13.1 Recursion: Introduction

An ***algorithm*** is a sequence of steps for solving a problem. For example, an algorithm for making lemonade is:

Figure 13.1.1: Algorithms are like recipes.

Make lemonade:

- Add sugar to pitcher
- Add lemon juice
- Add water
- Stir

Some problems can be solved using a recursive algorithm. A ***recursive algorithm*** solves a problem by breaking that problem into smaller subproblems, solving these subproblems, and combining the solutions.

Figure 13.1.2: Mowing the lawn can be broken down into a recursive process.

- Mow the lawn
  - Mow the frontyard
    - Mow the left front
    - Mow the right front
  - Mow the backyard
    - Mow the left back
    - Mow the right back

An algorithm that is defined by repeated applications of the same algorithm on smaller problems is a ***recursive*** algorithm. The mowing algorithm consists of applying the mowing algorithm on smaller pieces of the yard.

At some point, a recursive algorithm must describe how to actually do something, known as the ***base case***. The mowing algorithm could thus be written as:

- *Mow* the lawn
  - *If lawn is less than 100 square meters*

- Push the lawnmower left-to-right in adjacent rows
- *Else*
  - Mow one half of the lawn
  - Mow the other half of the lawn

PARTICIPATION ACTIVITY 13.1.1: Recursion.

Which are recursive definitions/algorithms?

1) Helping N people:

   If N is 1, help that person.
   Else, help the first N/2 people, then help the second N/2 people.

   - True
   - False

2) Driving to the store:

   Go 1 mile.
   Turn left on Main Street.
   Go 1/2 mile.

   - True
   - False

3) Sorting envelopes by zipcode:

   If N is 1, done.
   Else, find the middle zipcode. Put all zipcodes less than the middle zipcode on the left, all greater ones on the right. Then sort the left, then sort the right.

   - True
   - False

13.2 Recursive functions

A function may call other functions, including calling itself. A function that calls itself is a ***recursive function***.

PARTICIPATION ACTIVITY 13.2.1: A recursive function example.

Animation captions:

1. The first call to CountDown( ) function comes from main. Each call to CountDown( ) effectively creates a new "copy" of the executing function, as shown on the right.
2. Then, the CountDown( ) function calls itself. CountDown(1) similarly creates a new "copy" of th executing function.
3. CountDown( ) function calls itself once more.
4. That last instance does not call CountDown( ) again, but instead returns. As each instance returns, that copy is deleted.

Each call to CountDown() effectively creates a new "copy" of the executing function, as shown on the right. Returning deletes that copy.

The example is for demonstrating recursion; counting down is otherwise better implemented with a loop.

Recursion may be direct, such as f() itself calling f(), or indirect, such as f() calling g() and g() calling f().

PARTICIPATION ACTIVITY 13.2.2: Thinking about recursion.

Refer to the above CountDown example for the following.

1) How many times is CountDown() called if main() calls CountDown(5)?

Check Show answer

2) How many times is CountDown() called if main() calls CountDown(0)?

Check Show answer

3) Is there a difference in how we define the parameters of a recursive versus

non-recursive function? Answer yes or no.

Check Show answer

CHALLENGE ACTIVITY 13.2.1: Calling a recursive function.

Write a statement that calls the recursive function BackwardsAlphabet() with parameter startingLetter.

```
#include <stdio.h>

void BackwardsAlphabet(char currLetter){
   if (currLetter == 'a') {
      printf("%c\n", currLetter);
   }
   else{
      printf("%c ", currLetter);
      BackwardsAlphabet(currLetter - 1);
   }
   return;
}

int main(void) {
   char startingLetter = '-';

   startingLetter = 'z';

   /* Your solution goes here  */
```

Run

13.3 Recursive algorithm: Search

Consider a guessing game program where a friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly. What algorithm would you use to minimize the number of guesses?

A first try might implement an algorithm that simply guesses in increments of 1:

- Is it 0? Higher

- Is it 1? Higher
- Is it 2? Higher

This algorithm requires too many guesses (50 on average). A second try might implement an algorithm that guesses by 10s and then by 1s:

- Is it 10? Higher
- Is it 20? Higher
- Is it 30? Lower
- Is it 21? Higher
- Is it 22? Higher
- Is it 23? Higher

This algorithm does better but still requires about 10 guesses on average: 5 to find the correct tens digit and 5 to guess the correct ones digit. An even better algorithm uses a binary search. A ***binary search*** algorithm begins at the midpoint of the range and halves the range after each guess. For example:

- Is it 50 (the middle of 0-100)? Lower
- Is it 25 (the middle of 0-50)? Higher
- Is it 37 (the middle of 25-50)? Lower
- Is it 31 (the middle of 25-37).

After each guess, the binary search algorithm is applied again, but on a smaller range, i.e., the algorithm is recursive.

PARTICIPATION ACTIVITY 13.3.1: Binary search: A well-known recursive algorithm.

Animation captions:

1. A friend thinks of a number from 0 to 100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly.
2. Using a binary search algorithm, you begin at the midpoint of the lower range. (highVal + lowVa / 2 = (100 + 0) / 2, or 50.
3. The number is lower. The algorithm divides the range in half, then chooses the midpoint of that range.
4. After each guess, the binary search algorithm is applied, halving the range and guessing the midpoint or the corresponding range.
5. A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the guessing range.

A recursive function is a natural match for the recursive binary search algorithm. A function GuessNumber(lowVal, highVal) has parameters that indicate the low and high sides of the

guessing range. The function guesses at the midpoint of the range. If the user says lower, the function calls GuessNumber(lowVal, midVal). If the user says higher, the function calls GuessNumber(midVal + 1, highVal)[mid].

