"To Iterate is Human, to Recurse, Divine"

- James O. Coplien

CSE102
Computer Programming with C

2016-2018 Spring Semester

Recursion

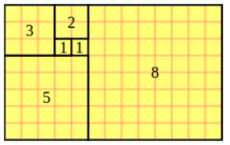
Largely adapted from J.R. Hanly, E.B. Koffman, F.E. Sevilgen, and others…

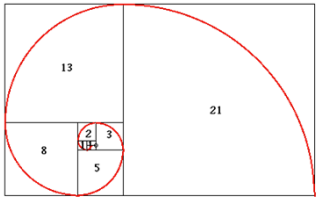
Fibonacci Numbers

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
•••

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





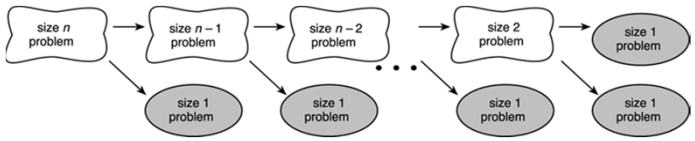
Functions in C

```
#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);
}
```


Recursive Functions

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

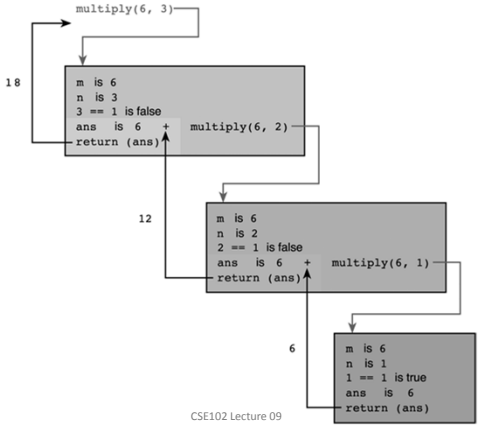
Splitting a Problem into Smaller Problems



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

Counting occurrences of `'s'` in

`Mississippi sassafras`

If I could just get someone to count the s's in this list

. . .then the number of s's is either that number or 1 more, depending on whether the first letter is an s.

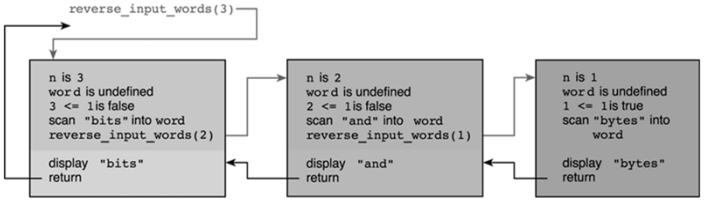
Count a Character in a String

```
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. }
```


Function reverse_input_words

```
1.  /*
2.   *  Take n words as input and print them in reverse order on separate lines.
3.   *  Pre: n > 0
4.   */
5.  void
6.  reverse_input_words(int n)
7.  {
8.        char word[WORDSIZ];  /*  local variable for storing one word            */
9.
10.       if (n <= 1) {   /* simple case: just one word to get and print          */
11.
12.             scanf("%s", word);
13.             printf("%s\n", word);
14.
15.       } else {  /* get this word; get and print the rest of the words in
16.                    reverse order; then print this word                        */
17.
18.             scanf("%s", word);
19.             reverse_input_words(n - 1);
20.             printf("%s\n", word);
21.       }
22. }
```

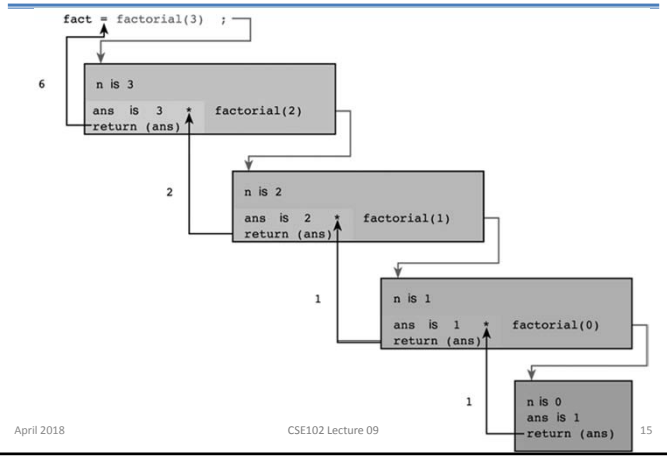

reverse_input_words(3): "bits" "and" "bytes"



Sequence of Events for Trace

Call `reverse_input_words` with n equal to 3.
- Scan the first word (`"bits"`) into `word`.
- Call `reverse_input_words` with n equal to 2.
  - Scan the second word (`"and"`) into `word`.
  - Call `reverse_input_words` with n equal to 1.
    - Scan the third word (`"bytes"`) into `word`.
    - Display the third word (`"bytes"`).
    - Return from third call.
  - Display the second word (`"and"`).
  - Return from second call.
- Display the first word (`"bits"`).
- Return from original call.

