**Nauseating Numbers**

Time for more problems! As you progress through this set of problems, the difficulty will ramp up. There are no RSpec tests for this exercise, so you'll want to create your own ruby file to code and test your work against the examples provided.

The solution is available [here](https://aao-alpha.s3-us-west-1.amazonaws.com/assets/topics/advanced_ruby/projects/nauseating_numbers_solution.rb.zip), but please do not look at it until you have attempted all of the problems!

**Phase 1: No big deal.**

**strange\_sums**

Write a method strange\_sums that accepts an array of numbers as an argument. The method should return a count of the number of distinct pairs of elements that have a sum of zero. You may assume that the input array contains unique elements.

Examples

p strange\_sums([2, -3, 3, 4, -2]) # 2

p strange\_sums([42, 3, -1, -42]) # 1

p strange\_sums([-5, 5]) # 1

p strange\_sums([19, 6, -3, -20]) # 0

p strange\_sums([9]) # 0

**pair\_product**

Write a method pair\_product that accepts an array of numbers and a product as arguments. The method should return a boolean indicating whether or not a pair of distinct elements in the array result in the product when multiplied together. You may assume that the input array contains unique elements.

Examples

p pair\_product([4, 2, 5, 8], 16) # true

p pair\_product([8, 1, 9, 3], 8) # true

p pair\_product([3, 4], 12) # true

p pair\_product([3, 4, 6, 2, 5], 12) # true

p pair\_product([4, 2, 5, 7], 16) # false

p pair\_product([8, 4, 9, 3], 8) # false

p pair\_product([3], 12) # false

**rampant\_repeats**

Write a method rampant\_repeats that accepts a string and a hash as arguments. The method should return a new string where characters of the original string are repeated the number of times specified by the hash. If a character does not exist as a key of the hash, then it should remain unchanged.

Examples

p rampant\_repeats('taco', {'a'=>3, 'c'=>2}) # 'taaacco'

p rampant\_repeats('feverish', {'e'=>2, 'f'=>4, 's'=>3}) # 'ffffeeveerisssh'

p rampant\_repeats('misispi', {'s'=>2, 'p'=>2}) # 'mississppi'

p rampant\_repeats('faarm', {'e'=>3, 'a'=>2}) # 'faaaarm'

**perfect\_square?**

Write a method perfect\_square? that accepts a number as an argument. The method should return a boolean indicating whether or not the argument is a perfect square. A perfect square is a number that is the product of some number multiplied by itself. For example, since 64 = 8 \* 8 and 144 = 12 \* 12, 64 and 144 are perfect squares; 35 is not a perfect square.

Examples

p perfect\_square(1) # true

p perfect\_square(4) # true

p perfect\_square(64) # true

p perfect\_square(100) # true

p perfect\_square(169) # true

p perfect\_square(2) # false

p perfect\_square(40) # false

p perfect\_square(32) # false

p perfect\_square(50) # false

**Phase 2: Nothing you can't handle.**

**anti\_prime?**

Write a method anti\_prime? that accepts a number as an argument. The method should return true if the given number has more divisors than all positive numbers less than the given number. For example, 24 is an anti-prime because it has more divisors than any positive number less than 24. Math Fact: Numbers that meet this criteria are also known as [highly composite numbers](https://en.wikipedia.org/wiki/Highly_composite_number).

Examples

p anti\_prime?(24) # true

p anti\_prime?(36) # true

p anti\_prime?(48) # true

p anti\_prime?(360) # true

p anti\_prime?(1260) # true

p anti\_prime?(27) # false

p anti\_prime?(5) # false

p anti\_prime?(100) # false

p anti\_prime?(136) # false

p anti\_prime?(1024) # false

**matrix\_addition**

Let a 2-dimensional array be known as a "matrix". Write a method matrix\_addition that accepts two matrices as arguments. The two matrices are guaranteed to have the same "height" and "width". The method should return a new matrix representing the sum of the two arguments. To add matrices, we simply add the values at the same positions:

# programmatically

[[2, 5], [4, 7]] + [[9, 1], [3, 0]] => [[11, 6], [7, 7]]

