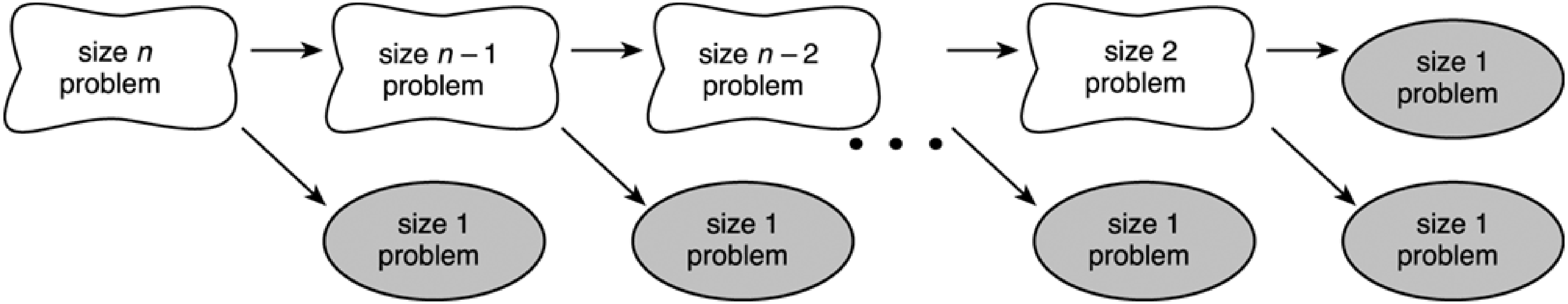
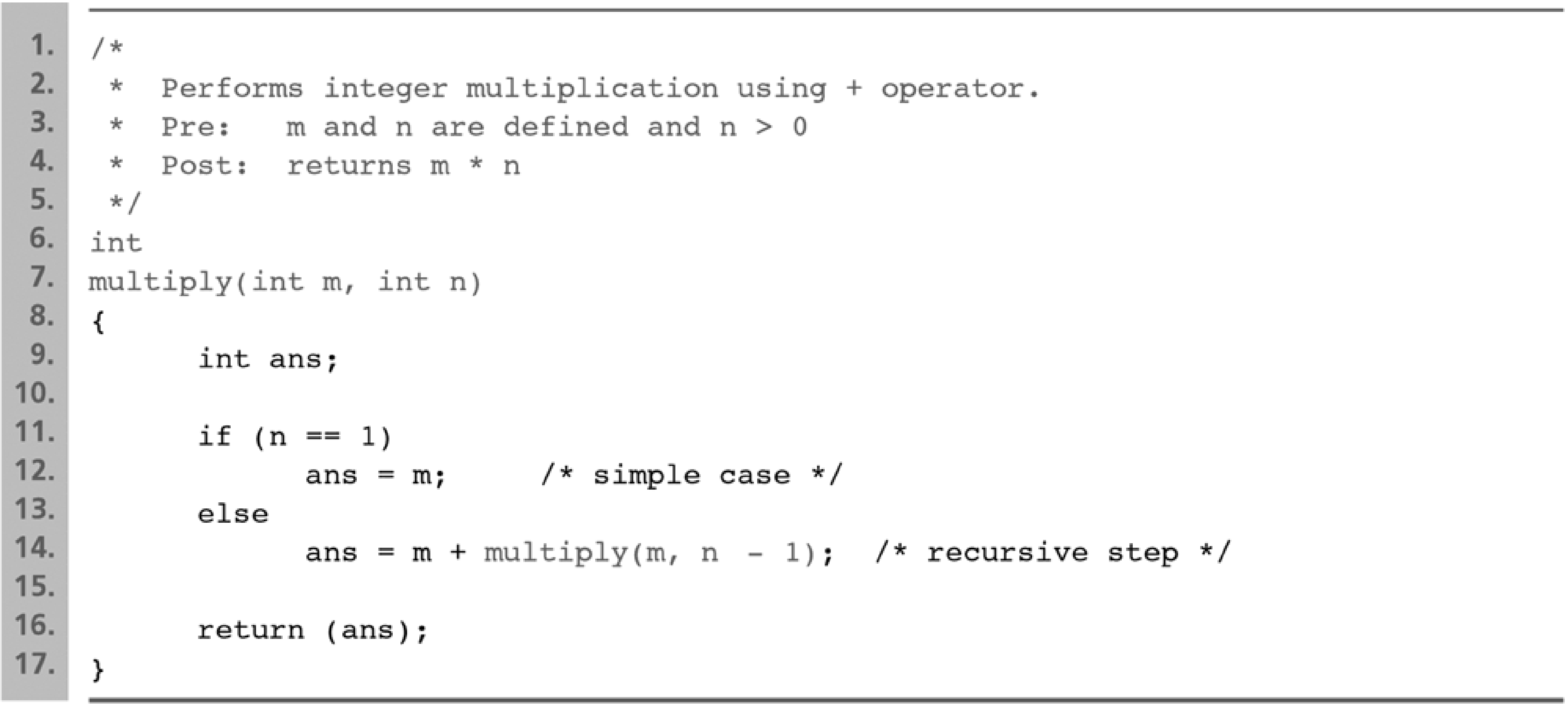
Computer Programming

Recursion

Splitting a Problem into Smaller Problems

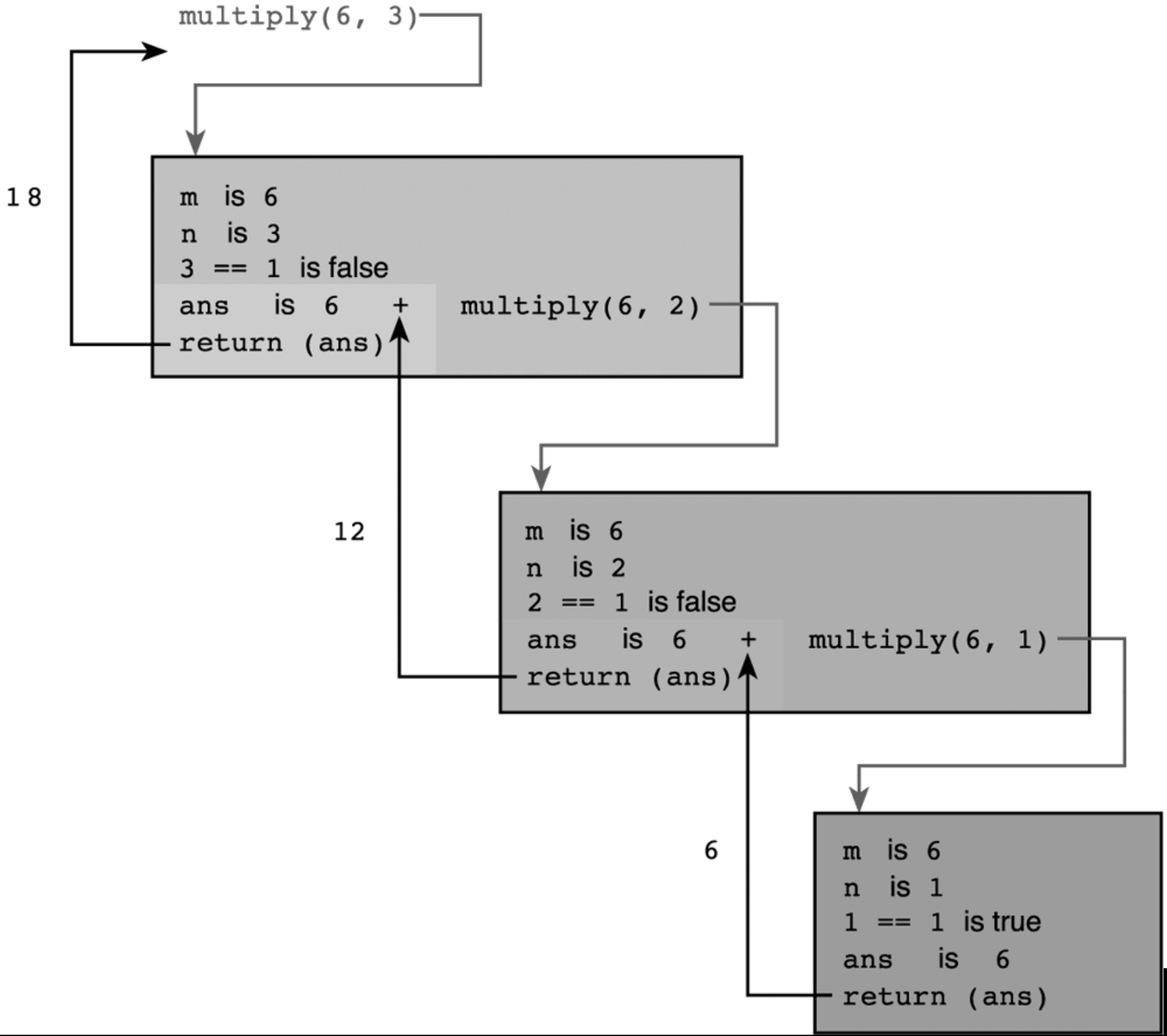


Recursive Function multiply



```
/*
 *  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;

      if (n == 1)
            ans = m;     /* simple case */
      else
            ans = m + multiply(m, n - 1);  /* recursive step */

      return (ans);
}
```

Trace of Function multiply



multiply(6, 3)

18

m is 6
n is 3
3 == 1 is false
ans is 6 + multiply(6, 2)
return (ans)

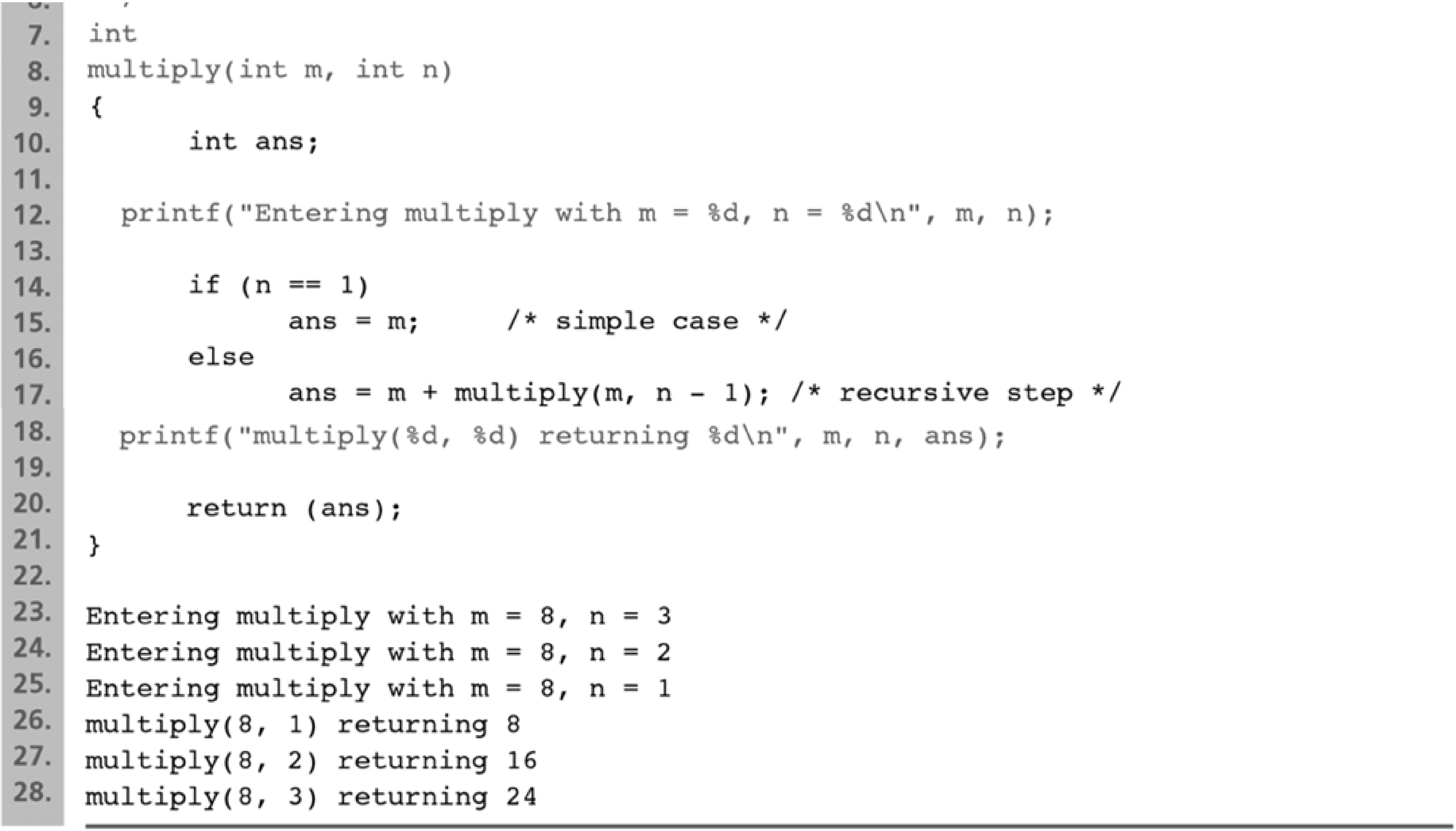
12

m is 6
n is 2
2 == 1 is false
ans is 6 + multiply(6, 1)
return (ans)

6

m is 6
n is 1
1 == 1 is true
ans is 6
return (ans)