Figure 13.3.1: A recursive function Find() carrying out a binary search algorithm.


```
#include <stdio.h>

void GuessNumber(int lowVal, int highVal) {
   int midVal = 0;         // Midpoint of low and high value
   char userAnswer = '-'; // User response

   midVal = (highVal + lowVal) / 2;

   // Prompt user for input
   printf("Is it %d? (l/h/y): ",midVal);
   scanf(" %c", &userAnswer);

   if( (userAnswer != 'l') && (userAnswer != 'h') ) { // Base case: Found number
      printf("Thank you!\n");
   }
   else {                                             // Recursive case: split into lower OR
upper half
      if (userAnswer == 'l') {                        // Guess in lower half
         GuessNumber(lowVal, midVal);                 // Recursive call
      }
      else {                                          // Guess in upper half
         GuessNumber(midVal + 1, highVal);            // Recursive call
      }
   }

   return;
}

int main(void) {
   // Print game objective, user input commands
   printf("Choose a number from 0 to 100.\n");
   printf("Answer with:\n");
   printf("   l (your num is lower)\n");
   printf("   h (your num is higher)\n");
   printf("   any other key (guess is right).\n");

   // Call recursive function to guess number
   GuessNumber(0, 100);

   return 0;
}
```

```
Choose a number from 0 to 100.
Answer with:
   l (your num is lower)
   h (your num is higher)
   any other key (guess is right).
Is it 50? (l/h/y): l
Is it 25? (l/h/y): h
Is it 38? (l/h/y): l
Is it 32? (l/h/y): l
Is it 29? (l/h/y): h
Is it 31? (l/h/y): y
Thank you!
```

©zyBooks 04/05/18 21:47 261830
Julian Chan
WEBERCS2250ValleSpring2018

PARTICIPATION ACTIVITY 13.3.2: Binary search tree tool.

The following program guesses the hidden number known by the user. Assume the hidden number is 63.

```
#include <stdio.h>

void Find(int low, int high) {
   int mid; // Midpoint of low..high
   char answer;

   mid = (high + low) / 2;

   printf("Is it %d? (l/h/y): ", mid);
   scanf(" %c", &answer);

   if((answer != 'l') &&
      (answer != 'h')) { // Base case:
      printf("Thank you!\n");  // Found number!
   }
   else { // Recursive case: Guess in
          // lower or upper half of range
      if (answer == 'l') {  // Guess in lower half
         Find(low, mid);    // Recursive call
      }
      else {                // Guess in upper half
         Find(mid + 1, high); // Recursive call
      }
   }

   return;
}

int main(void) {
   printf("Choose a number from 0 to 100.\n");
   printf("Answer with:\n");
   printf("   l (your num is lower)\n");
   printf("   h (your num is higher)\n");
   printf("   any other key (guess is right).\n");

   Find(0, 100);

   return 0;
}
```


→ main()

```
int main(void) {
   printf("Choose a number from 0 to 100.\n");
   printf("Answer with:\n");
   printf("   l (your num is lower)\n");
   printf("   h (your num is higher)\n");
   printf("   any other key (guess is right).\n");

      Find(0, 100);

      return 0;
}
```

The recursive function has an if-else statement. The if branch ends the recursion, known as the ***base case***. The else branch has recursive calls. Such an if-else pattern is common in recursive functions.

Search is commonly performed to quickly find an item in a sorted list stored in an array or vector. Consider a list of attendees at a conference, whose names have been stored in alphabetical order in an array or vector. The following quickly determines whether a particular person is in attendance.

Figure 13.3.2: Recursively searching a sorted list.

```
#include <stdio.h>
#include <string.h>

/* Finds index of string in array of strings, else -1.
   Searches only with index range low to high
   Note: Upper/lower case characters matter
*/

const int NUM_ATTENDEES = 6;         // Number of total attendees
const int CHAR_LIMIT_PER_NAME = 50; // Limit size of attendee name

int FindMatch(char stringsList[NUM_ATTENDEES][CHAR_LIMIT_PER_NAME], char itemMatch[],
              int lowVal, int highVal) {
   int midVal = 0;    // Midpoint of low and high values
   int itemPos = -1;  // Position where item found, -1 if not found
   int rangeSize = 0; // Remaining range of values to search for match

   rangeSize = (highVal - lowVal) + 1;
   midVal = (highVal + lowVal) / 2;

   if (strcmp(stringsList[midVal], itemMatch) == 0) {    // Base case 1: item found at midVal
position
      itemPos = midVal;
   }
   else if (rangeSize == 1) {                            // Base case 2: match not found
      itemPos = -1;
   }
   else {                                                // Recursive case: search lower or
upper half
      if (strcmp(itemMatch, stringsList[midVal]) < 0) { // Search lower half, recursive call
         itemPos = FindMatch(stringsList, itemMatch, lowVal, midVal);
      }
      else {                                             // Search upper half, recursive call
```

```
      else {
         itemPos = FindMatch(stringsList, itemMatch, midVal + 1, highVal);
      }
   }

   return itemPos;
}

int main(void) {
   char attendeesList[NUM_ATTENDEES][CHAR_LIMIT_PER_NAME]; // List of attendees
   char attendeeName[50] = "";                             // Name of attendee to match. 50 is
CHAR_LIMIT_PER_NAME
   int matchPos = 0;                                       // Matched position in attendee list

   // Omitting part of program that adds attendees
   // Instead, we insert some sample attendees in sorted order
   strcpy(attendeesList[0],"Adams, Mary");
   strcpy(attendeesList[1],"Carver, Michael");
   strcpy(attendeesList[2],"Domer, Hugo");
   strcpy(attendeesList[3],"Fredericks, Carlos");
   strcpy(attendeesList[4],"Li, Jie");
   strcpy(attendeesList[5],"Zuckerberg, Al");

   // Prompt user to enter a name to find
   printf("Enter person's name: Last, First: ");
   fgets(attendeeName, CHAR_LIMIT_PER_NAME, stdin);
   attendeeName[strlen(attendeeName) - 1] = '\0'; // remove newline

   // Call function to match name, output results
   matchPos = FindMatch(attendeesList, attendeeName, 0, NUM_ATTENDEES - 1);
   if (matchPos >= 0) {
      printf("Found at position %d.\n",matchPos);
   }
   else {
      printf("Not found. \n");
   }

   return 0;
}
```



```
Enter person's name: Last, First: Meeks, Stan
Not found.

...

Enter person's name: Last, First: Adams, Mary
Found at position 0.

...

Enter person's name: Last, First: Li, Jie
Found at position 4.
```

FindMatch() restricts its search to elements within the range lowVal to highVal. main() initially passes a range of the entire list: 0 to (list size - 1). FindMatch() compares to the middle element, returning that element's position if matching. If not matching, FindMatch() checks if the window's size is just one element, returning -1 in that case to indicate the item was not found. If neither of those two base cases are satisfied, then FindMatch() recursively searches either the lower or upper half of the range as appropriate.

PARTICIPATION ACTIVITY 13.3.3: Recursive search algorithm.

Consider the above FindMatch() function for finding an item in a sorted list.

