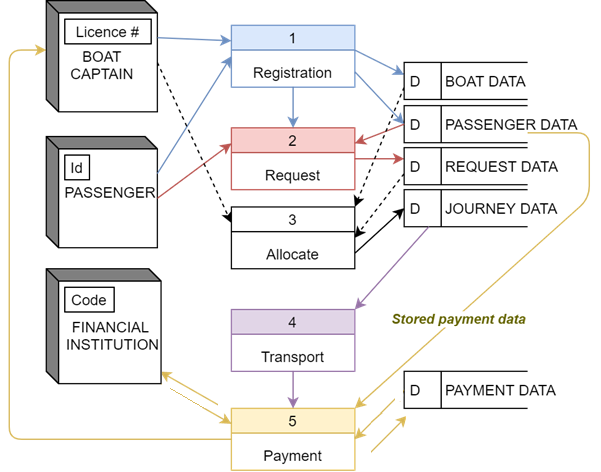
1. This is one potential solution of many, many solutions. To improve this further, annotate the data flow arrows. **Data flow arrows are not supposed to cross over each other**… sometimes it is hard though, so just do your best. More information on DFDs straight from the syllabus here: [**https://digisoln.com/technical\_representation/dfd**](https://digisoln.com/technical_representation/dfd) **IT IS WORTH A READ!**



1. Pseudocode sample (easier to read if you turn off spell check), assumes:
   1. ***length*** is pre-defined function that returns count of elements in an array
   2. ***ord*** and ***chr*** to convert from Unicode integers to characters
   3. ***mod*** as modulo division (remainder division)

BEGIN  
 INPUT plain\_text AS ARRAY  
 INPUT key1 AS ARRAY  
 INPUT key2 AS ARRAY  
 VARIABLE cipher\_text AS ARRAY  
 VARIABLE i AS INTEGER

i = 0  
 WHILE ( i < ***length***(key1) or i < ***length***(key2) )  
 base26\_char = ***ord***(plain\_text[i]) - ***ord***("A")  
 key1\_shift = ***ord***(key1[i]) - ***ord***("A")  
 key2\_shift = key2[i]  
 cipher\_num = (base26\_char + key1\_shift + key2\_shift) ***mod*** 26  
 cipher\_char = ***chr***( cipher\_num + ***ord***("A") )  
 cipher\_text[i] = cipher\_char  
 i = i + 1  
 END WHILE  
END

In Python (FYI):

|  |
| --- |
| plain\_text = ["Z","E","B","R","A"]  key1 = ["A","B","B","B","A"]  key2 = [1,2,1,2,1]  cipher\_text = []  i = 0  while i < len(key1): #or key2:  base26\_char = ord(plain\_text[i]) - ord("A")  key1\_shift = ord(key1[i]) - ord("A")  key2\_shift = key2[i]  cipher\_num = (base26\_char + key1\_shift + key2\_shift) % 26  cipher\_char = chr(cipher\_num+ord("A"))  cipher\_text.append(cipher\_char)  i=i+1  print(cipher\_text) |

If ["A", "I", "B"] generates ["W", "I", "Z"], then this is a total shift of [4, 0, 2].

Looking closer at the shift we see the values are (floor) divided by 2:

shift = (base – (key // 2)) mod 26

Given that floor division by 2 is used, the original shift values would have been **[8, 0, 4].**

key = BASE26(key1[counter])

This statement shows the key was in letters, therefore the original encryption key given would have been the characters representing [8,0,4] which are:

Encryption Key

|  |  |  |
| --- | --- | --- |
| **I** | **A** | **E** |

4.

BEGIN UNICODE\_CHARACTER(Base26 integer, Boolean upper):  
 IF (upper) THEN  
 RETURN character(integer + ordinal("A"))  
 ELSE  
 RETURN character(integer + ordinal("a"))  
 END IF  
END UNICODE\_CHARACTER

*Extension*: **BASE26** function without using the keyword **ELSE**:

BEGIN UNICODE\_CHARACTER(Base26 integer, Boolean upper):  
 IF (upper) THEN  
 RETURN character(integer + ordinal("A"))  
 END IF  
 RETURN character(integer + ordinal("a"))  
END UNICODE\_CHARACTER

5. The colours are shown:

* 1. Caesar Shift
  2. Polyalphabetic ciphers (e.g. Vigenère and Gronsfeld cipher)
  3. One-time Pad
  4. Hashing

|  |  |  |  |
| --- | --- | --- | --- |
| Plaintext: hi Pad: abcd  Ciphertext:  hj | Uses a series of interwoven Caesar ciphers based on the letters of a keyword | SHA-256 output will always have a fixed 256-bits length | Only has a maximum of 26 possible key combinations |
| Weakest of the 4 encryption or hashing techniques listed here | The length of the key guarantees that the ciphertext is not vulnerable due to repetition in the way that the Vigenère cipher is | A key of “N” or 13 will give the same message, whether shifting right or left (ROT13) | If plaintext is longer than the cipher key, wrap back in the key and reuse the letters of the key in order |
| taking an input string of any length and giving out an output of a fixed length | **Monoalphabetic**: cipher uses fixed substitution over the entire message | Provided the key is kept secret and never reused, this is the strongest form of encryption here, given its true randomness | A salt is random data that is used as an additional input to a one-way function, to defend against pre-computed hash matching |
| Identical to Vigenère cipher, except numbers are used as the key instead of letters | Used historically by KGB officers | one-way cryptographic algorithm | Plaintext: abcd Key: bc  Ciphertext:  bddf |

6.

