

Figure 10.1 Splitting a Problem into Smaller Problems

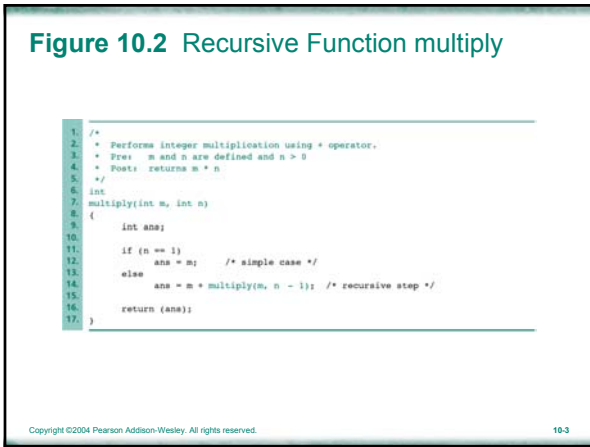


Figure 10.3 Thought Process of Recursive Algorithm Developer

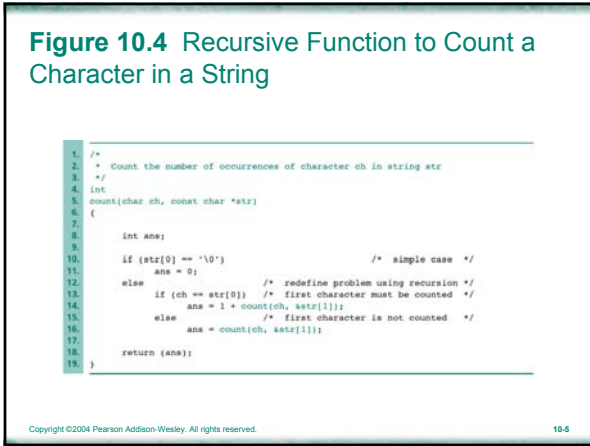
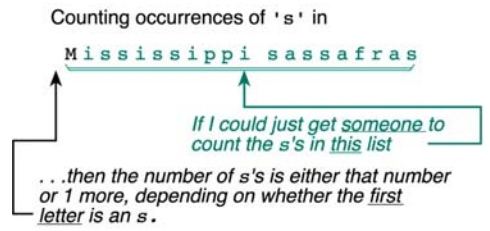


Figure 10.5 Trace of Function multiply

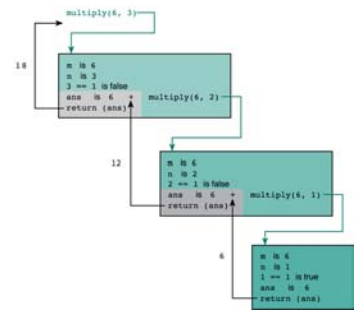


Figure 10.6 Function reverse_input_words

```

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 }

```

Figure 10.7 Trace of reverse_input_words(3) When the Words Entered are "bits" "and" "bytes"

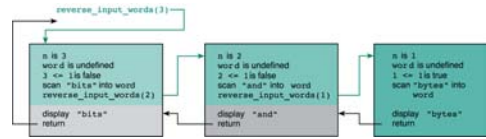


Figure 10.8 Sequence of Events for Trace of reverse_input_words(3)

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.

Figure 10.9 Recursive Function multiply with Print Statements to Create Trace and Output from multiply(8, 3)

```

1  /*
2  * *** Includes calls to print to trace execution ***
3  * Performs integer multiplication using * operator.
4  * Pre: m and n are defined and n >= 0
5  * Post: returns m * n
6  */
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 */

```

(continued)

Figure 10.9 Recursive Function multiply with Print Statements to Create Trace and Output from multiply(8, 3) (cont'd)

```

18     printf("multiply(%d, %d) returning %d\n", m, n, ans);
19     return (ans);
20 }
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

```

Figure 10.10 Recursive factorial Function

```

1  /*
2  * Computes 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 }

```

Figure 10.11 Trace of fact = factorial(3);