1) If a sorted list has elements 0 to 50 and the item being searched for is at element 6, how many times will FindMatch() be called?

Check Show answer

2) If an alphabetically ascending list has elements 0 to 50, and the item at element 0 is "Bananas", how many recursive calls to FindMatch() will be made during the failed search for "Apples"?

Check Show answer

PARTICIPATION ACTIVITY 13.3.4: Recursive calls.

A list has 5 elements numbered 0 to 4, with these letter values: 0: A, 1: B, 2: D, 3: E, 4: F.

1) To search for item C, the first call is FindMatch(0, 4). What is the second call to FindMatch()?

- FindMatch(0, 0)
- FindMatch(0, 2)
- FindMatch(3, 4)

2) In searching for item C, FindMatch(0, 2) is called. What happens next?

- Base case 1: item found at midVal.
- Base case 2: rangeSize == 1, so no match.
- Recursive call: FindMatch(2, 2)

Exploring further:

- Binary search from GeeksforGeeks.org

(*mid) Because midVal has already been checked, it need not be part of the new window, so midVal + 1 rather than midVal is used for the window's new low side, or midVal - 1 for the window's new high side. But the midVal - 1 can have the drawback of a non-intuitive base case (i.e., midVal < lowVal, because if the current window is say 4..5, midVal is 4, so the new window would be 4..4-1, or 4..3). rangeSize == 1 is likely more intuitive, and thus the algorithm uses midVal rather than midVal - 1. However, the algorithm uses midVal + 1 when searching higher, due to integer rounding. In particular, for window 99..100, midVal is 99 ((99 + 100) / 2 = 99.5, rounded to 99 due to truncation of the fraction in integer division). So the next window would again be 99..100, and the algorithm would repeat with this window forever. midVal + 1 prevents the problem, and doesn't miss any numbers because midVal was checked and thus need not be part of the window.

13.4 Adding output statements for debugging

Recursive functions can be particularly challenging to debug. Adding output statements can be helpful. Furthermore, an additional trick is to indent the print statements to show the current depth of recursion. The following program adds a parameter indent to a FindMatch() function that searches a sorted list for an item. All of FindMatch()'s print statements start with `printf("% ...", indentAmt, ...);`. Indent is typically some number of spaces. main() sets indent to three spaces. Each recursive call *adds* three more spaces. Note how the output now clearly shows the recursion depth.

Figure 13.4.1: Output statements can help debug recursive functions, especially if indented based on recursion depth.

```
#include <stdio.h>
#include <string.h>

/* Finds index of string in array of strings, else -1.
   Searches only with index range low to high
   Note: Upper/lower case characters matter
*/

const int NUM_ATTENDEES = 6;         // Number of total attendees
const int CHAR_LIMIT_PER_NAME = 50; // Limit size of attendee name

int FindMatch(char stringsList[NUM_ATTENDEES][CHAR_LIMIT_PER_NAME], char itemMatch[],
              int lowVal, int highVal, char indentAmt[CHAR_LIMIT_PER_NAME]) { // indentAmt used
for print debug
```

```
   int midVal = 0;           // Midpoint of low and high values
   int itemPos = -1;         // Position where item found, -1 if not found
   int rangeSize = 0;        // Remaining range of values to search for match
   char indentNext[50] = ""; // Stores next recursion level indentation amount

   // Update indent amount for next level of recursion
   strcpy(indentNext, indentAmt);
   strcat(indentNext, "   ");

   printf("%sFind() range %d %d\n", indentAmt, lowVal, highVal);
   rangeSize = (highVal - lowVal) + 1;
   midVal = (highVal + lowVal) / 2;

   if (strcmp(stringsList[midVal], itemMatch) == 0) {   // Base case 1: Found at mid
      printf("%sFound person.\n", indentAmt);
      itemPos = midVal;
   }
   else if (rangeSize == 1) {                           // Base case 2: Not found
      printf("%sPerson not found.\n", indentAmt);
      itemPos = -1;
   }
   else {                                               // Recursive case: search lower or
upper half
      if (strcmp(itemMatch, stringsList[midVal]) < 0) { // Search lower half, recursive call
         printf("%sSearching lower half.\n", indentAmt);
         itemPos = FindMatch(stringsList, itemMatch, lowVal, midVal, indentNext);
      }
      else {                                            // Search upper half, recursive call
         printf("%sSearching upper half.\n", indentAmt);
         itemPos = FindMatch(stringsList, itemMatch, midVal + 1, highVal, indentNext);
      }
   }

   printf("%sReturning pos = %d.\n", indentAmt, itemPos);
   return itemPos;
}

int main(void) {
   char attendeesList[NUM_ATTENDEES][CHAR_LIMIT_PER_NAME]; // List of attendees
   char attendeeName[50] = "";                             // Name of attendee to match. 50 is
CHAR_LIMIT_PER_NAME
   int matchPos = 0;                                       // Matched position in attendee list
   char indentLevel[50] = "";                              // Stores indentation (3 spaces)

   // Omitting part of program that adds attendees
   // Instead, we insert some sample attendees in sorted order
   strcpy(attendeesList[0],"Adams, Mary");
   strcpy(attendeesList[1],"Carver, Michael");
   strcpy(attendeesList[2],"Domer, Hugo");
   strcpy(attendeesList[3],"Fredericks, Carlos");
   strcpy(attendeesList[4],"Li, Jie");
   strcpy(attendeesList[5],"Zuckerberg, Al");

   // Prompt user to enter a name to find
   printf("Enter person's name: Last, First: ");
   fgets(attendeeName, CHAR_LIMIT_PER_NAME, stdin);
   attendeeName[strlen(attendeeName) - 1] = '\0'; // remove newline
   strcpy(indentLevel, "   ");

   // Call function to match name, output results
   matchPos = FindMatch(attendeesList, attendeeName, 0, NUM_ATTENDEES - 1, indentLevel);
   if (matchPos >= 0) {
      printf("Found at position %d.\n",matchPos);
   }
   else {
      printf("Not found. \n");
   }

   return 0;
}
```

```
Enter person's name: Last, First: Meeks, Stan
   Find() range 0 5
   Searching upper half.
      Find() range 3 5
      Searching upper half.
         Find() range 5 5
         Person not found.
         Returning pos = -1.
      Returning pos = -1.
   Returning pos = -1.
Not found.

...
Enter person's name: Last, First: Adams, Mary
   Find() range 0 5
   Searching lower half.
      Find() range 0 2
      Searching lower half.
         Find() range 0 1
         Found person.
         Returning pos = 0.
      Returning pos = 0.
   Returning pos = 0.
Found at position 0.
```


Some programmers like to leave the output statements in the code, commenting them out with "//" when not in use. The statements actually serve as a form of comment as well.

