“To Iterate is Human, to Recurse, Divine”

- James O. Coplien

CSE102
Computer Programming with C

2016-2017 Fall Semester

Recursion

Largely adapted from J.R. Hanly, E.B. Koffman, F.E. Sevilgen, and others…

Functions in C

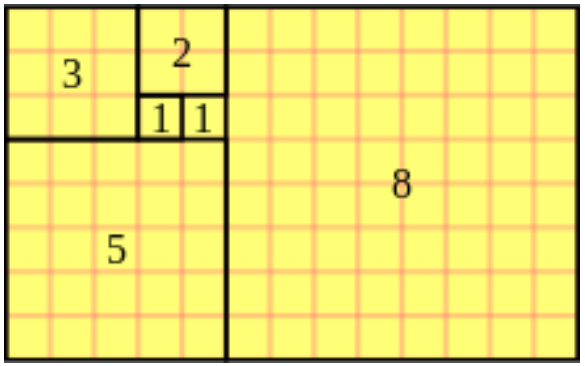
```
#include <stdio.h>

int f2(int x) {
  return x*2;
}

int f3(int x) {
  return f2(x)*3;
}

int f4(int x) {
   return f3(x)*4;
}

int f5(int x) {
   return f4(x)*5;
}

int f6(int x) {
   return f5(x)*6;
}

void main() {
   int a = f6(10);
}
```

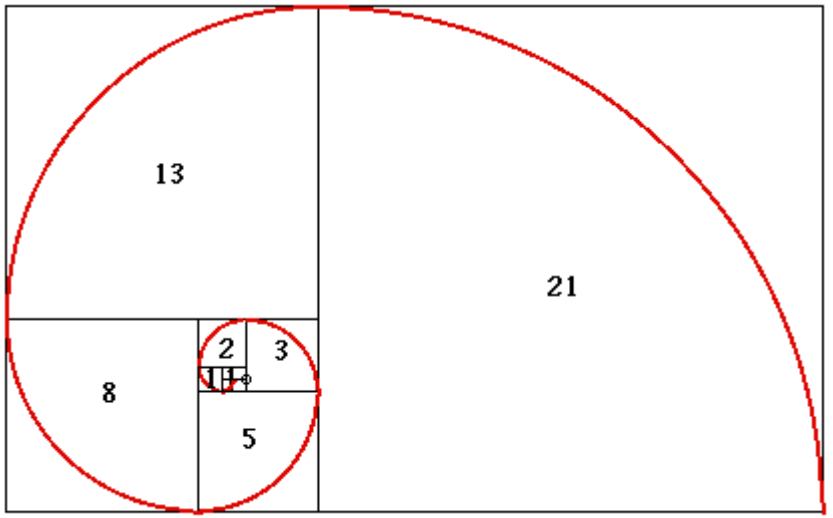
Fibonacci Numbers

1, 1, 2, 3, 5, 8, 13, 21,34, 55, 89, 144,…

1+1=2
1+2=3
2+3=5
3+5=8
5+8=13
8+13=21
13+21=34
21+34=55
• • •

$$f(n) = \begin{cases} n = 0 & 0 \\ n = 1 & 1 \\ n > 1 & f(n-1) + f(n-2) \end{cases}$$

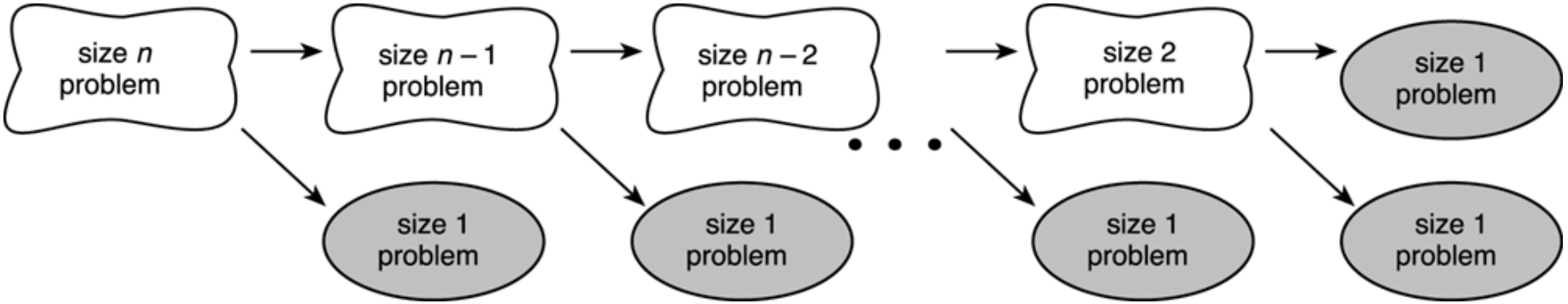




Recursive Functions

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot \text{fact}(n-1) & \text{if } n > 1 \end{cases}$$

