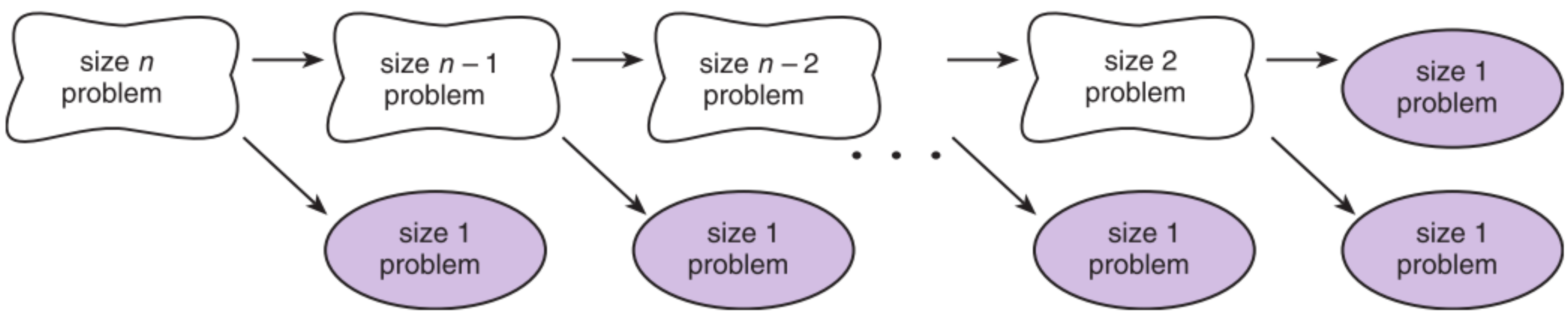
Stack of books

Function call stack

- Supports function call & return
- Supports creation, maintenance & destruction of each called function's local variables
- Keeps track of return addresses that each function needs to return control to the caller function
- Function call → an entry is pushed to stack
- Function return → an entry is popped from stack

Recursion

- A recursive function is a function that calls itself either directly or indirectly through another function.
- Nature of recursion
  - One or more simple cases of the problem have a straightforward, nonrecursive solution.
  - The other cases can be redefined in terms of problems that are closer to the simple cases.



Recursively calculating Factorial

- The factorial of a nonnegative integer *n*, written *n!* (pronounced "*n* factorial"), is the product
  - $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

  with 1! equal to 1, and 0! defined to be 1.
- A *recursive* definition of the factorial function is arrived at by observing the following relationship:

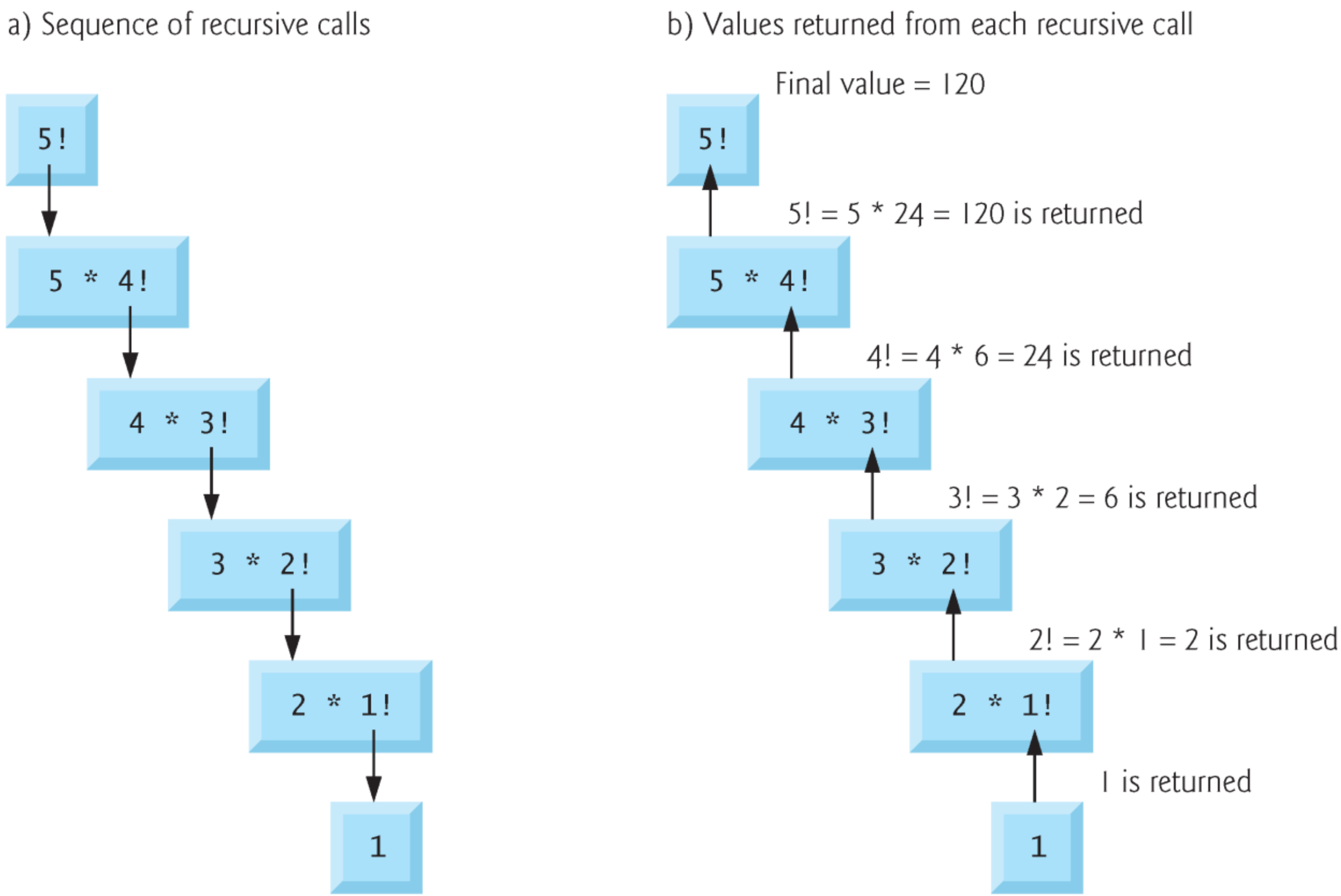
  $n! = n \cdot (n-1)!$
- Proof:

  $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$

  $n! = n \cdot ((n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1)$

  $n! = n \cdot ((n-1)!)$

Recursive evaluation of 5!



Recursive Factorial C Code (1)

```c
// Fig. 5.18: fig05_18.c
// Recursive factorial function.
#include <stdio.h>

unsigned long long int factorial(unsigned int number);

int main(void)
{
   // during each iteration, calculate
   // factorial(i) and display result
   for (unsigned int i = 0; i <= 21; ++i) {
      printf("%u! = %llu\n", i, factorial(i));
   }
}

```

Recursive Factorial C Code (2)

```c
// recursive definition of function factorial
unsigned long long int factorial(unsigned int number)
{
   // base case
   if (number <= 1) {
      return 1;
   }
   else { // recursive step
      return (number * factorial(number - 1));
   }
}
```

Recursive Factorial C Code (3) – Output

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768
```

Example Fibonacci Series by Recursion

- The Fibonacci series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …

- The Fibonacci series may be defined recursively as follows:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)
```

Recursive Fibonacci Series C Code (1)

```c
// Fig. 5.19: fig05_19.c
// Recursive fibonacci function
#include <stdio.h>

unsigned long long int fibonacci(unsigned int n); // function prototype

int main(void)
{
   unsigned int number; // number input by user

   // obtain integer from user
   printf("%s", "Enter an integer: ");
   scanf("%u", &number);

   // calculate fibonacci value for number input by user
   unsigned long long int result = fibonacci(number);

   // display result
   printf("Fibonacci(%u) = %llu\n", number, result);
}

```

Recursive Fibonacci Series C Code (2)