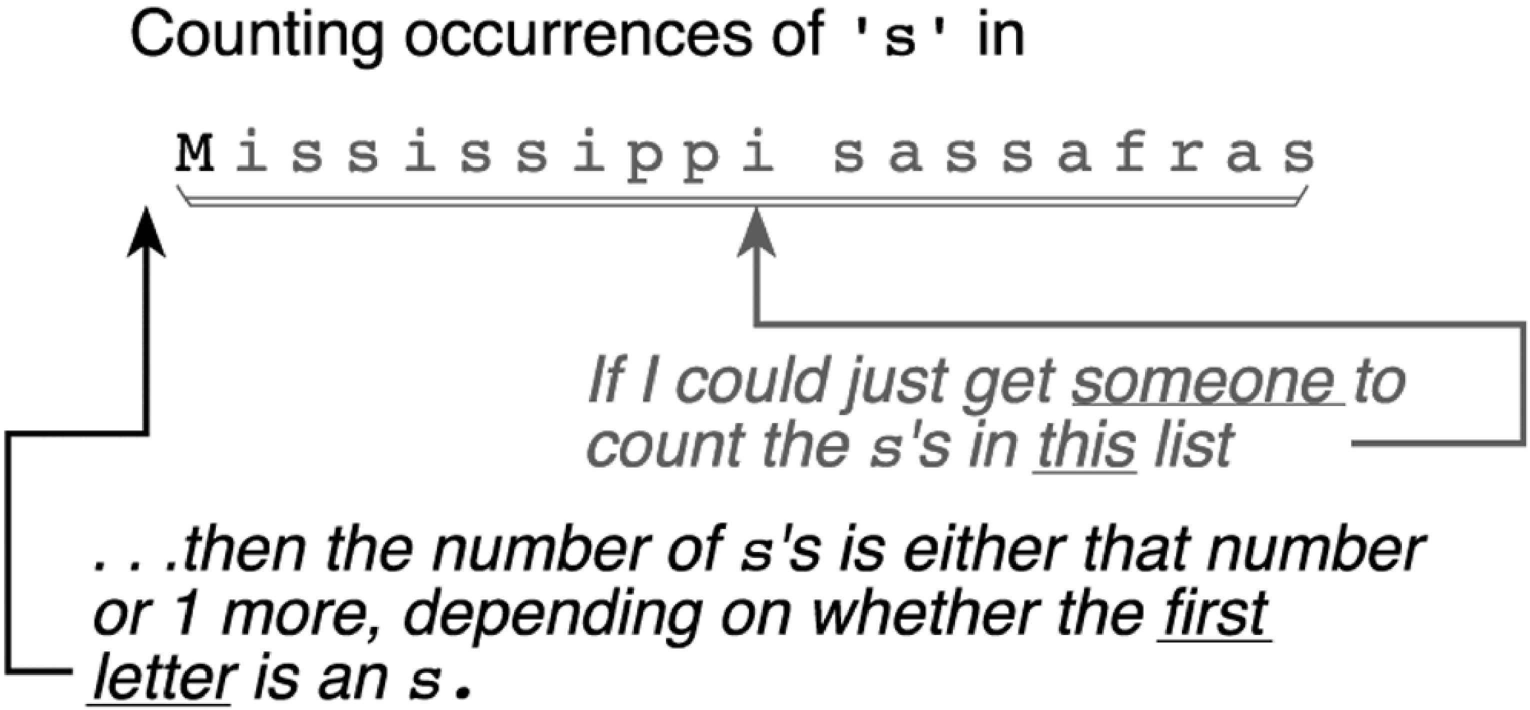
Output from multiply(8, 3)



```c
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
```

Recursive Algorithm Development



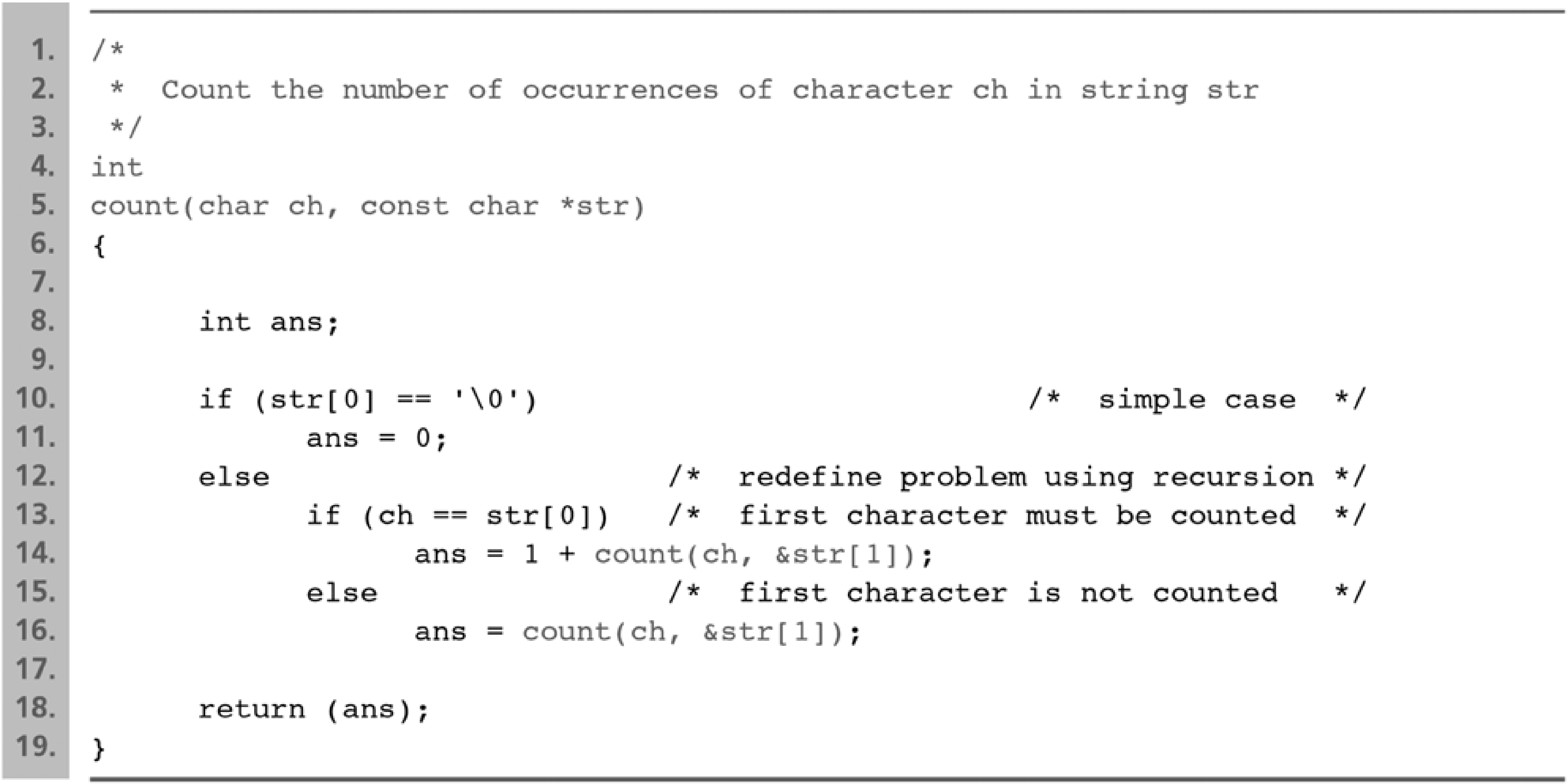
Counting occurrences of `'s'` in

`Mississippi sassafras`

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

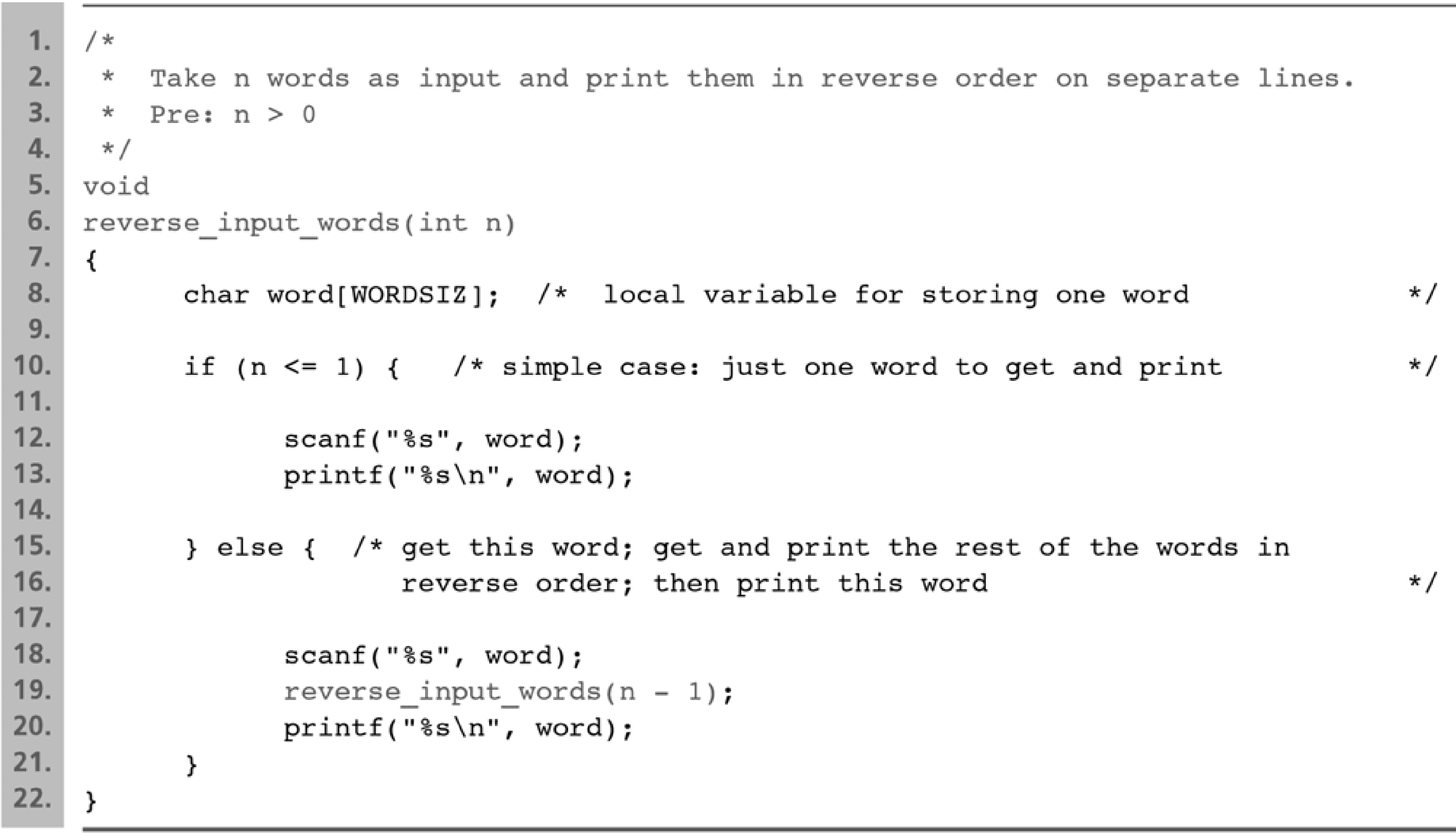
. . .then the number of s's is either that number or 1 more, depending on whether the first letter is an s.

Count a Character in a String



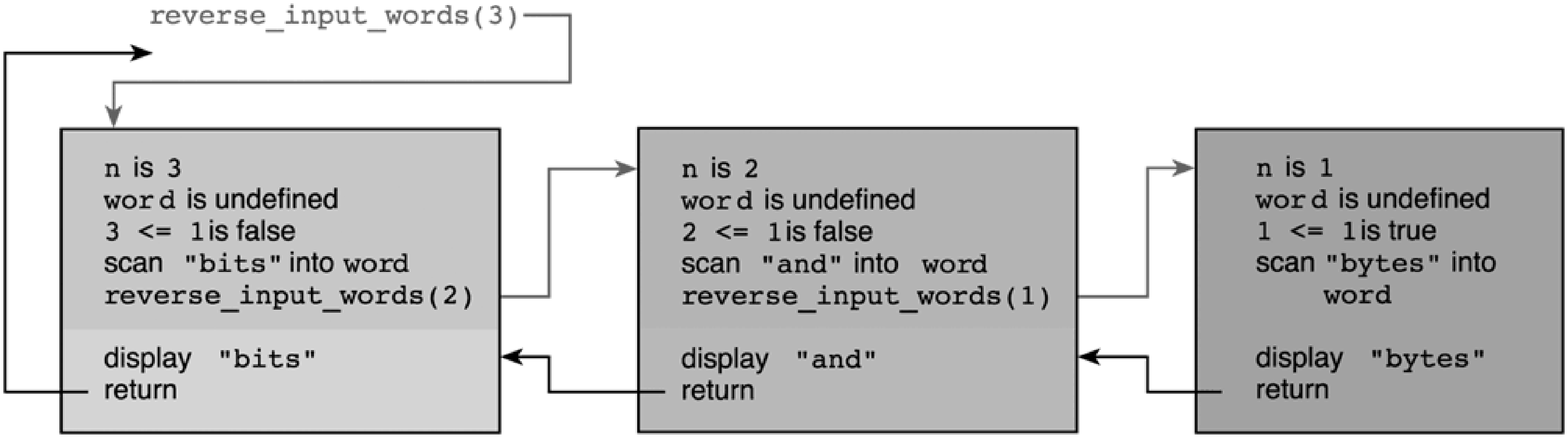
```
/*
 *  Count the number of occurrences of character ch in string str
 */
int
count(char ch, const char *str)
{

      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);
}
```

Function reverse_input_words



```
/*
 *  Take n words as input and print them in reverse order on separate lines.
 *  Pre: n > 0
 */
void
reverse_input_words(int n)
{
      char word[WORDSIZ];  /*  local variable for storing one word             */

      if (n <= 1) {   /* simple case: just one word to get and print           */

            scanf("%s", word);
            printf("%s\n", word);

      } else {  /* get this word; get and print the rest of the words in
                   reverse order; then print this word                         */

            scanf("%s", word);
            reverse_input_words(n - 1);
            printf("%s\n", word);
      }
}
```

