CS1010E: Programming Methodology

Take Home Lab 3: Functions & Recursions

08 Mar 2017

Preliminary: Maximum [#1]

Problem Description

This is a simple exercise of finding the maximum of several numbers given a function to find a maximum between two numbers. Assume the existence of the following function (*code them on your program*): `int max(int a, int b) { return a>b ? b : a; }`.

Final Objective

Given **four (4)** `integer` numbers a, b, c, and d, find the maximum of the four numbers by calling `int max(int a, int b);` at most **three (3)** times.

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

▷ -1000 $\leq$ `a, b, c, d` $\leq$ 1000 (*the four numbers*)

Tasks

The problem is split into 1 task(s). In the sample run, please note the following:

- ↩ is the *invisible* `[newline]` character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.

Task 1/1

Write a program that reads **four (4)** `integer` numbers and prints the largest of the four numbers.

Note:

- Obviously, this problem can be solved without using function
- The point of this practice is to solve it using function composition

Sample Run:

Inputs:	Outputs:
`-1000 0 -1 1000`	`1000↩`

Save your program in the file named `maximum1.c`. Submit your program to CodeCrunch.

Easy: ? [#2]

Problem Description

"In mathematics, the Minkowski question mark function (or the slippery devil's staircase), denoted by ?(x), is a function possessing various unusual fractal properties, defined by Hermann Minkowski (1904,pages 171172). It maps quadratic irrationals to rational numbers on the unit interval, via an expression relating the continued fraction expansions of the quadratics to the binary expansions of the rationals, given by Arnaud Denjoy in 1938. In addition, it maps rational numbers to dyadic rationals, as can be seen by a recursive definition closely related to the SternBrocot tree." – Wikipedia

While that may sound complicated, there is a simple rule that can be used to compute the ?(x) function. The recursive rule is defined in Formula 1 with two base cases defined in Formula 2 & 3.

$$?\left(\frac{p+r}{q+s}\right) = \frac{1}{2} \times \left(?\left(\frac{p}{q}\right) + ?\left(\frac{r}{s}\right)\right) \tag{1}$$

$$?\left(\frac{0}{1}\right) = 0 \tag{2}$$

$$?\left(\frac{1}{1}\right) = 1 \tag{3}$$

In all three formulas above, the value of p, q, r, and s are **integer** number. Note that ?(x) function is also defined for non-rational numbers, but we are not going to dwell into that nightmarish mess.

Final Objective

Given two numbers a and b, compute the value of $?\left(\frac{a}{b}\right)$ up to 2 decimal places.

Example

$?\left(\frac{2}{3}\right) = 0.75.$

$?\left(\frac{1}{2}\right) = 0.50.$

$?\left(\frac{1}{3}\right) = 0.25.$

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

- ▷ $1 \leq a \leq b$ (*the numerator*)
- ▷ $a \leq b \leq 100$ (*the denominator*)

Tasks

The problem is split into 1 task(s). In the sample run, please note the following:

- ↩ is the *invisible* [**newline**] character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.

Task 1/1

Write a program to read **two (2)** `integer` numbers p and q and print the value of ? $\left(\frac{p}{q}\right)$ up to **two (2)** decimal places.

Sample Run:

Inputs:	Outputs:
`2 3`	`0.75↩`

Save your program in the file named `minkowski1.c`. Submit your program to CodeCrunch.

Easy: Meta-Fibonacci [#3]

Problem Description

"The Hofstadter Q sequence is defined as follows:

$$Q(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ Q(n - Q(n-1)) + Q(n - Q(n-2)) & \text{if } n > 2 \end{cases} \tag{4}$$

"The first few terms of the sequence are: [1, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 8, 8, 8, 10, 9, 10, 11, 11, 12, ...] (sequence A005185 in the OEIS). Hofstadter named the terms of the sequence "Q numbers"; thus the Q number of 6 is 4. The presentation of the Q sequence in Hofstadter's book is actually the first known mention of a meta-Fibonacci sequence in literature.