Output is ['x', 'y', 'x', 'y', 'x'], which is useful to wrap key in Vigenère (would work for Gronsfeld too).

In Python (FYI):

def KEYWRAP(plaintext, key):  
 index = 0  
 counter = len(key)  
 while counter < len(plaintext):  
 value = key[index]  
 key.append(value) #key[counter] = value  
 if index < len(key):  
 index = index + 1  
 else:  
 index = 0  
 #END IF  
 counter = counter + 1  
 #END WHILE  
 return key  
#END KEYWRAP

key = ["x","y"]  
plaintext = ["h","e","l","l","o"]  
key = KEYWRAP(plaintext, key)  
print(key)

7. *Note*: the extension to this question suggests rewriting the algorithms so that so that they do not have to assume capital letters as plaintext. Use answer to question 4 if you want to do this. You could further modify for spaces and punctuation; however it must be stressed maintaining plain text **patterns** in encryption is a **bad idea** for security.

**Analysing the main difference between the 4:**

Caesar: the shift amount is constant (equivalent to a key of [3, 3, 3, 3, 3, ... ] )

**Caesar**  
   
 CONSTANT key = 3

Gronsfeld: the key is an array of variable integers, which wraps repeatedly the length of the plain text message:

**Gronsfeld**  
   
 INPUT key1 as Array  
 key1 = KEYWRAP(plaintext, key1)  
   
 ...  
 key = key1[counter]

Vigenère: same as Gronsfeld (i.e. key wraps length of message), but key is letters:

**Vigenère**  
  
 key = BASE26(key1[counter])

One-time Pad: Key will always be longer than message, so no need to wrap:

**One-time Pad**

INPUT key1 as Array

8. I only did some of the harder queries, hopefully you are OK with the rest:

Query XVIII:

SELECT COUNT(\*), category  
FROM books  
WHERE title LIKE "%the%" OR title LIKE "%The%"  
GROUP BY category

Query IX, X, XI, XII, XIII:

SELECT COUNT(\*), AVG(price), category  
FROM books  
WHERE price < 20  
GROUP BY category  
HAVING AVG(price) > 16 AND NOT(COUNT(\*)==1)  
ORDER BY AVG(price) DESC

Query XIV, XV:

SELECT title, category, price  
FROM books  
WHERE price > (  
 SELECT price  
 FROM books  
 WHERE title == "New Moon"  
)  
ORDER BY price DESC, category ASC

|  |  |
| --- | --- |
| *Statements executed in order:* | What does this SQL do? |
| CREATE TABLE 'sales' (  'book\_num' INTEGER,  'cust\_email' INTEGER,  PRIMARY KEY('book\_num','cust\_email')  ); | Creates table sales with composite primary key (i.e. the combination of book\_num and cust\_email will uniquely identify any row) |
| SELECT sales.cust\_email, books.title  FROM books INNER JOIN sales  ON books.num == sales.book\_num | Gives book titles and emails of **purchased** books only |
| SELECT s.cust\_email, b.title  FROM books b LEFT JOIN sales s  ON b.num == s.book\_num | Gives **all** book titles (even the ones that weren’t purchased) and the emails of customers that purchased those books |

9. see answer to question 7 about patterns. Patterns make it more obvious what you are trying to hide.

10. Explain the following algorithmic terms **using the samples** provided:

|  |  |  |
| --- | --- | --- |
| Sample | **Criteria**  *Terms* | Explanation |
| BEGIN module1  REPEAT  do long process  UNTIL job done  END module1  BEGIN module2  WHILE job not done  do long process  END WHILE  END module2 | **Efficiency**  *Pre-test vs Post-test loops* | module1 post-test happens minimum **ONCE**  module2 happens minimum of **ZERO** times (pre-test)  If long process is already done before module launch, it should be avoided. This one more efficient in this case. |
| BEGIN  x = INPUT team\_scoreA  y = INPUT team\_scoreB  winner1(x,y)  winner2(x,y) END  BEGIN winner1(a,b)  IF a > b THEN  PRINT "Winner Team A"  ELSE  PRINT "Winner Team B"  END IF END  BEGIN winner2(a,b)  IF a == b THEN  PRINT "Teams are drawn"  ELSE  IF a > b THEN  PRINT "Winner Team A"  ELSE  PRINT "Winner Team B"  END IF  END IF END | **Accuracy**  **Modularity**  *Selection - Multiple branches*  *Global vs Local variables* | winner1 does not account for draw, and incorrectly would reward Team B as winner in a draw.  winner2 accounts for draw  Therefore, winner2 more accurate algorithm in this case.  Modularity shown with both winner functions. Assuming x and y global (from main algorithm), a and b local to modules (these are passed in as parameters to the function calls). This means value of a in module1 cannot be accessed by module2. Values in x and y can (likely) be accessed in both modules, although better practise (perhaps) would be to label these GLOBAL. |