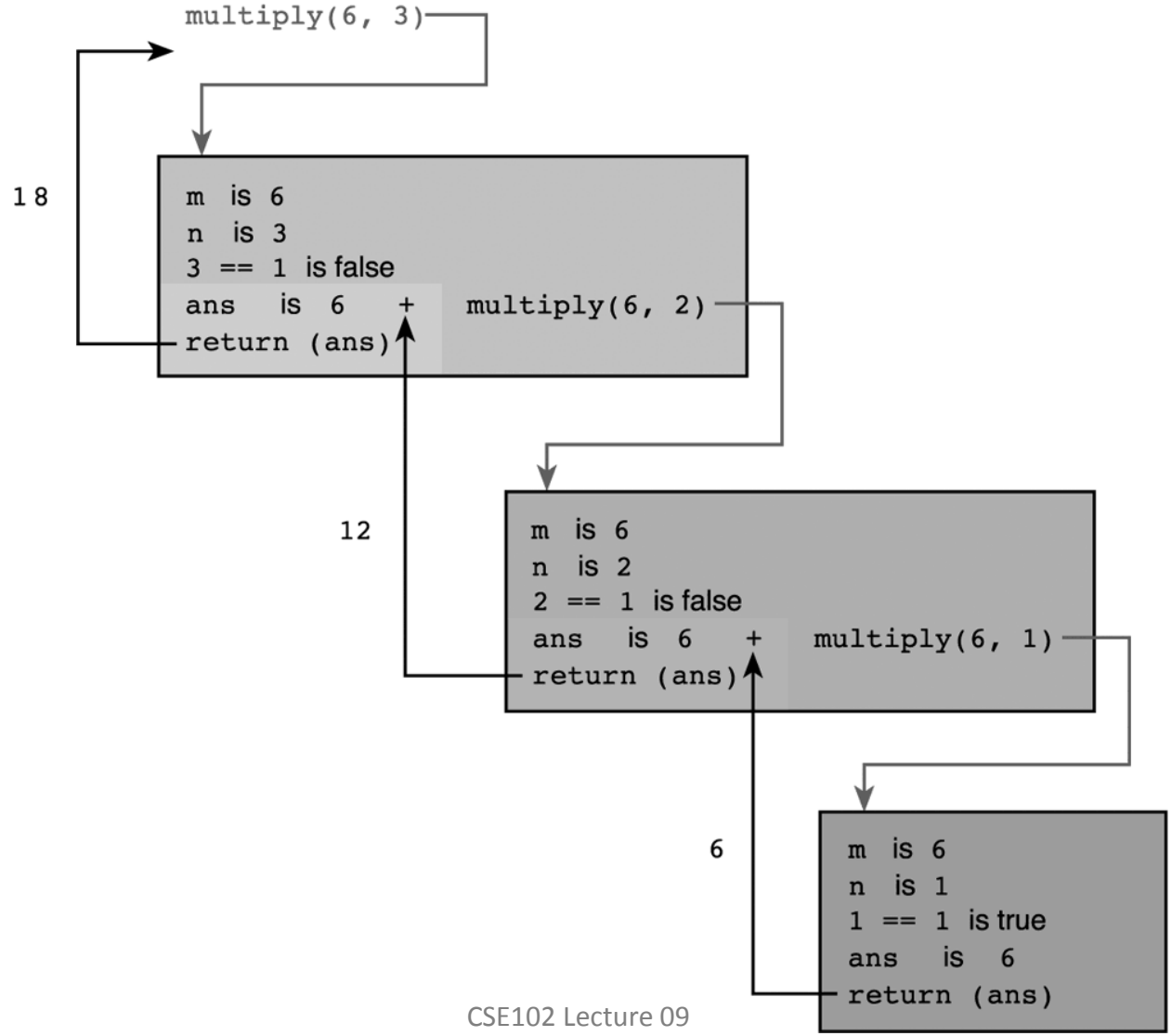
Splitting a Problem into Smaller Problems



Recursive Function multiply

```
1.  /*
2.   *  Performs integer multiplication using + operator.
3.   *  Pre:   m and n are defined and n > 0
4.   *  Post:  returns m * n
5.   */
6.  int
7.  multiply(int m, int n)
8.  {
9.        int ans;
10.
11.       if (n == 1)
12.             ans = m;      /* simple case */
13.       else
14.             ans = m + multiply(m, n  - 1);  /* recursive step */
15.
16.       return (ans);
17. }
```