"While the terms of the Fibonacci sequence are determined by summing the two preceding terms, the two preceding terms of a Q number determine how far to go back in the Q sequence to find the two terms to be summed. The indices of the summation terms thus depend on the Q sequence itself." – Wikipedia

We will implement the meta-fibonacci sequence as defined by Douglas Hofstadter. However, as a **twist**, you are not to print the *just* result but also the *number of times* the function $Q(n)$ is called for any value of n.

Final Objective

Given an `integer` number n, print the value of $Q(n)$ and the number of times the function $Q(n)$ is called.

Example

For $Q(1)$ and $Q(2)$, the result is 1. Furthermore, the number of times $Q(n)$ is called is exactly **once** for both. Consider $Q(3)$, we get the following call sequence

- $Q(3)$
  - $Q(n-1) = Q(2)$
    - `return 1`
  - $Q(n - Q(n-1)) = Q(n-1) = Q(2)$
    - `return 1`
  - $Q(n-2) = Q(1)$
    - `return 1`
  - $Q(n - Q(n-2)) = Q(n-1) = Q(2)$
    - `return 1`
  - `return 1 + 1 = 2`

Based on the visualization, the number of times the function is called is **five (5)**.

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

- ▷ $2 \leq$ `n` ≤ 30 (*the value of* n)
- ▷ You are guaranteed that the number of times $Q(n)$ is called fits in an `integer`

Tasks

The problem is split into 2 task(s). In the sample run, please note the following:

- ↩ is the *invisible* `[newline]` character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.

Task 1/2

Write a program to read a single **integer** number n and print $Q(n)$.

Sample Run:

Inputs:	Outputs:
5	3↩

Sample Run:

Inputs:	Outputs:
7	5↩

Save your program in the file named `metafib1.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp metafib1.c metafib2.c
```

Task 2/2

Write a program to read a single **integer** number n and print $Q(n)$ and the number of times $Q(n)$ is called. *Hint: use a global variable or* **integer** *pointer.*

Sample Run:

Inputs:	Outputs:
5	3 25↩

Sample Run:

Inputs:	Outputs:
7	5 93↩

Save your program in the file named `metafib2.c`. Submit your program to CodeCrunch.

Medium: Auto-Indentation [#4]

Problem Description

"The indent features of Vim are very helpful for indenting source code. This tip discusses settings that affect indentation. These settings mostly affect the automatic indentation which Vim inserts as you type, but also can be triggered manually with the = operator, so that you can easily Fix indentation in your buffer." – vim.wikia.com

Let us consider a simplified form of auto-indentation in VIM. In this problem, the input is the entire program as a single line. To simplify discussion, we start with terminology.

- Tabs: Tabs refer to the [space] at the beginning of a line
- Tab Number: The number of [space] in tabs
- Open Braces: Curly bracket of the form {
- Close Braces: Curly bracket of the form }

The indentation rule is described as follows:

- The initial tab number is 0
- After every read of open braces:
  - Print the open braces
  - Increase the tab number by 2
  - Print a [newline]
  - Print tabs with the new tab number
- After every read of close braces:
  - Decrease the tab number by 2
  - Print a [newline]
  - Print tabs with the new tab number
  - Print the close braces
  - Print a [newline]
  - Print tabs with the new tab number

In this problem, you are required to read **character**. Processing a **character** has been discussed in Lecture Notes 3 and the summary is listed below:

- Variables: **character** can be stored as either **char** or **int**
- Initializing: **character** *constant* is represented with a single quote (*e.g.,* `char ch = 'a';`)
- Reading: Reading **character** from user input can be done using `getchar` method (*e.g.,* `ch = getchar()`)
  - This will read all **character** *one-by-one* including a [space] and a [newline]
- Processing: Since **character** can be stored as **int**, any operation that can be done to **int** can be done to **character**
  - To compare **character** to a constant: `ch >= 'a' && ch <= 'z'` (*i.e., check if the character is lowercase*), `ch == 'z'` (*i.e., check if the character is* `z`)
- Printing: Printing **character** to the output can be done using `putchar` method (*e.g.,* `putchar(ch);`)

Final Objective

Given a sequence of **character** that *ends* with a [newline], perform auto-indentation with the rule described above.

Example

Consider the code `int main(void) { printf("Hello World\n"); return 0; }`, after auto-indentation, the code will be:

```
int main(void) {
  printf("Hello World\n"); return 0;
}
```

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

- ▷ $0 \leq 1 \leq \infty$ (*the length of the sequence*)
- ▷ The sequence will only contain *printable* characters
- ▷ The sequence will be terminated by a single `[newline]` at the end

Tasks

The problem is split into 3 task(s). In the sample run, please note the following:

- ↩ is the *invisible* `[newline]` character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.

Task 1/3

Write a program that reads the sequence print the sequence back.

Sample Run:

Inputs:

```
int main(void) { printf("Hello World\n"); return 0; }
```

Outputs:

```
int main(void) { printf("Hello World\n"); return 0; }↩
```

Save your program in the file named `indent1.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp indent1.c indent2.c
```

Task 2/3

Write a program that reads the sequence print the sequence such that a `[newline]` is printed *after* every open braces and a `[newline]` is printed *before* and *after* every close braces.

Hint: use the equality comparator

Sample Run:

Inputs:

```
int main(void) { printf("Hello World\n"); return 0; }
```

Outputs:

```
int main(void) {↩
 printf("Hello World\n"); return 0; ↩
}↩
```

Save your program in the file named `indent2.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 3*), copy your program using the following command:

```
cp indent2.c indent3.c
```

Task 3/3
Write a program that reads the sequence print the sequence with auto-indentation.
Hint: use the equality comparator and count the braces
Sample Run:
Inputs:

```
int main(void) { printf("Hello World\n"); return 0; }
```

Outputs:

```
int main(void) {↩
  printf("Hello World\n"); return 0; ↩
}↩
```

Save your program in the file named `indent3.c`. Submit your program to CodeCrunch.

Medium: Geometric Series [#5]

Problem Description

"In mathematics, a geometric series is a series with a constant ratio between successive terms. For example, the series $1 + 2 + 4 + 8 + 16 + \ldots$ is geometric, because each successive term can be obtained by multiplying the previous term by 2.

"Geometric series are among the simplest examples of infinite series with finite sums, although not all of them have this property. Historically, geometric series played an important role in the early development of calculus, and they continue to be central in the study of convergence of series. Geometric series are used throughout mathematics, and they have important applications in physics, engineering, biology, economics, computer science, queueing theory, and finance." – Wikipedia

The geometric series can be represented as follows:

$$G(x, n) = 1 + x + x^2 + x^3 + \ldots + x^n \tag{5}$$

It has a *closed-form* formula of:

$$G(x, n) = \frac{1 - x^{n+1}}{1 - x} \tag{6}$$

However, as the value of n increases, $G(x, n)$ increases *exponentially*. After a certain values of n, the sum is larger than the range for **integer**. To limit the result within the range of **integer**, we will use the `modulo` operator. We then modify the Formula 5 to the following:

$$G(x, n, m) = (1 + x + x^2 + x^3 + \ldots + x^n) \bmod m \tag{7}$$

While it may not *immediately* solve the problem since $1 + x + x^2 + x^3 + \ldots + x^n$ or even x^n can indeed be pretty large, there are certain properties of *modular arithmetic* that allows us to make the computation restricted within the range of **integer**. The two theorems from modular arithmetic is stated below:

$$(x + y) \bmod m = ((x \bmod m) + (y \bmod m)) \bmod m \tag{8}$$
$$(x \times y) \bmod m = ((x \bmod m) \times (y \bmod m)) \bmod m \tag{9}$$

Beyond this point, the information is not needed EXCEPT for those seeking challenge.
To solve the problem efficiently, we can abuse recursion. Firstly, note that x^n can be computed efficiently via the following recursive formula:

$$x^n = \begin{cases} 0 & \text{if } n = 0 \\ x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} & \text{if } n \text{ is even} \\ x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} \times x & \text{if } n \text{ is odd} \end{cases} \tag{10}$$

To make the recursion works efficiently, store the temporary value $x^{\lfloor n/2 \rfloor}$ so that you do not call the recursion twice. Next, let $S = H(x, n) = x + x^2 + x^3 + \ldots + x^n$. Then, $x^n \times S = x^n \times (x + x^2 + x^3 + \ldots + x^n)$. This means $S + (x^n \times S) = H(x, 2n)$. Thus, we have computed $H(x, 2n)$ while bypassing the computation for $H(x, n)$ to $H(x, 2n)$. This is the same idea as efficient x^n recursion above. Lastly, $G(x, n) = 1 + H(x, n)$.

