CS1010E: Programming Methodology

Take Home Lab 2: Selection & Repetition

15 Feb 2017

Preliminary: FooBar [#1]

Problem Description

FooBar is a simple exercise if printing `"foo"` when a number is divisible by 3, `"bar"` when a number is divisible by 5, and `"foobar"` when a number is divisible by both.

Final Objective

Given a list of number, print `"foo"` is the number is divisible by 3, print `"bar"` if the number is divisible by 5, and print `"foobar"` is the number is divisible by both. On any other number, print nothing.

Assumptions

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

- ▷ $3 \leq$ `N` ≤ 1000 (*the number of element in the list*)
- ▷ $1 \leq$ `val` $\leq 2^{30}$ (*the value of the number in the list*)

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 a list of **`integer`** and prints the FooBar output. Print each output line by line except when there is nothing to print. The first line of input is `N`, the number of element in the list followed by `N` lines of **`integer`** numbers representing the value. Sample Run:

Inputs:

```
5   | N: number of element      +-----------------> foo↩
1   | 1st element -> nothing    | +---------------> bar↩
3   | 2nd element -> foo -------+ | +-------------> foobar↩
5   | 3rd element -> bar ---------+ |
30  | 4th element -> foobar --------+
2   | 5th element -> nothing
```

Outputs:

Save your program in the file named `foobar1.c`. Submit your program to CodeCrunch.

Easy: Prime Number [#2]

Problem Description

"A prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself. A natural number greater than 1 that is not a prime number is called a composite number. For example, 5 is prime because 1 and 5 are its only positive integer factors, whereas 6 is composite because it has the divisors 2 and 3 in addition to 1 and 6. The fundamental theorem of arithmetic establishes the central role of primes in number theory: any integer greater than 1 can be expressed as a product of primes that is unique up to ordering. The uniqueness in this theorem requires excluding 1 as a prime because one can include arbitrarily many instances of 1 in any factorization, e.g., 3, 1×3, 1×1×3, etc. are all valid factorizations of 3."
– Wikipedia

We will categorize numbers based on whether or not the number is prime or composite.

Final Objective

Given a number, print `"prime"` if the number is a prime number. Otherwise print `"composite"`.

Example

The first 10 prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29.

Assumptions

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

▷ $2 \leq$ `num` ≤ 1000 (*the number to be checked*)

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 an **`integer`** number and prints whether it is a prime or composite number.

Sample Run:

Inputs:	Outputs:
23	prime↩

Sample Run:

Inputs:	Outputs:
4	composite↩

Sample Run:

Inputs:	Outputs:
29	prime↩

Sample Run:

Inputs:	Outputs:
1000	composite↩

Save your program in the file named `prime1.c`. Submit your program to CodeCrunch.

Easy: Collatz Conjecture [#3]

Problem Description

"The Collatz conjecture is a conjecture in mathematics named after Lothar Collatz. The conjecture is also known as the 3n + 1 conjecture, the Ulam conjecture (after Stanislaw Ulam), Kakutani's problem (after Shizuo Kakutani), the Thwaites conjecture (after Sir Bryan Thwaites), Hasse's algorithm (after Helmut Hasse), or the Syracuse problem; the sequence of numbers involved is referred to as the hailstone sequence or hailstone numbers (because the values are usually subject to multiple descents and ascents like hailstones in a cloud), or as wondrous numbers.

"The conjecture can be summarized as follows. Take any positive integer n. If n is even, divide it by 2 to get n / 2. If n is odd, multiply it by 3 and add 1 to obtain 3n + 1. Repeat the process (which has been called "Half Or Triple Plus One", or HOTPO) indefinitely. The conjecture is that no matter what number you start with, you will always eventually reach 1." – Wikipedia

In short, the rules are presented below in Formula 1.

$$n_{i+1} = \begin{cases} \frac{n_i}{2} & \text{if } n_i \text{ is } \textit{even} \\ 3 \times n_i + 1 & \text{if } n_i \text{ is } \textit{odd} \end{cases} \tag{1}$$

Continuously find n_{i+1} until it reaches 1 and record the number of steps required to reach 1.

Final Objective

Given a starting value of n, find the number of steps to reach 1 using the Formula 1.

Example

Start with number 3. The sequence of numbers obtained are: $3 \mapsto 10 \mapsto 5 \mapsto 16 \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1$. The number of steps required is equal to the number of "$\mapsto$" symbol: 7.

Assumptions

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

- ▷ $2 \leq n \leq 2^{30}$ (*the starting number*)
- ▷ You are guaranteed that all *intermediate* numbers never exceed the limit of **integer**

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 a single **integer** number and print 1 if the number is *even*. Otherwise, print 0.