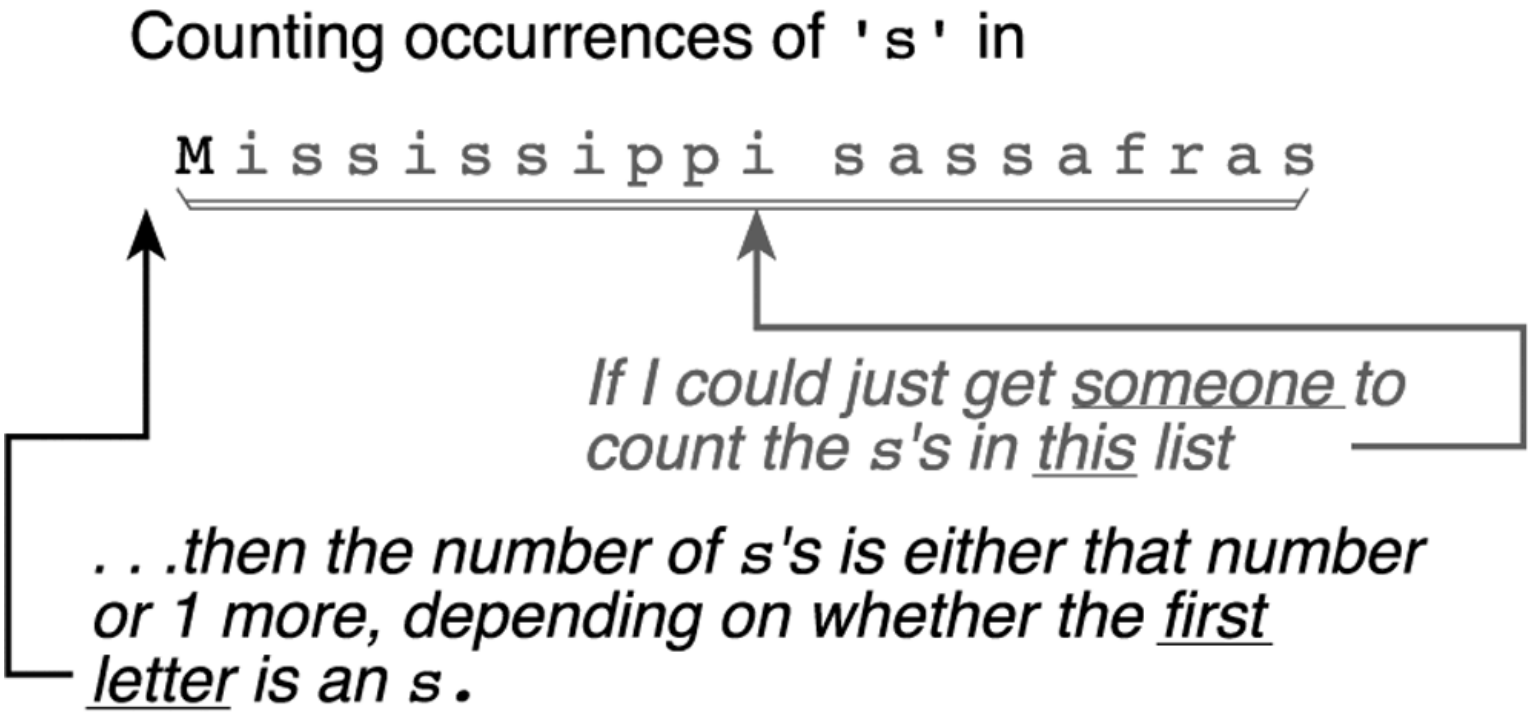
Trace of Function multiply



Output from multiply(8, 3)

```
7.  int
8.  multiply(int m, int n)
9.  {
10.       int ans;
11.
12.   printf("Entering multiply with m = %d, n = %d\n", m, n);
13.
14.       if (n == 1)
15.             ans = m;      /* simple case */
16.       else
17.             ans = m + multiply(m, n - 1); /* recursive step */
18.   printf("multiply(%d, %d) returning %d\n", m, n, ans);
19.
20.       return (ans);
21. }
22.
23. Entering multiply with m = 8, n = 3
24. Entering multiply with m = 8, n = 2
25. Entering multiply with m = 8, n = 1
26. multiply(8, 1) returning 8
27. multiply(8, 2) returning 16
28. multiply(8, 3) returning 24
```