Look at it another way, the following recursive formula holds:

$$H(x, n) = \begin{cases} 0 & \text{if } n = 0 \\ x & \text{if } n = 1 \\ H(x, \lfloor n/2 \rfloor) + \left(H(x, \lfloor n/2 \rfloor) \times x^{\lfloor n/2 \rfloor}\right) & \text{if } n \text{ is even} \\ H(x, \lfloor n/2 \rfloor) + \left(H(x, \lfloor n/2 \rfloor) \times x^{\lfloor n/2 \rfloor}\right) + x^n & \text{if } n \text{ is odd} \end{cases} \tag{11}$$

Final Objective

Given **three (3)** `integer` numbers x, n, and m, compute $G(x, n, m)$.

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

- ▷ $0 \leq \texttt{x} \leq 2^{14}$ (*the base*)
- ▷ $1 \leq \texttt{n} \leq 2^{10}$ (*the exponent*)
- ▷ $2 \leq \texttt{m} \leq 2^{14}$ (*the modulo*)

Tasks

The problem is split into 3 task(s). In the sample run, please note the following:

- ↩ is the *invisible* `[newline]` character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.

Task 1/3

Write a program to read **three (3)** `integer` numbers x, n, and m and print $x \times n$ mod m.

Sample Run:

Inputs:	Outputs:
`4 10 7`	`5↩`

Save your program in the file named `geometric1.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 2*), copy your program using the following command:
`cp geometric1.c geometric2.c`

Task 2/3

Write a program to read **three (3)** `integer` numbers x, n, and m and print x^n mod m.

Sample Run:

Inputs:	Outputs:
`4 10 7`	`4↩ | 4^10 = 1048576`

Save your program in the file named `geometric2.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 3*), copy your program using the following command:
`cp geometric2.c geometric3.c`

Task 3/3

Write a program to read **three (3)** `integer` numbers x, n, and m and print $G(x, n, m)$.

Sample Run:

Inputs:	Outputs:
`4 10 7`	`5↩`

Save your program in the file named `geometric3.c`. Submit your program to CodeCrunch.

Bonus Task

Write a program to read **three (3)** `integer` numbers x, n, and m and print $G(x, n, m)$ *efficiently*.

Sample Run:

Inputs:	Outputs:
`4 10 7`	`5↩`

Save your program in the file named `geometric4.c`. Submit your program to CodeCrunch.

Hard: Tower of Hanoi [#6]

Problem Description

"Although the three-peg version has a simple recursive solution as outlined above which has long been known, the optimal solution for the Tower of Hanoi problem with four pegs has not been verified until 2014, and in the case of more than four pegs the optimal solution is still an open problem." – Wikipedia

You have learned about the Tower of Hanoi problem and its recursive solution. We are now ready to generalize this to **four (4)** or more pegs. *Consider 4 pegs and n discs.* We separate this n discs into 2 parts: the top k discs and the bottom l discs.. Therefore, we have $k + l = n$.

Let's move the top k discs into one of the spare pegs. This peg will be occupied by k discs of decreasing size. We can move this top k discs by utilizing all the 4 pegs.

After we moved the top k discs, to move the bottom l discs into the destination peg, we cannot use the pegs with the top k discs. Thus, the entire problem has been reduced to move this bottom l discs using the standard movement of tower of hanoi. Lastly, we can then utilize all the 4 pegs to move the top k discs to the final peg. These steps are shown in Diagram 1.