Trace of fact = factorial(3);



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. }
```


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

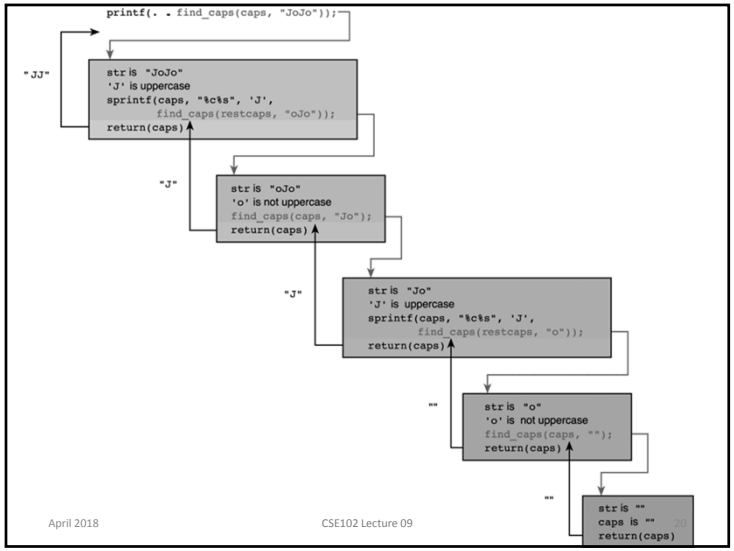
```
1.  /*
2.   *  Computes the nth Fibonacci number
3.   *  Pre: n > 0
4.   */
5.  int
6.  fibonacci(int n)
7.  {
8.        int ans;
9.
10.       if (n == 1  ||  n == 2)
11.             ans = 1;
12.       else
13.             ans = fibonacci(n - 2)  +  fibonacci(n - 1);
14.
15.       return (ans);
16.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


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. }
```


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. }
```

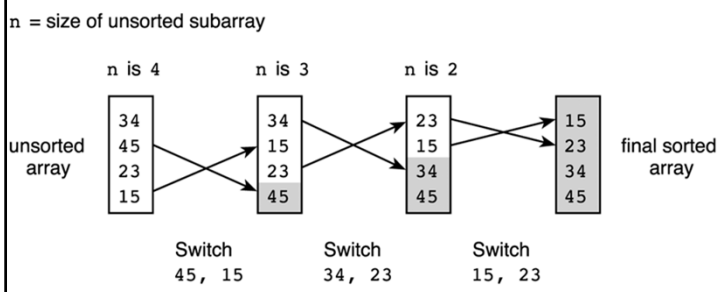



Sequence of Events

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.
```

Case Study: Recursive Set Operations

Sets represented as character strings

```
#define SETSIZ  65  /* 52 uppercase and lowercase letters, 10 digits,
                       {, }, and '\0'                                  */
#define TRUE    1
#define FALSE   0

int is_empty(const char *set);
int is_element(char ele, const char *set);
int is_set(const char *set);
int is_subset(const char *sub, const char *set);
char *set_union(char *result, const char *set1, const char *set2);
void print_with_commas(const char *str);
void print_set(const char *set);
char *get_set(char *set);
```

(continued)


```
int
main(void)
{
      char ele, set_one[SETSIZ], set_two[SETSIZ], set_three[SETSIZ];

      printf("A set is entered as a string of up to %d letters\n",
                  SETSIZ - 3);
      printf("and digits enclosed in {} ");
      printf("(no duplicate characters)\n");
      printf("For example, {a, b, c} is entered as {abc}\n");

      printf("Enter a set to test validation function> ");
      get_set(set_one);
      putchar('\n');
      print_set(set_one);
      if (is_set(set_one))
            printf(" is a valid set\n");
      else
            printf(" is invalid\n");

      printf("Enter a single character, a space, and a set> ");
      while(isspace(ele = getchar()));  /* gets first character after
                                           whitespace                 */
      get_set(set_one);
      printf("\n%c ", ele);
      if (is_element(ele, set_one))
            printf("is an element of ");
      else
            printf("is not an element of ");
      print_set(set_one);

      printf("\nEnter two sets to test set_union> ");
      get_set(set_one);
      get_set(set_two);
      printf("\nThe union of ");
      print_set(set_one);
      printf(" and ");
      print_set(set_two);
      printf(" is ");
      print_set(set_union(set_three, set_one, set_two));
      putchar('\n');

      return (0);
}
```



```

/*
 *  Determines if set is empty.  If so, returns 1;  if not, returns 0.
 */
int
is_empty(const char *set)
{
      return (set[0] == '\0');
}