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



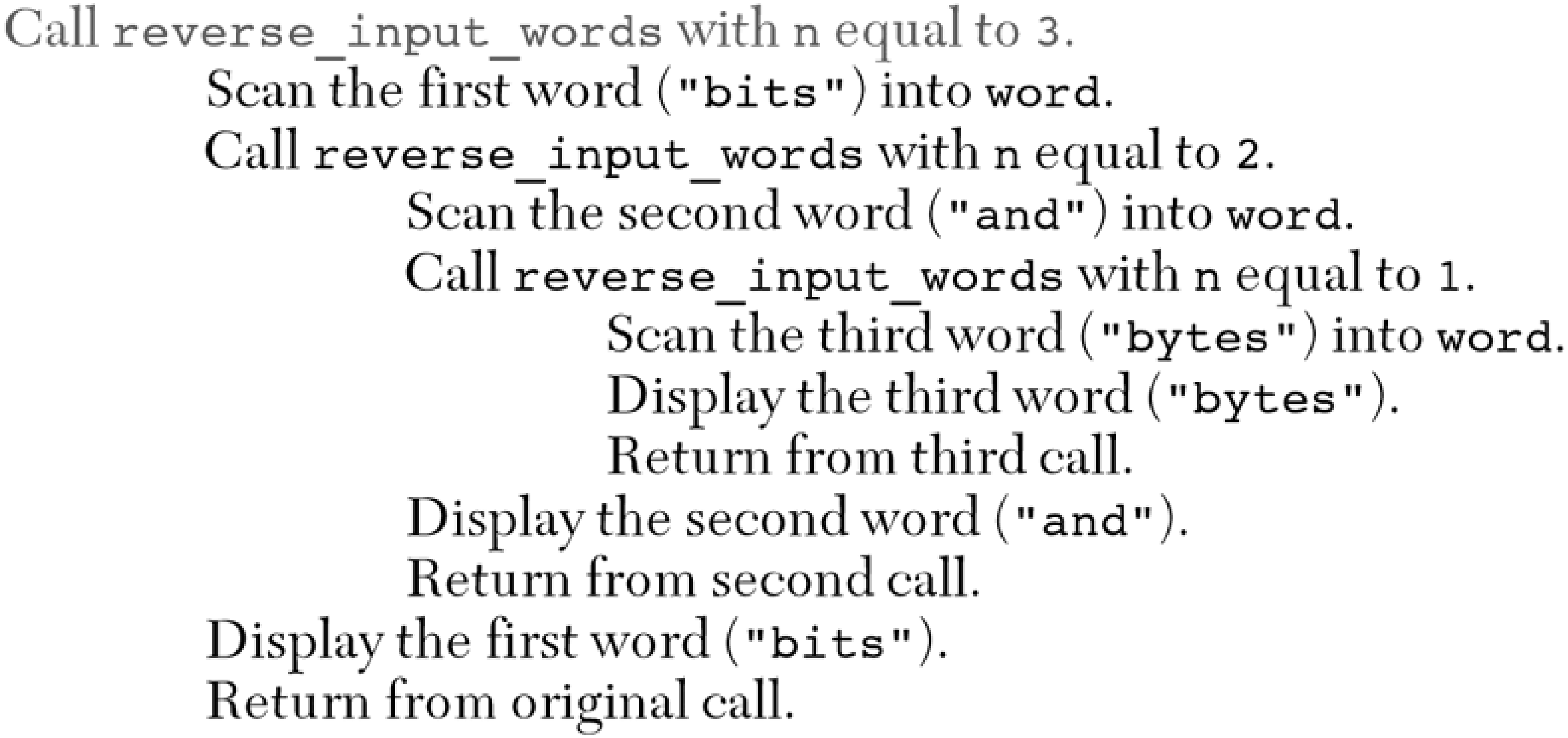
```
reverse_input_words(3)
```

```
n is 3
word is undefined
3 <= 1 is false
scan "bits" into word
reverse_input_words(2)

display "bits"
return
```

```
n is 2
word is undefined
2 <= 1 is false
scan "and" into word
reverse_input_words(1)

display "and"
return
```

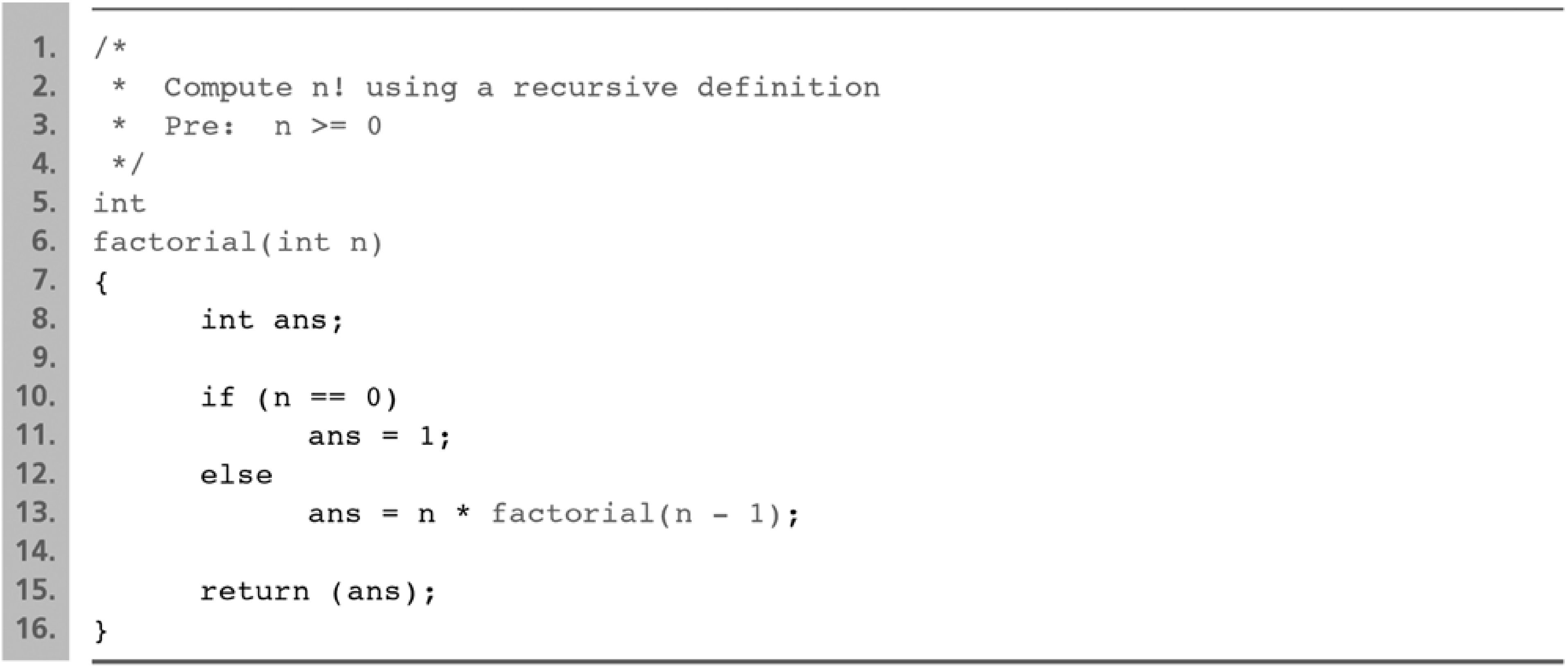
```
n is 1
word is undefined
1 <= 1 is true
scan "bytes" into
     word

display "bytes"
return
```

Sequence of Events for Trace



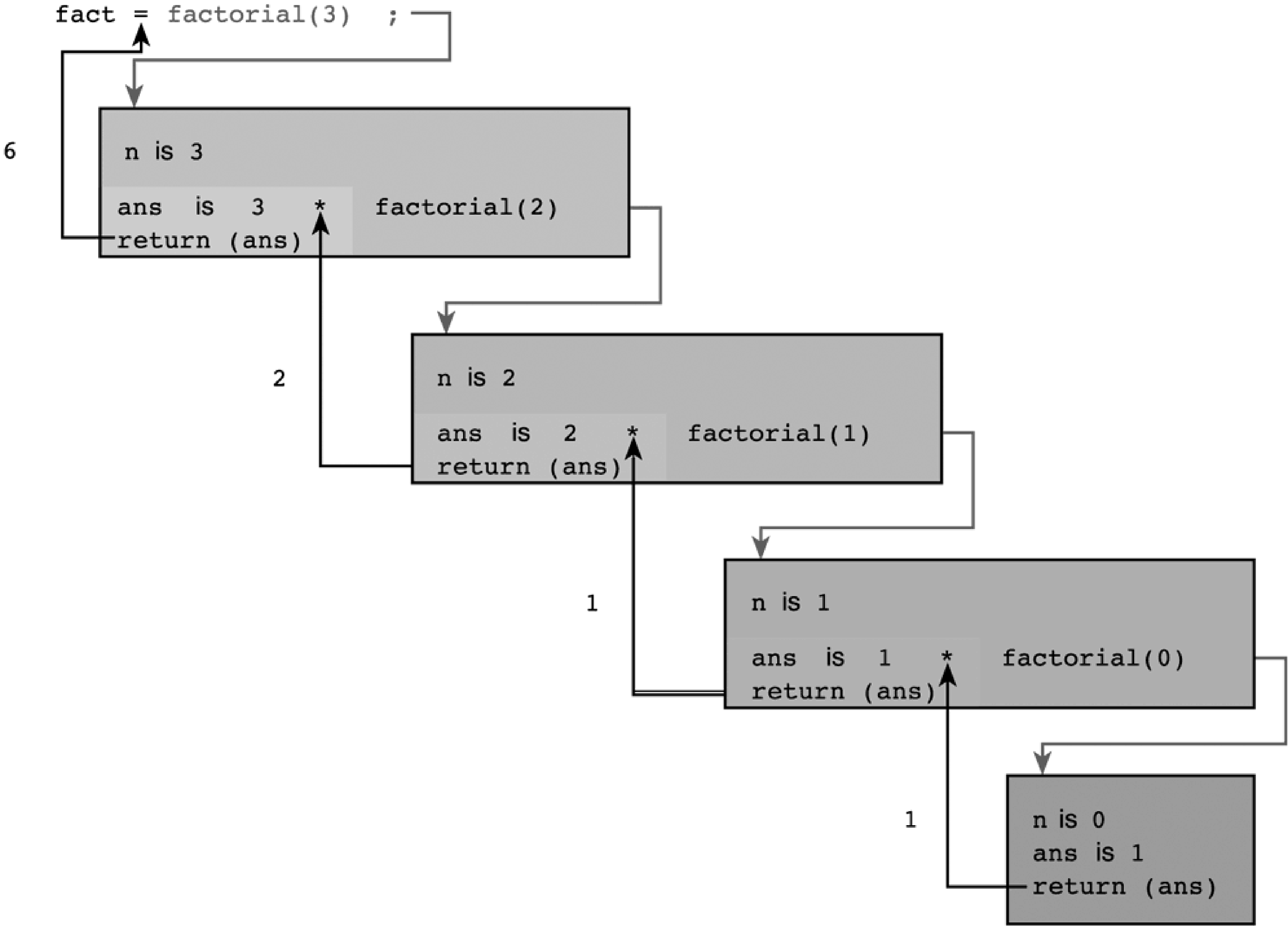
```
Call reverse_input_words with n equal to 3.
    Scan the first word ("bits") into word.
    Call reverse_input_words with n equal to 2.
        Scan the second word ("and") into word.
        Call reverse_input_words with n equal to 1.
            Scan the third word ("bytes") into word.
       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

Recursive factorial Function

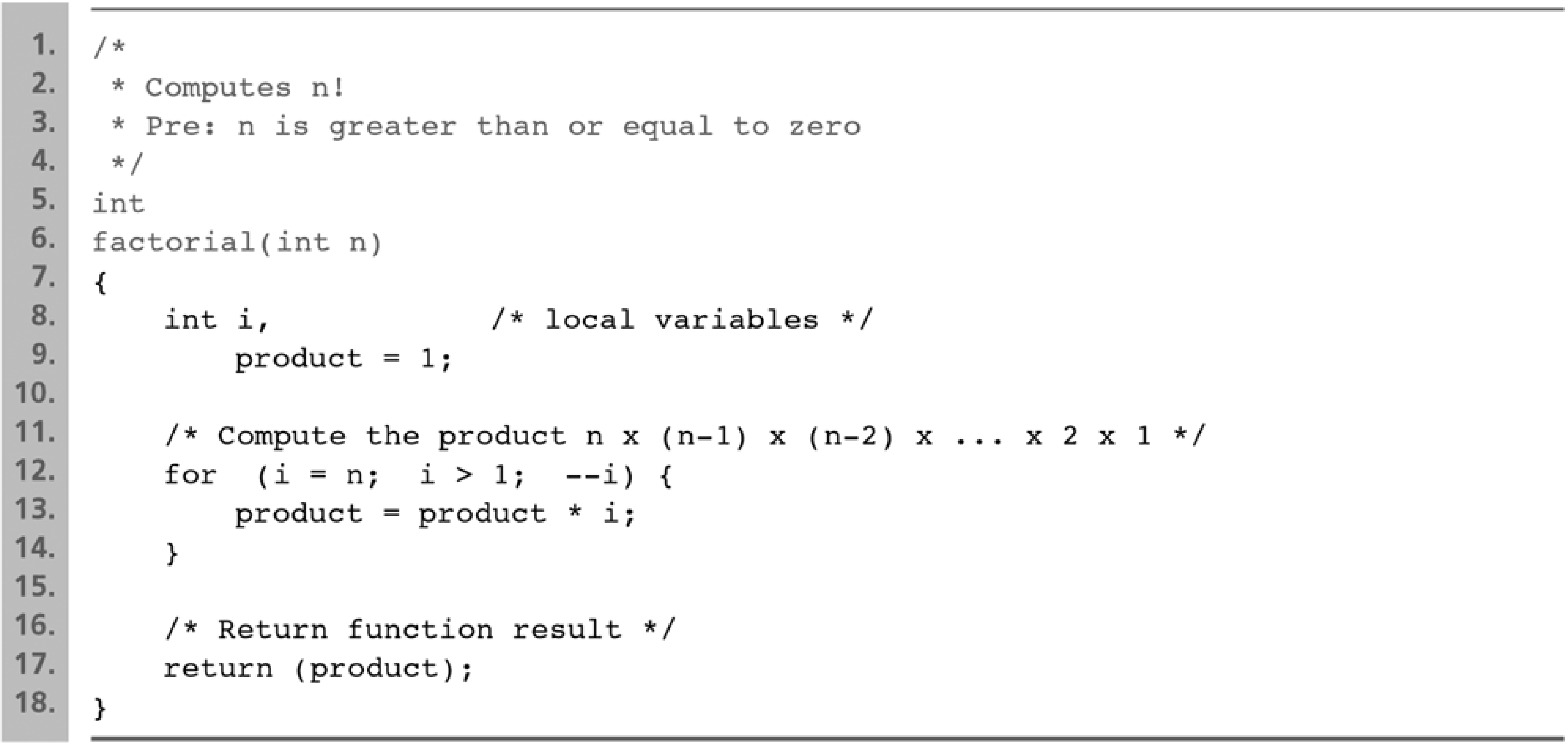