# structurally

2 5 + 9 1 => 11 6

4 7 3 0 7 7

Examples

matrix\_a = [[2,5], [4,7]]

matrix\_b = [[9,1], [3,0]]

matrix\_c = [[-1,0], [0,-1]]

matrix\_d = [[2, -5], [7, 10], [0, 1]]

matrix\_e = [[0 , 0], [12, 4], [6, 3]]

p matrix\_addition(matrix\_a, matrix\_b) # [[11, 6], [7, 7]]

p matrix\_addition(matrix\_a, matrix\_c) # [[1, 5], [4, 6]]

p matrix\_addition(matrix\_b, matrix\_c) # [[8, 1], [3, -1]]

p matrix\_addition(matrix\_d, matrix\_e) # [[2, -5], [19, 14], [6, 4]]

**mutual\_factors**

Write a method mutual\_factors that accepts any amount of numbers as arguments. The method should return an array containing all of the common divisors shared among the arguments. For example, the common divisors of 50 and 30 are [1, 2, 5, 10]. You can assume that all of the arguments are positive integers.

Examples

p mutual\_factors(50, 30) # [1, 2, 5, 10]

p mutual\_factors(50, 30, 45, 105) # [1, 5]

p mutual\_factors(8, 4) # [1, 2, 4]

p mutual\_factors(8, 4, 10) # [1, 2]

p mutual\_factors(12, 24) # [1, 2, 3, 4, 6, 12]

p mutual\_factors(12, 24, 64) # [1, 2, 4]

p mutual\_factors(22, 44) # [1, 2, 11, 22]

p mutual\_factors(22, 44, 11) # [1, 11]

p mutual\_factors(7) # [1, 7]

p mutual\_factors(7, 9) # [1]

**tribonacci\_number**

The tribonacci sequence is similar to that of [Fibonacci](https://en.wikipedia.org/wiki/Fibonacci_number). The first three numbers of the tribonacci sequence are 1, 1, and 2. To generate the next number of the sequence, we take the sum of the previous three numbers. The first six numbers of tribonacci sequence are:

1, 1, 2, 4, 7, 13, ... and so on

Write a method tribonacci\_number that accepts a number argument, n, and returns the n-th number of the tribonacci sequence.

Examples

p tribonacci\_number(1) # 1

p tribonacci\_number(2) # 1

p tribonacci\_number(3) # 2

p tribonacci\_number(4) # 4

p tribonacci\_number(5) # 7

p tribonacci\_number(6) # 13

p tribonacci\_number(7) # 24

p tribonacci\_number(11) # 274

**Phase 3: Now we're having fun.**

**matrix\_addition\_reloaded**

Write a method matrix\_addition\_reloaded that accepts any number of matrices as arguments. The method should return a new matrix representing the sum of the arguments. Matrix addition can only be performed on matrices of similar dimensions, so if all of the given matrices do not have the same "height" and "width", then return nil.

Examples

matrix\_a = [[2,5], [4,7]]

matrix\_b = [[9,1], [3,0]]

matrix\_c = [[-1,0], [0,-1]]

matrix\_d = [[2, -5], [7, 10], [0, 1]]

matrix\_e = [[0 , 0], [12, 4], [6, 3]]

p matrix\_addition\_reloaded(matrix\_a, matrix\_b) # [[11, 6], [7, 7]]

p matrix\_addition\_reloaded(matrix\_a, matrix\_b, matrix\_c) # [[10, 6], [7, 6]]

p matrix\_addition\_reloaded(matrix\_e) # [[0, 0], [12, 4], [6, 3]]

p matrix\_addition\_reloaded(matrix\_d, matrix\_e) # [[2, -5], [19, 14], [6, 4]]

p matrix\_addition\_reloaded(matrix\_a, matrix\_b, matrix\_e) # nil

p matrix\_addition\_reloaded(matrix\_d, matrix\_e, matrix\_c) # nil

**squarocol?**

Write a method squarocol? that accepts a 2-dimensional array as an argument. The method should return a boolean indicating whether or not any row or column is completely filled with the same element. You may assume that the 2-dimensional array has "square" dimensions, meaning it's height is the same as it's width.

