🛥 Uber Boats 🛥

In the current pandemic, unused passenger boats are operating a taxi service for residents.

First, a BOAT CAPTAIN must *sign up* for the system, supplying their *unique* marine license number, mobile phone number, password salt and hash, profile display name, and profile image link. This BOAT DATA must be recorded. Each time thereafter a BOAT CAPTAIN signs in, their *status* will be set to active. When they sign out, their *status* will be set to inactive.

A PASSENGER may sign up as well, by supplying CUSTOMER DATA including their *unique* mobile phone number, password salt and hash, and stored payment method and account debit number.

Once signed in, a PASSENGER can *lodge* a request for transportation. This transportation REQUEST DATA must be stored, and will contain passenger phone number, time/date stamp of request lodged, as well as origin and destination GPS latitude and longitude.

An active BOAT CAPTAIN can allocate themselves a transportation request if they are within a 1km radius of the origin GPS coordinates. The resulting allocation must be stored as JOURNEY DATA, and includes passengers unique account identifier (i.e. mobile phone number), boat captain unique identifier (i.e. marine license number), and meta data about the journey itself which will be null on first entry – such as feedback comment, feedback rating, and completion time/date stamp.

After the boat captain transports the passenger as per agreed service, the passengers unique account identifier (i.e. mobile phone number), total journey cost and payment status (default “PROCESSING”) is recorded as PAYMENT DATA. There may be a more eloquent solution to “link” these tables relationally not specified here.

Finally, the PAYMENT DATA is sent to Ubers FINANCIAL INSTITUTION (unspecified for this context) for processing, and includes all records in the PAYMENT DATA, as well as an authorisation token. Once the payments have been processed by the FINANCIAL INSTITUTION, the result triggers an SQL query in the PAYMENT DATA to set sent payments status to “PROCESSED”.

1. Draw a **DFD** to symbolise the flow of data through the above system. If you are feeling hungry for more logical and conceptual symbolisation, draw an ERD as well, indicating cardinality between entities, as well as field data types.
2. Generate the algorithm that determines the encryption process for the following block cipher:

Encryption

Process (?)

Block of ciphertext

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **H** | **D** | **U** | **B** |

Block of plain Text

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Z** | **E** | **B** | **R** | **A** |

Encryption Keys

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Key1** | **A** | **B** | **B** | **B** | **A** |
| **Key2** | **1** | **2** |  |  |  |

1. Determine the value of the Array variable key1 in the following block **decipher**. Note that // is the symbol used for DIV (aka floor division or integer division). It is used in this case as regular division would be however // ensures that an integer is returned (and not a float).

Block of plain Text

|  |  |  |
| --- | --- | --- |
| **W** | **I** | **Z** |

BEGIN
 VARIABLE counter = 0
 VARIABLE plaintext = []
 INPUT key1 as Array
 INPUT ciphertext as Array
 WHILE counter < length(ciphertext)
 base = BASE26(ciphertext[counter])
 key = BASE26(key1[counter])
 shift = (base – (key // 2)) mod 26
 plaintext[counter] = UNICODE\_CHARACTER(shift)
 counter = counter + 1
 END WHILE
END

BEGIN BASE26(Unicode char)
 RETURN ordinal(char) – ordinal("A")
END BASE26

BEGIN UNICODE\_CHARACTER(Base26 integer)
 RETURN character(integer + ordinal("A"))
END UNICODE\_CHARACTER

Block of ciphertext

|  |  |  |
| --- | --- | --- |
| **A** | **I** | **B** |

Encryption Key (?)

IAE

1. In the previous question, we did not account for upper- and lower-case letters. The BASE26 algorithm from the previous question has been modified to allow for this as follows:

BEGIN BASE26(Unicode char)
 IF uppercase(char) THEN
 RETURN ordinal(char) – ordinal("A")
 ELSE
 RETURN ordinal(char) – ordinal("a")
 END IF
END BASE26

Re-write the **UNICODE\_CHARACTER** algorithm from the previous question, so that the new **UNICODE\_CHARACTER** function takes 2 parameters: the Base26 integer value to convert, as well as a BOOLEAN value **upper** (either True or False), that determines whether the *return Unicode character* should be upper or lower case:

BEGIN UNICODE\_CHARACTER(Base26 integer, Boolean upper):
 ...