Sample Run:

Inputs:	Outputs:
3	0↩

Sample Run:

Inputs:	Outputs:
2	1↩

Save your program in the file named `collatz1.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp collatz1.c collatz2.c
```

Task 2/3

Write a program to read a single **integer** number and print the next number based on Formula 1.

Sample Run:

Inputs:	Outputs:
3	10↩

Sample Run:

Inputs:	Outputs:
2	1↩

Save your program in the file named `collatz2.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 3*), copy your program using the following command:

```
cp collatz2.c collatz3.c
```

Task 3/3

Write a program to read a single **integer** number and print the number of steps for repeated application of Formula 1 to reach 1.

Sample Run:

Inputs:	Outputs:
3	7↩

Sample Run:

Inputs:	Outputs:
2	1↩

Save your program in the file named `collatz3.c`. Submit your program to CodeCrunch.

Medium: Cosine [#4]

Problem Description

"In mathematics, the trigonometric functions (also called the circular functions) are functions of an angle. They relate the angles of a triangle to the lengths of its sides. Trigonometric functions are important in the study of triangles and modeling periodic phenomena, among many other applications.

"The most familiar trigonometric functions are the sine, cosine, and tangent. In the context of the standard unit circle (a circle with radius 1 unit), where a triangle is formed by a ray starting at the origin and making some angle with the x-axis, the sine of the angle gives the length of the y-component (the opposite to the angle or the rise) of the triangle, the cosine gives the length of the x-component (the adjacent of the angle or the run), and the tangent function gives the slope (y-component divided by the x-component). More precise definitions are detailed below. Trigonometric functions are commonly defined as ratios of two sides of a right triangle containing the angle, and can equivalently be defined as the lengths of various line segments from a unit circle. More modern definitions express them as infinite series or as solutions of certain differential equations, allowing their extension to arbitrary positive and negative values and even to complex numbers." – Wikipedia

The value of $cos(x)$ can be computed using Taylor series expansion as shown in Formula 2

$$cos(x) = \sum_{n=0}^{m} \left((-1)^n \times \frac{x^{2n}}{(2n)!} \right) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \tag{2}$$

When $m = \infty$, then we have the actual value of $cos(x)$. Otherwise, this value is an approximation up to m number of steps.

Final Objective

Given the value of x and m, compute $cos(x)$ on m step approximation, up to **three (3)** decimal places.

Example

$cos(2.0) = -0.416$ but the first step approximation is -1.000 and second step approximation is -0.333.

Assumptions

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

- ▷ $0 \leq$ `x` ≤ 2 (*the angle in radians*)
- ▷ $0 \leq$ `m` ≤ 100 (*the approximation steps*)

Restrictions

The following restriction(s) is/are imposed on the solution:

- ▷ You are not allowed to use <**math.h**> library

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 one **real** number x and one **integer** number m and print the number back such that the **real** number is printed up to **three (3)** d.p. **real** number.

Sample Run:

Inputs:	Outputs:
2 2	2.000 2↩

Save your program in the file named `cosine1.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp cosine1.c cosine2.c
```

Task 2/5

Write a program to read one **real** number x and one **integer** number m and print m number of lines corresponding to $(-1)^i$ for $i = 0, ..., m$ in **three (3)** d.p. **real** number.

Sample Run:

Inputs:

```
2 4
```

Outputs:

```
1.000↩
-1.000↩
1.000↩
-1.000↩
1.000↩
```

Save your program in the file named `cosine2.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 3*), copy your program using the following command:

```
cp cosine2.c cosine3.c
```

Task 3/5

Write a program to read one **real** number x and one **integer** number m and print m number of lines corresponding to $(-1)^i \times x^{2i}$ for $i = 0, ..., m$ in **three (3)** d.p. **real** number.

Sample Run:

Inputs:

```
2 4
```

Outputs:

```
1.000↩    | 2.00 ^ 0
-4.000↩   | 2.00 ^ 2
16.000↩   | 2.00 ^ 4
-64.000↩  | 2.00 ^ 6
256.000↩  | 2.00 ^ 8
```

Save your program in the file named `cosine3.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 4*), copy your program using the following command:

```
cp cosine3.c cosine4.c
```

Task 4/5

Write a program to read one **real** number x and one **integer** number m and print m number of lines corresponding to $(-1)^i \times \dfrac{x^{2i}}{(2i)!}$ for $i = 0, ..., m$ in **three (3)** d.p. **real** number.

Sample Run:

Inputs:

```
2 4
```

Outputs:

```
1.000↩    | 2.00 ^ 0
-2.000↩   | 2.00 ^ 2 / 2!
0.667↩    | 2.00 ^ 4 / 4!
-0.089↩   | 2.00 ^ 6 / 6!
0.006↩    | 2.00 ^ 8 / 8!
```

Save your program in the file named `cosine4.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 5*), copy your program using the following command:

```
cp cosine4.c cosine5.c
```

Task 5/5

Write a program to read one `real` number x and one `integer` number m and one `real` number of lines corresponding to $cos(x)$ up to m step approximation in **three (3)** d.p. `real` number.