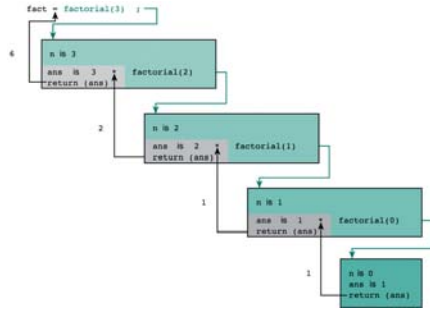


Figure 10.12 Iterative Function factorial

```

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. }
  
```

Figure 10.13 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. }
  
```

Figure 10.14 Program Using Recursive Function gcd

```

1. /*
2.  * Displays the greatest common divisor of two integers
3.  */
4.
5. #include <stdio.h>
6.
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)

Figure 10.14 Program Using Recursive Function gcd (cont'd)

```

23. int
24. main(void)
25. {
26.     int n1, n2;
27.
28.     printf("Enter two positive integers separated by a space: ");
29.     scanf("%d%d", &n1, &n2);
30.     printf("Their greatest common divisor is %d\n", gcd(n1, n2));
31.
32.     return (0);
33. }
34.
35. Enter two positive integers separated by a space> 24 84
36. Their greatest common divisor is 12
  
```

Figure 10.15 Recursive Function to 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[STRLEN]; /* 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%ks", str[0], find_caps(restcaps, &str[1]));
17.         else
18.             find_caps(caps, &str[1]);
19.
20.     return (caps);
21. }
  
```

Figure 10.16 Trace of Call to Recursive Function find_caps

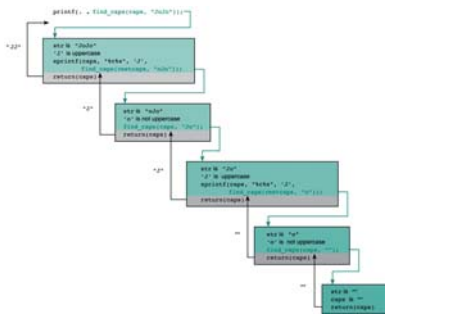


Figure 10.17 Sequence of Events for Trace of Call to find_caps from printf Statements

```

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,
call find_caps with input argument "".
Return "" from fifth call.
Return "o" from fourth call.
Complete execution of sprintf combining 'J' and "".
Return "Jo" from third call.
Return "Jo" from second call.
Complete execution of sprintf combining 'J' and "Jo".
Return "JoJo" from original call.
Complete call to printf to print capital letters in JoJo are JoJo.
    
```

Figure 10.18 Trace of Selection Sort

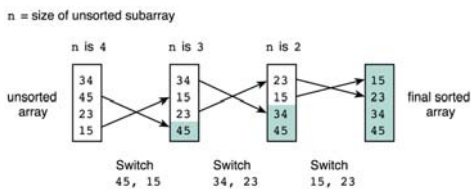


Figure 10.19 Recursive Selection Sort

```

1  /*
2  * Finds the largest value in list array[]..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 consider */
10
11 {
12     int temp; /* temporary variable for exchange */
13     int max_index; /* array subscript and loop control */
14     /* Find subscript of largest array value in max_index */
15     /* Save subscript of largest array value in max_index */
16     max_index = n - 1; /* assume last value is largest */
17     for (j = n - 2; j >= 0; --j)
18         if (array[j] > array[max_index])
19             max_index = j;
20
21     /* Unless largest value is already in last element, exchange
22     largest and last elements */
23     if (max_index != n - 1) {
24         temp = array[n - 1];
25         array[n - 1] = array[max_index];
26         array[max_index] = temp;
27     }
28 }
29
30 /*
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 */
    
```

Figure 10.19 Recursive Selection Sort (cont'd)

```

36. void
37. select_sort(int array[], /* input/output - array to sort */
38.             int n) /* input - number of array elements to sort */
39. {
40.     if (n > 1) {
41         place_largest(array, n);
42         select_sort(array, n - 1);
43     }
44 }
    