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

1. The following grid contains 4x4 squares. Each square contains a property relating to 1 of these 4 topics:
	1. Caesar Shift
	2. Polyalphabetic ciphers (e.g. Vigenère and Gronsfeld cipher)
	3. One-time Pad
	4. Hashing

Insert the corresponding letter of the topic into the square with the content that best matches the topic:

|  |  |  |  |
| --- | --- | --- | --- |
| Plaintext: hiPad: abcdCiphertext: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: abcdKey: bcCiphertext:bddf |

1. For the following algorithm:
	1. What is the output?
	2. Which classical, polyalphabetic cipher could this algorithm be useful for?

BEGIN
 VARIABLE key = ["x","y"]
 VARIABLE plaintext = ["h","e","l","l","o"]
 key = KEYWRAP(plaintext, key)
 PRINT key
END

BEGIN KEYWRAP(plaintext, key):
 index = 0
 counter = length(key)
 WHILE counter < length(plaintext)
 value = key[index]
 key[counter] = value
 IF index < length(key) THEN
 index = index + 1
 ELSE
 index = 0
 END IF
 counter = counter + 1
 END WHILE
 RETURN key
END KEYWRAP

1. All the following encryption algorithms in the **blue boxes** assume capital letter plaintext variables. All make use of the *helper* functions in the green box on the last page. **Analyse the main difference between the 4** – what makes each unique?

*Extension*: Re-write these so that they do not have to assume capital letters as plaintext... and what about spaces and punctuation marks?

BEGIN **One-time Pad**
 VARIABLE counter = 0
 INPUT plaintext as Array
 INPUT key1 as Array
 VARIABLE ciphertext = []
 WHILE counter < length(plaintext)
 base = BASE26(plaintext[counter])
 key = BASE26(key1[counter])
 shift = (base + key) mod 26
 ciphertext[counter] = UNICODE\_CHARACTER(shift)
 counter = counter + 1
 END WHILE
END **One-time Pad**

BEGIN **Caesar**
 VARIABLE counter = 0
 CONSTANT key = 3
 INPUT plaintext as Array
 VARIABLE ciphertext = []
 WHILE counter < length(plaintext)
 base = BASE26(plaintext[counter])
 shift = (base + key) mod 26
 ciphertext[counter] = UNICODE\_CHARACTER(shift)
 counter = counter + 1
 END WHILE
END **Caesar**

BEGIN **Gronsfeld**
 VARIABLE counter = 0
 INPUT plaintext as Array
 INPUT key1 as Array
 key1 = KEYWRAP(plaintext, key1)
 VARIABLE ciphertext = []
 WHILE counter < length(plaintext)
 base = BASE26(plaintext[counter])
 key = key1[counter]
 shift = (base + key) mod 26
 ciphertext[counter] = UNICODE\_CHARACTER(shift)
 counter = counter + 1
 END WHILE
END **Gronsfeld**

BEGIN **Vigenère**
 VARIABLE counter = 0
 INPUT plaintext as Array
 INPUT key1 as Array
 key1 = KEYWRAP(plaintext, key1)
 VARIABLE ciphertext = []
 WHILE counter < length(plaintext)
 base = BASE26(plaintext[counter])
 key = BASE26(key1[counter])
 shift = (base + key) mod 26
 ciphertext[counter] = UNICODE\_CHARACTER(shift)
 counter = counter + 1
 END WHILE
END **Vigenère**

BEGIN KEYWRAP(plaintext, key):
 index = 0
 counter = length(key)
 WHILE counter < length(plaintext)
 value = key[index]
 key[counter] = value
 IF index < length(key) THEN
 index = index + 1
 ELSE
 index = 0
 END IF
 counter = counter + 1
 END WHILE
 RETURN key
END KEYWRAP

