**Perilous Procs**

Let's find out how pro† you are with procs! As you progress through this set of problems, the difficulty will ramp up.

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

**Phase 1: Problems that are pleasant.**

**some?**

Write a method some? that accepts an array and a block as arguments. The method should return a boolean indicating whether or not at least one of the elements of the array returns true when given to the block. Solve this using Array#each.

Examples

p some?([3, 1, 11, 5]) { |n| n.even? } # false

p some?([3, 4, 11, 5]) { |n| n.even? } # true

p some?([8, 2]) { |n| n.even? } # true

p some?(['squash', 'corn', 'kale', 'carrot']) { |str| str[0] == 'p' } # false

p some?(['squash', 'corn', 'kale', 'potato']) { |str| str[0] == 'p' } # true

p some?(['parsnip', 'lettuce', 'pea']) { |str| str[0] == 'p' } # true

**exactly?**

Write a method exactly? that accepts an array, a number (n), and a block as arguments. The method should return a boolean indicating whether or not there are exactly n elements that return true when given to the block. Solve this using Array#each.

Examples

p exactly?(['A', 'b', 'C'], 2) { |el| el == el.upcase } # true

p exactly?(['A', 'B', 'C'], 2) { |el| el == el.upcase } # false

p exactly?(['A', 'B', 'C'], 3) { |el| el == el.upcase } # true

p exactly?(['cat', 'DOG', 'bird'], 1) { |el| el == el.upcase } # true

p exactly?(['cat', 'DOG', 'bird'], 0) { |el| el == el.upcase } # false

p exactly?(['cat', 'dog', 'bird'], 0) { |el| el == el.upcase } # true

p exactly?([4, 5], 3) { |n| n > 0 } # false

p exactly?([4, 5, 2], 3) { |n| n > 0 } # true

p exactly?([42, -9, 7, -3, -6], 2) { |n| n > 0 } # true

**filter\_out**

Write a method filter\_out that accepts an array and a block as arguments. The method should return a new array where elements of the original array are removed if they return true when given to the block. Solve this using Array#each.

Examples

p filter\_out([10, 6, 3, 2, 5 ]) { |x| x.odd? } # [10, 6, 2]

p filter\_out([1, 7, 3, 5 ]) { |x| x.odd? } # []

p filter\_out([10, 6, 3, 2, 5 ]) { |x| x.even? } # [3, 5]

p filter\_out([1, 7, 3, 5 ]) { |x| x.even? } # [1, 7, 3, 5]

**at\_least?**

Write a method at\_least? that accepts an array, a number (n), and a block as an arguments. The method should return a boolean indicating whether or not at least n elements of the array return true when given to the block. Solve this using Array#each.

Examples

p at\_least?(['sad', 'quick', 'timid', 'final'], 2) { |s| s.end\_with?('ly') }

# false

p at\_least?(['sad', 'quickly', 'timid', 'final'], 2) { |s| s.end\_with?('ly') }

# false

p at\_least?(['sad', 'quickly', 'timidly', 'final'], 2) { |s| s.end\_with?('ly') }

# true

p at\_least?(['sad', 'quickly', 'timidly', 'finally'], 2) { |s| s.end\_with?('ly') }

# true

p at\_least?(['sad', 'quickly', 'timid', 'final'], 1) { |s| s.end\_with?('ly') }

# true

p at\_least?(['sad', 'quick', 'timid', 'final'], 1) { |s| s.end\_with?('ly') }

# false

p at\_least?([false, false, false], 3) { |bool| bool }

# false

p at\_least?([false, true, true], 3) { |bool| bool }

# false

p at\_least?([true, true, true], 3) { |bool| bool }

# true

p at\_least?([true, true, true, true], 3) { |bool| bool }

# true

**every?**

Write a method every? that accepts an array and a block as arguments. The method should return a boolean indicating whether or not all elements of the array return true when given to the block. Solve this using Array#each.

Examples

p every?([3, 1, 11, 5]) { |n| n.even? } # false

p every?([2, 4, 4, 8]) { |n| n.even? } # true

p every?([8, 2]) { |n| n.even? } # true

p every?(['squash', 'corn', 'kale', 'carrot']) { |str| str[0] == 'p' } # false

p every?(['squash', 'pea', 'kale', 'potato']) { |str| str[0] == 'p' } # false

p every?(['parsnip', 'potato', 'pea']) { |str| str[0] == 'p' } # true

**at\_most?**

Write a method at\_most? that accepts an array, a number (n), and a block as arguments. The method should return a boolean indicating whether no more than n elements of the array return true when given to the block. Solve this using Array#each.