More advanced techniques for handling debug output exist too, such as ***conditional compilation*** (beyond this section's scope).

PARTICIPATION ACTIVITY 13.4.1: Recursive debug statements.

Refer to the above code using indented output statements.

1) The above debug approach requires an extra parameter be passed to indicate the amount of indentation.
   - True
   - False

2) Each recursive call should add a few spaces to the indent parameter.
   - True
   - False

3) The function should remove a few

spaces from the indent parameter before returning.

- True
- False

PARTICIPATION ACTIVITY 13.4.2: Output statements in a recursive function.

- Run the recursive program, and observe the output statements for debugging, and that the person is correctly not found.
- Introduce an error by changing `itemPos = -1` to `itemPos = 0` in the range size == 1 base case.
- Run the program, notice how the indented print statements help isolate the error of the person incorrectly being found.

Load default template... Run

```
#include <stdio.h>
#include <string.h>

/* Finds index of string in array of strings, else -1.
   Searches only with index range low to high
   Note: Upper/lower case characters matter
*/

const int NUM_ATTENDEES = 6;         // Number of total a
const int CHAR_LIMIT_PER_NAME = 50; // Limit size of att

int FindMatch(char stringsList[NUM_ATTENDEES][CHAR_LIMIT
              int lowVal, int highVal, char indentAmt[CH

   int midVal = 0;           // Midpoint of low and high
   int itemPos = -1;         // Position where item foun
   int rangeSize = 0;        // Remaining range of value
   char indentNext[50] = ""; // Stores next recursion le
```


13.5 Creating a recursive function

Creating a recursive function can be accomplished in two steps.

- **Write the base case** -- Every recursive function must have a case that returns a value without performing a recursive call. That case is called the ***base case***. A programmer may

write that part of the function first, and then test. There may be multiple base cases.

- **Write the recursive case** -- The programmer then adds the recursive case to the function.

The following illustrates for a simple function that computes the factorial of N (i.e. N!). The base case is N = 1 or 1! which evaluates to 1. The base case is written as `if (N <= 1) { fact = 1; }`. The recursive case is used for N > 1, and written as `else { fact = N * NFact( N - 1 ); }`.

PARTICIPATION ACTIVITY 13.5.1: Writing a recursive function for factorial: First write the base case, then add the recursive case.

Animation captions:

1. The base case, which returns a value without performing a recursive call, is written and tested first. If N is less than or equal to 1, then the NFact() function returns 1.
2. Next the recursive case, which calls itself, is written and tested. If N is greater than 1, then the NFact() function returns N * NFact(N - 1).

A common error is to not cover all possible base cases in a recursive function. Another common error is to write a recursive function that doesn't always reach a base case. Both errors may lead to infinite recursion, causing the program to fail.

Typically, programmers will use two functions for recursion. An "outer" function is intended to be called from other parts of the program, like the function `int CalcFactorial(int inVal)`. An "inner" function is intended only to be called from that outer function, for example a function `int _CalcFactorial(int inVal)` (note the "_"). The outer function may check for a valid input value, e.g., ensuring inVal is not negative, and then calling the inner function. Commonly, the inner function has parameters that are mainly of use as part of the recursion, and need not be part of the outer function, thus keeping the outer function more intuitive.

PARTICIPATION ACTIVITY 13.5.2: Creating recursion.

1) Recursive functions can be accomplished in one step, namely repeated calls to itself.

   - ○ True
   - ○ False

2) A recursive function with parameter N counts up from any negative number to 0. An appropriate base case would be `N == 0`.

   - ○

True

False

3) A recursive function can have two base cases, such as `N == 0` returning 0, and `N == 1` returning 1.

True

False

Before writing a recursive function, a programmer should determine:

1. Does the problem naturally have a recursive solution?
2. Is a recursive solution better than a non-recursive solution?

For example, computing N! (N factorial) does have a natural recursive solution, but a recursive solution is not better than a non-recursive solution. The figure below illustrates how the factorial computation can be implemented as a loop. Conversely, binary search has a natural recursive solution, and that solution may be easier to understand than a non-recursive solution.

Figure 13.5.1: Non-recursive solution to compute N!

```
for (i = inputNum; i > 1; --i) {
   facResult = facResult * i;
}
```

PARTICIPATION ACTIVITY 13.5.3: When recursion is appropriate.

1) N factorial (N!) is commonly implemented as a recursive function due to being easier to understand and executing faster than a loop implementation.

True

False

PARTICIPATION ACTIVITY 13.5.4: Output statements in a recursive function.

Implement a recursive function to determine if a number is prime. Skeletal code is provided in the IsPrime function.

Load default template...

Run

```
#include <stdio.h>

// Returns 0 if value is not prime, 1 if value is prime
int IsPrime(int testVal, int divVal)
{
   // Base case 1: 0 and 1 are not prime, testVal is not

   // Base case 2: testVal only divisible by 1, testVal

   // Recursive Case
      // Check if testVal can be evenly divided by divVa
      // Hint: use the % operator

      // If not, recursive call to isPrime with testVal
   return 0;
}

int main(void){
```


CHALLENGE ACTIVITY 13.5.1: Recursive function: Writing the base case.

Write code to complete DoublePennies()'s base case. Sample output for below program:

```
Number of pennies after 10 days: 1024
```

Note: These activities may test code with different test values. This activity will perform three tests, with startingPennies = 1 and userDays = 10, then with startingPennies = 1 and userDays = 40, then with startingPennies = 1 and userDays = 1. See "How to Use zyBooks".

Also note: If the submitted code has an infinite loop, the system will stop running the code after a few seconds, and report "Program end never reached." The system doesn't print the test case that caused the reported message.

```
#include <stdio.h>

// Returns number of pennies if pennies are doubled numDays times
long long DoublePennies(long long numPennies, int numDays){
   long long totalPennies = 0;

   /* Your solution goes here  */

   else {
     totalPennies = DoublePennies((numPennies * 2), numDays - 1);
     }

   return totalPennies;
}
```

```
15
16 // Program computes pennies if you have 1 penny today,
17 // 2 pennies after one day, 4 after two days, and so on
18 int main(void) {
19    long long startingPennies = 0;
```

Run

CHALLENGE ACTIVITY 13.5.2: Recursive function: Writing the recursive case.