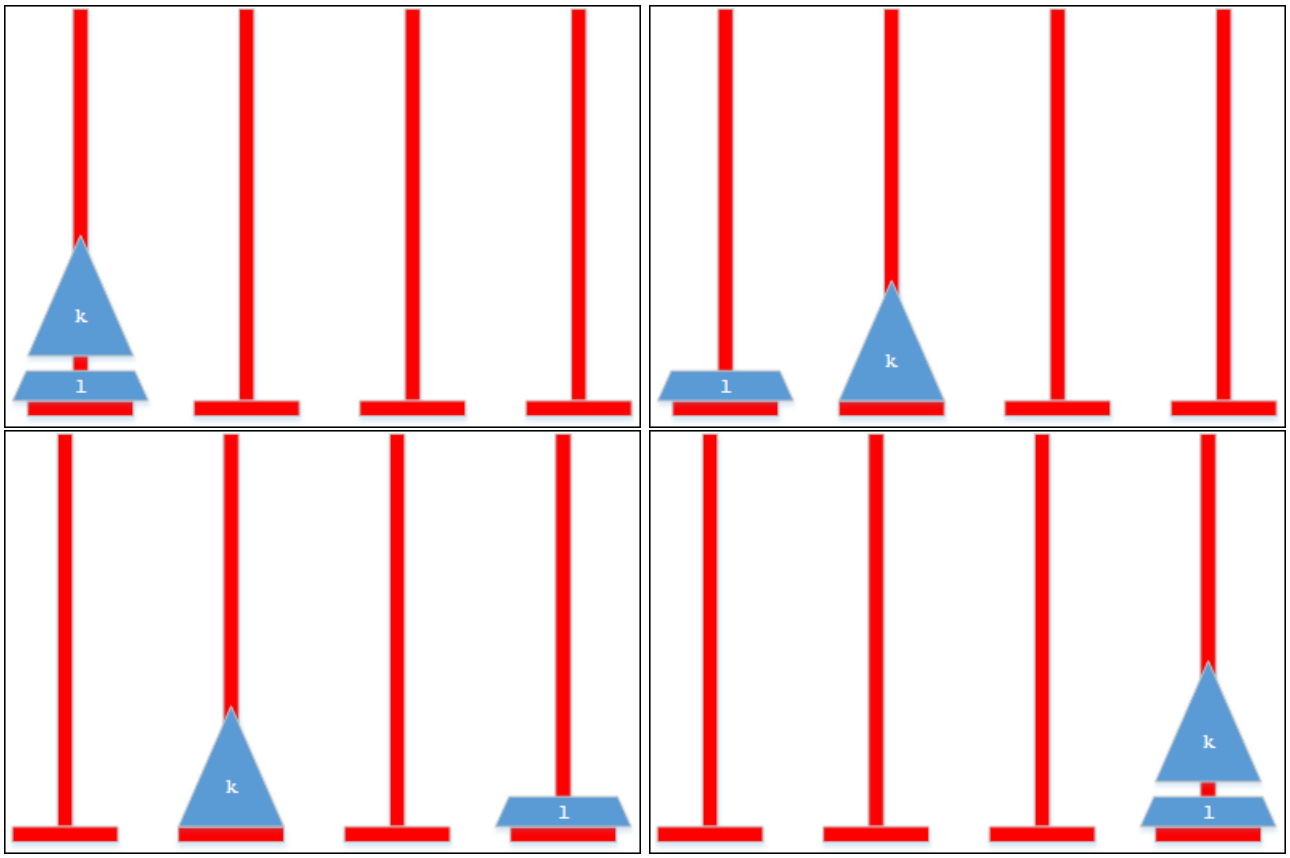

Diagram 1: Solving Tower of Hanoi Puzzle. Step 1: Partition (*top-left*). Step 2: Move top k with 4 pegs (*top-right*). Step 3: Move bottom l with 3 pegs (*bottom-left*). Step 4: Move top k with 4 pegs (*bottom-right*).

Once you have understood how to solve it, we want to find the *minimum* number of moves required to solve the Reve's Puzzle with 4 or more pegs. The generalization to p pegs is similar, but instead of moving the middle l using 3 pegs, you can move it with $p - 1$ pegs. Once you have reached $p = 3$ pegs, then it is reduced to the standard tower of hanoi problem with a known number of minimum moves. According to the algorithm above, we will then need to find the *optimal partition* k and l such that $k + l = n$ and to move the top k discs using p pegs and bottom l using $p - 1$ pegs as described above is the *minimum*. The easiest is to *repeat the recursion* over the possible partition.