Examples

p at\_most?([-4, 100, -3], 1) { |el| el > 0 } # true

p at\_most?([-4, -100, -3], 1) { |el| el > 0 } # true

p at\_most?([4, 100, -3], 1) { |el| el > 0 } # false

p at\_most?([4, 100, 3], 1) { |el| el > 0 } # false

p at\_most?(['r', 'q', 'e', 'z'], 2) { |el| 'aeiou'.include?(el) } # true

p at\_most?(['r', 'i', 'e', 'z'], 2) { |el| 'aeiou'.include?(el) } # true

p at\_most?(['r', 'i', 'e', 'o'], 2) { |el| 'aeiou'.include?(el) } # false

**first\_index**

Write a method first\_index that accepts an array and a block as arguments. The method should return the index of the first element of the array that returns true when giben to the block. If no element of returns true, then the method should return nil. Solve this using Array#each.

Examples

p first\_index(['bit', 'cat', 'byte', 'below']) { |el| el.length > 3 } # 2

p first\_index(['bitten', 'bit', 'cat', 'byte', 'below']) { |el| el.length > 3 } # 0

p first\_index(['bitten', 'bit', 'cat', 'byte', 'below']) { |el| el.length > 6 } # nil

p first\_index(['bit', 'cat', 'byte', 'below']) { |el| el[0] == 'b' } # 0

p first\_index(['bit', 'cat', 'byte', 'below']) { |el| el.include?('a') } # 1

p first\_index(['bit', 'cat', 'byte', 'below']) { |el| el[0] == 't' } # nil

**Phase 2: The proc thickens.**

**xnor\_select**

Write a method xnor\_select that accepts an array and two procs as arguments. The method should return a new array containing elements of the original array that either return true for both procs or return false for both procs.

Examples

is\_even = Proc.new { |n| n % 2 == 0 }

is\_odd = Proc.new { |n| n % 2 != 0 }

is\_positive = Proc.new { |n| n > 0 }

p xnor\_select([8, 3, -4, -5], is\_even, is\_positive) # [8, -5]

p xnor\_select([-7, -13, 12, 5, -10], is\_even, is\_positive) # [-7, -13, 12]

p xnor\_select([-7, -13, 12, 5, -10], is\_odd, is\_positive) # [5, -10]

**filter\_out!**

Rewrite your previous filter\_out method to mutate the input array instead of returning a new array. That is, write a method filter\_out! that accepts an array and a block as args. The method should remove elements of the input array that return true when given to the block. Solve this without using Array.reject!.

Examples

arr\_1 = [10, 6, 3, 2, 5 ]

filter\_out!(arr\_1) { |x| x.odd? }

p arr\_1 # [10, 6, 2]

arr\_2 = [1, 7, 3, 5 ]

filter\_out!(arr\_2) { |x| x.odd? }

p arr\_2 # []

arr\_3 = [10, 6, 3, 2, 5 ]

filter\_out!(arr\_3) { |x| x.even? }

p arr\_3 # [3, 5]

arr\_4 = [1, 7, 3, 5 ]

filter\_out!([1, 7, 3, 5 ]) { |x| x.even? }

p arr\_4 # [1, 7, 3, 5]

**multi\_map**

Write a method multi\_map that accepts an array, an optional number (n), and a block as arguments. The method should return a new array where each element of the original array is repeatedly run through the block n times. If the number argument is not passed in, then the the elements should be run through the block once.

Examples

p multi\_map(['pretty', 'cool', 'huh?']) { |s| s + '!'} # ["pretty!", "cool!", "huh?!"]

p multi\_map(['pretty', 'cool', 'huh?'], 1) { |s| s + '!'} # ["pretty!", "cool!", "huh?!"]

p multi\_map(['pretty', 'cool', 'huh?'], 3) { |s| s + '!'} # ["pretty!!!", "cool!!!", "huh?!!!"]

p multi\_map([4, 3, 2, 7], 1) { |num| num \* 10 } # [40, 30, 20, 70]

p multi\_map([4, 3, 2, 7], 2) { |num| num \* 10 } # [400, 300, 200, 700]

p multi\_map([4, 3, 2, 7], 4) { |num| num \* 10 } # [40000, 30000, 20000, 70000]

**proctition**

Write a method proctition that accepts an array and a block as arguments. The method should return a new array where the elements that return true when given to the block come before all of the elements that return false when given to the block. This means that the new array has two partitions; the order among elements in the same partition should be the same as their relative order in the input array.

