# Gold Problem 2: Sorting out Caesar!

**Copied from:**
**https://www.cs.hmc.edu/twiki/bin/view/CS5/CaesarCipherGold**
**on 3/22/2017**

## [60 points; individual or pair] filename: `hw3pr2.py`

This problem asks you to write several functions using *functional programming*, i.e., conditionals, recursion, and/or list comprehensions.

For each one, be sure to

- name the function as specified ***including capitalization*** - this helps us test them smoothly
- include a docstring that briefly explains the function's inputs and what it does

---

## Function to write #1: `encipher( S, n )`

Write the function `encipher( S , n )` that takes as input a string `S` and a non-negative integer `n` between 0 and 25. Then, `encipher` should return a new string in which the letters in `S` have been "rotated" by `n` characters forward in the alphabet, wrapping around as needed.

For this problem, you should assume that upper-case letters are "rotated" to upper-case letters, lower-case letters are "rotated" to lower-case letters, and all non-alphabetic characters are left *unchanged*. For example, if we were to shift the letter `'y'` by 3, we would get `'b'` and if we were to shift the letter `'Y'` by 3 we would get `'B'`. (In python, you can use the test `if 'a' <= c <= 'z':` to determine if a character `c` is between `'a'` and `'z'` in the alphabet.)

You can write `encipher` any way you like as long as you use functional programming -- that is, feel free to use conditionals, recursion, and/or list comprehensions.

You might use the class suggestion of writing a helper function that "rotates" a single character by `n` spots, wrapping around the alphabet as appropriate. In lecture we looked at how that might work. Then, you could use this helper function to encipher your string. It's up to you how you do this!

That said, for rotating, keep in mind that the built-in functions `ord` and `chr` convert from single-character strings to integers and back:

- For example, `ord('a')` outputs `97`
- and `chr(97)` outputs `'a'`.

Remember that

- uppercase letters wrap around the alphabet to uppercase letters
- lowercase letters wrap always to lowercase letters
- non-letters do not wrap at all!

## Hints, part 1... **Write `rot(c,n)`!**

---

- Write a function `rot(c,n)` that rotates `c`, a single character, forward by `n` spots in the alphabet.
- We wrote `rot13(c)` in class -- it's very close to `rot(c,n)`!
- Remember that you'll need to wrap the alphabet (as `rot13` did) and ***leave non-alphabetic characters unchanged***
- Test out your `rot(c,n)` function to make sure it works:

```
•        rot('a',2)   -->  'c'
•        rot('y',2)   -->  'a'
•        rot('A',3)   -->  'D'
•        rot('Y',3)   -->  'B'
         rot(' ',4)   -->  ' '
```

## Hints, part 2 **If you have `rot(c,n)`, you're nearly there!**

---

- With `rot(c,n)`, this problem is identical to the dna_to_rna (`transcribe`) problem!

- That is, you can handle one letter at a time (using `rot(c,n)`) in just the same way... .
- Alternatively, you can use a list comprehension to apply `rot(c,n)` many times.
- If you do use a list comprehension, then use `list_to_str` (below) to get back to a string!

**`['H','e','l','p','!']`**

If you have a list of characters and want a string, feel free to use this function (copy it to your file) to convert from list to string:

```
def list_to_str( L ):
    """ L must be a list of characters; then,
        this returns a single string from them
    """
    if len(L) == 0: return ''
    return L[0] + list_to_str( L[1:] )
```

Here's how to test `list_to_str`: `list_to_str( ['c','s','5','!'] )` should return `'cs5!'`

Some encipher examples:

```
In [1]: encipher('xyza', 1)
Out[1]: 'yzab'

In [2]: encipher('Z A', 1)
Out[2]: 'A B'

In [3]: encipher('*ab?', 1)
Out[3]: '*bc?'

In [4]: encipher('This is a string!', 1)
Out[4]: 'Uijt jt b tusjoh!'

In [5]: encipher('Caesar cipher? I prefer Caesar salad.', 25)
Out[5]: 'Bzdrzq bhogdq? H oqdedq Bzdrzq rzkzc.'
```

## Function to write #2: `decipher( S )`

On the other hand, `decipher( S )` will be given a string of English text already shifted by some amount. Then, `decipher` should return, to the best of its ability, the *original* English string, which will be some rotation (possibly `0`) of the input `S`.