```
/*
 *  Compute n! using a recursive definition
 *  Pre:  n >= 0
 */
int
factorial(int n)
{
      int ans;

      if (n == 0)
            ans = 1;
      else
            ans = n * factorial(n - 1);

      return (ans);
}
```

Trace of fact = factorial(3);

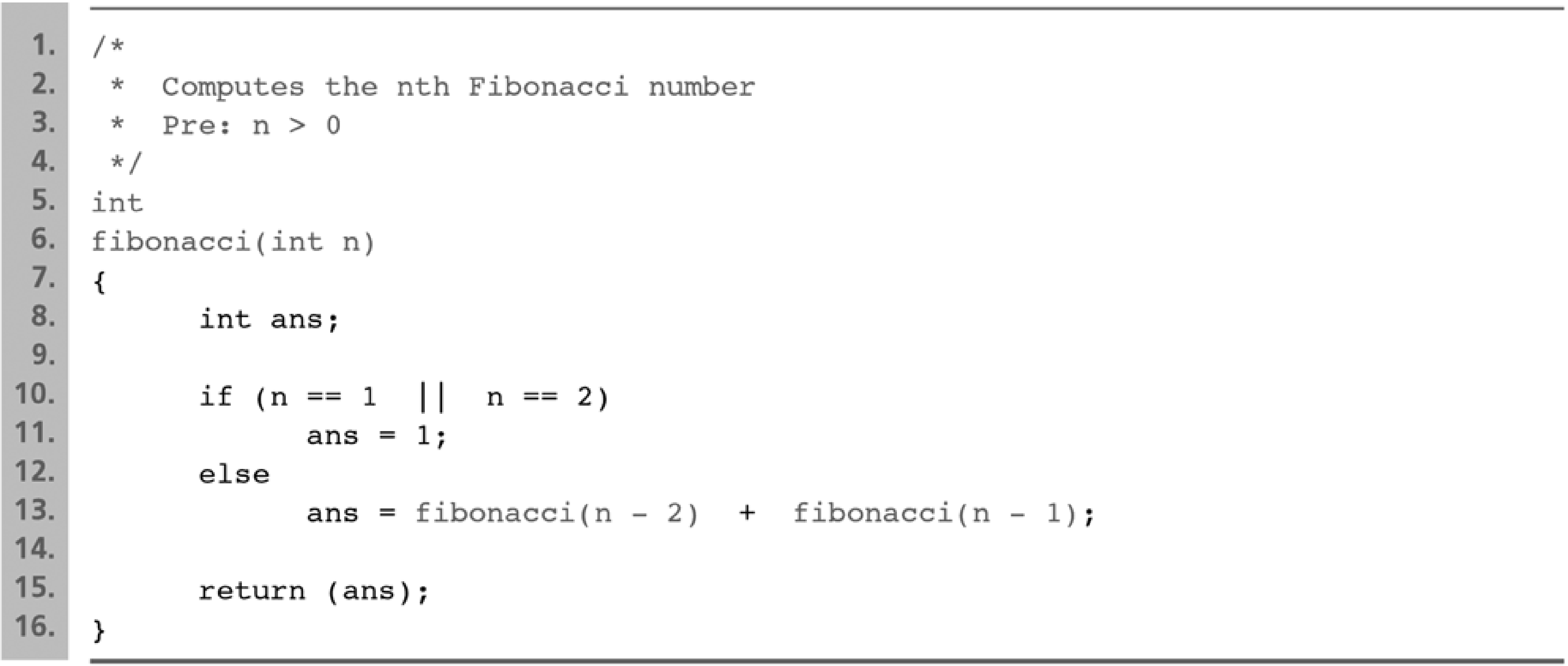


Iterative Function factorial



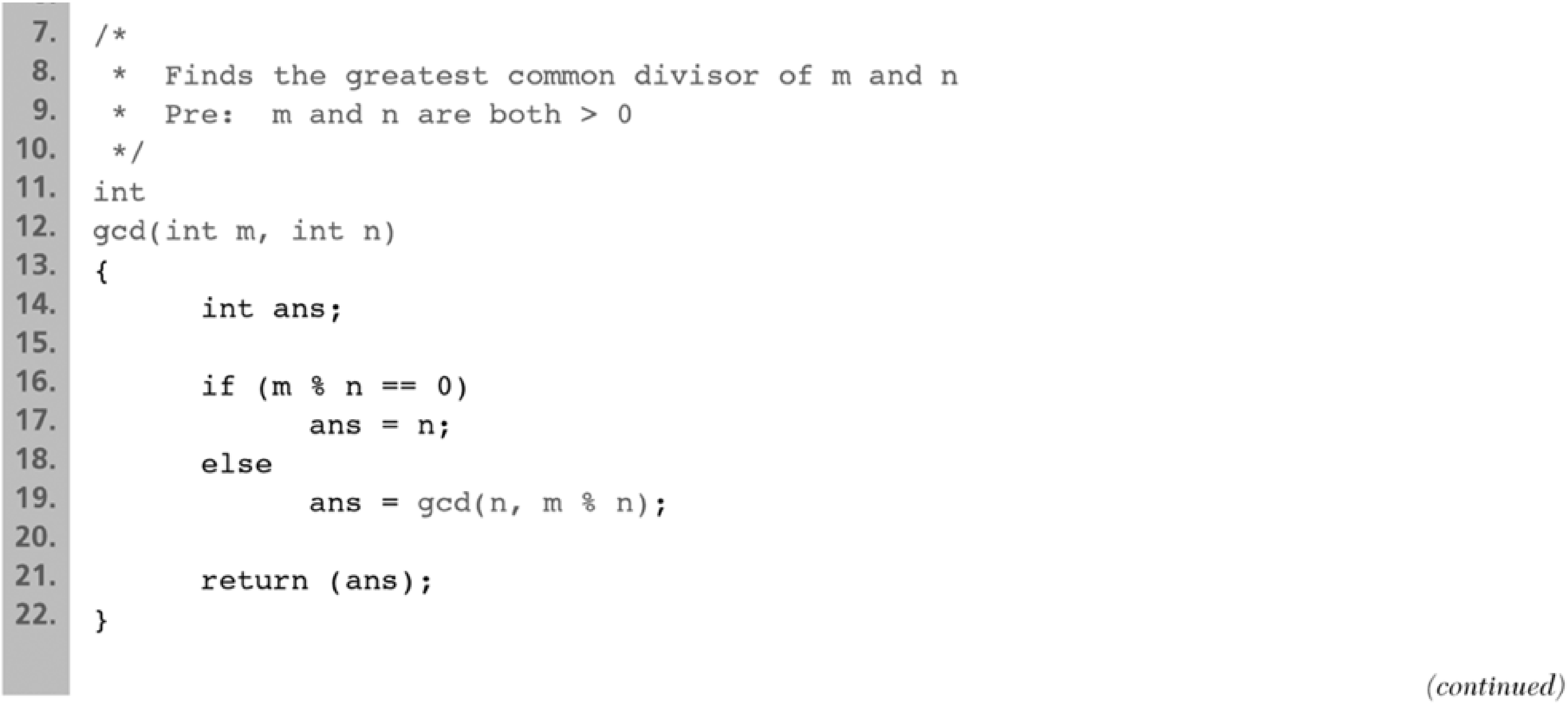
```
/*
 * Computes n!
 * Pre: n is greater than or equal to zero
 */
int
factorial(int n)
{
    int i,            /* local variables */
        product = 1;

    /* Compute the product n x (n-1) x (n-2) x ... x 2 x 1 */
    for  (i = n;  i > 1;  --i) {
        product = product * i;
    }

    /* Return function result */
    return (product);
}
```

Recursive Function fibonacci



```
/*
 *   Computes the nth Fibonacci number
 *   Pre: n > 0
 */
int
fibonacci(int n)
{
      int ans;

      if (n == 1  ||  n == 2)
            ans = 1;
      else
            ans = fibonacci(n - 2)  +  fibonacci(n - 1);

      return (ans);
}
```

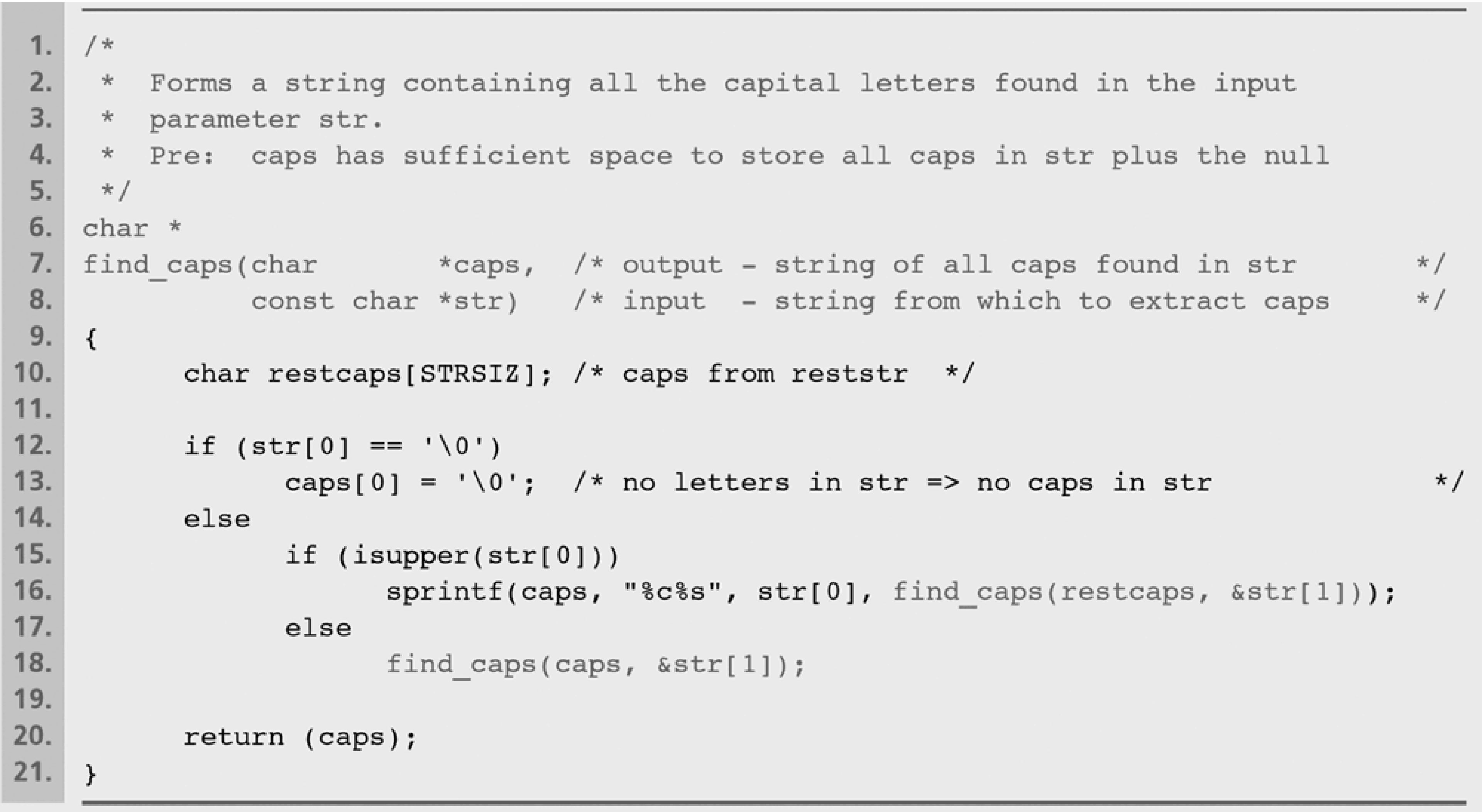
Recursive Function gcd



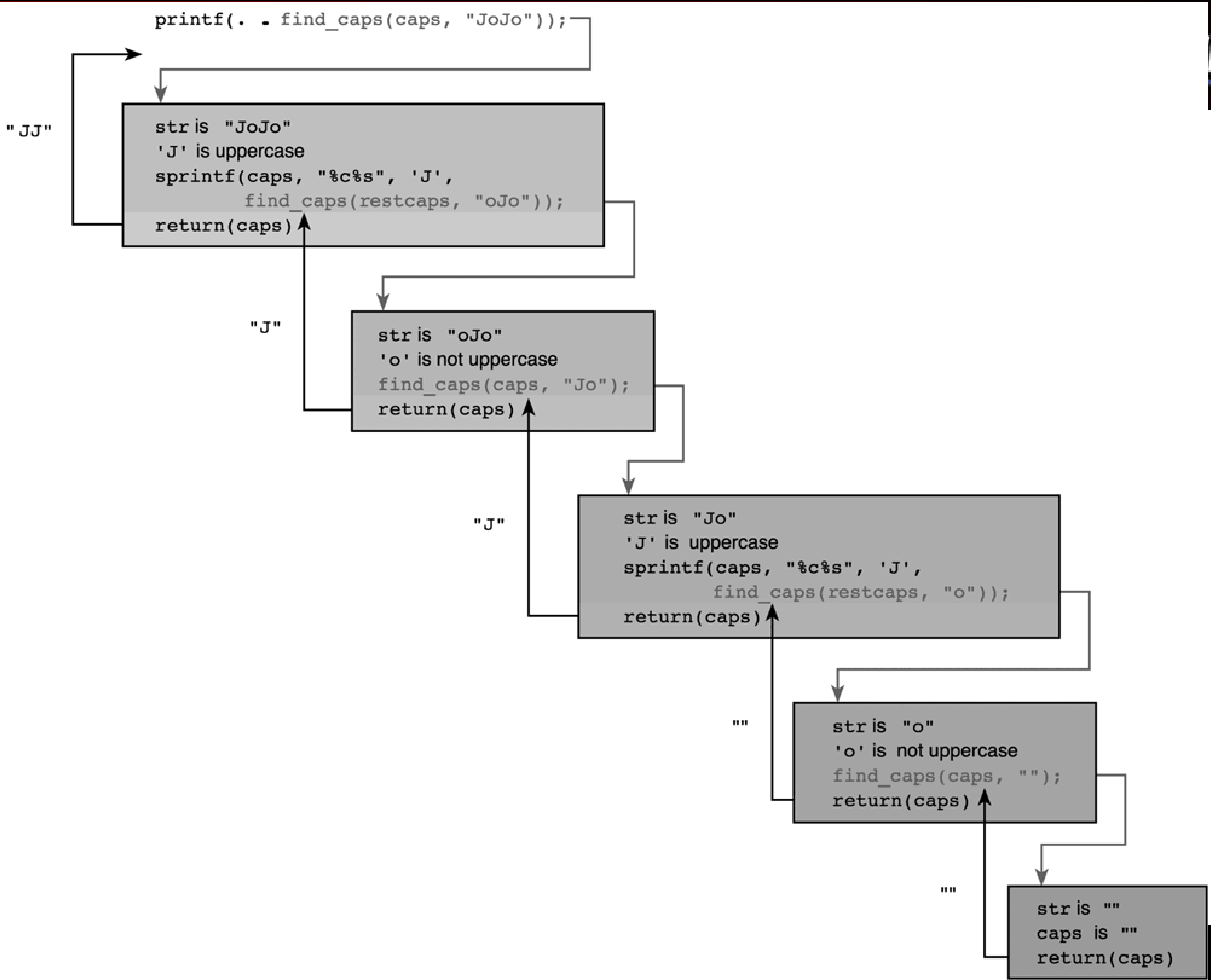
```
/*
 *  Finds the greatest common divisor of m and n
 *  Pre:  m and n are both > 0
 */
int
gcd(int m, int n)
{
      int ans;