/*
 *  Determines if ele is an element of set.
 */
int
is_element(char       ele,     /* input - element to look for in set         */
           const char *set)    /* input - set in which to look for ele       */
{
      int ans;

      if (is_empty(set))
            ans = FALSE;
      else if (set[0] == ele)
            ans = TRUE;
      else
            ans = is_element(ele, &set[1]);

      return (ans);
}

```



```
/*
 *  Determines if string value of set represents a valid set (no duplicate
 *  elements)
 */

int
is_set (const char *set)
{
      int ans;

      if (is_empty(set))
            ans = TRUE;
      else if (is_element(set[0], &set[1]))
            ans = FALSE;
      else
            ans = is_set(&set[1]);
      return (ans);
}

/*
 *  Determines if value of sub is a subset of value of set.
 */
int
is_subset(const char *sub, const char *set)
{
      int ans;

      if (is_empty(sub))
            ans = TRUE;
      else if (!is_element(sub[0], set))
            ans = FALSE;
      else
            ans = is_subset(&sub[1], set);

      return (ans);
}
```

```
/*
 *   Finds the union of set1 and set2.
 *   Pre:  size of result array is at least SETSIZ;
 *         set1 and set2 are valid sets of characters and digits
 */
char *
set_union(char       *result,  /* output - space in which to store
          const char *set1,    /* input  - sets whose                */
          const char *set2)    /*          union is being formed     */
{
      char temp[SETSIZ];       /* local variable to hold result of call
                                  to set_union embedded in sprintf call      */

      if (is_empty(set1))
            strcpy(result, set2);
      else if (is_element(set1[0], set2))
            set_union(result, &set1[1], set2);
      else
            sprintf(result, "%c%s", set1[0],
                    set_union(temp, &set1[1], set2));

      return (result);
}

/*
 *  Displays a string so that each pair of characters is separated by a
 *  comma and a space.
 */
void
print_with_commas(const char *str)
{
      if (strlen(str) == 1) {
            putchar(str[0]);
      } else {
            printf("%c, ", str[0]);
            print_with_commas(&str[1]);
      }
}
```

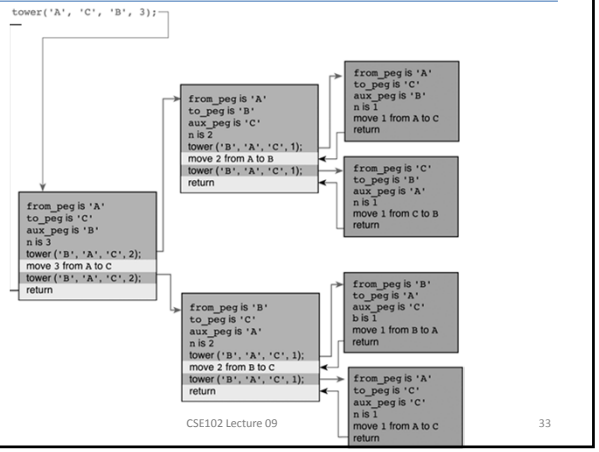

Towers of Hanoi


```
/*
 *  Displays a string in standard set notation.
 *  e.g.  print_set("abc") outputs {a, b, c}
 */
void
print_set(const char *set)
{
      putchar('{');
      if (!is_empty(set))
            print_with_commas(set);
      putchar('}');
}

/*
 *  Gets a set input as a string with brackets (e.g., {abc})
 *  and strips off the brackets.
 */
char *
get_set(char *set)   /* output - set string without brackets {}              */
{
      char inset[SETSIZ];

      scanf("%s", inset);
      strncpy(set, &inset[1], strlen(inset) - 2);
      set[strlen(inset) - 2] = '\0';
      return (set);
}
```


Recursive Function tower

```
/*
 *  Displays instructions for moving n disks from from_peg to to_peg using
 *  aux_peg as an auxiliary.  Disks are numbered 1 to n (smallest to
 *  largest). Instructions call for moving one disk at a time and never
 *  require placing a larger disk on top of a smaller one.
 */
void
tower(char from_peg,    /* input - characters naming           */
      char to_peg,      /*         the problem's               */
      char aux_peg,     /*         three pegs                  */
      int n)            /* input - number of disks to move     */
{
      if (n == 1) {
            printf("Move disk 1 from peg %c to peg %c\n", from_peg, to_peg);
      } else {
            tower(from_peg, aux_peg, to_peg, n - 1);
            printf("Move disk %d from peg %c to peg %c\n", n, from_peg, to_peg);
            tower(aux_peg, to_peg, from_peg, n - 1);
      }
}
```

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
```