

Practice Problems in Julia for Basic Syntax

These should be doable for all of you. This is just some warm up exercise, practice and try to solve we will talk about the solutions on Monday.

Question 1. Write a function **divisible_two_three(x)**, which for any **x** will do the following.

- a. check whether **x** is divisible by 2 and not by 3
- b. check whether **x** is divisible by 3 and not by 3
- c. check whether **x** is divisible by 2 and 3,
- d. check whether **x** is neither divisible by 2 nor by 3,

(Hint: It is a simple function which will check different **if else** conditions. To check the divisibility you can use the remainder operator **%**, e.g., if **x** is divisible by **5** then it means **x % 5 == 0** will be **true**, i.e., there is no remainder)

```
# After you define the function you can check whether your function is working like this

julia> divisible_two_three(21.3)
# Neither divisible by 3 nor by 2

julia> divisible_two_three(5)
# Neither divisible by 3 nor by 2

julia> divisible_two_three(21)
# Divisible by 3 and not by 2

julia> divisible_two_three(6)
# Divisible by 2 and 3

julia> divisible_two_three(26)
# Divisible by 2 and not by 3

julia> divisible_two_three(0)
# Divisible by 2 and 3
```

Question 2. Write a function **count_vowels(s)** which will take any string **s** in English alphabets, and will count the number of vowels there (You already know but the vowels but here they are: **a, e, i, o, u**)

```
# YOu can check the function like this way

julia> count_vowels("laslkasas")
# The number of vowels is 3

julia> count_vowels("laksaksasklas")
```

```
# The number of vowels is 4

julia> count_vowels("ooiiuubbnmmzddssgghhaqwweellaakkjja")
# The number of vowels is 12
```

Question 3. Write a function **count_ints(x)** such that for any vector of integers **x**, it should return the total number of integers between 0 to 9.

```
# You can check your function like this
julia> x = [1, 3, 4, 5]
# 4-element Array{Int64,1}:
#  1
#  3
#  4
#  5

julia> count_ints(x)
# 4

julia> x = [1, 3, 44, 6, 88, 101]
# 6-element Array{Int64,1}:
#  1
#  3
# 44
#  6
# 88
# 101

julia> count_ints(x)
# 3
```

Question 4. Write a function **find_cube_root(x)** which will only accept integer **x** and find the integer cube root. In other words, if **x** is a perfect cube it will return the cube root. If **x** is not a perfect cube, then it will print it is not. You can only use the basic functionalities, like loops, if else, and mathematical operations and also you are allowed to use **abs** function in julia.

(Hint: Start from a initial guess, lets say $root = 0$, ideal situation is we want $(root)^3 = x$. So one idea is, with a loop we can increment the $root$ by 1 till $(root)^3 < x$. And then after the loop, the resulting root will be the cube root if x is a perfect cube. However if after the loop, $(root)^3 \neq x$, then you know x is not a perfect cube. You should think about negative values as well.)

```
# here is how you can check

julia> find_cube_root(12)
# 12 is not a perfect cube
```

```
julia> find_cube_root(8)
# The cube root of 8 is 2

julia> find_cube_root(9)
# 9 is not a perfect cube

julia> find_cube_root(27)
# The cube root of 27 is 3

julia> find_cube_root(-27)
# The cube root of -27 is -3
```

Question 5: In this exercise you need to write a function `primes_to(n)` which will find all the prime numbers less than given a value `n` using Sieve of Eratosthenes algorithm.

So what are prime numbers? A prime number (or a prime) is a natural number greater than 1 that is not a product of two smaller natural numbers. This means if we divide this by any other natural number other than 1 and itself, the result will not be another natural number (of course excluding 1 itself). For example, 2, 3, 5, 7 are all primes, and on the other hand 1, 4, 6, 8 are not (You can read more about primes if you want!)

How can we find all the prime numbers less than or equal to a given integer `n`? One way to follow what is known as *Eratosthenes* method, you can find some details here about the algorithm, at [wiki page](#). You can also watch the explanation of how the algorithm at this [nice video by Eddie Woo](#).

- Try to implement first without the refinements
- Then try with the refinements.

Here is psuedo-code without the refinements

```
input: an integer n > 1.
output: all prime numbers from less than n.

Let,
A = an array of Boolean values of size n, initially all set to true.
A[1] = false

p = 2

while p ≤ n do
    if A[p] is true
        j = 2p
        while j ≤ n
            A[j] = false
            j = j + p
```

```
p = p + 1
```

```
return all p such that A[p] is true.
```

Hints:

- 1. you can create an array of full of Bool value **true** different ways. Two easy solutions are using **ones** function in Julia (make sure to apply the correct type) or **fill** function in Julia
- 2. To return "all p such that A[p] is true" you can use **findall** function.
- 3. But of course you can try other ways too.

```
# here is to check
```

```
julia> primes_to(10)'  
# 1×4 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:  
#  2  3  5  7
```

```
julia> primes_to(20)'  
# 1×8 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:  
#  2  3  5  7 11 13 17 19
```

```
julia> primes_to(5)'  
# 1×3 LinearAlgebra.Adjoint{Int64,Array{Int64,1}}:  
#  2  3  5
```