Final Objective

Given **two (2)** `integer` numbers n and p corresponding to the number of discs and pegs, find the minimum number of moves to solve the tower of hanoi for n discs and p pegs according to the algorithm above.

Example

The minimum for $n = 4$ and $p = 3$ is 15 moves.
The minimum for $n = 16$ and $p = 4$ is 161 moves.
The minimum for $n = 21$ and $p = 4$ is 321 moves.
The minimum for $n = 21$ and $p = 5$ is 127 moves.

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

- ▷ $0 \leq \mathsf{n} \leq 30$ (*the number of discs*)
- ▷ $3 \leq \mathsf{p} \leq 9$ (*the number of pegs*)

Tasks

The problem is split into 5 task(s). In the sample run, please note the following:

- ↩ is the *invisible* `[newline]` character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.

Task 1/5

Write a program to read **two (2)** `integer` numbers n and p and print the number back.

Sample Run:

Inputs:	Outputs:
`4 3`	`4 3↩`

Sample Run:

Inputs:	Outputs:
`21 5`	`21 5↩`

Save your program in the file named `hanoi1.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp hanoi1.c hanoi2.c
```

Task 2/5

Write a program to read **two (2)** `integer` numbers n and p and print the value $2^n - 1$ if $p = 3$. Otherwise, it prints n.

Sample Run:

Inputs:	Outputs:
`4 3`	`15↩`

Sample Run:

Inputs:	Outputs:
`21 5`	`21↩`

Save your program in the file named `hanoi2.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 3*), copy your program using the following command:

```
cp hanoi2.c hanoi3.c
```

Task 3/5

Write a program to read **two (2)** `integer` numbers n and p and print the value $2^n - 1$ if $p = 3$. Otherwise, it sets $n' = \lfloor n/2 \rfloor$ and repeats the entire process with (n', p) and another one with $(n - n', p - 1)$.

Sample Run:

Inputs:	Outputs:
`4 3`	`15↩`

Sample Run:

Inputs:	Outputs:
`21 5`	`10↩ | n' = 21/2 = 10, p = 5` `11↩ | n-n' = 11, p = 4`

Sample Run:

Inputs:	Outputs:
`21 4`	`10↩  | n' = 21/2 = 10, p = 4` `2047↩ | n-n' = 11, p = 3 -> 2^11 - 1`

Save your program in the file named `hanoi3.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 4*), copy your program using the following command:

```
cp hanoi3.c hanoi4.c
```

Task 4/5

Write a program to read **two (2)** `integer` numbers n and p and print the value $2^n - 1$ if $p = 3$. Otherwise, it sets $n' = \lfloor n/2 \rfloor$ and repeats *recursively* the entire process with (n', p) and another one with $(n - n', p - 1)$ until either $p = 3$, $n' = 0$, or $n' = 1$. In all steps, print only when $p = 3$.

Sample Run:

Inputs:	Outputs:
`4 3`	`15↩`

Sample Run:

Inputs:	Outputs:

`21 5`	`3↩  | n' = 2, p = 3`
	`7↩  | n' = 3, p = 3`
	`7↩  | n' = 3, p = 3`
	`63↩ | n' = 6, p = 3`

Sample Run:

Inputs:	Outputs:

`21 4`	`1↩  | n' = 1,  p = 3`
	`7↩  | n' = 3,  p = 3`
	`31↩ | n' = 5,  p = 3`
	`2047↩ | n' = 11, p = 3`

Save your program in the file named `hanoi4.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 5*), copy your program using the following command:

```
cp hanoi4.c hanoi5.c
```

Task 5/5

Write a program to read **two (2)** `integer` numbers n and p and print the *presumed* minimum number of steps to solve the tower of hanoi problem with n discs and p pegs using the method discussed above.

Sample Run:

Inputs:	Outputs:
4 3	15↩

Sample Run:

Inputs:	Outputs:
16 4	161↩

Sample Run:

Inputs:	Outputs:
21 4	321↩

Sample Run:

Inputs:	Outputs:
21 5	127↩

Save your program in the file named `hanoi5.c`. Submit your program to CodeCrunch.