Write code to complete PrintFactorial()'s recursive case. Sample output if userVal is 5:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

```
1  #include <stdio.h>
2
3  void PrintFactorial(int factCounter, int factValue){
4     int nextCounter = 0;
5     int nextValue = 0;
6
7     if (factCounter == 0) {            // Base case: 0! = 1
8        printf("1\n");
9     }
10    else if (factCounter == 1) {       // Base case: Print 1 and result
11       printf("%d = %d\n", factCounter, factValue);
12    }
13    else {                             // Recursive case
14       printf("%d * ", factCounter);
15       nextCounter = factCounter - 1;
16       nextValue = nextCounter * factValue;
17
18       /* Your solution goes here  */
19
```

Run

13.6 Recursive math functions

Fibonacci sequence

Recursive functions can solve certain math problems, such as computing the Fibonacci sequence. The ***Fibonacci sequence*** is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.; starting with 0, 1, the pattern is to compute the next number by adding the previous two numbers.

Below is a program that outputs the Fibonacci sequence values step-by-step, for a user-entered number of steps. The base case is that the program has output the requested number of steps. The recursive case is that the program needs to compute the number in the Fibonacci sequence.

Figure 13.6.1: Fibonacci sequence step-by-step.

```
#include <stdio.h>

/* Output the Fibonacci sequence step-by-step.
   Fibonacci sequence starts as:
   0 1 1 2 3 5 8 13 21 ... in which the first
   two numbers are 0 and 1 and each additional
   number is the sum of the previous two numbers
*/

void ComputeFibonacci(int fibNum1, int fibNum2, int runCnt) {

   printf("%d + %d = %d\n", fibNum1, fibNum2, fibNum1 +
fibNum2);

   if (runCnt <= 1) { // Base case: Ran for user specified
                      // number of steps, do nothing
   }
   else {             // Recursive case: compute next value
      ComputeFibonacci(fibNum2, fibNum1 + fibNum2, runCnt -
1);
   }

   return;
}

int main(void) {
   int runFor = 0; // User specified number of values
computed

   // Output program description
   printf("This program outputs the\n");
   printf("Fibonacci sequence step-by-step,\n");
   printf("starting after the first 0 and 1.\n\n");

   // Prompt user for number of values to compute
   printf("How many steps would you like? ");
   scanf("%d", &runFor);

   // Output first two Fibonacci values, call recursive
function
   printf("0\n1\n");
   ComputeFibonacci(0, 1, runFor);

   return 0;
}
```

```
This program outputs the
Fibonacci sequence step-by-
step,
starting after the first 0 and
1.

How many steps would you like?
10
0
1
0+1= 1
1+1= 2
1+2= 3
2+3= 5
3+5= 8
5+8= 13
8+13= 21
13+21= 34
21+34= 55
34+55= 89
```


PARTICIPATION ACTIVITY 13.6.1: Recursive Fibonacci.

Complete ComputeFibonacci() to return F_N, where F_0 is 0, F_1 is 1, F_2 is 1, F_3 is 2, F_4 is 3, and continuing: F_N is F_{N-1} + F_{N-2}. Hint: Base cases are N == 0 and N == 1.

```
#include <stdio.h>

int ComputeFibonacci(int N) {

   printf("FIXME: Complete this function.\n");
   printf("Currently just returns 0.\n");


   return 0;
}

int main(void) {
   int N = 4; // F_N, starts at 0

   printf("F_%d is %d\n", N, ComputeFibonacci(N));

   return 0;
}
```


Run

Greatest common divisor (GCD)

Recursion can solve the greatest common divisor problem. The ***greatest common divisor*** (GCD) is the largest number that divides evenly into two numbers, e.g. GCD(12, 8) = 4. One GCD algorithm (described by Euclid around 300 BC) subtracts the smaller number from the larger number until both numbers are equal. Ex:

- GCD(12, 8): Subtract 8 from 12, yielding 4.
- GCD(4, 8): Subtract 4 from 8, yielding 4.
- GCD(4, 4): Numbers are equal, return 4

The following recursively computes the GCD of two numbers. The base case is that the two numbers are equal, so that number is returned. The recursive case subtracts the smaller number from the larger number and then calls GCD with the new pair of numbers.

Figure 13.6.2: Calculate greatest common divisor of two numbers.

```c
#include <stdio.h>

/* Determine the greatest common divisor
   of two numbers, e.g. GCD(8, 12) = 4
*/
int GCDCalculator(int inNum1, int inNum2) {
   int gcdVal = 0; // Holds GCD results

   if(inNum1 == inNum2) {     // Base case: Numbers are equal
      gcdVal = inNum1;        // return value
   }
   else {                     // Recursive case: subtract
smaller from larger
      if (inNum1 > inNum2) { // call fucntion with new values
         gcdVal = GCDCalculator(inNum1 - inNum2, inNum2);
      }
      else {
         gcdVal = GCDCalculator(inNum1, inNum2 - inNum1);
      }
   }

   return gcdVal;
}

int main(void) {
   int gcdInput1 = 0; // First input to GCD calc
   int gcdInput2 = 0; // Second input to GCD calc
   int gcdOutput = 0; // Result of GCD

   // Print program function
   printf("This program outputs the greatest \n");
   printf("common divisor of two numbers.\n");

   // Prompt user for input
   printf("Enter first number: ");
   scanf("%d", &gcdInput1);

   printf("Enter second number: ");
   scanf("%d", &gcdInput2);

   // Check user values are > 1, call recursive GCD function
   if ((gcdInput1 < 1) || (gcdInput2 < 1)) {
      printf("Note: Neither value can be below 1.\n");
   }
   else {
      gcdOutput = GCDCalculator(gcdInput1, gcdInput2);
      printf("Greatest common divisor = %d\n", gcdOutput);
   }

   return 0;
}
```

```
This program outputs the
greatest
common divisor of two
numbers.
Enter first number: 12
Enter second number: 8
Greatest common divisor = 4

...

This program outputs the
greatest
common divisor of two
numbers.
Enter first number: 456
Enter second number: 784
Greatest common divisor = 8

...

This program outputs the
greatest
common divisor of two
numbers.
Enter first number: 0
Enter second number: 10
Note: Neither value can be
below 1.
```


PARTICIPATION ACTIVITY 13.6.2: Recursive GCD example.

