README

April 18, 2016

prep$_{\textbf{ruby challenges}}$

The Ruby challenge problems from the Markup and Coding course of the Viking Code School Prep Work

1 Ruby Calisthenics

1.1 Power

Write a method *power* which takes two integers (*base* and *exponent*) and returns the *base* raised to the power of *exponent*. Do not use Ruby's "**" operator for this!

```
> power(3,4)
=> 81 # (3*3*3*3)
```

```
def power(base,exponent)
  # returns base raised to the power of exponent without the use of ** operator

  a = base
  b = exponent
  c = []

  b.times do
    c.push a
  end

  c.inject(1) {|product, n| product * n}
end

p power(3,4)
```

1.2 Factorial

Write a method *factorial* which takes a number and returns the product of every number up to the current number multiplied together.

```
> factorial(5)
=> 120 # from 1*2*3*4*5

def factorial(n)
  # Int => Int
  # Takes a number and returns the product of every number up to
  # the current number multiplied together

  a = []

  n.downto(1).each do |i|
    a.push i
  end

  return a.inject(1) {|product, n| product * n}

end

p factorial(5)
```

1.3 Uniques

Write a method *uniques* which takes an array of items and returns the array without any duplicates. Don't use Ruby's *uniq* method.

```
uniques([1,5,"frog",2,1,3,"frog"])
=> [1,5,"frog",2,3]

def uniques(array)
  # Array of Items => Array of Items
  # Takes an array, returns array with duplicate items removed.
  # Write without uniq

  no_dupes = []
  couples = array.combination(2)
  groups = array.group_by{|e| e}
  groups.each do |g|
    no_dupes.push(g[0])
  end

  return no_dupes
end

p uniques([1,5,"frog",2,1,3,"frog"])
```

1.4 Combinations

Write a method *combinations* which takes two arrays of strings and returns an array with all of the combinations of the items in them, listing the first items first.

```
> combinations(["on","in"],["to","rope"])
=> ["onto","onrope","into","inrope"]

def combinations(ary1,ary2)
  # Ary(Str), Ary(Str) => Ary(Str)
  # Takes two arrays of strings, returns an array with all of the combinations
  # of the items in them, listing the first item first.

  a = ary1
  b = ary2

  c = []

  a.each do |s|
    b.each do |x|
      c.push "#{s}#{x}"
    end
  end

  p c
end

combinations(["on","in"],["to","rope"])
```

1.5 Primes

Write a method is_{prime}*?* which takes in a number and returns *true* if it is a prime number.

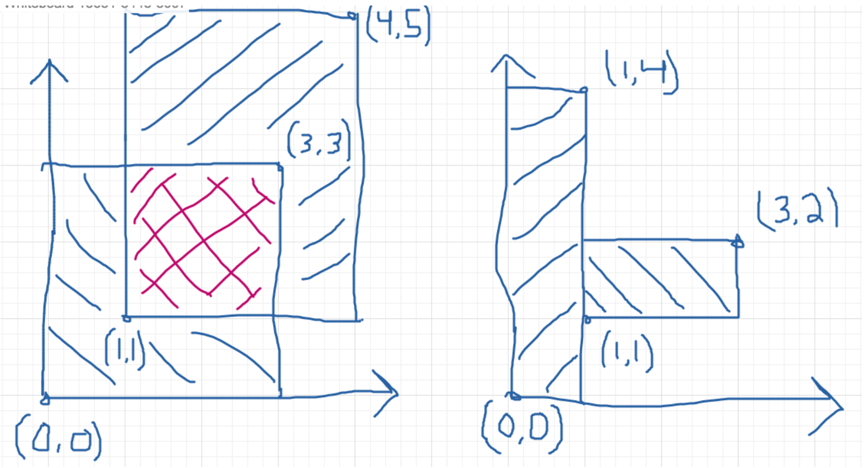
```
> is_prime?(7)
=> true
> is_prime?(14)
=> false

def is_prime?(i)
  range = (i-1).downto(2)

  range.each do |a|
    #p i%a == 0
  end

  p range.any? {|a| i%a == 0}
end

is_prime?(7)
```

1.6 Rectangle Overlap

Write a method *overlap* which takes two rectangles defined by the coordinates of their corners, e.g. *[[0,0],[3,3]]* and *[[1,1],[4,6]]*, and determines whether they overlap. You can assume all coordinates are positive integers.

```
> overlap( [ [0,0],[3,3] ], [ [1,1],[4,5] ] )
=> true
> overlap( [ [0,0],[1,4] ], [ [1,1],[3,2] ] )
=> false
```

It doesn't count as overlapping if their edges touch but they do not otherwise overwrite each other. As expressed by a sixth grade student:



```
def overlap(a,b)
  # Array(Coordinates), Array(Coordinates) => Boolean

  # a = [[0,0],[3,3]]
  ax1 = a[0][0]
  ay1 = a[0][1]
  ax2 = a[1][0]
  ay2 = a[1][1]

  awidth = ax2-ax1
  aheight = ay2-ay1
  aarea = awidth*aheight

  # b = [[1,1],[4,5]]
  bx1 = b[0][0]
  by1 = b[0][1]
  bx2 = b[1][0]
  by2 = b[1][1]

  bwidth = bx2-bx1
  bheight = by2-by1
  barea = bwidth*bheight

  #( [ [0  , 0  ],[3  , 3  ] ], [ [1  , 1  ],[4  , 5  ] ] )
  #( [ [ax1, ay1],[ax2, ay2] ], [ [bx1, by1],[bx2, by2] ] )
  if bx1 < ax2 && by1 < ay2
    true
  end
end

overlap( [ [0,0],[3,3] ], [ [1,1],[4,5] ] )
overlap( [ [0,0],[1,4] ], [ [1,1],[3,2] ] )

# further development needed to explore every case
```

2 A Bigger Challenge: The Counting Game

2.1 NB: Software Engineering

`https://www.vikingcodeschool.com/software-engineering-basics`

- "logic" way through problems
    - pseudocoding ("whiteboarding")
    - modular design and engineering best practices
    - 4-step engineering problem solving approach
        1. Understand the problem
        2. Plan a solution
        3. Carry out that plan
        4. Examine your results for accuracy
    - Agile development
        - project management technique / development philosophy
        - teams commonly work in short (1-2 week) sprints
        - XP and SCRUM, Agile techniques
            - short cycle times
            - frequent client/user interaction
            - keeps project focused on relevant tasks
            - XP
            - pair programming
            - pairing developers together at workstations
        - keep software user-driven
        - TDD