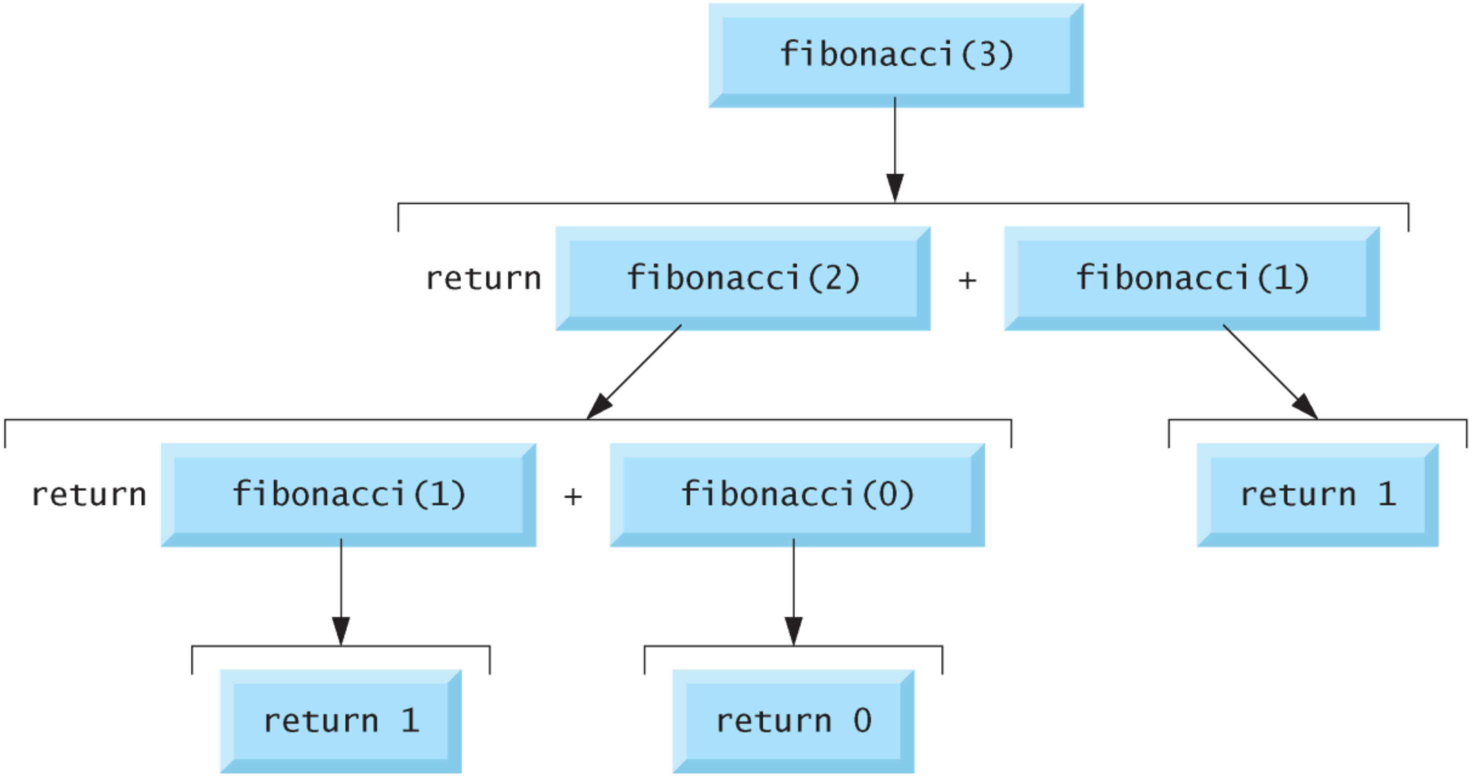
```c
// Recursive definition of function fibonacci
unsigned long long int fibonacci(unsigned int n)
{
   // base case
   if (0 == n || 1 == n) {
      return n;
   }
   else { // recursive step
      return fibonacci(n - 1) + fibonacci(n - 2);
   }
}
```

```
Enter an integer: 0
Fibonacci(0) = 0
```

```
Enter an integer: 1
Fibonacci(1) = 1
```

```
Enter an integer: 2
Fibonacci(2) = 1
```

Recursive calls



Recursion vs Iteration

- Both iteration and recursion are based on a control statement: Iteration uses a repetition statement; recursion uses a *selection statement*.

- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition statement; recursion achieves repetition through *repeated function calls*.

- Iteration and recursion each involve a *termination test*: Iteration terminates when the *loop-continuation condition fails*; recursion when a *base case is recognized*.

Recursion is expensive

- It *repeatedly* invokes the mechanism, and consequently the *overhead, of function calls*.
- This can be expensive in both processor time and memory space.
- Each recursive call causes *another copy* of the function to be created; this can consume *considerable memory*.
- The amount of memory in a computer is finite, so only a certain amount of memory can be used to store stack frames on the function call stack.
- If more function calls occur than can have their stack frames stored on the function call stack, a *fatal* error known as a stack overflow occurs.

Class Discussion

- Write a C Program to find product of 2 Numbers using Recursion
- Example:
  - Multiply 6 by 3
  - Divide it into two problems:
    1. Multiply 6 by 2
    2. Add 6 to the result of problem 1
  - Split problem 1 into 2 smaller problems:
    1. Multiply 6 by 2
       a) Multiply 6 by 1
       b) Add 6 to the result of problem 1a)
    2. Add 6 to the result of problem 1

