Write a method most\_vowels that takes in a sentence string and returns the word of the sentence that contains the most vowels.

def most\_vowels(sentence)

vowels = "aeiou"

my\_hash = Hash.new(0)

words = sentence.split(" ")

words.each do |ele|

ele.each\_char do |char|

if vowels.include?(char)

my\_hash[ele] += 1

end

end

end

sorted = my\_hash.sort\_by { |k,v| v}

return sorted[-1][0]

end

print most\_vowels("what a wonderful life") #=> "wonderful"

**Prime**

Write a method prime? that takes in a number and returns a boolean, indicating whether the number is prime. A prime number is only divisible by 1 and itself.

def prime?(num)

if num < 2

return false

end

(2...num).each do |multiple|

if num % multiple == 0

return false

end

end

return true

end

puts prime?(2) #=> true

puts prime?(5) #=> true

puts prime?(11) #=> true

puts prime?(4) #=> false

puts prime?(9) #=> false

puts prime?(-5) #=> false

def greatest\_factor\_array(arr)

my\_arr = arr.map do |ele|

if ele % 2 == 0

findfactor(ele / 2)

else

ele

end

end

return my\_arr

end

**Greatest Factor Array**

Write a method greatest\_factor\_array that takes in an array of numbers and returns a new array where every even number is replaced with it's greatest factor. A greatest factor is the largest number that divides another with no remainder. For example the greatest factor of 16 is 8. (For the purpose of this problem we won't say the greatest factor of 16 is 16, because that would be too easy, ha)

def findfactor(ele)

most = 0

(2..ele).each do |factor|

if ele % factor == 0

most = factor

end

end

return most

end

**Perfect Square**

Write a method perfect\_square? that takes in a number and returns a boolean indicating whether it is a perfect square. A perfect square is a number that results from multiplying a number by itself. For example, 9 is a perfect square because 3 *3 = 9, 25 is a perfect square because 5* 5 = 25.

def perfect\_square?(num)

i = 1

while i < num

if i \* i == num

return true

end

i += 1

end

return false

end

puts perfect\_square?(5) #=> false

puts perfect\_square?(12) #=> false

puts perfect\_square?(30) #=> false

puts perfect\_square?(9) #=> true

puts perfect\_square?(25) #=> true

**Triple Sequence**

Write a method triple\_sequence that takes in two numbers, start and length. The method should return an array representing a sequence that begins with start and is length elements long. In the sequence, every element should be 3 times the previous element. Assume that the length is at least 1.

def triple\_sequence(start, length)

seq = [start]

while seq.length < length

seq << seq[-1] \* 3

end

return seq

end

def triple\_sequence(start, length)

my\_arr = [start]

i = 0

num = start

while i < length -1

my\_arr << (num \*= 3)

i += 1

end

return my\_arr

end

print triple\_sequence(2, 4) # => [2, 6, 18, 54]

puts

print triple\_sequence(4, 5) # => [4, 12, 36, 108, 324]

puts

**Summation Sequence**

A number's summation is the sum of all positive numbers less than or equal to the number. For example: the summation of 3 is 6 because 1 + 2 + 3 = 6, the summation of 6 is 21 because 1 + 2 + 3 + 4 + 5 + 6 = 21. Write a method summation\_sequence that takes in a two numbers: start and length. The method should return an array containing length total elements. The first number of the sequence should be the start number. At any point, to generate the next element of the sequence we take the summation of the previous element. You can assume that length is not zero.

def summation\_sequence(start, length)

my\_seq = [start]

while my\_seq.length < length

count = 0

(1..my\_seq[-1]).each do |ele|

count += ele

end

my\_seq << count

end

return my\_seq

end

print summation\_sequence(3, 4) # => [3, 6, 21, 231]

puts

print summation\_sequence(5, 3) # => [5, 15, 120]

def fibonacci(length)

if length == 0

return []

elsif length == 1

return [1]

else

i = 0

fib\_arr = [1,1]

while fib\_arr.length < length

fib\_arr << (fib\_arr[-1] + fib\_arr[-2])

i += 1

end

end

return fib\_arr

end

print fibonacci(0) # => []

puts

print fibonacci(1) # => [1]

puts

print fibonacci(6) # => [1, 1, 2, 3, 5, 8]

puts

print fibonacci(8) # => [1, 1, 2, 3, 5, 8, 13, 21]

puts

**Caesar Cipher**

Write a method caesar\_cipher that takes in a string and a number. The method should return a new string where every character of the original is shifted num characters in the alphabet.

def caesar\_cipher(str, num)

new\_str = ""

alphabet = "abcdefghijklmnopqrstuvwxyz"

alpha\_list = alphabet.split("")

str.each\_char do |char|

ind = alpha\_list.index(char)

new\_str += alpha\_list[(ind + num) % 26]

end

return new\_str

end

puts caesar\_cipher("apple", 1) #=> "bqqmf"

puts caesar\_cipher("bootcamp", 2) #=> "dqqvecor"

puts caesar\_cipher("zebra", 4) #=> "difve"

Write a method vowel\_cipher that takes in a string and returns a new string where every vowel becomes the next vowel in the alphabet.

def vowel\_cipher(string)

str = ""

vowels = "aeiou".split("")

string.each\_char do |char|

if vowels.include?(char)

newvowel = vowels.index(char) + 1

str += vowels[newvowel%vowels.length]

else

str += char

end

end

return str

end

def vowel\_cipher(string)

vowels = "aeiou"

new\_chars = string.split("").map do |char|

if vowels.include?(char)

old\_idx = vowels.index(char)

new\_idx = old\_idx + 1

vowels[new\_idx % vowels.length]

else

char

end

end

return new\_chars.join("")

end

puts vowel\_cipher("bootcamp") #=> buutcemp

puts vowel\_cipher("paper cup") #=> pepir cap

**Double Letter Count**

Write a method that takes in a string and returns the number of times that the same letter repeats twice in a row.

