CSc 372 — Comparative Programming Languages
Fall 2026 (McCann)

https://cs.arizona.edu/classes/cs372/fall26/

Program #1: Ruby

Due Date: September 30th, 2026, at the beginning of class

— Reminder —

In this class, we are allowing you to use AI tools (LLMs, Generative AIs, IDE AI plug–ins, etc.) to help you write code on the programming assignments. However, we *strongly recommend* that you use such tools for assistance only when you need to *learn* something, such as the cause of a mystery syntax error or the semantics of an unfamiliar language feature. Please don't ask AI to write code for you! Why not? Because we ask code–writing questions on the exams, and you will not be allowed to use AI (or any electronic devices) on tests. The purpose of each programming assignment is to help you learn the basics of the language we just covered. So: Write the code yourself, turning to AI for help as little as possible. Looking for friendly guidance from a human being? The TAs and I are here to help, both during office hours and via posts on Piazza!

Overview: In this class, we study specific languages in some detail, but not all detail. Our purpose is to study the design and philosophies of the languages. In the case of Ruby, we are focusing on the object–oriented characteristics. The best way to understand those features is to write programs that use them. Before you can do that, you also need to be able to use the basics of the language. That is why this assignment is a collection of programming exercises, ranging from the straight–forward to the more involved. Between the lecture material and this assignment, we hope that you will become comfortable enough with Ruby to continue to explore additional features of the language on your own, should you choose to do so.

Assignment: For each numbered task below, write a Ruby 3.2.3 (or earlier) program, named as stated, that accomplishes it.

1. **Pell Numbers and $\sqrt{2}$.**

```
$ ruby pell.rb 7
The first 7 Pell number approximations of sqrt(2) are:

1/1, 3/2, 7/5, 17/12, 41/29, 99/70, 239/169

In decimal, 239/169 differs from Math.sqrt(2) by 1.2378941142587863e-05.
```

You are likely familiar with the Fibonacci sequence: $f_0 = 0, f_1 = 1$, and $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$. The Pell sequence is a slight variation of that recurrence: $P_0 = 0, P_1 = 1$, and $P_n = 2P_{n-1} + P_{n-2}$ for $n \geq 2$. Pell numbers were of historical interest because the ratio $\frac{P_{n-1}+P_n}{P_n}$ is an approximation of the value $\sqrt{2}$.

Write a complete, well–documented Ruby program named `pell.rb` that accepts a positive integer n from the command line and computes and displays the first n (starting with $n = 1$) of the $(P_{n-1} + P_n)/P_n$ ratios, followed by the difference between the n-th ratio and Ruby's `Math.sqrt(2)`. Use the output format shown above.

(Continued...)

2. **Semiprimes**.

```
$ ruby semiprimes.rb 0 3
0
$ ruby semiprimes.rb 25 50
9
```

Prime numbers are integers ≥ 2 whose divisors are just 1 and themselves. (Correct: 1 is not a prime number.) Because each non–prime (a.k.a. *composite*) integer n must have a divisor that is $\leq \sqrt{n}$, the primality of an integer can be tested easily (though inefficiently!) via repeated divisions.

The *prime factorization* of an integer $n \geq 2$ is the collection of prime numbers whose product is n. For example, the prime factorization of 12 is $2 \cdot 2 \cdot 3$ and of 15 is $3 \cdot 5$. Prime numbers are their own prime factorizations. Computing n's prime factorization can be accomplished by trial division by the sequence of primes in increasing order. Consider 42. We can divide it by the first prime (2). The result, 21, cannot be divided by two again, but can be divided by the second prime (3), producing 7, another prime. Thus, 42's prime factorization is $2 \cdot 3 \cdot 7$.

A *semiprime* is an integer $s \geq 2$ such that the prime factorization of s contains exactly two (not necessarily distinct) primes. Thus, 15 is a semiprime but 12 and 42 are not.

Write a complete, well–documented Ruby program named `semiprimes.rb` that accepts two non–negative integer arguments a and b, $a \leq b$, and determines the quantity of values in the range $[a, b]$ that are semiprimes.

3. **Generalized Siamese Magic Squares**.

```
$ ruby magic.rb 3 -4 3
17 -4 11
 2  8 14
 5 20 -1
```

A *normal magic square* of size $n \times n$ is constructed of the integers $1..n^2$ arranged such that the sums of the values in each row, column, and both diagonals are identical. If n is odd, the Siamese method, a.k.a. the De la Loubère method, can be used to construct a magic square easily.

Here's how to do it. In general, the value $i + 1$ is placed in the location immediately above and to the right of the location holding value i. If moving above and right leaves the boundary of the square, we 'wrap around' to the bottom row and/or leftmost column as needed. We begin by placing the value 1 in the center location of the top row. To place 2, we move up and right. However, moving up takes us outside of the square, so we wrap around to the bottom row, where we can then move to the right to place 2 in the column immediately right of 1's column. Final exception: If the above–right location is already occupied, place the next value immediately below the last. Following this algorithm fills a 3×3 square as shown below left:

A normal 3 x 3 magic square ⇒

8	1	6
3	5	7
4	9	2

17	-4	11
2	8	14
5	20	-1

⇐ A modified Siamese method square

The Siamese method works for any arithmetic sequence of integers. Recall that an arithmetic sequence is defined by an initial value a_1 and a common difference d such that $a_{i+1} = a_i + d$. If $a_1 = -4$ and $d = 3$, the first nine members of the sequence are -4, -1, 2, 5, 8, 11, 14, 17, 20. Using those values in place of $1..n^2$ produces the non–normal magic square given above right.

Write a complete, well–documented Ruby program named `magic.rb` that accepts three integers from the command line — n (the size of the square), a_1, and d — and generates and displays the corresponding non–normal Siamese method magic square. Display the square as 'squarishly' as you are able, respecting the magnitudes of the square and the values contained therein.