Class Discussion

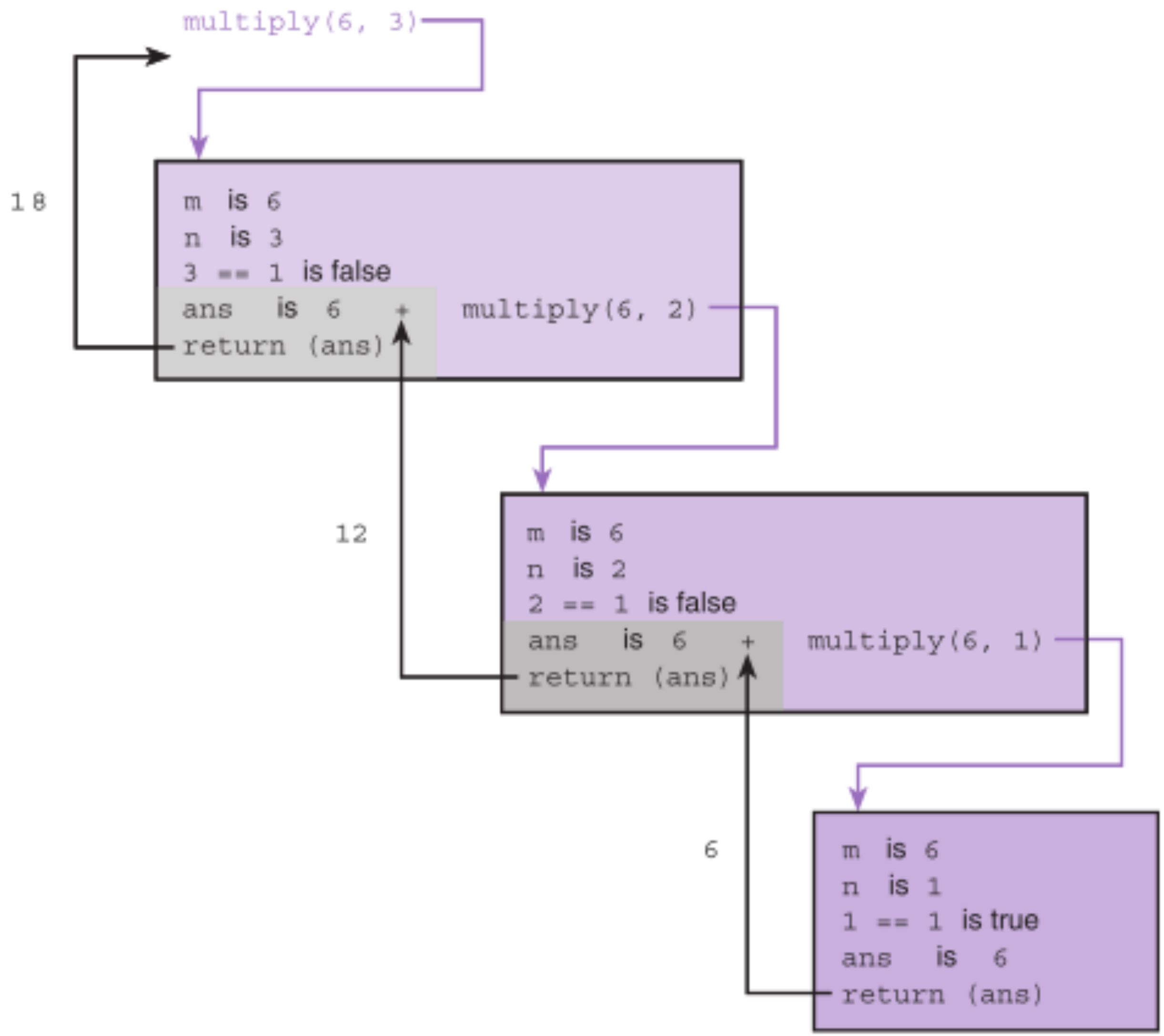
- Write a C Program to find product of 2 Numbers using Recursion
- Example:
  - Multiply 6 by 3
  - Divide it into two problems:
    1. Multiply 6 by 2
    2. Add 6 to the result of problem 1
  - Split problem 1 into 2 smaller problems:
    1. Multiply 6 by 2
       a) Multiply 6 by 1
       b) Add 6 to the result of problem 1a)
    2. Add 6 to the result of problem 1

- Generalization:
  - If n is 1,
    - ans is m.
  - Else
    - ans is m + multiply(m-1)

Trace Multiply

FIGURE 9.5

Trace of Function Multiply



Recursive Multiply

FIGURE 9.9 Recursive Function multiply with Print Statements to Create Trace and Output from multiply(8, 3)

```
/*
 * *** Includes calls to printf to trace execution ***
 * Performs integer multiplication using + operator.
 * Pre: m and n are defined and n > 0
 * Post: returns m * n
 */
int
multiply(int m, int n)
{
      int ans;

  printf("Entering multiply with m = %d, n = %d\n", m, n);

      if (n == 1)
            ans = m; /* simple case */
      else
            ans = m + multiply(m, n - 1); /* recursive step */
  printf("multiply(%d, %d) returning %d\n", m, n, ans);

      return (ans);
}

Entering multiply with m = 8, n = 3
Entering multiply with m = 8, n = 2
Entering multiply with m = 8, n = 1
multiply(8, 1) returning 8
multiply(8, 2) returning 16
multiply(8, 3) returning 24
```