def double\_letter\_count(string)

count = 0

string.each\_char.with\_index do |char, index|

if string[index + 1] == string[index]

count += 1

end

end

return count

end

puts double\_letter\_count("the jeep rolled down the hill") #=> 3

puts double\_letter\_count("bootcamp") #=> 1

**Adjacent Sum**

Write a method adjacent\_sum that takes in an array of numbers and returns a new array containing the sums of adjacent numbers in the original array. See the examples.

def adjacent\_sum(arr)

i = 1

new\_arr = []

while i < arr.length

new\_arr << arr[i] + arr[i - 1]

i += 1

end

return new\_arr

end

print adjacent\_sum([3, 7, 2, 11]) #=> [10, 9, 13], because [ 3+7, 7+2, 2+11 ]

puts

print adjacent\_sum([2, 5, 1, 9, 2, 4]) #=> [7, 6, 10, 11, 6], because [2+5, 5+1, 1+9, 9+2, 2+4]

puts

def adjacent\_sum(arr)

new\_arr = []

arr.each\_with\_index do |ele, i|

if i != arr.length - 1

new\_arr << arr[i] + arr[i + 1]

end

end

return new\_arr

end

print adjacent\_sum([3, 7, 2, 11]) #=> [10, 9, 13], because [ 3+7, 7+2, 2+11 ]

puts

print adjacent\_sum([2, 5, 1, 9, 2, 4]) #=> [7, 6, 10, 11, 6], because [2+5, 5+1, 1+9, 9+2, 2+4]

puts

def pyramid\_sum(base)

new\_arr = [base]

i = 1

while i < base.length

new\_arr << adjacent\_sum(new\_arr[-1])

i+= 1

end

return new\_arr.reverse()

end

**Pyramid Sum**

Write a method pyramid\_sum that takes in an array of numbers representing the base of a pyramid. The function should return a 2D array representing a complete pyramid with the given base. To construct a level of the pyramid, we take the sum of adjacent elements of the level below.

def adjacent\_sum(arr)

i = 1

new\_arr = []

while i < arr.length

new\_arr << arr[i] + arr[i - 1]

i += 1

end

return new\_arr

end

def pyramid\_sum(base)

pyramid = [base]

while pyramid.length < base.length

prev\_level = pyramid[0]

next\_level = adjacent\_sum(prev\_level)

pyramid.unshift(next\_level)

end

return pyramid

end

def adjacent\_sum(arr)

new\_arr = []

arr.each\_with\_index do |ele, i|

if i != arr.length - 1

new\_arr << arr[i] + arr[i + 1]

end

end

return new\_arr

end

print pyramid\_sum([1, 4, 6]) #=> [[15], [5, 10], [1, 4, 6]]

puts

print pyramid\_sum([3, 7, 2, 11]) #=> [[41], [19, 22], [10, 9, 13], [3, 7, 2, 11]]

puts

**All Else Equal**

Write a method all\_else\_equal that takes in an array of numbers. The method should return the element of the array that is equal to half of the sum of all elements of the array. If there is no such element, the method should return nil.

def all\_else\_equal(arr)

master = 0

arr.each do |ele|

master += ele

end

master /= 2

if arr.include?(master)

return master

else

return nil

end

end

p all\_else\_equal([2, 4, 3, 10, 1]) #=> 10, because the sum of all elements is 20

p all\_else\_equal([6, 3, 5, -9, 1]) #=> 3, because the sum of all elements is 6

p all\_else\_equal([1, 2, 3, 4]) #=> nil, because the sum of all elements is 10 and there is no 5 in the array

def all\_else\_equal(arr)

sum = sum\_array(arr)

arr.each do |ele|

if ele == sum / 2.0

return ele

end

end

return nil

end

def sum\_array(arr)

sum = 0

arr.each do |num|

sum += num

end

return sum

end

**Anagrams**

Write a method anagrams? that takes in two words and returns a boolean indicating whether or not the words are anagrams. Anagrams are words that contain the same characters but not necessarily in the same order. Solve this without using .sort

def anagrams?(word1, word2)

word1arr = word1.split("")

word2.each\_char do |char|

if !word1arr.include?(char)

return false

end

end

return true

end

puts anagrams?("cat", "act") #=> true

puts anagrams?("restful", "fluster") #=> true

puts anagrams?("cat", "dog") #=> false

puts anagrams?("boot", "bootcamp") #=> false

def anagrams?(word1, word2)

return char\_count(word1) == char\_count(word2)

end

def char\_count(word)

count = Hash.new(0)

word.each\_char { |char| count[char] += 1 }

return count

end

def same\_char\_collapse(str)

collapsible = true

**Same Char Collapse**

Write a method same\_char\_collapse that takes in a string and returns a collapsed version of the string. To collapse the string, we repeatedly delete 2 adjacent characters that are the same until there are no such adjacent characters. If there are multiple pairs that can be collapsed, delete the leftmost pair. For example, we take the following steps to collapse "zzzxaaxy": zzzxaaxy -> zxaaxy -> zxxy -> zy

while collapsible

words = str.split("")

collapsible = false

words.each.with\_index do |var, idx|

if words[idx] == words[idx+1]

words[idx] = ""

words[idx+1] = ""

collapsible = true

end

end

str = words.join("")

end

return str

end

puts same\_char\_collapse("zzzxaaxy") #=> "zy"

# because zzzxaaxy -> zxaaxy -> zxxy -> zy

puts same\_char\_collapse("uqrssrqvtt") #=> "uv"

# because uqrssrqvtt -> uqrrqvtt -> uqqvtt -> uvtt -> uv