**BACKGROUND**

A *XOR cipher* is an encryption method on which we take a text file, convert the bytes to ASCII, then XOR each byte with a given value, taken from a secret key. The advantage with the XOR function is that using the same encryption key on the cipher text, restores the plain text; for example, 65 XOR 42 = 107, then 107 XOR 42 = 65.

A common way to use a XOR cipher is with relatively long text and a relatively short key, on which case the key is repeated cyclically throughout the message:

‘apoquahgbabfp’ XOR ‘xyz’ =

‘a’ XOR ‘x’

‘p’ XOR ‘y’

‘o’ XOR ‘z’

‘q’ XOR ‘x’

‘u’ XOR ‘y’, ...

In this project, you are required to build an **encryption breaking server** that gets an **encrypted text** and a **key size** on a web form - **without** the key itself - and works to find the encryption key using a brute-force lookup.

**Please note.** This project is intended to test both raw development and general technical skills. For the latter part you will be expected to work in a linux OS environment and do various system configurations. Please treat these parts as integral to the project.

**WORK ENVIRONMENT**

You are encouraged to write this exercise in Python and run it in a Linux environment, but you can choose a different setup if you’re not comfortable with Python. Please try to choose based on the following priority:

1. Python under Linux/Mac - most preferred.
2. Python on any other operating system - 2nd priority
3. Any other language/framework - 3rd priority.

**Please take your time to get comfortable, make sure you can comfortably write code and run it.**

**PART A**

This part should take around 60 minutes.

1. Write a function **xor\_decrypt(encrypted\_text, key)** that gets a list of numbers and a key (as string) and decrypts this list into a text (as string again). Make sure that xor\_decrypt(xor\_encrypt(text, key), key) == text.  
   Example:  
    xor\_decrypt([24, 24, 26, 30, 28], "yz") -> “abcde”  
   Explanation:  
    ‘y’ = 121 -> 24 XOR 121 = 97 -> 97 = ‘a’ (or in short: 24 XOR ‘y’ = ‘a’)  
    24 XOR ‘z’ = ‘b’  
    26 XOR ‘y’ = ‘c’  
    30 XOR ‘z’ = ‘d’  
    28 XOR ‘y’ = ‘e’
2. Write a function **guess\_key(encrypted\_text, key\_size)** that tries to break a XOR encryption by a brute-force attack under the following assumptions:
   1. The decrypted text contains a standard English text with merely letters, numbers, punctuation marks etc. You can implement some basic result filters based on this assumption
   2. The encryption key consists of *key\_size* lower-case English characters.

The function will return a list of possible matches as pairs of key and  
 possibly-decrypted text.

**PART B**

This part should take around 30 minutes. However, it may take a bit longer if you need to familiar yourself with the environment/frameworks in use

1. The goal is to create a “Crack My Password” application, using the methods created in the last part.
2. We created for you a frontend that after you initialize it should be deployed on [http://localhost:3000](http://localhost:3000/). You can find the frontend in the **vantage-exam-frontend.zip** file provided, which can be extracted and deployed using “**yarn install &** **yarn start”**. The UI has an ability to specify an encrypted text and key size.
3. Set up a HTTP server in port **5000** with any framework you want (can be Django, Flask, Sanic, Tornado..). The HTTP server should be able to respond to the following HTTP endpoint:  
   **Request should look like:**  
   URL: /bruteforce/  
   METHOD: POST  
   Body (using JSON): {“text”: “my-text”, “size”: 1}  
     
   **Response should look like:**Headers: {“Content-Type”: “application/json”}  
   Body (using JSON): [[“aa”, “asdrt#21”], [“ab”, “@4sawet”]]  
     
   As you can see, the response is a list of possible matches. For each potential match please show the key and the resulting text after decryption.
4. Run your server, and make sure the client works with your backend. In case of trouble, please troubleshoot the issue.
5. You should also receive a file named “encrypted.txt” with encrypted text. It was encrypted using a key of size 3. Use the “Crack My Password” page to find the key for this text. This may require you to add more validations to guess\_key in order to output more focused results.

**PART C**

Those are just some topics to consider for the follow up discussion (no need to implement anything, just write down some notes for yourself):

1. In **guess\_key** you implemented basic result filters. Can you think of more complex filters you could have written assuming you will implement this is a full flagged sysytem?
2. What would happen if a large key\_size is used? How many keys and results would your system generate, and what would be the implications for the frontend and backend?
3. Assuming we would like to be able to support large key\_size (say, 10 or 12). How can we design a system (frontend and backend) which will be able to provide a user with results and allow them to go over them, without using an unreasonable amount of resources on both ends.

Final reminder: this test is yours and yours only. We trust you to do it alone and struggle with its various components. You can, and **should** (of course), consult with online resources.