Class Discussion

- Raising an integer to an integer power
- Example:
  - 3^3
  - Divide it into two problems:
    1. 3^2
    2. Multiply 3 to the result of problem 1
  - Split problem 1 into 2 smaller problems:
    1. 3^2
       a) 3^1
       b) Multiply 3 to the result of problem 1a)
    2. Multiply 3 to the result of problem 1

Class Discussion

- Raising an integer to an integer power
- Example:
  - 3^3
  - Divide it into two problems:
    1. 3^2
    2. Multiply 3 to the result of problem 1
  - Split problem 1 into 2 smaller problems:
    1. 3^2
       a) 3^1
       b) Multiply 3 to the result of problem 1a)
    2. Multiply 3 to the result of problem 1

- Generalization:
  - If n is 1,
    - ans is m.
  - Else
    - ans is m * power(m,n)

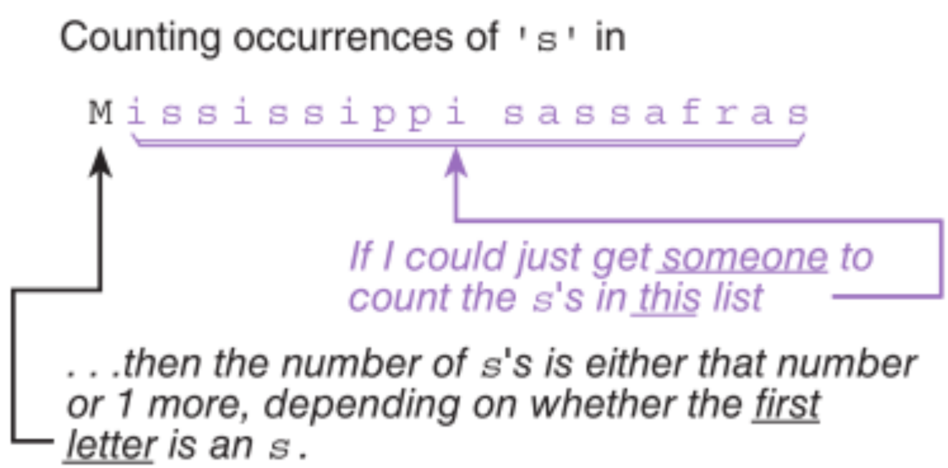
Count by Recursion

- Develop a function to count the number of times a particular character appears in a string.

count('s', "Mississippi sassafrs");

FIGURE 9.3

Thought Process of Recursive Algorithm Developer



Counting Occurences Code (1)

FIGURE 9.4 Counting Occurrences of a Character in a String

```
/*
 * Counting occurrences of a letter in a string.
 */

#include <stdio.h>

int count(char ch, const char *str);

int
main(void)
{
     char str[80];          /* string to be processed */
     char target;           /* character counted */
     int my_count;

     printf("Enter up to 79 characters.\n");
     gets(str);             /* read in the string */

     printf("Enter the character you want to count: ");
     scanf("%c", &target);

     my_count = count(target, str);
     printf("The number of occurrences of %c in\n\"%s\"\nis %d\n",
            target, str, my_count);

```

(continued)

Counting Occurences Code (2)

FIGURE 9.4 (continued)

```
      return (0);
}

/*
 * Counts the number of times ch occurs in string str.
 * Pre:  Letter ch and string str are defined.
 */
int
count(char ch, const char *str)
{
      int ans;

      if (str[0] == '\0')                              /* simple case */
          ans = 0;
      else                    /* redefine problem using recursion */
         if (ch == str[0]) /* first character must be counted */
             ans = 1 + count(ch, &str[1]);
         else                 /* first character is not counted */
             ans = count(ch, &str[1]);

      return (ans);
}

Enter up to 79 characters.
this is the string I am testing
Enter the character you want to count: t
The number of occurrences of t in
"this is the string I am testing" is 5
```

Iteration vs Recursion

- Iteration
  - When the problem is simple
  - When solution is not inherently recursive
  - The stack space available to a thread is often much less than the space available in the heap, Recursive algorithms require more stack space than iterative algorithms.
- Recursion
  - When the problem is complex
  - When the solution is inherently recursive

Iteration vs Recursion