(Continued...)

4. **Word Count**.

```
$ cat onetwo.txt
one: 1
two: 2
$ od -c onetwo.txt
0000000   o   n   e   :       1  \n   t   w   o   :       2  \n
0000016
$ ruby wordcount.rb onetwo.txt
2 4 14 onetwo.txt

$ cat kaydijkstra.txt
"I don't know how many of you have ever met Dijkstra, but you probably know
that arrogance in computer science is measured in nano-Dijkstras." -- Alan Kay
$ ruby wordcount.rb kaydijkstra.txt
2 27 155 kaydijkstra.txt
```

A standard Linux utility program is `wc`, short for Word Count. `wc` gives users the size of a text file as quantities of the lines, words, and bytes it contains. (`wc` can do more, but our version will stop there.) Counting lines is done by counting 'end of line' characters (`\n` (ASCII 10) for Linux text files). Counting words is almost as straight–forward: Any sequence of characters not interrupted by 'whitespace' (spaces, tabs, and `\n` characters) is a word. Whitespace characters are characters, and are counted as such.

Write a complete, well–documented Ruby program named `wordcount.rb` that accepts a text file name on the command line and implements that much of `wc`'s functionality.

5. **Word_Assistant**.

```
$ irb
irb(main):001:0> load 'word_assistant.rb'
=> true
irb(main):002:0> w = Word_assistant.new "consonant"
=> #<Word:0x00007ed2a5b2dff8 @word="consonant">
irb(main):003:0> w.to_s
=> "consonant"
irb(main):004:0> w.qty_of_syllables
=> 3
irb(main):005:0> w.syllablize
=> "con/so/nant"
```

Ruby is an OO language; we ought to ask you to do something at least vaguely OO!

English is a famously inconsistent language, thanks to multiple inheritance from other languages and mutations over time. One difficult task: Dividing a word into syllables by examining its characters. Your job: Write the `word_assistant` class to count and display estimates of the syllables in a word.

`word_assistant`'s constructor accepts a string containing a single English word. The class provides two 'syllablizing' methods:

- `qty_of_syllables` — Returns an integer that is an estimate of the quantity of syllables in the word represented by this `word_assistant` object.

  Syllables must contain at least one vowel (a, e, i, o, u, and y, for our purposes). But, many words have pairs of vowels that represent just one vowel sound, such as the 'ou' in "sound". Also, many words end in a silent 'e', such as the word "before". With that in mind, here's how you are to estimate the number of syllables in a word: Count the vowels. Subtract one from that count for each adjacent pair of vowels in the word. If the word ends with an 'e', subtract one more. The result is the estimate of the quantity of syllables in the word. For example, "instantiable" contains 5 vowels, including one pair of vowels and the trailing 'e'. We estimate its quantity of syllables to be 5 - 1 - 1 = 3. That's not the actual number of syllables, but that's what this algorithm's estimate

is. If your estimate is less than one, return one. Words with three vowels in sequence (e.g., "quail") have two pair: "ua" and "ai".

- `syllablize` — Returns a string representation of this object's word that has forward slashes added to divide the word into its estimated syllables. For example, if the word is "identify", the returned string will be "i/den/ti/fy", including the forward slashes but without the quotes.

  Here's how your method is to identify syllables: We'll consider just two situations, commonly known as the 'vccv' and 'vcv' rules. As you scan a word from left to right, if you encounter a vowel followed by two consonants (non–vowels) followed by a vowel, add a slash between the consonants. For example, "hidden" contains this pattern and is divided into two syllables: "hid/den". If you see the 'vcv' pattern, divide before the consonant. Thus, "consonant", which exhibits both patterns, would be divided like this: "con/so/nant". Note that `syllablize`'s result could show a different quantity of syllables than `qty_of_syllables`'s result. That's OK.

Data: The input expectations for the programs are given with the program descriptions, above.

Output: Output details for each program are stated in the Assignment section, above. Please ask (preferably on Piazza) if you need additional details.

Hand In: You are required to submit your completed program files (your `.rb` files) using the `turnin` facility on lectura. The submission folder is `cs372p1`. Instructions are available from the document of submission instructions linked to the class web page. In particular, because we will be *grading* your program on lectura, it needs to *run* on lectura, so be sure to *test* it on lectura. Submit all files as–is, *without* packaging them into `.zip`, `.jar`, `.tar`, etc., files.

Want to Learn More?

- https://en.wikipedia.org/wiki/Pell_number — More than you'd care to know about Pell numbers.
- https://en.wikipedia.org/wiki/Semiprime — A more reasonable amount of info about semiprimes.
- https://en.wikipedia.org/wiki/Siamese_method — One guess!
- `man wc` or `info wc` — Log into lectura and type either command for all of `wc`'s options.
- https://en.wikipedia.org/wiki/Syllable — Way, way more than you want to know about syllabus!

Additional Hints, Reminders, and Requirements:

- There is a list of Ruby resources on the class web page, and other suggestions (irb, Googling) were mentioned in class. Working through statement syntax, execution errors, etc., are chores that we expect you to do on your own, as you would have to if you were learning a language on your own. But, if you get really stuck, the TA(s) and I are here to help.
- Worried about features of Ruby that we haven't (or won't) cover? You can either wait for those features to be covered, and/or you can explore them on your own. You're welcome to use features of Ruby beyond what we'll cover in class, so long as you are sticking to the expectations of the assignment. For example, if you get excited about handling even–sided magic squares, great, but make sure that your program also does what this assignment asks it to do.
- **Start Early!** We don't expect that any of these programs will be conceptually difficult for you, but they will probably take some time to complete, due to your inexperience with the language.