Recursive Algorithm Development



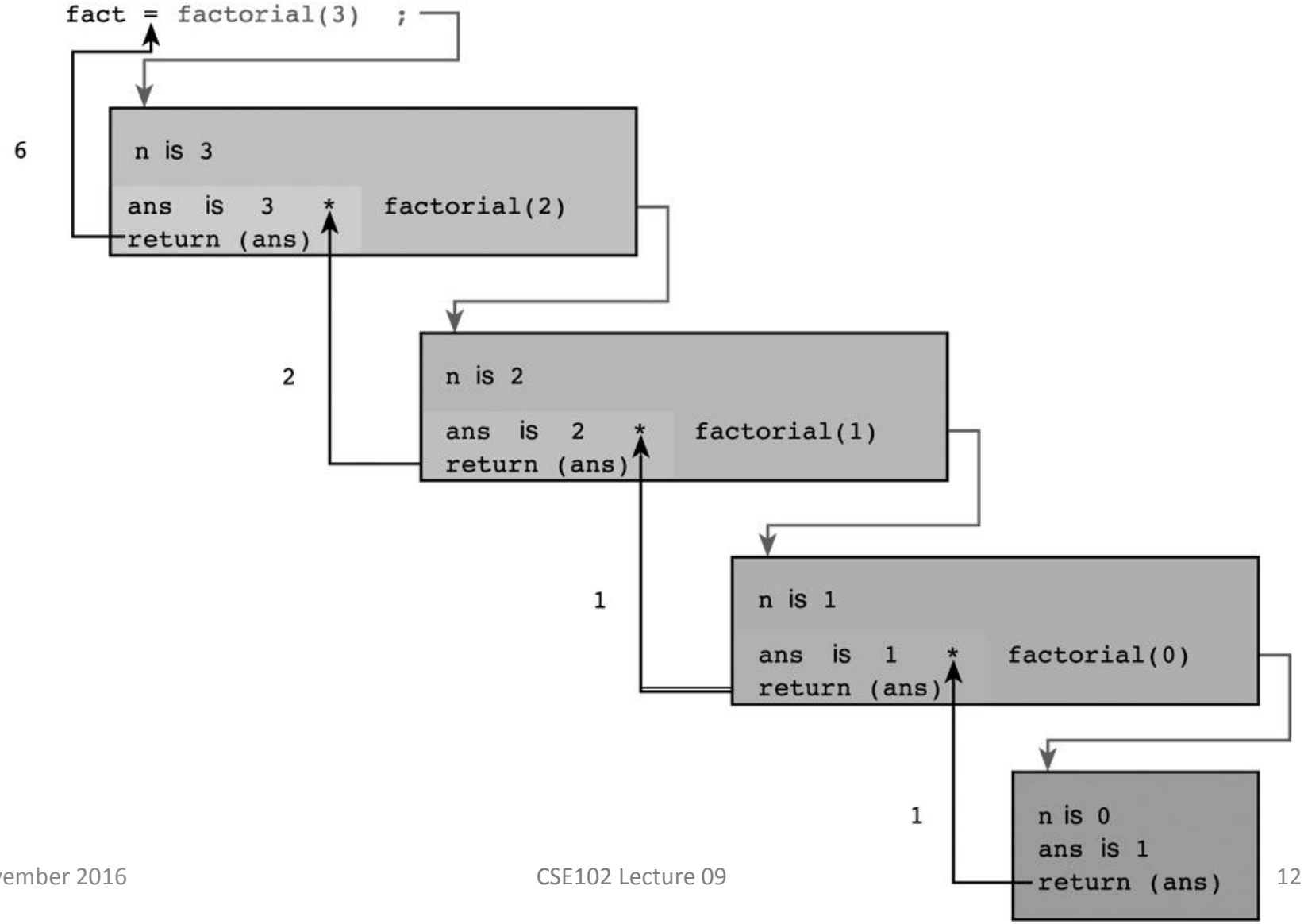
Count a Character in a String

```
1.  /*
2.   *  Count the number of occurrences of character ch in string str
3.   */
4.  int
5.  count(char ch, const char *str)
6.  {
7.
8.        int ans;
9.
10.       if (str[0] == '\0')                          /*  simple case  */
11.             ans = 0;
12.       else                        /*  redefine problem using recursion */
13.             if (ch == str[0])     /*  first character must be counted  */
14.                   ans = 1 + count(ch, &str[1]);
15.             else                  /*  first character is not counted   */
16.                   ans = count(ch, &str[1]);
17.
18.       return (ans);
19. }
```

Recursive factorial Function

```
1.  /*
2.   *  Compute n! using a recursive definition
3.   *  Pre:  n >= 0
4.   */
5.  int
6.  factorial(int n)
7.  {
8.        int ans;
9.
10.       if (n == 0)
11.             ans = 1;
12.       else
13.             ans = n * factorial(n - 1);
14.
15.       return (ans);
16. }
```

Trace of fact = factorial(3);



Iterative Function factorial

```
1.  /*
2.   * Computes n!
3.   * Pre: n is greater than or equal to zero
4.   */
5.  int
6.  factorial(int n)
7.  {
8.      int i,              /* local variables */
9.          product = 1;
10.
11.     /* Compute the product n x (n-1) x (n-2) x ... x 2 x 1 */
12.     for  (i = n;  i > 1;  --i) {
13.         product = product * i;
14.     }
15.
16.     /* Return function result */
17.     return (product);
18. }
```