Examples

p squarocol?([

[:a, :x , :d],

[:b, :x , :e],

[:c, :x , :f],

]) # true

p squarocol?([

[:x, :y, :x],

[:x, :z, :x],

[:o, :o, :o],

]) # true

p squarocol?([

[:o, :x , :o],

[:x, :o , :x],

[:o, :x , :o],

]) # false

p squarocol?([

[1, 2, 2, 7],

[1, 6, 6, 7],

[0, 5, 2, 7],

[4, 2, 9, 7],

]) # true

p squarocol?([

[1, 2, 2, 7],

[1, 6, 6, 0],

[0, 5, 2, 7],

[4, 2, 9, 7],

]) # false

**squaragonal?**

Write a method squaragonal? that accepts 2-dimensional array as an argument. The method should return a boolean indicating whether or not the array contains all of the same element across either of its diagonals. You may assume that the 2-dimensional array has "square" dimensions, meaning it's height is the same as it's width.

Examples

p squaragonal?([

[:x, :y, :o],

[:x, :x, :x],

[:o, :o, :x],

]) # true

p squaragonal?([

[:x, :y, :o],

[:x, :o, :x],

[:o, :o, :x],

]) # true

p squaragonal?([

[1, 2, 2, 7],

[1, 1, 6, 7],

[0, 5, 1, 7],

[4, 2, 9, 1],

]) # true

p squaragonal?([

[1, 2, 2, 5],

[1, 6, 5, 0],

[0, 2, 2, 7],

[5, 2, 9, 7],

]) # false

**pascals\_triangle**

[Pascal's triangle](https://en.wikipedia.org/wiki/Pascal%27s_triangle) is a 2-dimensional array with the shape of a pyramid. The top of the pyramid is the number 1. To generate further levels of the pyramid, every element is the sum of the element above and to the left with the element above and to the right. Nonexisting elements are treated as 0 when calculating the sum. For example, here are the first 5 levels of Pascal's triangle:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

Write a method pascals\_triangle that accepts a positive number, n, as an argument and returns a 2-dimensional array representing the first n levels of pascal's triangle.

Examples

p pascals\_triangle(5)

# [

# [1],

# [1, 1],

# [1, 2, 1],

# [1, 3, 3, 1],

# [1, 4, 6, 4, 1]

# ]

p pascals\_triangle(7)

# [

# [1],

# [1, 1],

# [1, 2, 1],

# [1, 3, 3, 1],

# [1, 4, 6, 4, 1],

# [1, 5, 10, 10, 5, 1],

# [1, 6, 15, 20, 15, 6, 1]

# ]

**Phase 4: Nauseating.**

**mersenne\_prime**

A "Mersenne prime" is a prime number that is one less than a power of 2. This means that it is a prime number with the form 2^x - 1, where x is some exponent. For example:

* 3 is a Mersenne prime because it is prime and 3 = 2^2 - 1
* 7 is a Mersenne prime because it is prime and 7 = 2^3 - 1
* 11 is *not* a Mersenne prime because although it is prime, it does not have the form 2^x - 1

The first three Mersenne primes are 3, 7, and 31. Write a method mersenne\_prime that accepts a number, n, as an argument and returns the n-th Mersenne prime.

Examples

p mersenne\_prime(1) # 3

p mersenne\_prime(2) # 7

p mersenne\_prime(3) # 31

p mersenne\_prime(4) # 127

p mersenne\_prime(6) # 131071

**triangular\_word?**

A [triangular number](https://en.wikipedia.org/wiki/Triangular_number) is a number of the form (i \* (i + 1)) / 2 where i is some positive integer. Substituting i with increasing integers gives the triangular number sequence. The first five numbers of the triangular number sequence are 1, 3, 6, 10, 15. Below is a breakdown of the calculations used to obtain these numbers:

| **i** | **(i \* (i + 1)) / 2** |
| --- | --- |
| 1 | 1 |
| 2 | 3 |
| 3 | 6 |
| 4 | 10 |
| 5 | 15 |

We can encode a word as a number by taking the sum of its letters based on their position in the alphabet. For example, we can encode "cat" as 24 because c is the 3rd of the alphabet, a is the 1st, and t is the 20th:

3 + 1 + 20 = 24

Write a method triangular\_word? that accepts a word as an argument and returns a boolean indicating whether or not that word's number encoding is a triangular number. You can assume that the argument contains lowercase letters.