Sample Run:

Inputs:	Outputs:
`2 0`	`1.000↩`

Sample Run:

Inputs:	Outputs:
`2 1`	`-1.000↩`

Sample Run:

Inputs:	Outputs:
`2 2`	`-0.333↩`

Sample Run:

Inputs:	Outputs:
`2 3`	`-0.422↩`

Sample Run:

Inputs:	Outputs:
`2 4`	`-0.416↩`

Save your program in the file named `cosine5.c`. Submit your program to CodeCrunch.

Hard: Normalized Mean [#5]

Problem Description

Given n numbers t_1, t_2, ..., t_n, we define normalized mean using Formula 3.

$$\bar{t}_{mean} = \sum_{i=1}^{n} \left(\frac{\bar{t}_i}{n} \right) \tag{3}$$

Where each of $\bar{t}_i$ is defined in Formula 4

$$\bar{t}_i = \frac{t_i - t_{min}}{t_{max} - t_{min}} \tag{4}$$

Such that t_{max} is the largest number from all t_1, t_2, ..., t_n and t_{min} is the smallest number from all t_1, t_2, ..., t_n. In other words, $t_{max} = max\{t_1, t_2, ..., t_n\}$ and $t_{min} = min\{t_1, t_2, ..., t_n\}$. Unfortunately, you do not know how many numbers there are.

Final Objective

Given an *arbitrary* number of **integer** numbers, compute the normalized mean from all the numbers up to **two (2)** decimal places.

Example

The normalized mean from the sequence: 1, 2, 3, 4, 5 = 0.50.

Assumptions

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

- ▷ -1000 ≤ `t_i` ≤ 1000 (*the number in the arbitrary sequence*)
- ▷ You are guaranteed that there is at least one non-zero number in the sequence

Restrictions

The following restriction(s) is/are imposed on the solution:

- ▷ You cannot use array (*since you do not know how many numbers are there in the first place*).

Tasks

The problem is split into 6 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 1/6

Write a program to read an arbitrary number of **integer** and print the number back as **two (2)** d.p. **real** numbers line by line. *Hint: use* `while(scanf("%d", &t_i) != EOF) { /* code */ }` *for input reading.*

Sample Run:

Inputs:	Outputs:
1	1.00↩
2	2.00↩
3	3.00↩
4	4.00↩
5	5.00↩

Save your program in the file named `normalized1.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp normalized1.c normalized2.c
```

Task 2/6
Write a program to read an arbitrary number of **integer** and print the number of **integer** read.
Sample Run:

Inputs:	Outputs:
1 2 3 4 5	5↩

Save your program in the file named `normalized2.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 3*), copy your program using the following command:
`cp normalized2.c normalized3.c`

Task 3/6
Write a program to read an arbitrary number of **integer** and print the largest number of **integer** read as **two (2)** d.p. **real** number.
Sample Run:

Inputs:	Outputs:
1 2 3 4 5	5.00↩

Save your program in the file named `normalized3.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 4*), copy your program using the following command:
`cp normalized3.c normalized4.c`

Task 4/6
Write a program to read an arbitrary number of **integer** and print the smallest number of **integer** read as **two (2)** d.p. **real** number.
Sample Run:

Inputs:	Outputs:
1 2 3 4 5	1.00↩

Save your program in the file named `normalized4.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 5*), copy your program using the following command:
`cp normalized4.c normalized5.c`

Task 5/6
Write a program to read an arbitrary number of **integer** and print the sum of number of **integer** read as **two (2)** d.p. **real** number.
Sample Run:

Inputs:	Outputs:
1 2 3 4 5	15.00↩

Save your program in the file named `normalized5.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 6*), copy your program using the following command:
`cp normalized5.c normalized6.c`

Task 6/6
Write a program to read an arbitrary number of **integer** and print the normalized mean of number of **integer** read as **two (2)** d.p. **real** number. *Hint: rearrange the equation.*
Sample Run:

Inputs:	Outputs:
1 2 3 4 5	0.50↩

Save your program in the file named `normalized6.c`. Submit your program to CodeCrunch.

Hard: Newton's Method [#6]

Problem Description

"In numerical analysis, Newton's method (also known as the Newton-Raphson method), named after Isaac Newton and Joseph Raphson, is a method for finding successively better approximations to the roots (or zeroes) of a real-valued function.

$x : f(x) = 0$

"The Newton-Raphson method in one variable is implemented as follows:

The method starts with a function f defined over the real numbers x, the function's derivative f', and an initial guess x_0 for a root of the function f. If the function satisfies the assumptions made in the derivation of the formula and the initial guess is close, then a better approximation x_1 is:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

"Geometrically, (x_1, 0) is the intersection of the x-axis and the tangent of the graph of f at (x_0, f(x_0)). The process is repeated as:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

until a sufficiently accurate value is reached." – Wikipedia

We represent a *mathematical* function $f(x) = ax^4 + bx^3 + cx^2 + dx + e$ as **five (5)** `real` numbers a, b, c, d, and e. The derivative of $f(x)$ (*i.e.*, $f'(x)$) is computed such that for each $n \times x^m$, the derivative is $(n \times m) \times x^{m-1}$. The function $f(x)$ is computed for each value of x by replacing x with the given value. For instance given $f(x) = 4x^4 + 2x^3 - 2$, $f(3) = 376$.

Final Objective

Given a, b, c, d, e, and the starting guess x_0 find the minimum/maximum/inflexion point using Newton's method described above up to **two (2)** decimal places. We say that the approximation is *good enough* when two consecutive guesses differs by less than 0.001.

Example

The minimum value of $f(x) = 4x^4 + 2x^3 - x^2 - 3x - 2$ is -3.25753 when $x = 0.529583$.

Assumptions

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

- ▷ $1 \leq$ `a` $\leq 2^{30}$ (*the coefficient of the first term in polynomial*)
- ▷ $-2^{30} \leq$ `b,c,d,e` $\leq 2^{30}$ (*the coefficient of other terms in polynomial*)
- ▷ $-1000 \leq$ `x0` ≤ 1000 (*the starting guess*)
- ▷ $f(x) = 0$ is guaranteed to be found
- ▷ $f'(x) = 0$ is guaranteed to be found
- ▷ $f(x)$ is guaranteed to be differentiable

Restrictions

The following restriction(s) is/are imposed on the solution:

- ▷ You are only allowed to use `pow()` and `abs()` from the `<math.h>` library

Tasks

The problem is split into 6 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 1/6
Write a program to four **real** numbers a, b, c, d, e, and x_0 and print the function $f(x) = ax^2+bx+c$ with each coefficient printed up to **two (2)** decimal place. Print only when the coefficient is either greater than 0.001 or smaller than -0.001. Furthermore, print the sign correctly. Note the extra [**space**] after the last coefficient.
Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 1`	`4.00x^4 + 2.00x^3 - 1.00x^2 - 3.00x - 2.00 ↩`

Save your program in the file named `newton1.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 2*), copy your program using the following command:
`cp newton1.c newton2.c`

Task 2/6
Write a program to four **real** numbers a, b, c, d, e, and x_0 and print the value of $f(x_0)$ up to **two (2)** decimal place.
Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 1`	`0.00↩`

Save your program in the file named `newton2.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 3*), copy your program using the following command:
`cp newton2.c newton3.c`

Task 3/6
Write a program to four **real** numbers a, b, c, d, e, and x_0 and print the value of $f'(x_0)$ up to **two (2)** decimal place.
Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 1`	`17.00↩ | f'(x) = 16x^3 + 6x^2 - 2x - 3`

Save your program in the file named `newton3.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 4*), copy your program using the following command:
`cp newton3.c newton4.c`

Task 4/6
Write a program to four **real** numbers a, b, c, d, e, and x_0 and print the value of x_1 using the Newton's method up to **two (2)** decimal place.
Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 1`	`1↩ | x0 - f(x0)/f'(x0) = 1 - (0/17) = 1`

Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 2`	`1.53↩`

Save your program in the file named `newton4.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 5*), copy your program using the following command:
`cp newton4.c newton5.c`

Task 5/6

Write a program to four **real** numbers a, b, c, d, e, and x_0 and print the value of x_n using the Newton's method up to **two (2)** decimal place such that x_n is a *good enough* approximation for $f(x_n) = 0$.

Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 1`	`1.00↩`

Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 2`	`1.00↩`

Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 3`	`1.00↩`

Save your program in the file named `newton5.c`. Submit your program to CodeCrunch. To proceed to the next task (*e.g., task 6*), copy your program using the following command:

```
cp newton5.c newton6.c
```

Task 6/6

Write a program to four **real** numbers a, b, c, d, e, and x_0 and print the value of x_n such that it is the minimum value of $f(x)$. The minimum value of $f(x)$ is computed by finding $x : f'(x) = 0$. In other words, use Newton's method on the derivative of $f(x)$ to find the value of x_n and print $f(x_n)$.

Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 1`	`-3.26↩`

Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 2`	`-3.26↩`

Sample Run:

Inputs:	Outputs:
`4 2 -1 -3 -2 3`	`-3.26↩`

Save your program in the file named `newton6.c`. Submit your program to CodeCrunch.