1) How many calls are made to GCDCalculator() function for input values 12 and 8?

- 1
- 2
- 3

2) What is the base case for the GCD algorithm?

- When both inputs to the function are equal.
- When both inputs are greater than 1.
- When inNum1 > inNum2.

Exploring further:

- Fibonacci number from Wolfram.
- Greatest Common Divisor from Wolfram.

CHALLENGE ACTIVITY 13.6.1: Writing a recursive math function.

Write code to complete RaiseToPower(). Sample output if userBase is 4 and userExponent is 2 is shown below. Note: This example is for practicing recursion; a non-recursive function, or using the built-in function pow(), would be more common.

```
4^2 = 16
```

```
#include <stdio.h>

int RaiseToPower(int baseVal, int exponentVal){
   int resultVal = 0;

   if (exponentVal == 0) {
      resultVal = 1;
   }
   else {
      resultVal = baseVal * /* Your solution goes here  */;
   }

   return resultVal;
}

int main(void) {
```

```
17    int userBase = 0;
18    int userExponent = 0;
19
```

Run

13.7 Recursive exploration of all possibilities

Recursion is a powerful technique for exploring all possibilities, such as all possible reorderings of a word's letters, all possible subsets of items, all possible paths between cities, etc. This section provides several examples.

Word scramble

Consider printing all possible combinations (or "scramblings") of a word's letters. The letters of abc can be scrambled in 6 ways: abc, acb, bac, bca, cab, cba. Those possibilities can be listed by making three choices: Choose the first letter (a, b, or c), then choose the second letter, then choose the third letter. The choices can be depicted as a tree. Each level represents a choice. Each node in the tree shows the unchosen letters on the left, and the chosen letters on the right.

PARTICIPATION ACTIVITY 13.7.1: Exploring all possibilities viewed as a tree of choices.

Animation captions:

1. Consider printing all possible combinations of a word's letters. Those possibilities can be listed by choosing the first letter, then the second letter, then the third letter.
2. The choices can be depicted as a tree. Each level represents a choice.
3. A recursive exploration function is a natural match to print all possible combinations of a string's letters. Each call to the function chooses from the set of unchosen letters, continuing until no unchosen letters remain.

The tree guides creation of a recursive exploration function to print all possible combinations of a string's letters. The function takes two parameters: unchosen letters, and already chosen letters. The base case is no unchosen letters, causing printing of the chosen letters. The recursive case calls the function once for each letter in the unchosen letters. The above animation depicts how the recursive algorithm traverses the tree. The tree's leaves (the bottom nodes) are the base cases.

The following program prints all possible ordering of the letters of a user-entered word.

Figure 13.7.1: Scramble a word's letters in every possible way.

```
#include <stdio.h>
#include <string.h>

const int MAX_ARR_SIZE = 50;                          // Word size limit

void RemoveFromIndex(char* origString, int remLoc); // Remove letter at location i from string
c
void InsertAtIndex(char* origString, char* addChar,
                   int addLoc);                       // Add letter n to location i of string c

/* Output every possible combination of a word.
   Each recursive call moves a letter from
   remainLetters" to scramLetters".
*/
void ScrambleLetters(char* remainLetters,  // Remaining letters
                     char* scramLetters) { // Scrambled letters

   char tmpString[2] = ""; // Using c string for access to strcat
   int i = 0;              // Loop index

   if(strlen(remainLetters) == 0) {            // Base case: All letters used
      printf("%s\n",scramLetters);
   }
   else {                                      // Recursive case: move a letter from
                                               // remaining to scrambled letters
      for (i = 0; i < strlen(remainLetters); ++i) {

         // Move letter to scrambled letters
         tmpString[0] = remainLetters[i];
         strcat(scramLetters, tmpString);
         RemoveFromIndex(remainLetters, i);

         ScrambleLetters(remainLetters, scramLetters);

         // Put letter back in remaining letters
         scramLetters[strlen(scramLetters)-1]='\0';
         InsertAtIndex(remainLetters, tmpString, i);
      }
   }
   return;
}

int main(void) {
   char wordScramble[50] = "";   // User defined word to scramble. 50 is MAX_ARR_SIZE
   char finishScramble[50] = ""; // Temp string already scrambled. 50 is MAX_ARR_SIZE

   // Init strings
   strcpy(wordScramble, "");
   strcpy(finishScramble, "");

   // Prompt user for input
   printf("Enter a word to be scrambled: ");
   scanf("%s", wordScramble);

   // Call recursive function
   ScrambleLetters(wordScramble, finishScramble);

   return 0;
}

// Remove letter at location remLoc from string origString
void RemoveFromIndex(char* origString, int remLoc) {
   char tmpString[50] = "";      // Temp string to extract char. 50 is MAX_ARR_SIZE

   strcpy(tmpString, "");                   // Init string
```

```
   strncat(tmpString, origString, remLoc); // Copy before location remLoc
   strncat(tmpString, origString + remLoc + 1,
           strlen(origString) - remLoc);   // Copy after location remLoc
   strcpy(origString, tmpString);          // Copy back to orignal string
}

// Add letter addChar to location addLoc of string origString
void InsertAtIndex(char* origString, char* addChar, int addLoc) {
   char tmpString[50] = "";       // Temp string to add char. 50 is MAX_ARR_SIZE

   strcpy(tmpString,"");                   // Init string
   strncat(tmpString, origString, addLoc); // Copy before location addLoc
   strncat(tmpString, addChar, 1);         // Copy letter addChar to location addLoc
   strncat(tmpString, origString + addLoc,
           strlen(origString) - addLoc);   // Copy after location addLoc
   strcpy(origString, tmpString);
}
```

```
Enter a word to be scrambled: cat
cat
cta
act
atc
tca
tac
```


PARTICIPATION ACTIVITY 13.7.2: Letter scramble.

1) What is the output of ScrambleLetters("xy", "")? Determine your answer by manually tracing the code, not by running the program.

- yx xy
- xx yy xy yx
- xy yx

Shopping spree

Recursion can find all possible subsets of a set of items. Consider a shopping spree in which a person can select any 3-item subset from a larger set of items. The following program prints all possible 3-item subsets of a given larger set. The program also prints the total price of each subset.

ShoppingBagCombinations() has a parameter for the current bag contents, and a parameter for the remaining items from which to choose. The base case is that the current bag already has 3 items, which prints the items. The recursive case moves one of the remaining items to the bag, recursively calling the function, then moving the item back from the bag to the remaining items.