Recursive Function fibonacci

```
1.  /*
2.   *  Computes the nth Fibonacci number
3.   *  Pre: n > 0
4.   */
5.  int
6.  fibonacci(int n)
7.  {
8.        int ans;
9.
10.       if (n == 1  ||  n == 2)
11.             ans = 1;
12.       else
13.             ans = fibonacci(n - 2)  +  fibonacci(n - 1);
14.
15.       return (ans);
16. }
```

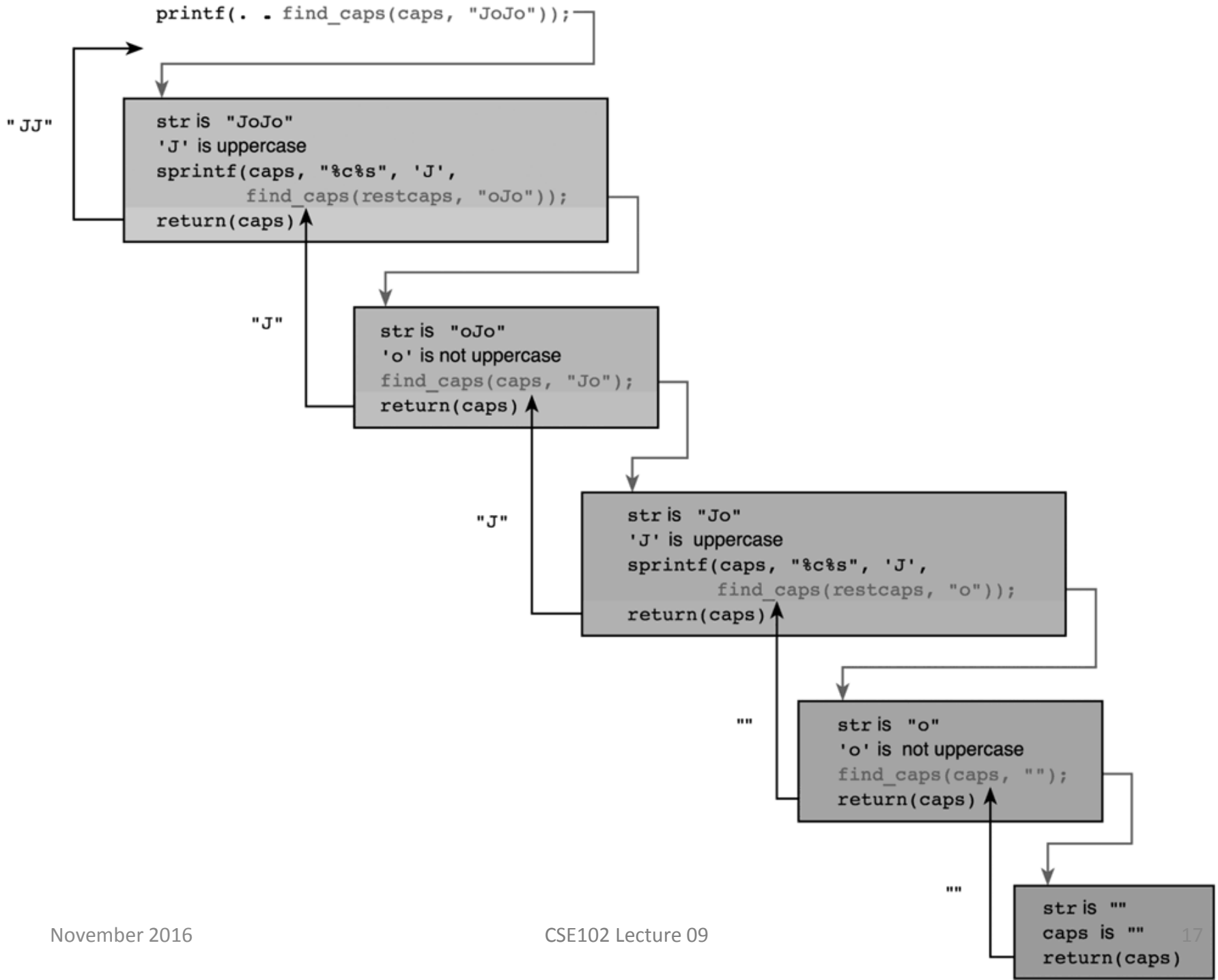
Recursive Function gcd

```
7.  /*
8.   *  Finds the greatest common divisor of m and n
9.   *  Pre:  m and n are both > 0
10.  */
11. int
12. gcd(int m, int n)
13. {
14.       int ans;
15.
16.       if (m % n == 0)
17.             ans = n;
18.       else
19.             ans = gcd(n, m % n);
20.
21.       return (ans);
22. }
```