Examples

p triangular\_word?('abc') # true

p triangular\_word?('ba') # true

p triangular\_word?('lovely') # true

p triangular\_word?('question') # true

p triangular\_word?('aa') # false

p triangular\_word?('cd') # false

p triangular\_word?('cat') # false

p triangular\_word?('sink') # false

**consecutive\_collapse**

Write a method consecutive\_collapse that accepts an array of numbers as an argument. The method should return a new array that results from continuously removing consecutive numbers that are adjacent in the array. If multiple adjacent pairs are consecutive numbers, remove the leftmost pair first. For example:

[3, 4, 1] -> [1]

* because 3 and 4 are consecutive and adjacent numbers, so we can remove them

[1, 4, 3, 7] -> [1, 7]

* because 4 and 3 are consecutive and adjacent numbers, so we can remove them

[3, 4, 5] -> [5]

* because 4 and 3 are consecutive and adjacent numbers, we don't target 4 and 5 since we should prefer to remove the leftmost pair

We can apply this rule repeatedly until we cannot collapse the array any further:

# example 1

[9, 8, 4, 5, 6] -> [4, 5, 6] -> [6]

# example 2

[3, 5, 6, 2, 1] -> [3, 2, 1] -> [1]

Code examples

# p consecutive\_collapse([3, 4, 1]) # [1]

# p consecutive\_collapse([1, 4, 3, 7]) # [1, 7]

# p consecutive\_collapse([9, 8, 2]) # [2]

# p consecutive\_collapse([9, 8, 4, 5, 6]) # [6]

# p consecutive\_collapse([1, 9, 8, 6, 4, 5, 7, 9, 2]) # [1, 9, 2]

# p consecutive\_collapse([3, 5, 6, 2, 1]) # [1]

# p consecutive\_collapse([5, 7, 9, 9]) # [5, 7, 9, 9]

# p consecutive\_collapse([13, 11, 12, 12]) # []

**pretentious\_primes**

Write a method pretentious\_primes that takes accepts an array and a number, n, as arguments. The method should return a new array where each element of the original array is replaced according to the following rules:

* when the number argument is positive, replace an element with the n-th nearest prime number that is greater than the element
* when the number argument is negative, replace an element with the n-th nearest prime number that is less than the element

For example:

* if element = 7 and n = 1, then the new element should be 11 because 11 is the nearest prime greater than 7
* if the element = 7 and n = 2, then the new element should be 13 because 13 is the 2nd nearest prime greater than 7
* if the element = 7 and n = -1, then the new element should be 5 because 5 is the nearest prime less than 7
* if the element = 7 and n = -2, then the new element should be 3 because 3 is the 2nd nearest prime less than 7

Note that we will always be able to find a prime that is greater than a given number because there are an infinite number of primes (this is given by [Euclid's Theorem](https://en.wikipedia.org/wiki/Euclid%27s_theorem)). However, we may be unable to find a prime that is smaller than a given number, because 2 is the smallest prime. When a smaller prime cannot be calculated, replace the element with nil.

Examples

# p pretentious\_primes([4, 15, 7], 1) # [5, 17, 11]

# p pretentious\_primes([4, 15, 7], 2) # [7, 19, 13]

# p pretentious\_primes([12, 11, 14, 15, 7], 1) # [13, 13, 17, 17, 11]

# p pretentious\_primes([12, 11, 14, 15, 7], 3) # [19, 19, 23, 23, 17]

# p pretentious\_primes([4, 15, 7], -1) # [3, 13, 5]

# p pretentious\_primes([4, 15, 7], -2) # [2, 11, 3]

# p pretentious\_primes([2, 11, 21], -1) # [nil, 7, 19]

# p pretentious\_primes([32, 5, 11], -3) # [23, nil, 3]

# p pretentious\_primes([32, 5, 11], -4) # [19, nil, 2]

# p pretentious\_primes([32, 5, 11], -5) # [17, nil, nil]