```

Figure 10.20 Recursive Set Operations on Sets Represented as Character Strings

```

1  /*
2  * Functions to perform basic operations on sets of characters
3  * represented as strings. Note: "rest of set" is represented as
4  * aset[], which is indeed the address of the rest of the set excluding
5  * the first element. This efficient representation, which does not
6  * re-copy the rest of the set, is an acceptable subtlety reference in
7  * these functions only because the "rest of the set" is always passed
8  * strictly as an input argument.
9  */
10
11 #include <stdio.h>
12 #include <string.h>
13 #include <ctype.h>
14
15 #define REPSIZ 45 /* 52 uppercase and lowercase letters, 10 digits,
16                  ( , ) and '\b' */
17 #define TRUE 1
18 #define FALSE 0
19
20 int is_empty(const char *set);
21 int is_element(char ele, const char *set);
22 int is_set(const char *set);
23 int is_subset(const char *sub, const char *set);
24 char *set_union(char *result, const char *set1, const char *set2);
25 void print_set(const char *str);
26 void print_set(const char *set);
27 char *get_set(const char *set);
    
```

Figure 10.20
Recursive
Set Operations
on Sets
Represented
as Character
Strings (cont'd)

```

187 /*
188  * Try to test set operations functions.
189  */
190 int
191 main(void)
192 {
193     char s1a[SETSIZE], s1b[SETSIZE], s1c[SETSIZE];
194
195     printf("Set s1 entered as a string of up to %d letters:\n",
196           SETSIZE - 1);
197     printf("and digits enclosed in {} ");
198     printf("on digit(s) character(s):\n");
199     printf("For example, {a, b, c} is entered as {abc}:\n");
200
201     printf("Enter a set to test validation function: ");
202     get_set(s1a);
203     printf("%s\n", s1a);
204     print_set(s1a);
205     if (!is_valid_set(s1a))
206         printf("is a valid set!\n");
207     else
208         printf("is invalid!\n");
209
210     printf("Enter a single character, a space, and a set:\n");
211     while (getchar() != '\n')
212         continue; /* get first character after whitespace */
213     get_set(s1a);
214     printf("%s\n", s1a);
215     if (!is_element(s1a, s1a))
216         printf("is an element of %s\n");
217     else
218         printf("is not an element of %s\n");
219     print_set(s1a);
220
221     printf("Enter two sets to test set_union: ");
222     get_set(s1a);
223     get_set(s1b);
224     printf("The union of %s\n");
225     print_set(s1a);
226     printf("and %s\n");
227     print_set(s1b);
228 }

```

Figure 10.20
Recursive
Set Operations
on Sets
Represented
as Character
Strings (cont'd)

```

229
230     printf(" is ");
231     print_set(s1a);
232     print_set(s1b);
233     print_set(s1c);
234 }
235
236 /*
237  * Determine if set s1 is empty. If so, return 1; if not, return 0.
238  */
239 int
240 is_empty_set(char *set)
241 {
242     return (set[0] == '\0');
243 }
244
245 /*
246  * Determine if s1a is an element of set.
247  */
248 int
249 is_element(char *s1a, /* input - element to look for in set
250                * set - input - set in which to look for s1a
251                */
252           char *set)
253 {
254     int end;
255
256     if (!is_empty(set))
257         end = strlen(set);
258     else if (set[0] == '\0')
259         end = 0;
260     else
261         end = is_element(s1a, set[0]);
262
263     return (end);
264 }
265
266 /*
267  * Determine if string value of set represents a valid set (no duplicate
268  * elements)
269  */

```

Figure 10.20
Recursive
Set Operations
on Sets
Represented
as Character
Strings (cont'd)

