1. **(10%, Coin Tossing)** Write a program that simulates coin tossing. For each toss of the coin the program should print Heads or Tails. Let the program toss the coin 1000 times, and count the number of times each side of the coin appears. Print the results. The program should call a separate function flip that takes no arguments and return 0 for tails and 1 for heads.

[Note: If the program realistically simulates the coin tossing, then each side of the coin should appear approximately half the time for a total of approximately 50 heads and 50 tails.]

1. **(10%, Guess the Number)** Write a C program that plays the game of “guess the number” as follows: Your program chooses the number to be guessed by selecting an integer at random in the range 1 to 1000. The program then types:

|  |
| --- |
| I have a number between 1 and 1000  Can you guess my number?  Please type your first guess. |

The player the types a first guess. The program responds with one of the following:

|  |
| --- |
| 1. Excellent! You guessed the number.   Would you like to play again (y or n)?   1. Too low! Try again. 2. Too high! Try again. |

If the player’s guess is incorrect, your program should loop until the player finally gets the number. Your program should keep telling the player Too high or Too low to help the player “zero in” on the correct answer.

1. **(10%, Recursive Exponentiation)** Write a recursive function power(base, exponent) that when invoked returns . For example, power(3, 4)= 3\* 3\* 3\* 3. Assume that exponent is an integer greater than or equal to 1. Write a main function to prompt the user to enter the base and exponent and display the result.
2. **(10%, Least Common Multiple)** The *lease common multiple (LCM)* of two integers is the smallest positive integer that is a multiple of both the numbers. Write function lcm that returns the least common multiple of two integers. Write a program that prompts the user to enter two integer numbers and get the LCM by your lcm function. Display the result on the screen.
3. **(Fibonacci)** The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, ….

Begins with the terms 0 and 1 and has the property that each succeeding term is the sum of the two preceding terms.

1. **(15%)** Write a nonrecurrive function fibonacci(n) that calculates the nth Fibonacci number. Use unsigned int for the function’s parameter and unsigned long long int for its return type.
2. **(15%)** Write a program to determine the largest Fibonacci number that can be printed on your system.
3. **(15%, Craps Game Modification)** Modify the craps program of Fig. 5.14 to allow wagering. Package as a function the portion of the program that runs one games of craps. Initialize variable **bankBalance** to 1000 dollars. Prompt the player to enter a wager. Use a while loop to check that wager is less than or equal to **bankBalance**, and if not, prompt the user to reenter wager until a valid wager is entered. After a correct wager is entered, run one game of craps. If the player wins, increase bankBalance by wager and print the new bankBalance. If the player loses, decrease bankBalance by wager, print the new balance, check whether bankBalance has become zero, and if so, print the message, “Sorry. You busted!”
4. ***(15%, Towers of Hanoi)*** Every budding computer scientist must grapple with certain classic problems, and the Towers of Hanoi (see the following figure) is one of the most famous of these. Legend has it that in a temple in the Far East, priests are attempting to move a stack of disks from one peg to another. The initial stack had 64 disks threaded onto one peg and arranged from bottom to top by decreasing size. The priests are attempting to move the stack from this peg to a second peg under the constraints that exactly one disk is moved at a time, and at no time may a larger disk be placed above a smaller disk. A third peg is available for temporarily holding the disks. Supposedly the world will end when the priests complete their task, so there is little incentive for us to facilitate their efforts.



Let us assume that the priests are attempting to move the disks from peg 1 to peg 3. We wish to develop an algorithm that will print the precise sequence of disk-to-disk peg transfers. If we were to approach this problem with conventional methods, we would rapidly find ourselves hopelessly knotted up in managing the disks. Instead, if we attack the problem with recursion in mind, it immediately becomes tractable. Moving *n* disks can be viewed in terms of moving only *n* – 1 disks (and hence the recursion) as follows:

a) Move *n* – 1 disks from peg 1 to peg 2, using peg 3 as a temporary holding area.

b) Move the last disk (the largest) from peg 1 to peg 3.

c) Move the *n* – 1 disks from peg 2 to peg 3, using peg 1 as a temporary holding area.

The process ends when the last task involves moving *n* = 1 disk, i.e., the base case. This is accomplished by trivially moving the disk without the need for a temporary holding area.

Write a program to solve the Towers of Hanoi problem. Use a recursive function with four parameters:

a) The number of disks to be moved

b) The peg on which these disks are initially threaded

c) The peg to which this stack of disks is to be moved

d) The peg to be used as a temporary holding area

Your program should print the precise instructions it will take to move the disks from the starting peg to the destination peg. For example, to move a stack of three disks from peg 1 to peg 3, your program should print the following series of moves:

1 3 (This means move one disk from peg 1 to peg 3.)

1 2

3 2

1 3

2 1

2 3

1 3

**Please show your results for moving five disks from peg 1 to peg 3.**