Figure 13.7.2: Shopping spree in which a user can fit 3 items in a shopping bag.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

typedef struct Item_struct {
   char itemName[50]; // Name of item
   int priceDollars;  // Price of item
} Item;

const int TOTAL_ITEMS = 4;                    // Total number of items
available
const unsigned int MAX_SHOPPING_BAG_SIZE = 3; // Max number of items in
shopping bag

/* Output every possible combination of items that
   fit in a shopping bag. Each recursive call moves
   one item into the bag.
*/
void ShoppingBagCombinations(Item* currBag,        // Bag contents
                             Item* remainingItems, // Available items
                             bool* beenAdded,      // Items already in
shopping bag
                             int bagCnt) {         // Current shopping
bag size
   int bagValue = 0; // Cost of items in shopping bag
   int i = 0;        // Loop index

   if (bagCnt == MAX_SHOPPING_BAG_SIZE) { // Base case: Shopping bag full
      bagValue = 0;
      for(i = 0; i < bagCnt; ++i) {
         printf("%s  ", currBag[i].itemName);
         bagValue += currBag[i].priceDollars;
      }
      printf("= $%d\n", bagValue);
   }
   else {                                  // Recursive case: move one
      for (i = 0; i < TOTAL_ITEMS; ++i) { // item to bag
         if (!beenAdded[i]) {
            // Move item to bag
            beenAdded[i] = true;
            currBag[bagCnt] = remainingItems[i];

            ShoppingBagCombinations(currBag, remainingItems,
                                    beenAdded, bagCnt + 1);

            // Take item out of bag
            beenAdded[i] = false;
         }
      }
   }

   return;
}

int main(void) {
   Item* possibleItems = NULL; // Possible shopping items
   Item* shoppingBag = NULL;   // Current shopping bag
   bool* itemBeenAdded = NULL; // Track if item already in bag
   Item tmpGroceryItem;        // Temp item

   possibleItems = (Item*)malloc(sizeof(Item) * TOTAL_ITEMS);
   shoppingBag = (Item*)malloc(sizeof(Item) * TOTAL_ITEMS);
   itemBeenAdded = (bool*)malloc(sizeof(bool) * TOTAL_ITEMS);

   // No items added yet
   itemBeenAdded[0] = false;
   itemBeenAdded[1] = false;
   itemBeenAdded[2] = false;
   itemBeenAdded[3] = false;
```

```
Milk  Belt  Toys  = $45
Milk  Belt  Cups  = $38
Milk  Toys  Belt  = $45
Milk  Toys  Cups  = $33
Milk  Cups  Belt  = $38
Milk  Cups  Toys  = $33
Belt  Milk  Toys  = $45
Belt  Milk  Cups  = $38
Belt  Toys  Milk  = $45
Belt  Toys  Cups  = $55
Belt  Cups  Milk  = $38
Belt  Cups  Toys  = $55
Toys  Milk  Belt  = $45
Toys  Milk  Cups  = $33
Toys  Belt  Milk  = $45
Toys  Belt  Cups  = $55
Toys  Cups  Milk  = $33
Toys  Cups  Belt  = $55
Cups  Milk  Belt  = $38
Cups  Milk  Toys  = $33
Cups  Belt  Milk  = $38
Cups  Belt  Toys  = $55
Cups  Toys  Milk  = $33
Cups  Toys  Belt  = $55
```

```
    // Populate grocery with different items
    strcpy(tmpGroceryItem.itemName, "Milk");
    tmpGroceryItem.priceDollars = 2;
    possibleItems[0] = tmpGroceryItem;

    strcpy(tmpGroceryItem.itemName, "Belt");
    tmpGroceryItem.priceDollars = 24;
    possibleItems[1] = tmpGroceryItem;

    strcpy(tmpGroceryItem.itemName, "Toys");
    tmpGroceryItem.priceDollars = 19;
    possibleItems[2] = tmpGroceryItem;

    strcpy(tmpGroceryItem.itemName, "Cups");
    tmpGroceryItem.priceDollars = 12;
    possibleItems[3] = tmpGroceryItem;

    // Try different combinations of three items
    ShoppingBagCombinations(shoppingBag, possibleItems,
                            itemBeenAdded, 0);