      if (m % n == 0)
            ans = n;
      else
            ans = gcd(n, m % n);

      return (ans);
}
```

(continued)

Extract Capital Letters from a String



```
/*
 *  Forms a string containing all the capital letters found in the input
 *  parameter str.
 *  Pre:  caps has sufficient space to store all caps in str plus the null
 */
char *
find_caps(char       *caps,  /* output - string of all caps found in str      */
          const char *str)   /* input  - string from which to extract caps    */
{
      char restcaps[STRSIZ]; /* caps from reststr  */

      if (str[0] == '\0')
            caps[0] = '\0';  /* no letters in str => no caps in str             */
      else
            if (isupper(str[0]))
                  sprintf(caps, "%c%s", str[0], find_caps(restcaps, &str[1]));
            else
                  find_caps(caps, &str[1]);

      return (caps);
}
```



```
printf(. . find_caps(caps, "JoJo"));
```

```
str is  "JoJo"
'J' is uppercase
sprintf(caps, "%c%s", 'J',
         find_caps(restcaps, "oJo"));
return(caps)
```

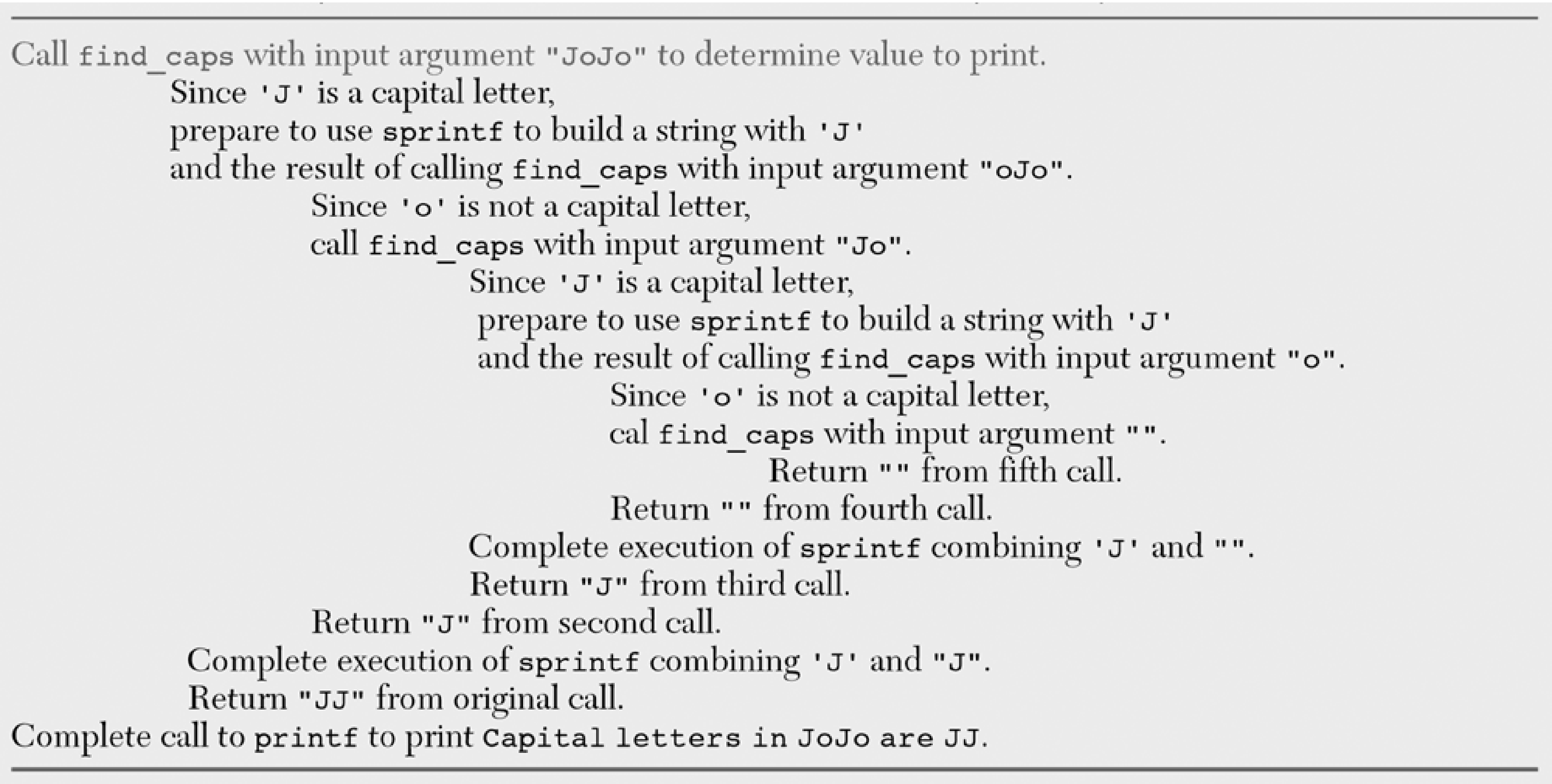
```
str is  "oJo"
'o' is not uppercase
find_caps(caps, "Jo");
return(caps)
```

```
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)
```

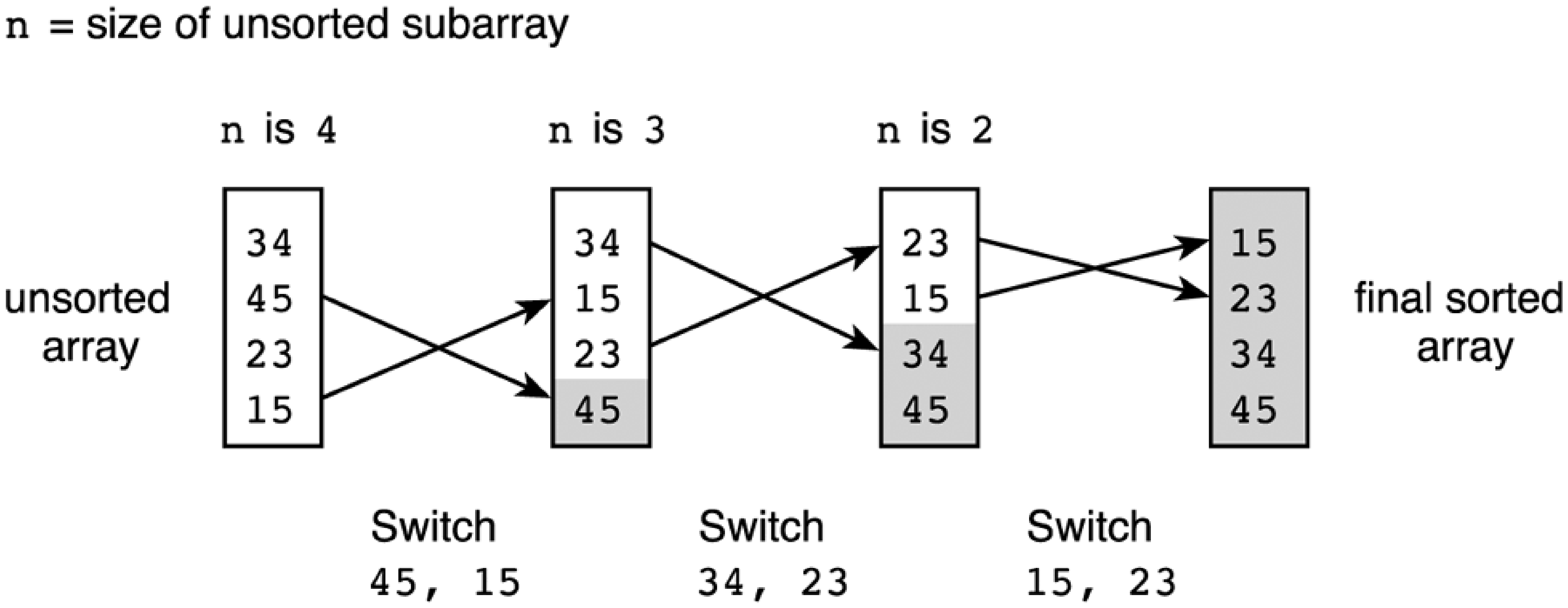
```
str is  ""
caps is  ""
return(caps)
```

Sequence of Events

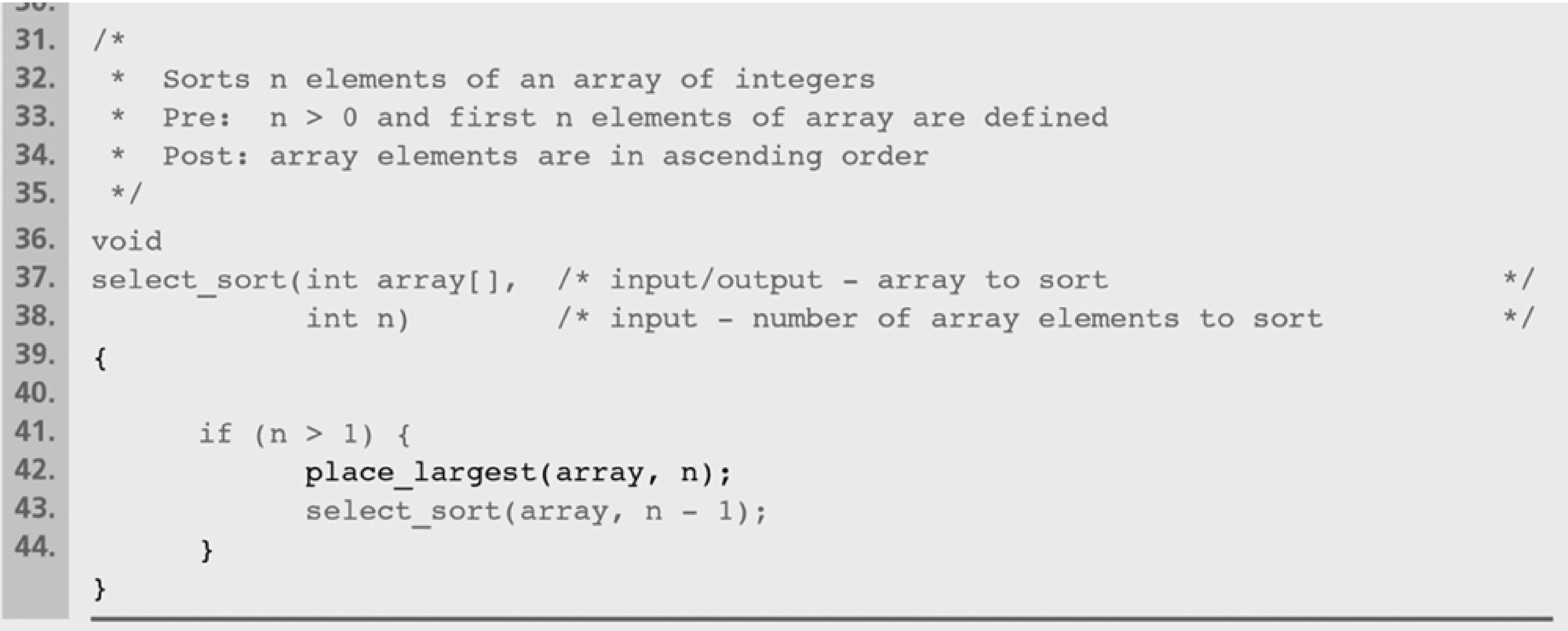