Examples

p proctition([4, -5, 7, -10, -2, 1, 3]) { |el| el > 0 }

# [4, 7, 1, 3, -5, -10, -2]

p proctition([7, 8, 3, 6, 10]) { |el| el.even? }

# [8, 6, 10, 7, 3]

p proctition(['cat','boot', 'dog', 'bug', 'boat']) { |s| s[0] == 'b' }

# ["boot", "bug", "boat", "cat", "dog"]

**Phase 3: Perilous.**

**selected\_map!**

Write a method selected\_map! that accepts an array and two procs as arguments. The method should replace the elements that return true when passed into the first proc with their return values when they are passed into the second proc. This method should mutate the input array and return nil.

Examples

is\_even = Proc.new { |n| n.even? }

is\_positive = Proc.new { |n| n > 0 }

square = Proc.new { |n| n \* n }

flip\_sign = Proc.new { |n| -n }

arr\_1 = [8, 5, 10, 4]

p selected\_map!(arr\_1, is\_even, square) # nil

p arr\_1 # [64, 5, 100, 16]

arr\_2 = [-10, 4, 7, 6, -2, -9]

p selected\_map!(arr\_2, is\_even, flip\_sign) # nil

p arr\_2 # [10, -4, 7, -6, 2, -9]

arr\_3 = [-10, 4, 7, 6, -2, -9]

p selected\_map!(arr\_3, is\_positive, square) # nil

p arr\_3 # [-10, 16, 49, 36, -2, -9]

**chain\_map**

Write a method that accepts any value and an array of procs as an argument. The method should return the final result of feeding the value through all of the procs. For example, if the array contains three procs, then:

* the value is given to the first proc
* the result of the first proc is given to the second proc
* the result of the second proc is given to the third proc
* the result of third proc is the final result

Examples

add\_5 = Proc.new { |n| n + 5 }

half = Proc.new { |n| n / 2.0 }

square = Proc.new { |n| n \* n }

p chain\_map(25, [add\_5, half]) # 15.0

p chain\_map(25, [half, add\_5]) # 17.5

p chain\_map(25, [add\_5, half, square]) # 225

p chain\_map(4, [square, half]) # 8

p chain\_map(4, [half, square]) # 4

**proc\_suffix**

Write a method proc\_suffix that accepts a sentence and a hash as arguments. The hash contains procs as keys and suffix strings as values. The method should return a new sentence where each word of the original sentence is appended with a suffix if the **original** word returns true when given to the corresponding proc key. If an original word returns true for multiple procs, then the suffixes should be appended in the order that they appear in the input hash.

Examples

contains\_a = Proc.new { |w| w.include?('a') }

three\_letters = Proc.new { |w| w.length == 3 }

four\_letters = Proc.new { |w| w.length == 4 }

p proc\_suffix('dog cat',

contains\_a => 'ly',

three\_letters => 'o'

) # "dogo catlyo"

p proc\_suffix('dog cat',

three\_letters => 'o',

contains\_a => 'ly'

) # "dogo catoly"

p proc\_suffix('wrong glad cat',

contains\_a => 'ly',

three\_letters => 'o',

four\_letters => 'ing'

) # "wrong gladlying catlyo"

p proc\_suffix('food glad rant dog cat',

four\_letters => 'ing',

contains\_a => 'ly',

three\_letters => 'o'

) # "fooding gladingly rantingly dogo catlyo"

**proctition\_platinum**

Write a method proctition\_platinum that accepts an array and any number of additional procs as arguments. The method should return a hash where the keys correspond to the number of procs passed in.

* For example, if three procs are passed in, then the hash should have the keys 1, 2, and 3.

The values associated with each key should be an array containing the elements of the input array that return true when passed into the corresponding proc.

* For example, this means that the array that corresponds to the key 2 should contain the elements that return true when passed into the second proc.

