Report: Candidate Number: 21908

New features:

PermutationKey

- Full error checking for anything the user can input in PermutationKey Class Constructor and Swap function

Attack

Dictionary Functions

- New property 'dictionary'
- Create a custom Dictionary and add/remove as many words/numbers/dates as you like, uppercase or lowercase
- Determine how many words in dictionary are in ciphertext when decrypted with the current key. Displays results in a clean table to the user (see below)

Changes to constructor (Attack) to make class more user friendly. User can now input,

- A=Attack(secret)
- A=Attack(secret,key)
- A=Attack(secret,key,dictionary)
- A=Attack(secret,key,dictionary,past)

Other changes

- When you change the key (for example, by using swap function) the new key displays, and the ciphertext decrypted with that key.
- There is now full error checking for anything the user inputs for all methods in Attack Class.

Useful applications of dictionary functions:

- Useful for spotting frequency of certain 'n' letter words in text. For example, if you suspect that an encrypted word 'FND' is 'AND' in English. You can 'DictionaryAttack' 'FND' to find its frequency of appearance in the text. If the frequency is high, you can then 'DictionaryAttack' the letter 'F' on its own (since 'A' is a very common word) and see if this also has a high frequency of occurrence. If they both have a high frequency, you can then be very sure that 'swap(key,'a','f')' will move you closer to deciphering the key/text.
- Text that a permutation key does not encrypt can also be put into the dictionary, for example dates or numbers (as a string). This may have some useful applications in determining the format of the text.

Features that would have been implemented if more time had been given:

- Improve the method 'attack' so that it uses 'DictionaryAttack' or 'lettercount' to find letters in the ciphertext that are alone (surrounded by a space), counts them and then assigns 'a' to the most common frequency and 'I' too the second most common.
- Use 'Lists' to store text in dictionary. This would have been much more efficient when appending and removing cells (lists would have an O(1) when adding words rather than O(k) when using append).

Example tests in command window

```
A=Attack(secret,'VFCPKGQNHEUYZSOAWBDRXMITJL',{'Can','Two','why','we'})
A=A.AddDictionary('they','when','28/03/2021')
A.DictionaryAttack
A=A.RemoveDictionary('can','why')
A.dictionary
A.past
```

Word_In_Dictionary	How_Many_Times_Word_Appears_In_Ciphertext_Decrypted_With_key
{'CAN' }	9
{'TWO' }	2
{'WHY' }	22
{'WE' }	11
{'THEY' }	7
{'WHEN' }	5
{'28/03/2021'}	0

Efficiency and functionality

To determine how efficient my functions are I decided to make a testing function using MATLAB's inbuilt profiling system. (Figure 1)

As you can see my PermutationKey code seems very efficient. This was expected since I pre-allocated the exact space I needed for every array, rather than using the append function. For example, the append function could have made encryption/decryption rather inefficient for text with many characters. However, there is room for improvement, the constructor has a computation complexity of $O(k^2)$. This could be improved by using a merge sort ($O(klogk)$) to replace the double for-loops. Unfortunately, 'msort' was integrated into the 'permutation.m' file, so I couldn't use it without calling 'permutation'. However, since the maximum length of the for-loops was 26, it was still fairly efficient (PermutationKey called 36 times 0.003s).

Functions 'Lettercount' and 'sample' are as efficient as they can be. However, because of my constructor changes, after computing 'swap' and 'undo', they are both ran through the constructor where error checking is done. This is inefficient because the inputs are confirmed to be valid. This is done since the key has changed so my code decrypts the ciphertext with the new key automatically (new feature). However, this doesn't bother me too much since the time to compute this algorithm is still fast at 0.005s. There is room for improvement in terms of efficiency when it comes to methods 'AddDictionary' and 'RemoveDictionary'. Both these methods have a computational complexity of $O(k^2)$, which is due to the error checking and appending of cells. A better method, with better computational complexity would have been to use the class 'List' (discussed above). 'DictionaryAttack' efficiency could have been improved by evaluating the text in chunks, but other than that the output of the function/functionality is very useful to the user. I am overall very happy with the usability and functionality of the additional methods.

Function Name	Calls	Total Time (s)	Self Time* (s)
tests	1	0.084	0.011
PermutationKey>PermutationKey.encryption	15	0.028	0.028
PermutationKey>PermutationKey.decryption	6	0.024	0.001
Attack>Attack.DictionaryAttack	1	0.020	0.003
Attack>Attack.attack	1	0.011	0.001
Attack>Attack.Attack	6	0.010	0.008
Attack>Attack.lettercount	2	0.007	0.007
permutation	1	0.006	0.000
permutation>msort	51	0.006	0.002
Attack>Attack.swap	1	0.005	0.002
permutation>merge	25	0.004	0.004
PermutationKey>PermutationKey.swap	3	0.003	0.003
Attack>Attack.AddDictionary	1	0.003	0.002
...tationKey>PermutationKey.PermutationKey	36	0.003	0.003
Attack>Attack.undo	1	0.003	0.001
Attack>Attack.RemoveDictionary	1	0.002	0.002
PermutationKey>PermutationKey.invertion	12	0.002	0.001
Attack>Attack.sample	1	0.002	0.001
PermutationKey>PermutationKey.mtimes	5	0.001	0.001
List>List.List	7	0.001	0.001
List>List.isNil	1	0.000	0.000
ciphertext	1	0.000	0.000

Figure 1: Run times for 'Attack' and 'PermutationKey'. Note: Table operations hidden.