```
Call find_caps with input argument "JoJo" to determine value to print.
        Since 'J' is a capital letter,
        prepare to use sprintf to build a string with 'J'
        and the result of calling find_caps with input argument "oJo".
                Since 'o' is not a capital letter,
                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.
```

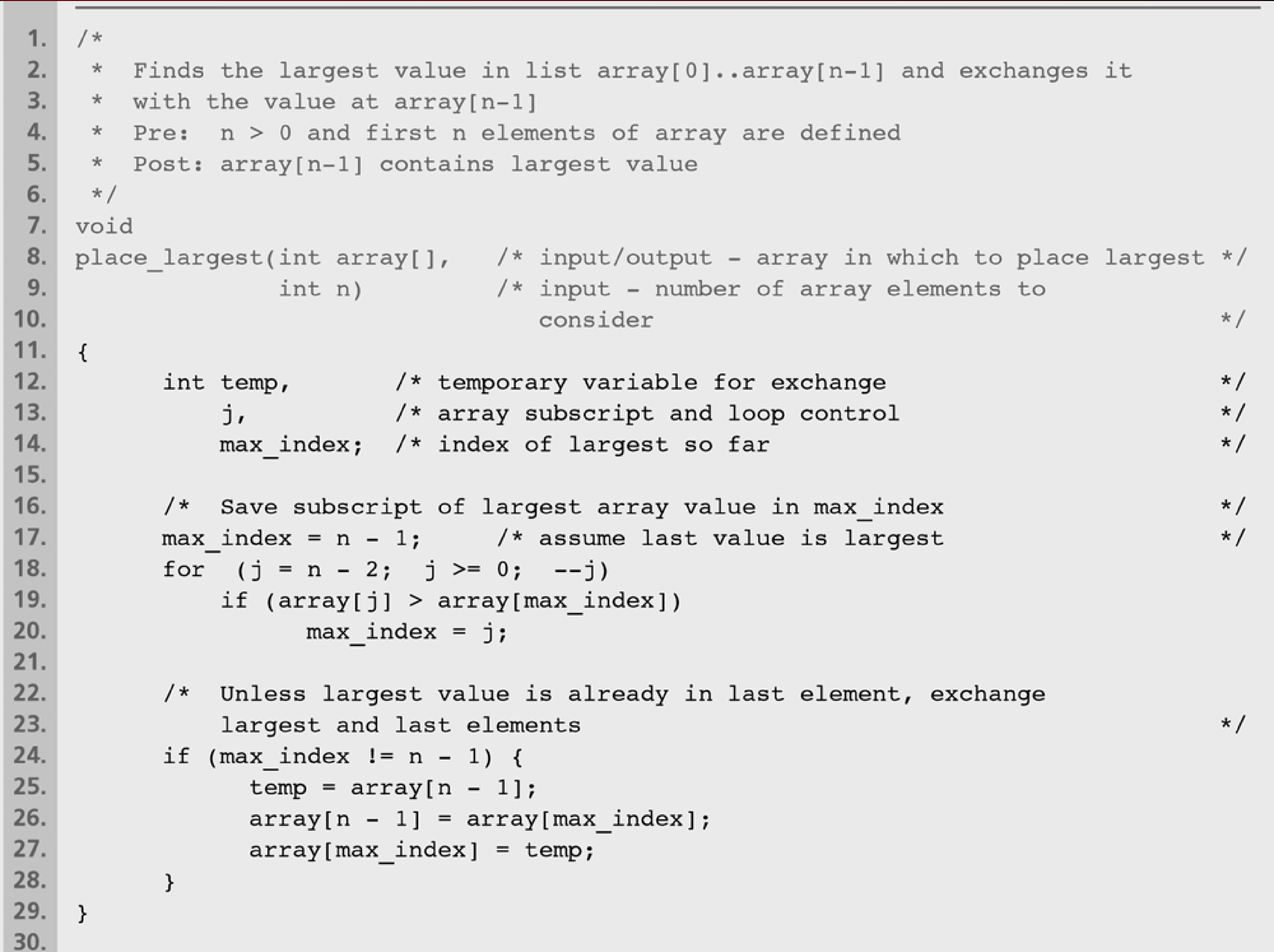
Trace of Selection Sort



Recursive Selection Sort



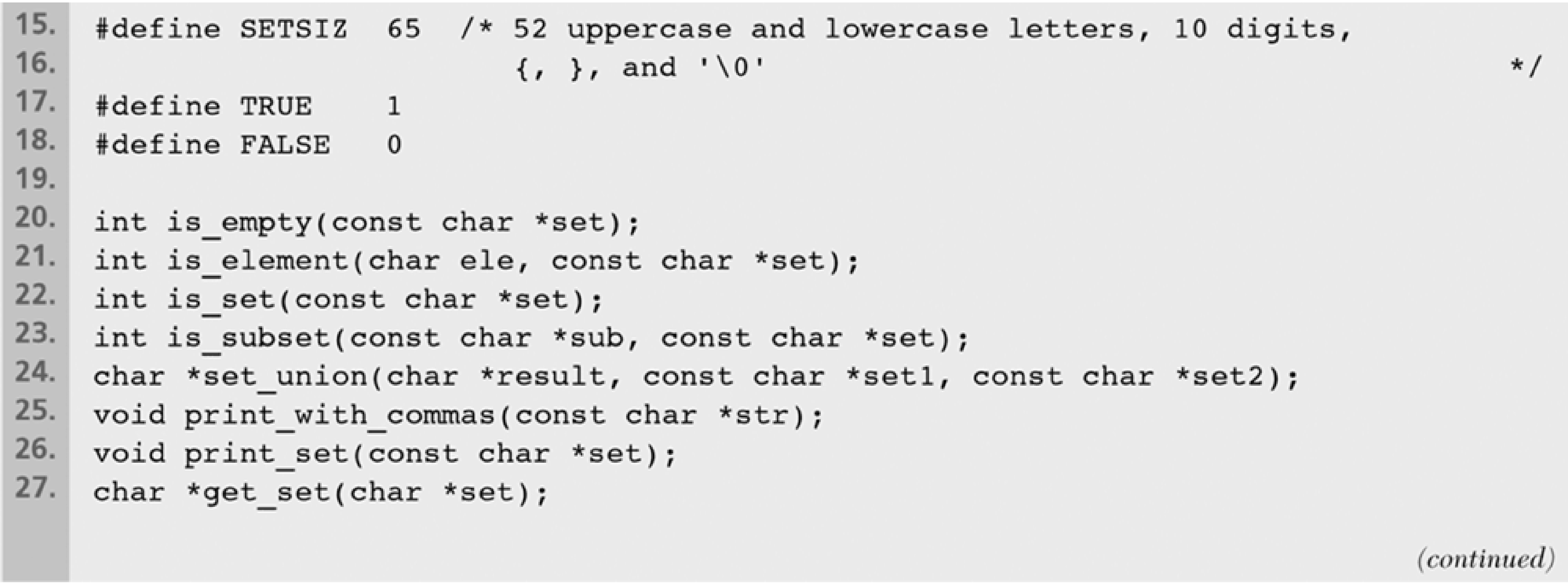
```
/*
 *  Sorts n elements of an array of integers
 *  Pre:  n > 0 and first n elements of array are defined
 *  Post: array elements are in ascending order
 */
void
select_sort(int array[],  /* input/output - array to sort                    */
            int n)        /* input - number of array elements to sort        */
{

      if (n > 1) {
            place_largest(array, n);
            select_sort(array, n - 1);
      }
}
```



```
/*
 *  Finds the largest value in list array[0]..array[n-1] and exchanges it
 *  with the value at array[n-1]
 *  Pre:  n > 0 and first n elements of array are defined
 *  Post: array[n-1] contains largest value
 */
void
place_largest(int array[],   /* input/output - array in which to place largest */
              int n)         /* input - number of array elements to
                                consider                                       */
{
      int temp,       /* temporary variable for exchange                       */
          j,          /* array subscript and loop control                      */
          max_index;  /* index of largest so far                               */

      /*  Save subscript of largest array value in max_index                   */
      max_index = n - 1;     /* assume last value is largest                   */
      for  (j = n - 2;  j >= 0;  --j)
          if (array[j] > array[max_index])
                max_index = j;

      /*  Unless largest value is already in last element, exchange
          largest and last elements                                            */
      if (max_index != n - 1) {
            temp = array[n - 1];
            array[n - 1] = array[max_index];
            array[max_index] = temp;
      }
}
```

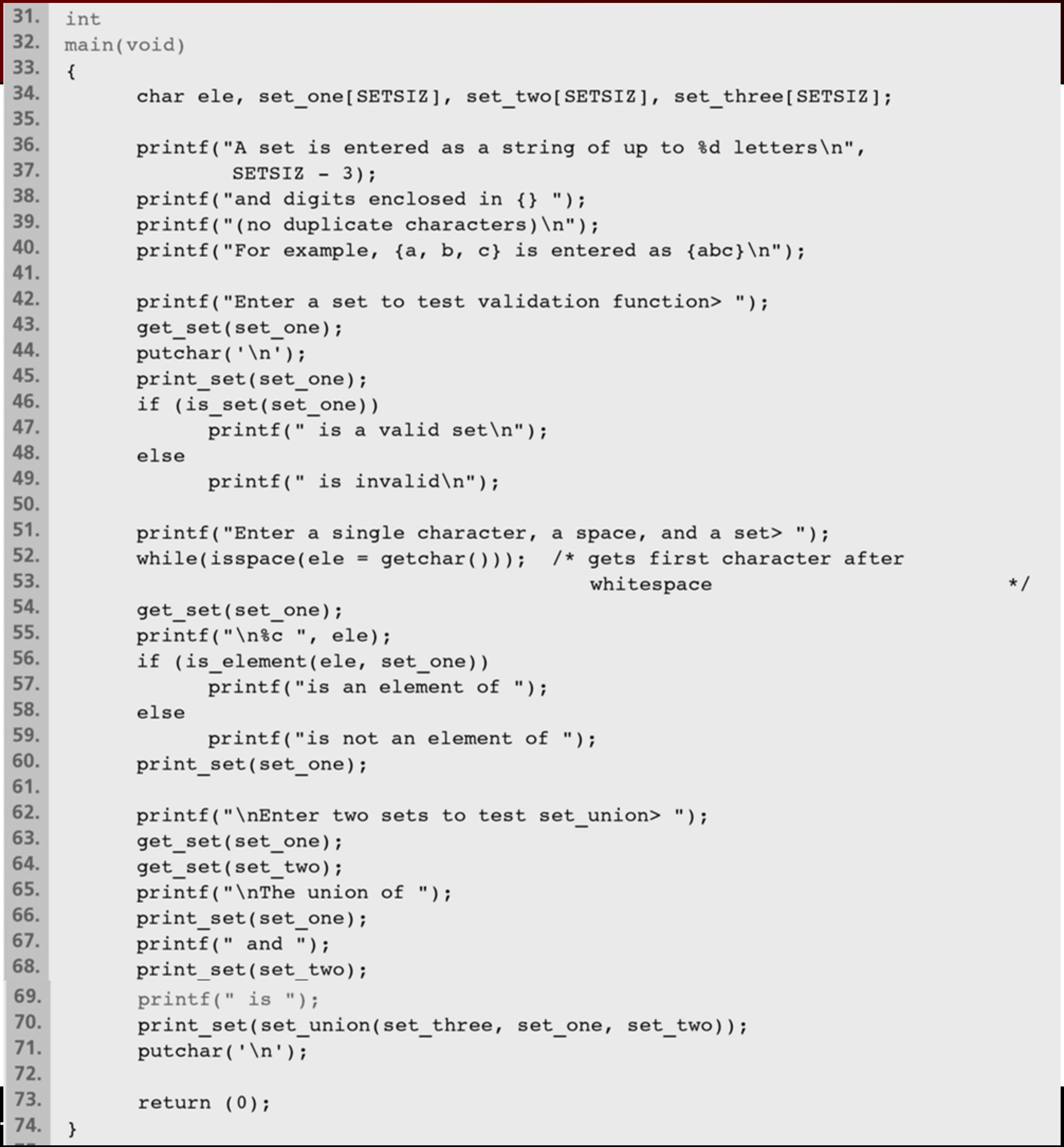
Case Study: Recursive Set Operations

- Sets represented as character strings

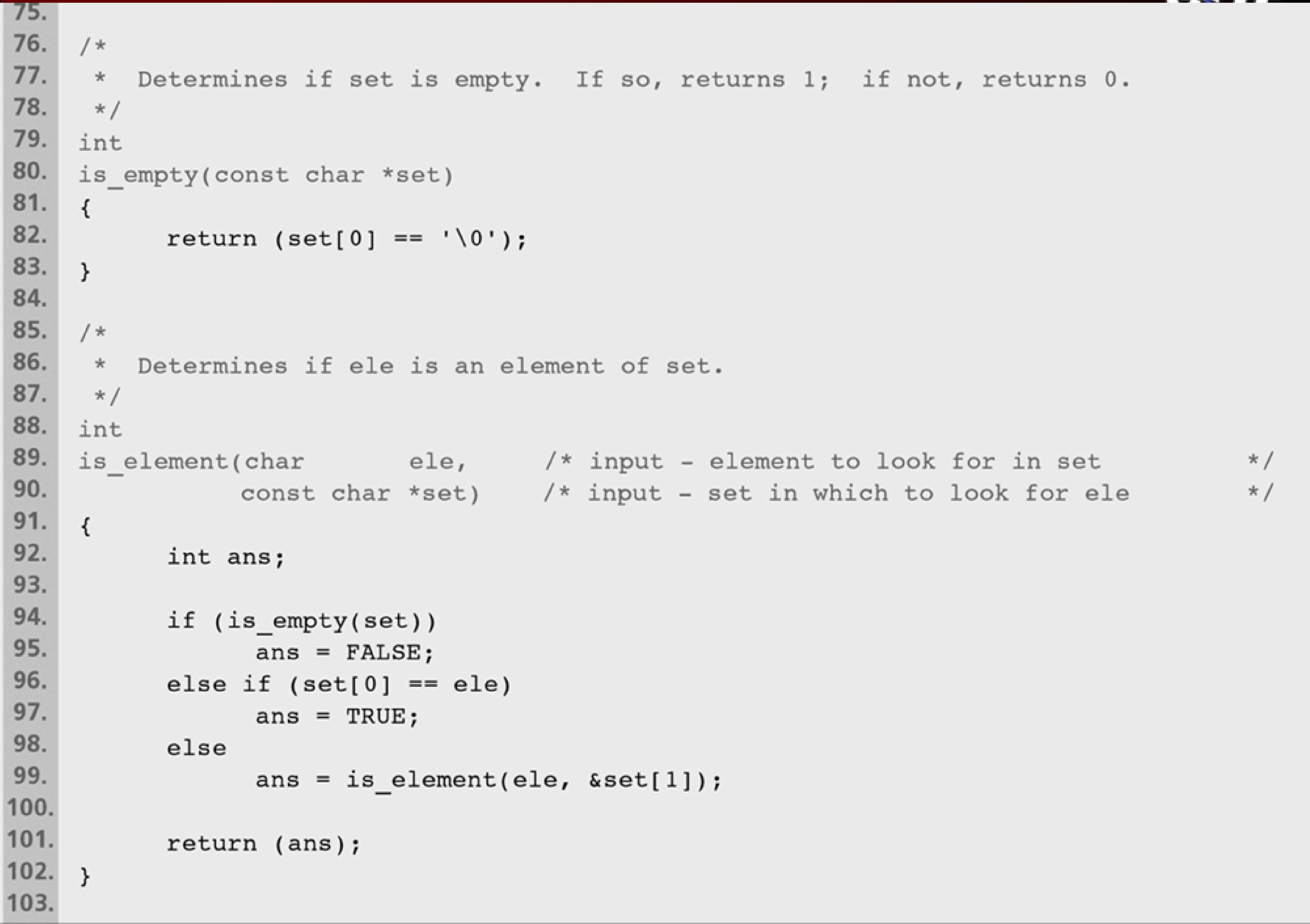