```

270
271 int
272 is_subset(char *set1, /* input - set whose
273                  * subset is being tested
274                  */
275          char *set2)
276 {
277     int end1;
278
279     if (!is_empty(set1))
280         end1 = strlen(set1);
281     else if (set1[0] == '\0')
282         end1 = 0;
283     else
284         end1 = is_subset(set1, set2);
285
286     return (end1);
287 }
288
289 /*
290  * Determine if value of set1 is a subset of value of set2.
291  */
292 int
293 is_subset(char *set1, /* input - set whose
294                  * subset is being tested
295                  */
296          char *set2)
297 {
298     int end1;
299
300     if (!is_empty(set1))
301         end1 = strlen(set1);
302     else if (set1[0] == '\0')
303         end1 = 0;
304     else
305         end1 = is_subset(set1, set2);
306
307     return (end1);
308 }
309
310 /*
311  * Finds the union of set1 and set2.
312  * Pre: size of result array is at least SETSIZE.
313  * Post: set1 and set2 are valid sets of characters and digits
314  */
315 char
316 set_union(char *result, /* output - space in which to store
317                  * string result
318                  */

```

Figure 10.20
Recursive
Set Operations
on Sets
Represented
as Character
Strings (cont'd)

```

319
320     char *set1, /* input - set whose
321                * subset is being tested
322                */
323     char *set2, /* input - set in which to look for s1a
324                */
325     char temp[SETSIZE]; /* local variable to hold result of call
326                        * to set_union embedded in printf call
327                        */
328     if (!is_empty(set1))
329         strcpy(temp, set1);
330     else if (set1[0] == '\0')
331         set_union(temp, set2);
332     else
333         printf("Error: %s\n", set1);
334
335     return (temp);
336 }
337
338 /*
339  * Displays a string so that each pair of characters is separated by a
340  * comma and a space.
341  */
342 void
343 print_with_commas(char *str)
344 {
345     if (strlen(str) == 1)
346         putchar(str[0]);
347     else
348         printf("%c, ", str[0]);
349         print_with_commas(str + 1);
350 }
351
352 /*
353  * Displays a string in standard set notation.
354  * Pre: print_set(char *) outputs {a, b, c}
355  */
356 void
357 print_set(char *set)
358 {

```

Figure 10.20 Recursive Set Operations on Sets Represented as Character Strings (cont'd)

```

187.     putchar('\n');
188.     if (!is_empty(set))
189.         print_with_commas(set);
190.     putchar('\n');
191. }
192.
193. /*
194.  * Gets a set input as a string with brackets (e.g., {abc})
195.  * and strips off the brackets.
196.  */
197. char
198. get_set(char *set) /* output - set string without brackets {} */
199. {
200.     char inset[SETSIZE];
201.
202.     scanf("%s", inset);
203.     strncpy(set, inset + 1, strlen(inset) - 2);
204.     set[strlen(inset) - 2] = '\0';
205.     return (set);
206. }

```

Figure 10.21 Towers of Hanoi



Figure 10.22 Towers of Hanoi After Steps 1 and 2

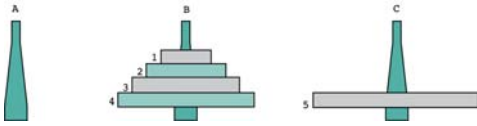


Figure 10.23 Towers of Hanoi After Steps 1, 2, 3.1, and 3.2

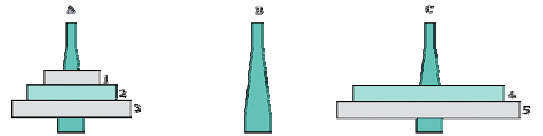


Figure 10.24 Recursive Function tower

```

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  }

```

Figure 10.25 Trace of tower ('A', 'C', 'B', 3);

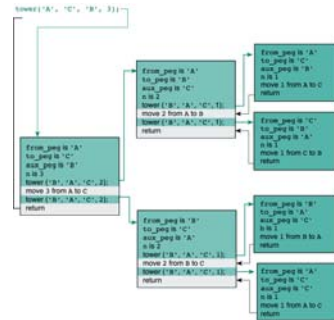


Figure 10.26 Output Generated by tower ('A', 'C', 'B', 3);

```

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

```

Figure 10.27 Grid with Three Blobs