If an element returns true for multiple procs, then it should only be placed into the array that corresponds to the proc that appears first in the arguments.

Examples

is\_yelled = Proc.new { |s| s[-1] == '!' }

is\_upcase = Proc.new { |s| s.upcase == s }

contains\_a = Proc.new { |s| s.downcase.include?('a') }

begins\_w = Proc.new { |s| s.downcase[0] == 'w' }

p proctition\_platinum(['WHO', 'what', 'when!', 'WHERE!', 'how', 'WHY'], is\_yelled, contains\_a)

# {1=>["when!", "WHERE!"], 2=>["what"]}

p proctition\_platinum(['WHO', 'what', 'when!', 'WHERE!', 'how', 'WHY'], is\_yelled, is\_upcase, contains\_a)

# {1=>["when!", "WHERE!"], 2=>["WHO", "WHY"], 3=>["what"]}

p proctition\_platinum(['WHO', 'what', 'when!', 'WHERE!', 'how', 'WHY'], is\_upcase, is\_yelled, contains\_a)

# {1=>["WHO", "WHERE!", "WHY"], 2=>["when!"], 3=>["what"]}

p proctition\_platinum(['WHO', 'what', 'when!', 'WHERE!', 'how', 'WHY'], begins\_w, is\_upcase, is\_yelled, contains\_a)

# {1=>["WHO", "what", "when!", "WHERE!", "WHY"], 2=>[], 3=>[], 4=>[]}

**procipher**

Write a method procipher that accepts a sentence and a hash as arguments. The hash contains procs as both keys and values. The method should return a new sentence where each word of the input sentence is changed by a value proc if the **original** word returns true when passed into the key proc. If an original word returns true for multiple key procs, then the value proc changes should be applied in the order that they appear in the hash.

Examples

is\_yelled = Proc.new { |s| s[-1] == '!' }

is\_upcase = Proc.new { |s| s.upcase == s }

contains\_a = Proc.new { |s| s.downcase.include?('a') }

make\_question = Proc.new { |s| s + '???' }

reverse = Proc.new { |s| s.reverse }

add\_smile = Proc.new { |s| s + ':)' }

p procipher('he said what!',

is\_yelled => make\_question,

contains\_a => reverse

) # "he dias ???!tahw"

p procipher('he said what!',

contains\_a => reverse,

is\_yelled => make\_question

) # "he dias !tahw???"

p procipher('he said what!',

contains\_a => reverse,

is\_yelled => add\_smile

) # "he dias !tahw:)"

p procipher('stop that taxi now',

is\_upcase => add\_smile,

is\_yelled => reverse,

contains\_a => make\_question

) # "stop that??? taxi??? now"

p procipher('STOP that taxi now!',

is\_upcase => add\_smile,

is\_yelled => reverse,

contains\_a => make\_question

) # "STOP:) that??? taxi??? !won"

**picky\_procipher**

Write a method picky\_procipher that accepts a sentence and a hash as arguments. The hash contains procs as both keys and values. The method should return a new sentence where each word of the input sentence is changed by a value proc if the **original** word returns true when passed into the key proc. *If an original word returns true for multiple key procs, then only the value proc that appears earliest in the hash should be applied.*

Examples

is\_yelled = Proc.new { |s| s[-1] == '!' }

is\_upcase = Proc.new { |s| s.upcase == s }

contains\_a = Proc.new { |s| s.downcase.include?('a') }

make\_question = Proc.new { |s| s + '???' }

reverse = Proc.new { |s| s.reverse }

add\_smile = Proc.new { |s| s + ':)' }

p picky\_procipher('he said what!',

is\_yelled => make\_question,

contains\_a => reverse

) # "he dias what!???"

p picky\_procipher('he said what!',

contains\_a => reverse,

is\_yelled => make\_question

) # "he dias !tahw"

p picky\_procipher('he said what!',

contains\_a => reverse,

is\_yelled => add\_smile

) # "he dias !tahw"

p picky\_procipher('stop that taxi now',

is\_upcase => add\_smile,

is\_yelled => reverse,

contains\_a => make\_question

) # "stop that??? taxi??? now"

p picky\_procipher('STOP that taxi!',

is\_upcase => add\_smile,

is\_yelled => reverse,

contains\_a => make\_question

) # "STOP:) that??? !ixat"

† pun intended