```
#define SETSIZ  65  /* 52 uppercase and lowercase letters, 10 digits,
                       {, }, and '\0'                                         */
#define TRUE    1
#define FALSE   0

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);
```

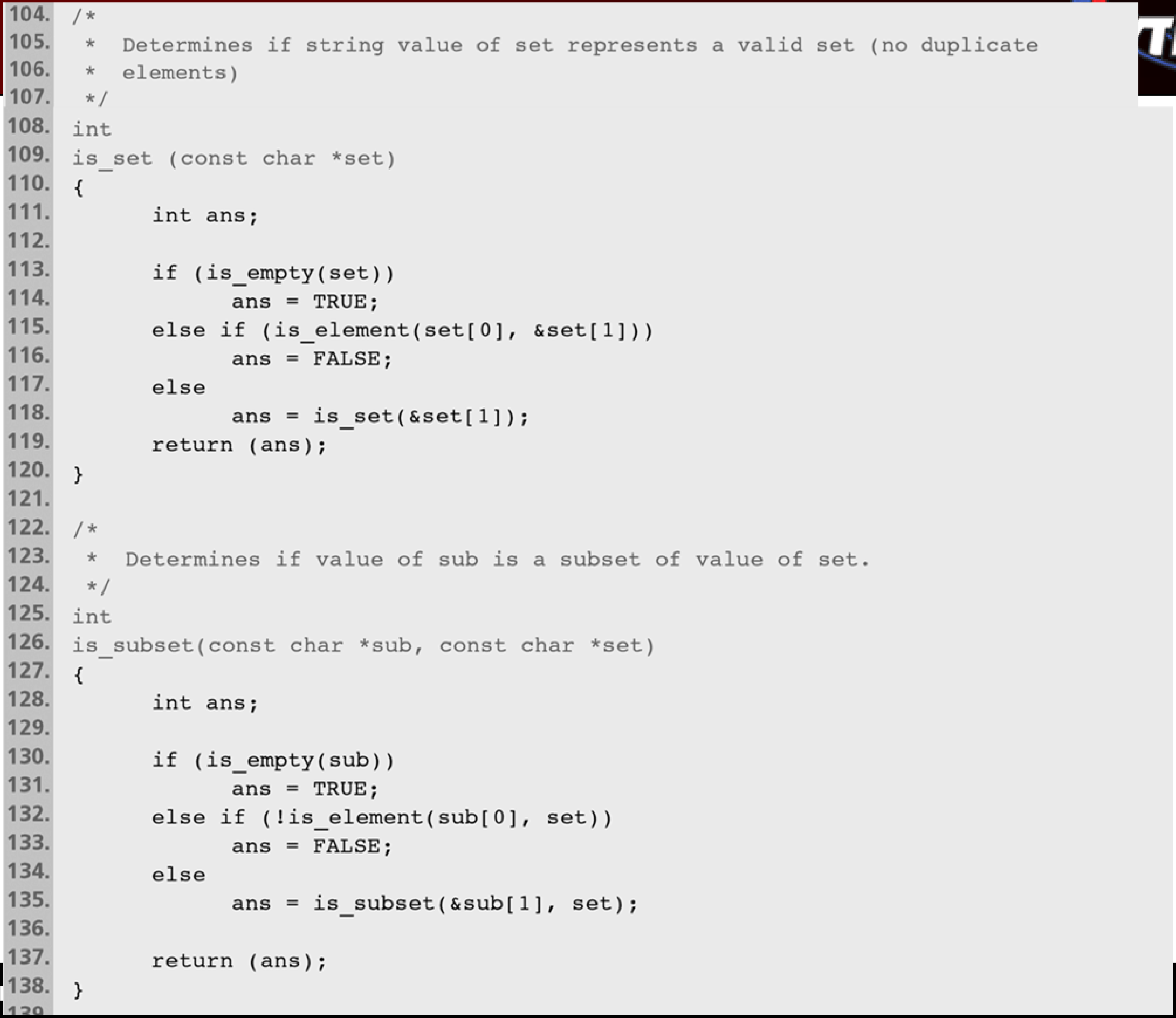
(continued)



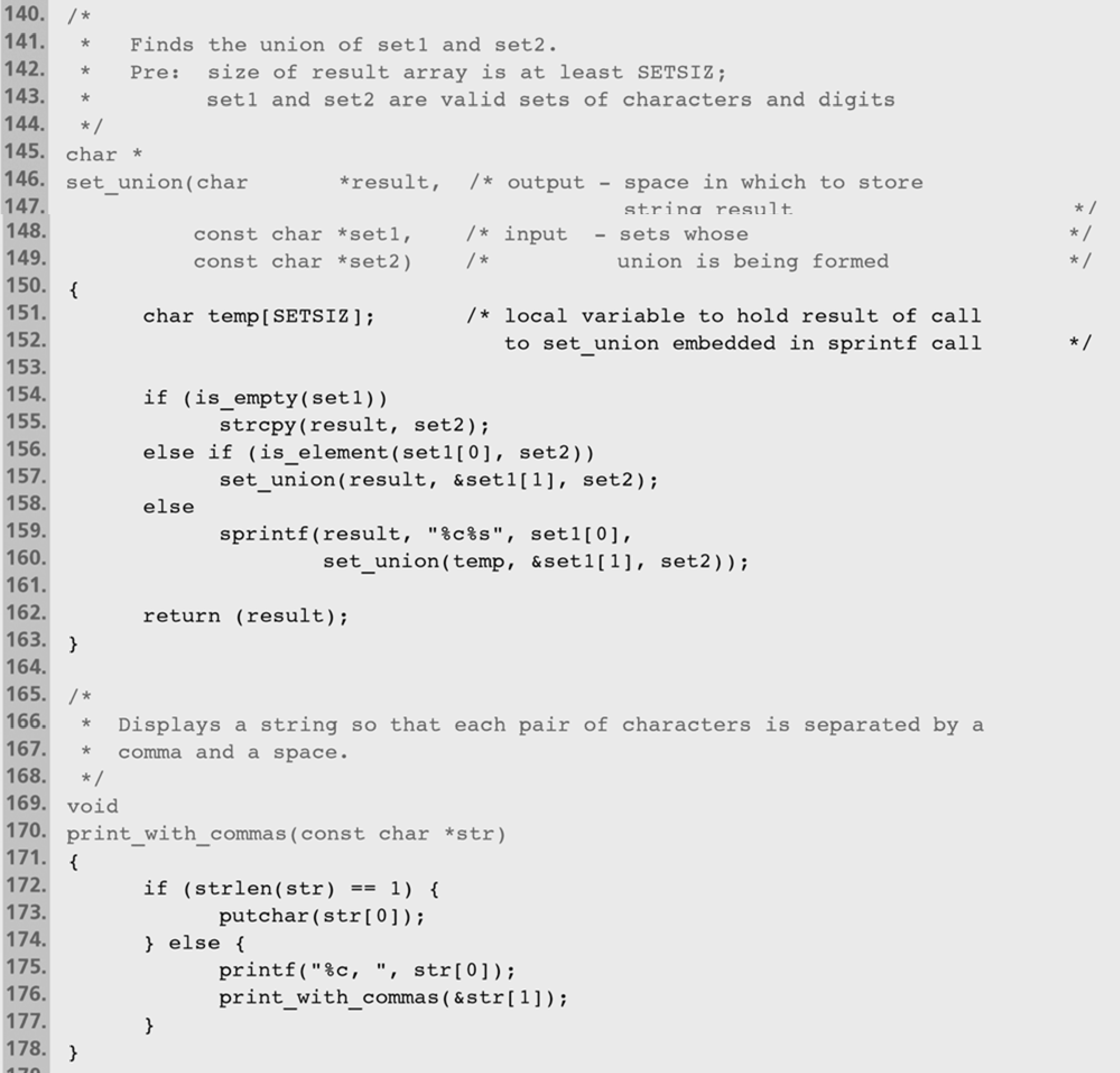
```c
int
main(void)
{
        char ele, set_one[SETSIZ], set_two[SETSIZ], set_three[SETSIZ];

        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);
}
```



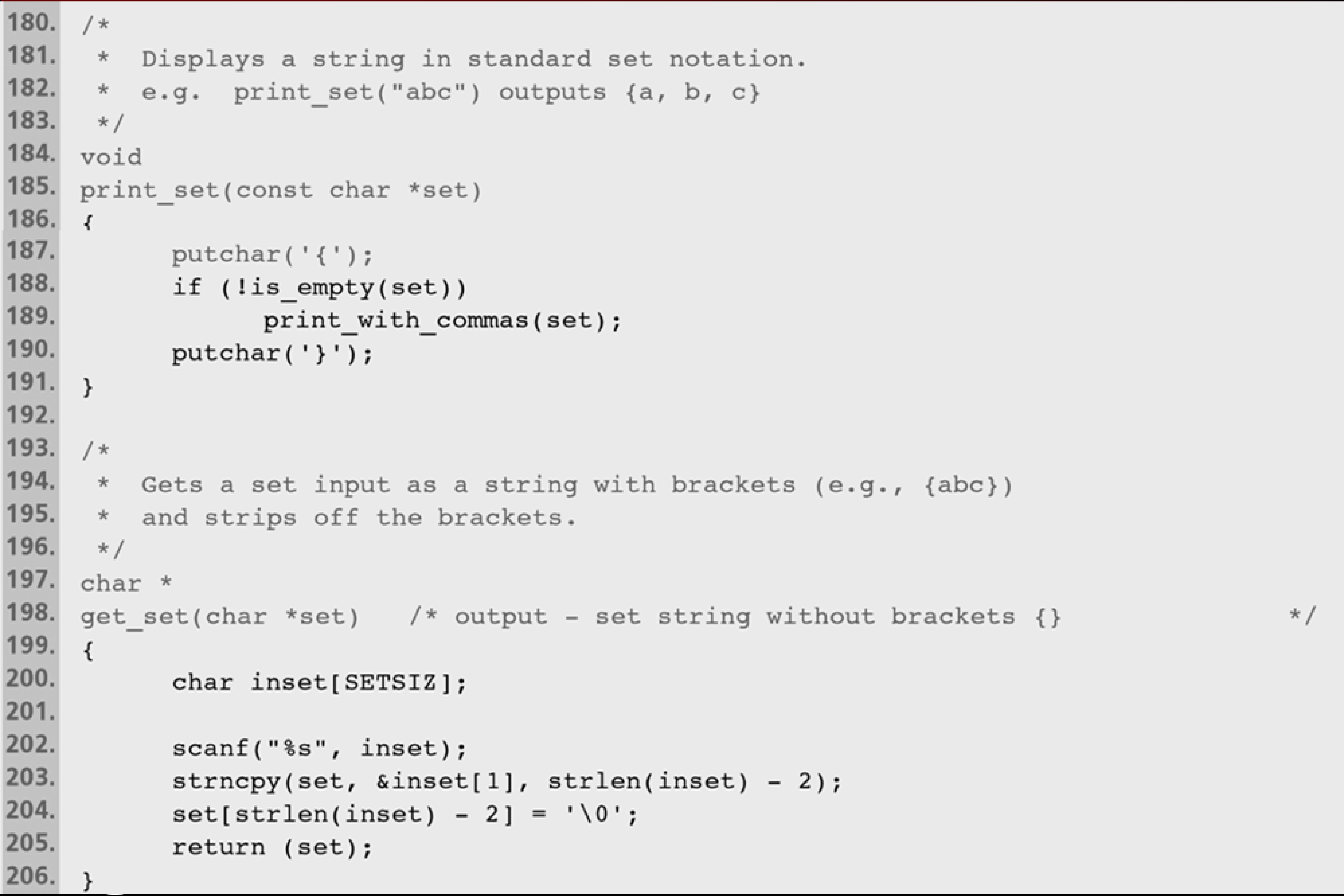
```c