(continued)

Extract Capital Letters from a String

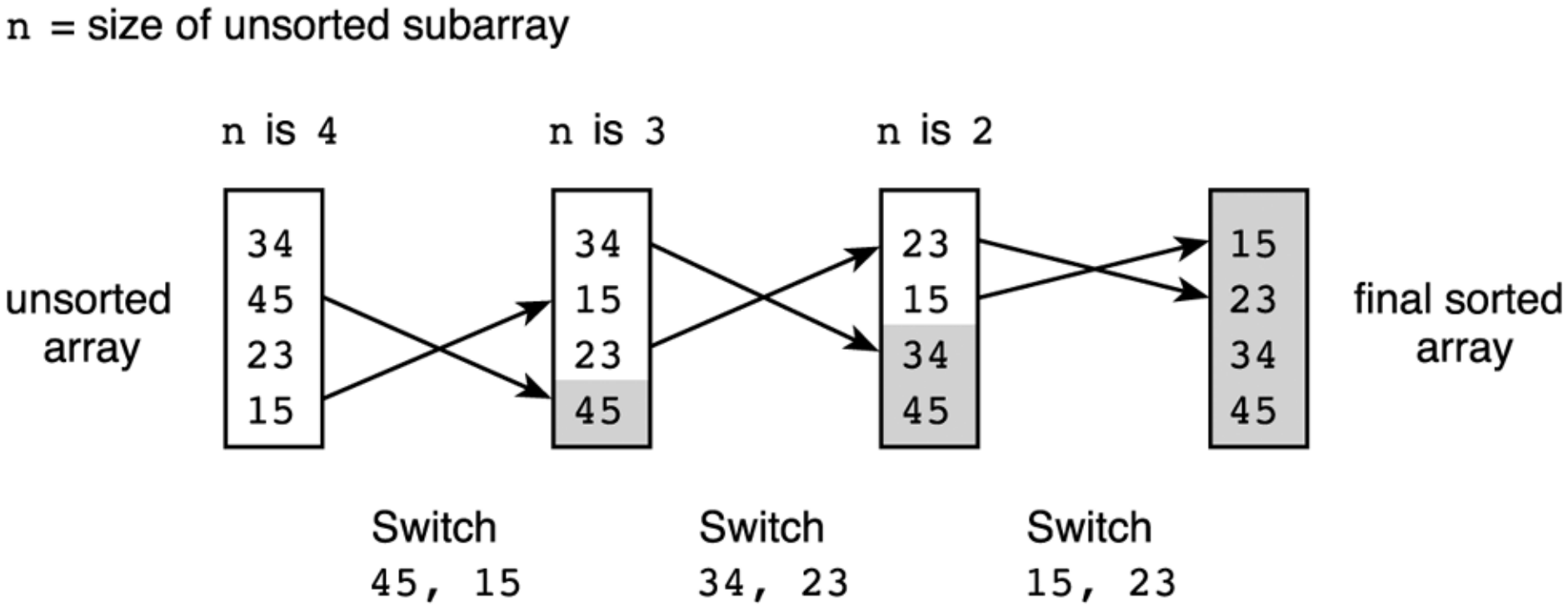
```
1.  /*
2.   *  Forms a string containing all the capital letters found in the input
3.   *  parameter str.
4.   *  Pre:  caps has sufficient space to store all caps in str plus the null
5.   */
6.  char *
7.  find_caps(char       *caps,  /* output - string of all caps found in str       */
8.            const char *str)   /* input  - string from which to extract caps     */
9.  {
10.       char restcaps[STRSIZ]; /* caps from reststr  */
11.
12.       if (str[0] == '\0')
13.             caps[0] = '\0';  /* no letters in str => no caps in str             */
14.       else
15.             if (isupper(str[0]))
16.                   sprintf(caps, "%c%s", str[0], find_caps(restcaps, &str[1]));
17.             else
18.                   find_caps(caps, &str[1]);
19.
20.       return (caps);
21. }
```


printf(. . find_caps(caps, "JoJo"));
"JJ"
str is "JoJo"
'J' is uppercase
sprintf(caps, "%c%s", 'J',
find_caps(restcaps, "oJo"));
return(caps)
"J"
str is "oJo"
'o' is not uppercase
find_caps(caps, "Jo");
return(caps)
"J"
str is "Jo"
'J' is uppercase
sprintf(caps, "%c%s", 'J',
find_caps(restcaps, "o"));
return(caps)
""
str is "o"
'o' is not uppercase
find_caps(caps, "");
return(caps)
""
str is ""
caps is ""
return(caps)