BEGIN BASE26(Unicode char)
 RETURN ordinal(char) – ordinal("A")
END BASE26

BEGIN UNICODE\_CHARACTER(Base26 integer)
 RETURN character(integer + ordinal("A"))
END UNICODE\_CHARACTER

1. Write SQL queries using the *bookshop.db* database file – <https://digisoln.com/flask/sqlite/getbookshopdb> – to:
	1. Show a list of titles and prices
	2. Show a list of titles and prices of books in either fiction or picture book categories
	3. Show a list of fiction books that cost less than $15
	4. Where’s Wally is on sale at 50% off. How much is it?
	5. Count the number of books for sale authored by Rowling, J.K.
	6. Count the number of books with the word “The” or “the” in the title
	7. Count the number of books in each category
	8. Count the number of books in each category with the word “The” or “the” in the title
	9. Count the number of books in each category that cost less than $20
	10. Count the number of books in each category that cost less than $20, and show the average price of the books remaining in these categories (i.e. the average price of all the books in the categories that cost less than $20)
	11. For the previous query, exclude categories that have an average price of books greater than $16
	12. For the previous query, also exclude categories that have a COUNT of books in the category with only 1 book (i.e. less than 2 books)
	13. For the previous query, order the results by AVG(price) descending
	14. Find the price of New Moon, and in the same query use that price to give a list of all books (title, category and price) that are more expensive than New Moon
	15. Order the previous query by price DESC, then 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')); |  |
| INSERT into sales('book\_num', 'cust\_email') values (7, "me@my.com"); |  |
| UPDATE booksSET category = "comedy"WHERE title = "Twilight" |  |
| DELETE FROM booksWHERE category == "cooking" |  |
| SELECT sales.cust\_email, books.titleFROM books INNER JOIN salesON books.num == sales.book\_num |  |
| SELECT s.cust\_email, b.titleFROM books b LEFT JOIN sales sON b.num == s.book\_num |  |

1. Another implementation of the upper-case handling is shown below.

|  |
| --- |
| Note the Digital Solutions syllabus joins the words **ENDIF** and **ENDWHILE**. This has not been done here to avoid excessive spelling errors underlined in a Word document by default. |

**Question**: Why would you generally want to avoid *case sensitivity*, *punctuation marks* and *spaces* when generating an encrypted cipher text?

 E.g.: “Hi Sarah. I will meet you at D'Aguilar National Park.” >> “Kl Vdudk. L zloo phhw brx dw G'Djxlodu Qdwlrqdo Sdun.”

BEGIN **Caesar**
 VARIABLE counter = 0
 **VARIABLE flag = FALSE**
 CONSTANT key = 3
 INPUT plaintext as Array
 VARIABLE ciphertext = []
 WHILE counter < *length*(plaintext)
 IF *isuppercase*(plaintext[counter]) THEN
 **flag = TRUE**
 END IF
 base = BASE26(plaintext[counter], **flag**)
 shift = (base + key) mod 26
 ciphertext[counter] = UNICODE\_CHARACTER(shift, **flag**)
 counter = counter + 1
 END WHILE
END **Caesar**

BEGIN BASE26(Unicode char, Boolean upper)
 IF upper THEN
 RETURN ordinal(char) – ordinal("A")
 END IF
 RETURN ordinal(char) – ordinal("a")
END BASE26

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

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

|  |  |  |
| --- | --- | --- |
| Sample | **Criteria***Terms* | Explanation |
| BEGIN module1 REPEAT do long process UNTIL job doneEND module1BEGIN module2 WHILE job not done do long process END WHILEEND module2 | **Efficiency***Pre-test vs Post-test loops* |  |
| BEGIN x = INPUT team\_scoreA y = INPUT team\_scoreB winner1(x,y) winner2(x,y)ENDBEGIN winner1(a,b) IF a > b THEN PRINT "Winner Team A" ELSE PRINT "Winner Team B" END IFENDBEGIN 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 IFEND | **Accuracy****Modularity***Selection - Multiple branches**Global vs Local variables* |  |