/*
 *  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);
}

```



```c
/*
 *   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);
}
```

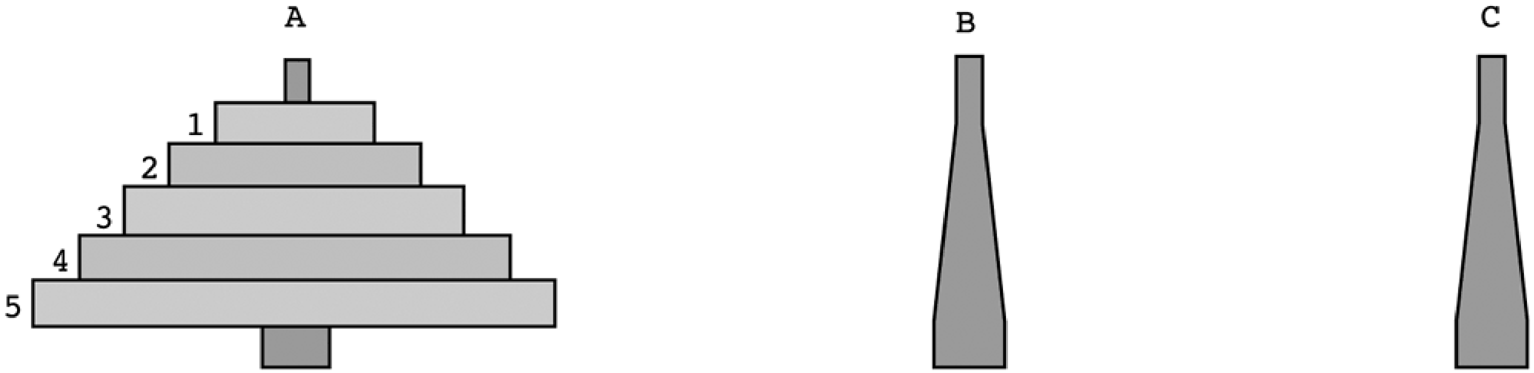


```
/*
 *   Finds the union of set1 and set2.
 *   Pre:  size of result array is at least SETSIZ;
 *         set1 and set2 are valid sets of characters and digits
 */
char *
set_union(char       *result,  /* output - space in which to store
                                          string result                */
          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]);
      }
}
```

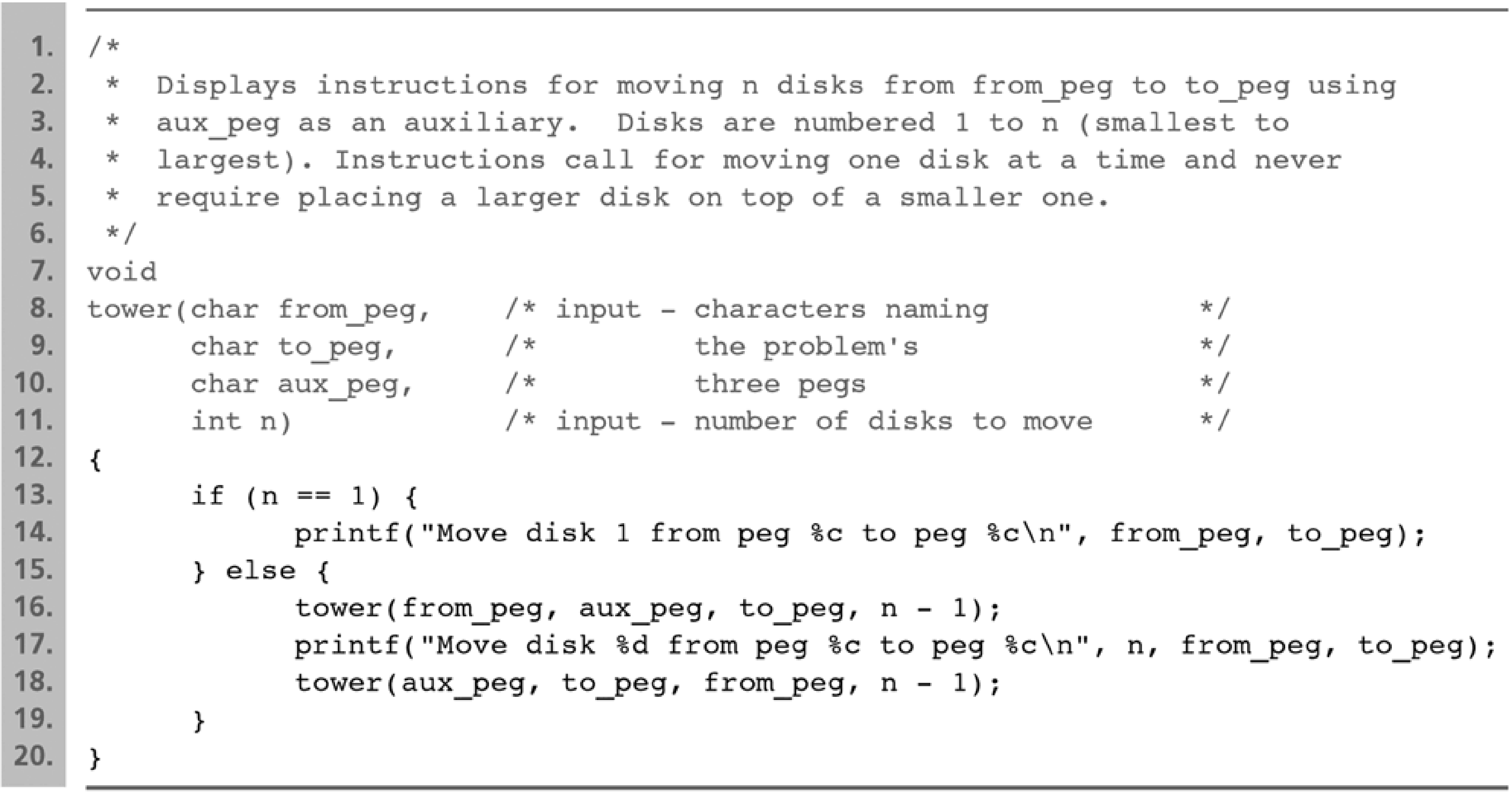


```
/*
 *  Displays a string in standard set notation.
 *  e.g.  print_set("abc") outputs {a, b, c}
 */
void
print_set(const char *set)
{
        putchar('{');
        if (!is_empty(set))
                print_with_commas(set);
        putchar('}');
}

/*
 *  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);
}
```

Towers of Hanoi

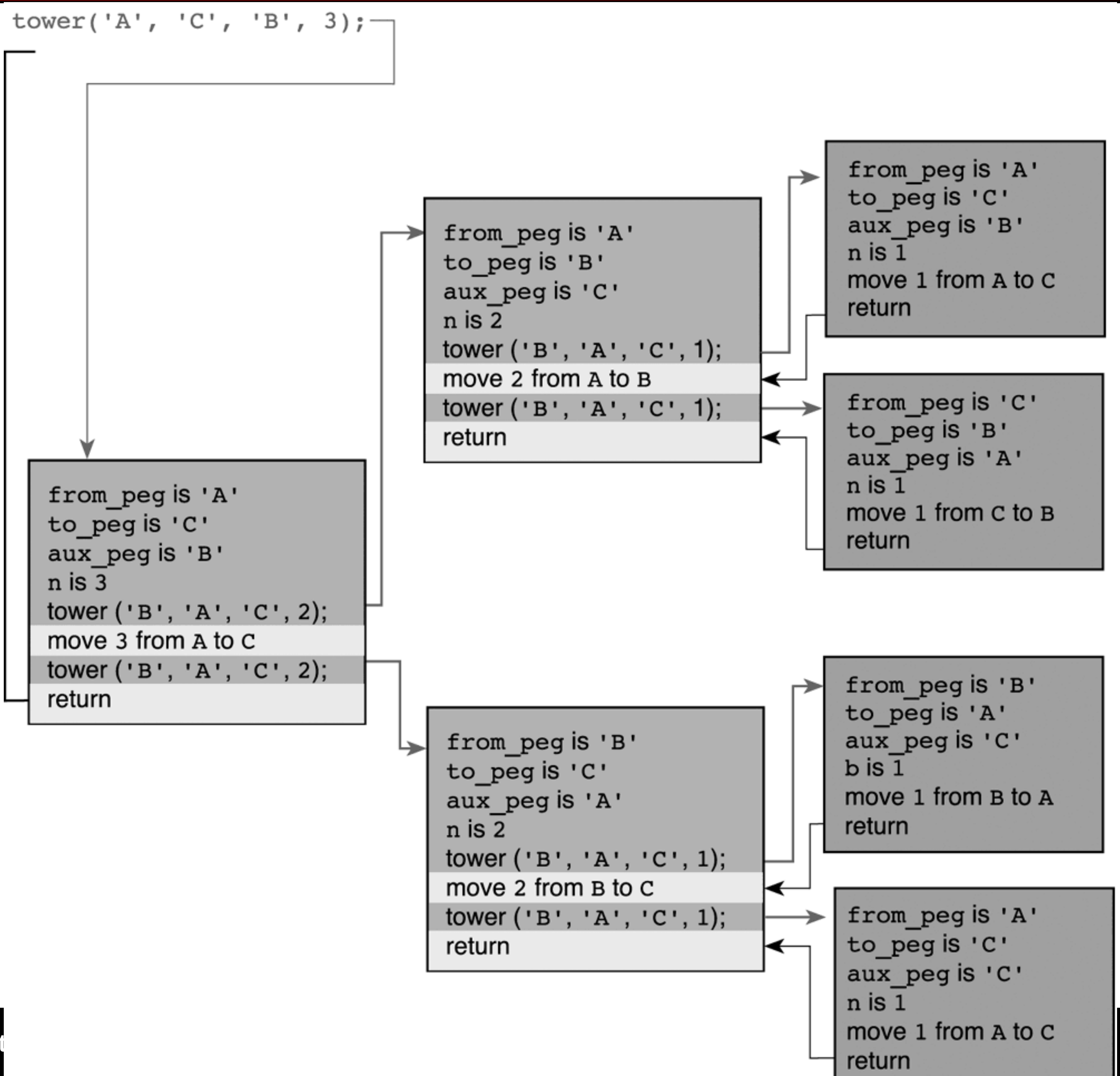


Recursive Function tower



```
/*
 *  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);



```
tower('A', 'C', 'B', 3);
```

from_peg is 'A'
to_peg is 'C'
aux_peg is 'B'
n is 3
tower ('B', 'A', 'C', 2);
move 3 from A to C
tower ('B', 'A', 'C', 2);
return

from_peg is 'A'
to_peg is 'B'
aux_peg is 'C'
n is 2
tower ('B', 'A', 'C', 1);
move 2 from A to B
tower ('B', 'A', 'C', 1);
return

from_peg is 'A'
to_peg is 'C'
aux_peg is 'B'
n is 1
move 1 from A to C
return

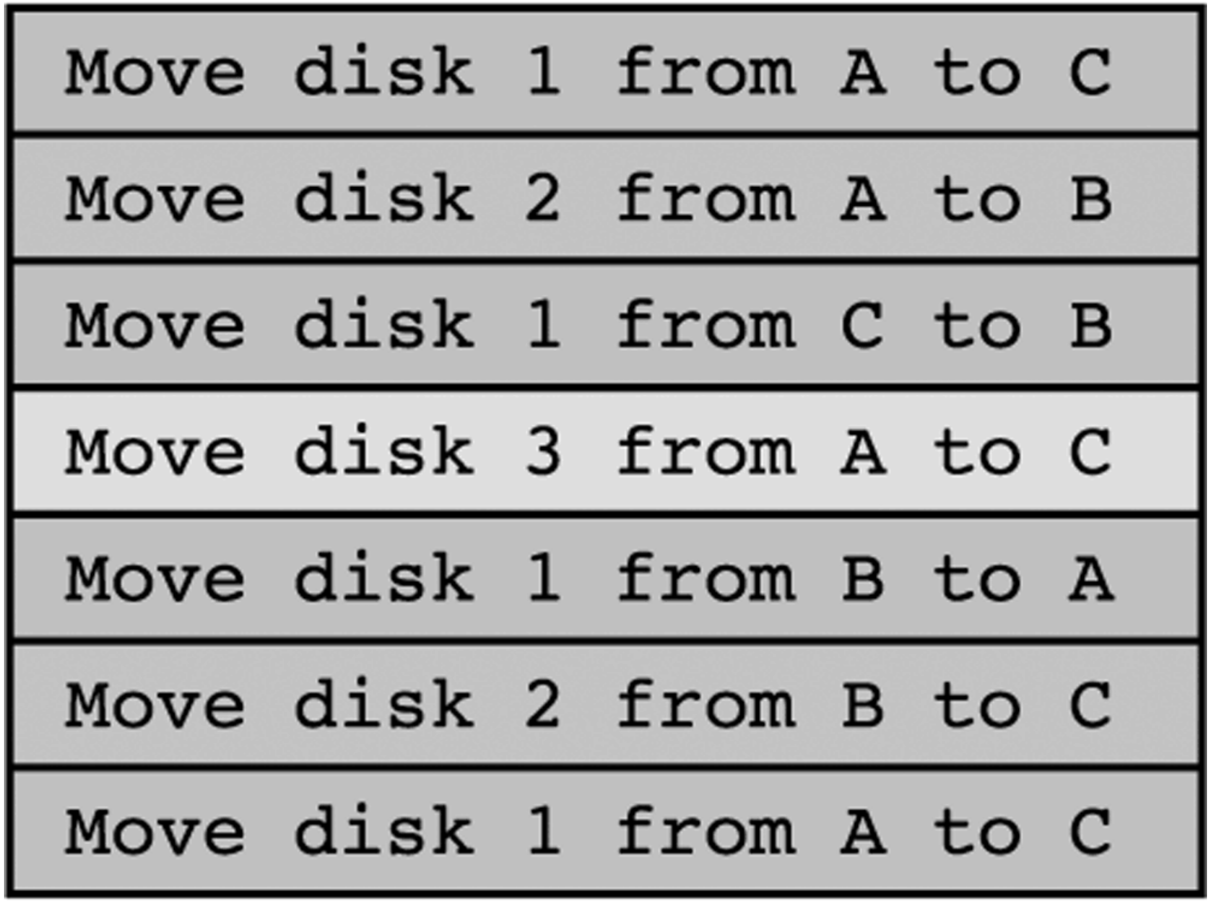
from_peg is 'C'
to_peg is 'B'
aux_peg is 'A'
n is 1
move 1 from C to B
return

from_peg is 'B'
to_peg is 'C'
aux_peg is 'A'
n is 2
tower ('B', 'A', 'C', 1);
move 2 from B to C
tower ('B', 'A', 'C', 1);
return

from_peg is 'B'
to_peg is 'A'
aux_peg is 'C'
b is 1
move 1 from B to A
return

from_peg is 'A'
to_peg is 'C'
aux_peg is 'C'
n is 1
move 1 from A to C
return

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
```