Sequence of Events

Call `find_caps` with input argument `"JoJo"` to determine value to print.
Since `'J'` is a capital letter,
prepare to use `sprintf` to build a string with `'J'`
and the result of calling `find_caps` with input argument `"oJo"`.
Since `'o'` is not a capital letter,
call `find_caps` with input argument `"Jo"`.
Since `'J'` is a capital letter,
prepare to use `sprintf` to build a string with `'J'`
and the result of calling `find_caps` with input argument `"o"`.
Since `'o'` is not a capital letter,
cal `find_caps` with input argument `""`.
Return `""` from fifth call.
Return `""` from fourth call.
Complete execution of `sprintf` combining `'J'` and `""`.
Return `"J"` from third call.
Return `"J"` from second call.
Complete execution of `sprintf` combining `'J'` and `"J"`.
Return `"JJ"` from original call.
Complete call to `printf` to print `Capital letters in JoJo are JJ`.

Trace of Selection Sort

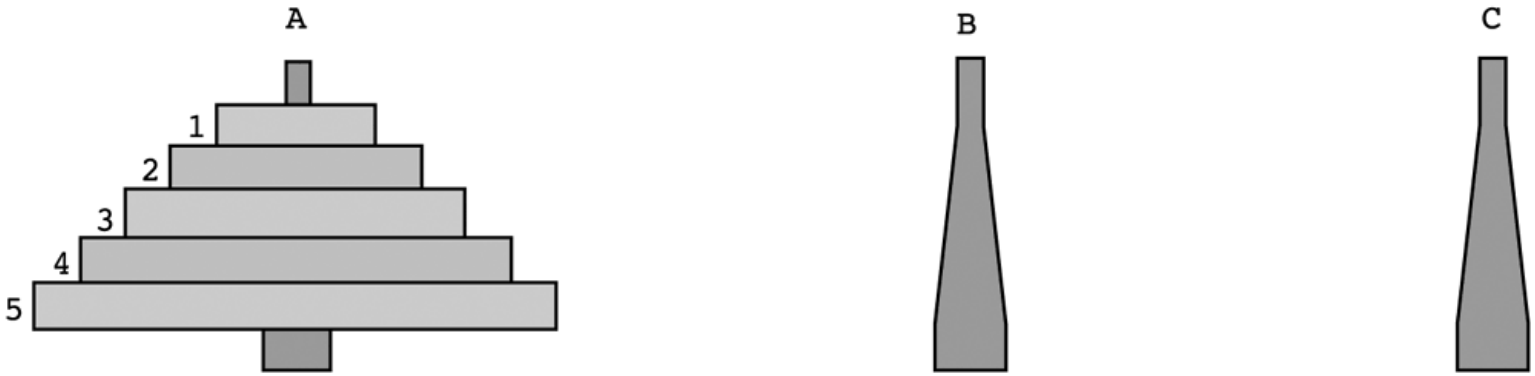


Recursive Selection Sort

```
31.  /*
32.   *  Sorts n elements of an array of integers
33.   *  Pre:  n > 0 and first n elements of array are defined
34.   *  Post: array elements are in ascending order
35.   */
36.  void
37.  select_sort(int array[],  /* input/output - array to sort                    */
38.               int n)        /* input - number of array elements to sort       */
39.  {
40.
41.        if (n > 1) {
42.              place_largest(array, n);
43.              select_sort(array, n - 1);
44.        }
     }
```

```
1.  /*
2.   *  Finds the largest value in list array[0]..array[n-1] and exchanges it
3.   *  with the value at array[n-1]
4.   *  Pre:  n > 0 and first n elements of array are defined
5.   *  Post: array[n-1] contains largest value
6.   */
7.  void
8.  place_largest(int array[],   /* input/output - array in which to place largest */
9.                int n)         /* input - number of array elements to
10.                                  consider                                      */
11. {
12.       int temp,        /* temporary variable for exchange                     */
13.           j,           /* array subscript and loop control                    */
14.           max_index;   /* index of largest so far                             */
15.
16.       /*  Save subscript of largest array value in max_index                  */
17.       max_index = n - 1;      /* assume last value is largest                 */
18.       for  (j = n - 2;  j >= 0;  --j)
19.           if (array[j] > array[max_index])
20.                 max_index = j;
21.
22.       /*  Unless largest value is already in last element, exchange
23.           largest and last elements                                           */
24.       if (max_index != n - 1) {
25.             temp = array[n - 1];
26.             array[n - 1] = array[max_index];
27.             array[max_index] = temp;
28.       }
29. }
30.
```