    return 0;
}
```


PARTICIPATION ACTIVITY 13.7.3: All letter combinations.

1) When main() calls ShoppingBagCombinations(), how many items are in the remainingItems list?

   - None
   - 3
   - 4

2) When main() calls ShoppingBagCombinations(), how many items are in currBag list?

   - None
   - 1
   - 4

3) After main() calls ShoppingBagCombinations(), what happens first?

   - The base case prints Milk, Belt, Toys.
   - The function bags one item,

makes recursive call.

- The function bags 3 items, makes recursive call.

4) After ShoppingBagCombinations() returns back to main(), how many items are in the remainingItems list?

- None
- 4

5) How many recursive calls occur before the first combination is printed?

- None
- 1
- 3

6) What happens if main() only put 2, rather than 4, items in the possibleItems list?

- Base case never executes; nothing printed.
- Infinite recursion occurs.

Traveling salesman

Recursion is useful for finding all possible paths. Suppose a salesman must travel to 3 cities: Boston, Chicago, and Los Angeles. The salesman wants to know all possible paths among those three cities, starting from any city. A recursive exploration of all travel paths can be used. The base case is that the salesman has traveled to all cities. The recursive case is to travel to a new city, explore possibilities, then return to the previous city.

Figure 13.7.3: Find distance of traveling to 3 cities.


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

const int NUM_CITIES = 3; // Number of cities

int cityDistances[3][3];  // Distance between cities
char cityNames[3][50];    // City names

/* Output every possible travel path.
   Each recursive call moves to a new city.
*/
```

```
void TravelPaths(int* currPath, int* toVisit,
                 bool* haveBeen, int cityCnt) {
   int totalDist = 0; // Total distance given current path
   int i = 0;         // Loop index

   if (cityCnt == NUM_CITIES) { // Base case: Visited all cities
      totalDist = 0;            // return total path distance
      for (i = 0; i < cityCnt; ++i) {
         printf("%s   ", cityNames[currPath[i]]);

         if (i > 0) {
            totalDist += cityDistances[currPath[i - 1]][currPath[i]];
         }
      }

      printf("= %d\n", totalDist);
   }
   else {                       // Recursive case: pick next city
      for (i = 0; i < NUM_CITIES; ++i) {
         if (!haveBeen[i]) {
            // Add city to travel path
            haveBeen[i] = true;
            currPath[cityCnt] = toVisit[i];

            TravelPaths(currPath, toVisit, haveBeen, cityCnt+1);

            // Remove city from travel path
            haveBeen[i] = false;
         }
      }
   }

   return;
}

int main(void) {
   int* currPath = NULL;  // Current path traveled
   int* toVisit = NULL;   // Cities left to visit
   bool* haveBeen = NULL; // City already visited

   // Initialize distances array
   cityDistances[0][0] = 0;
   cityDistances[0][1] = 960;  // Boston-Chicago
   cityDistances[0][2] = 2960; // Boston-Los Angeles
   cityDistances[1][0] = 960;  // Chicago-Boston
   cityDistances[1][1] = 0;
   cityDistances[1][2] = 2011; // Chicago-Los Angeles
   cityDistances[2][0] = 2960; // Los Angeles-Boston
   cityDistances[2][1] = 2011; // Los Angeles-Chicago
   cityDistances[2][2] = 0;

   strcpy(cityNames[0], "Boston");
   strcpy(cityNames[1], "Chicago");
   strcpy(cityNames[2], "Los Angeles");

   currPath = (int*)malloc(sizeof(int) * NUM_CITIES);
   toVisit = (int*)malloc(sizeof(int) * NUM_CITIES);
   haveBeen = (bool*)malloc(sizeof(bool) * NUM_CITIES);

   toVisit[0] = 0;
   toVisit[1] = 1;
   toVisit[2] = 2;

   haveBeen[0] = false;
   haveBeen[1] = false;
   haveBeen[2] = false;

   // Explore different paths
   TravelPaths(currPath, toVisit, haveBeen, 0);
```

```
    return 0;
}
```

```
Boston   Chicago   Los Angeles   = 2971
Boston   Los Angeles   Chicago   = 4971
Chicago   Boston   Los Angeles   = 3920
Chicago   Los Angeles   Boston   = 4971
Los Angeles   Boston   Chicago   = 3920
Los Angeles   Chicago   Boston   = 2971
```


PARTICIPATION ACTIVITY 13.7.4: Recursive exploration.

1) You wish to generate all possible 3-letter subsets from the letters in an N-letter word (N>3). Which of the above recursive functions is the closest?
   - ShoppingBagCombinations
   - ScrambleLetters
   - main()

Exploring further:

- Recursive Algorithms from khanacademy.org

13.8 Stack overflow

Recursion enables an elegant solution to some problems. But, for large problems, deep recursion can cause memory problems. Part of a program's memory is reserved to support function calls. Each function call places a new ***stack frame*** on the stack, for local parameters, local variables, and more function items. Upon return, the frame is deleted.

Deep recursion could fill the stack region and cause a ***stack overflow***, meaning a stack frame extends beyond the memory region allocated for stack, Stack overflow usually causes the program to crash and report an error like: segmentation fault, access violation, or bad access.

PARTICIPATION ACTIVITY 13.8.1: Recursion causing stack overflow.

Animation captions:

1. Deep recursion may cause stack overflow, causing a program to crash.

The animation showed a tiny stack region for easy illustration of stack overflow.

The size of parameters and local variables results in a larger stack frame. Large vectors, arrays, or strings declared as local variables, or passed by copy, can lead to faster stack overflow.

A programmer can estimate recursion depth and stack size to determine whether stack overflow might occur. Sometime a non-recursive algorithm must be developed to avoid stack overflow.

PARTICIPATION ACTIVITY 13.8.2: Stack overflow.

1) A memory's stack region can store at most one stack frame.
   - ○ True
   - ○ False

2) The size of the stack is unlimited.
   - ○ True
   - ○ False

3) A stack overflow occurs when the stack frame for a function call extends past the end of the stack's memory.
   - ○ True
   - ○ False

4) The following recursive function will result in a stack overflow.

```
int RecAdder(int inValue) {
   return RecAdder(inValue + 1);
}
```

   - ○ True
   - ○ False

13.9 C example: Recursively output permutations

PARTICIPATION ACTIVITY 13.9.1: Recursively output permutations.

The below program prints all permutations of an input string of letters, one permutation per line. Ex: The six permutations of "cab" are:

```
cab
cba
acb
abc
bca
bac
```

Below, the PermuteString function works recursively by starting with the first character and permuting the remainder of the string. The function then moves to the second character and permutes the string consisting of the first character and the third through the end of the string, and so on.

1. Run the program and input the string "cab" (without quotes) to see that the above output is produced.
2. Modify the program to print the permutations in the opposite order, and also to output a permutation count on each line.
3. Run the program again and input the string cab. Check that the output is reversed.
4. Run the program again with an input string of abcdef. Why did the program take longer to produce the results?

```
#include <stdio.h>
#include <string.h>

// FIXME: Use a static variable to count permutations. Why must it be static?

void RemoveFromIndex(char* origString, int remLoc);
void InsertAtIndex(char* origString, char* addChar, int addLoc);

void PermuteString(char* remainLetters, char* permutedLetters) {
   char tmpString[2] = "";
   int i = 0;

   tmpString[1] = '\0';
   if (strlen(remainLetters) == 0) {            // Base case: All letters used
      // FIXME: add count for each permutation
      printf("%s\n", permutedLetters);
   }
   else {                                       // Recursive case: move a letter from
```

```
19                                                  // remaining to permuted letters
```

```
cab
```

Run

PARTICIPATION ACTIVITY 13.9.2: Recursively output permutations (solution).

Below is the solution to the above problem.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  static int permutationCount = 0;                    // For counting permutations
5
6  void RemoveFromIndex(char* origString, int remLoc);
7  void InsertAtIndex(char* origString, char* addChar, int addLoc);
8
9  void PermuteString(char* remainLetters,     // Remaining letters
10                    char* permutedLetters) { // Permuted letters
11    char tmpString[2] = "";
12    int i = 0;
13
14    tmpString[1] = '\0';
15    if (strlen(remainLetters) == 0) {            // Base case: All letters used
16       ++permutationCount;                       // Counting permutations
17       printf("%d) %s\n", permutationCount, permutedLetters);
18    }
19    else {                                       // Recursive case: move a letter from
```

```
cab
abcdef
```

Run