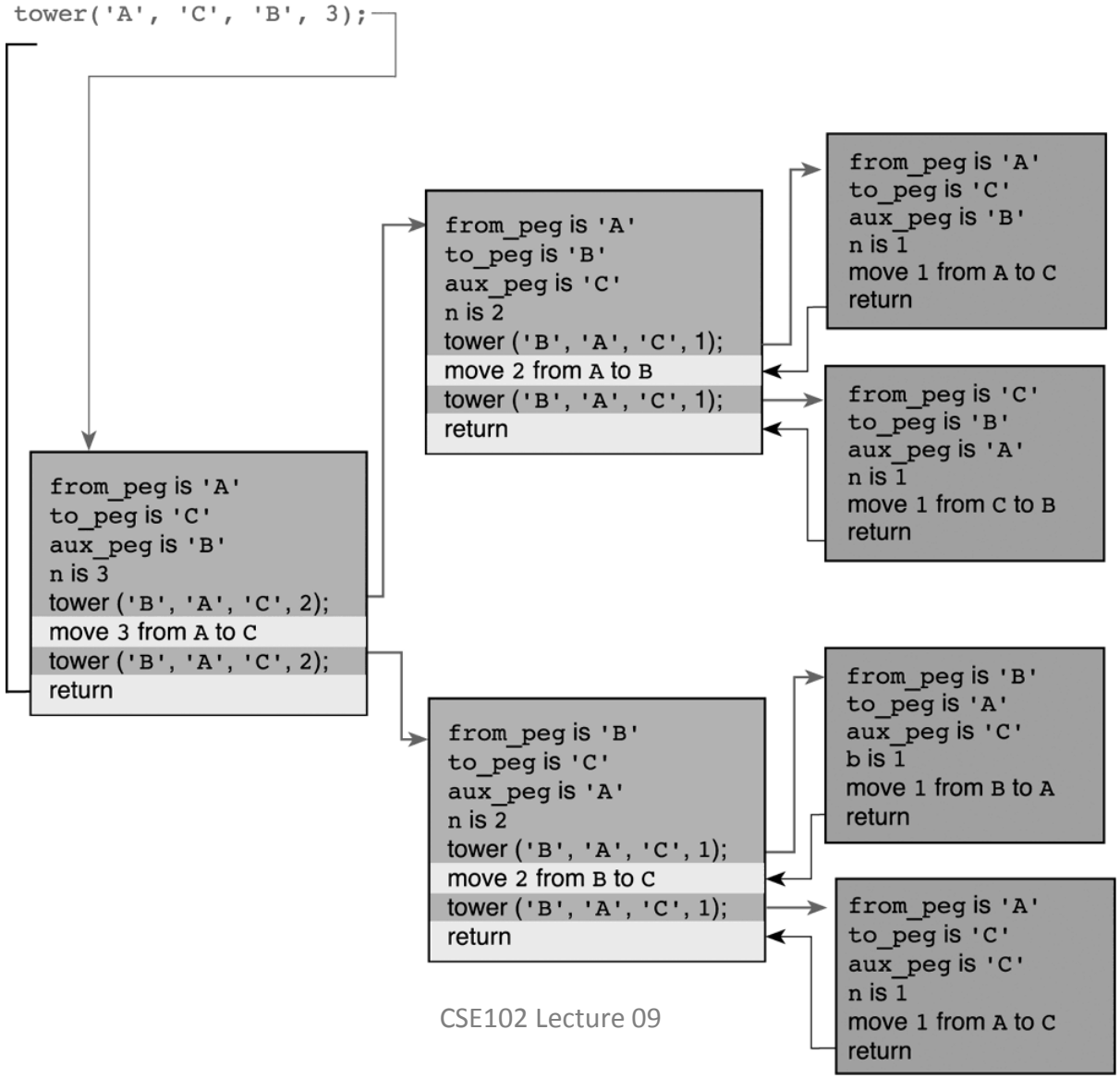
Towers of Hanoi



Recursive Function tower

```
1.  /*
2.   *  Displays instructions for moving n disks from from_peg to to_peg using
3.   *  aux_peg as an auxiliary.  Disks are numbered 1 to n (smallest to
4.   *  largest). Instructions call for moving one disk at a time and never
5.   *  require placing a larger disk on top of a smaller one.
6.   */
7.  void
8.  tower(char from_peg,    /* input - characters naming            */
9.        char to_peg,     /*         the problem's                */
10.       char aux_peg,    /*         three pegs                   */
11.       int n)           /* input - number of disks to move      */
12. {
13.       if (n == 1) {
14.             printf("Move disk 1 from peg %c to peg %c\n", from_peg, to_peg);
15.       } else {
16.             tower(from_peg, aux_peg, to_peg, n - 1);
17.             printf("Move disk %d from peg %c to peg %c\n", n, from_peg, to_peg);
18.             tower(aux_peg, to_peg, from_peg, n - 1);
19.       }
20. }
```

Trace of tower ('A', 'C', 'B', 3);



Output of tower('A', 'C', 'B', 3);

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```