**Note**: some strings have more than one English "deciphering." What's more, it is difficult or impossible to handle very short strings correctly. Thus, your `decipher` function *does not have to be perfect*. However, it should work almost all of the time on long stretches of English text, e.g., sentences of 8+ words. On a single word or short phrase, you will not lose any credit for not getting the correct deciphering!

Hints:

- A good place to start is to create a line ***with every possible ENCODING***, something like this:
- `    L = [     __________          for n in range(26) ]`
- Then, you will want to use the `LoL` "list of lists" technique in which each element of `L` gets a score. You might want to look back at how that worked...
- `    LoL = [     __________          for x in L ]`
- It's entirely up to you how you might want to score "Englishness." See below for some starting points... .
- To be specific, take a look at the `bestWord` example that found the word with the greatest scrabble-score in a list of words. That's not so far from what you want here!
- Then, go back and take a look at the min/max lecture to see how to handle the `LoL` "list of lists"

One approach you could try is to use letter frequencies -- a function providing those frequencies is provided below -- feel free to cut-and-paste it into your HW file. Scrabble scores have also been suggested in the past...! You're welcome to use some additional "heuristics" (rules of thumb) of your own design. Also, you are welcome to write one or more small "helper" functions that will assist in writing `decipher`.

However you approach it, **be sure** to describe whatever strategies you used in writing your `decipher` function in a short comment above your `decipher` function.

Some decipher examples:

```
In [1]: decipher('Bzdrzq bhogdq? H oqdedq Bzdrzq rzkzc.')
Out[1]: 'Caesar cipher? I prefer Caesar salad.'

In [2]: decipher('Hu lkbjhapvu pz doha ylthpuz hmaly dl mvynla '\
               'lclyfaopun dl ohcl slhyulk.')
Out[2]: 'An education is what remains after we forget everything we have
learned.'

In [3]: decipher('Onyx balks')
Out[3]: 'Edon rqbai'  # mine is wrong! This is OK here...
```

Note that the last example shows that our decipherer gets some short phrases wrong -- ***this is completely OK!***. Your decipherer should get more and more phrases correct, the longer they get, but it does not have to get single words or short phrases -- after all, for short strings, there are likely to be rotations that have more "English-y" letters than the original!

Here is a letter-probability function and its source:

```
# table of probabilities for each letter...
def letProb( c ):
    """ if c is the space character or an alphabetic character,
        we return its monogram probability (for english),
        otherwise we return 1.0 We ignore capitalization.
        Adapted from

http://www.cs.chalmers.se/Cs/Grundutb/Kurser/krypto/en_stat.html
    """
    if c == ' ': return 0.1904
    if c == 'e' or c == 'E': return 0.1017
    if c == 't' or c == 'T': return 0.0737
    if c == 'a' or c == 'A': return 0.0661
    if c == 'o' or c == 'O': return 0.0610
    if c == 'i' or c == 'I': return 0.0562
    if c == 'n' or c == 'N': return 0.0557
    if c == 'h' or c == 'H': return 0.0542
    if c == 's' or c == 'S': return 0.0508
    if c == 'r' or c == 'R': return 0.0458
    if c == 'd' or c == 'D': return 0.0369
    if c == 'l' or c == 'L': return 0.0325
    if c == 'u' or c == 'U': return 0.0228
    if c == 'm' or c == 'M': return 0.0205
    if c == 'c' or c == 'C': return 0.0192
    if c == 'w' or c == 'W': return 0.0190
    if c == 'f' or c == 'F': return 0.0175
    if c == 'y' or c == 'Y': return 0.0165
    if c == 'g' or c == 'G': return 0.0161
    if c == 'p' or c == 'P': return 0.0131
    if c == 'b' or c == 'B': return 0.0115
    if c == 'v' or c == 'V': return 0.0088
    if c == 'k' or c == 'K': return 0.0066
    if c == 'x' or c == 'X': return 0.0014
    if c == 'j' or c == 'J': return 0.0008
    if c == 'q' or c == 'Q': return 0.0008
    if c == 'z' or c == 'Z': return 0.0005
    return 1.0
```

## Function to write #3: `blsort( L )`: Binary-list sorting...

Design and write a function named `blsort( L )`, which will take in a list `L` and should output a list with the same elements as `L`, but in ascending order. However, `blsort` **ONLY NEEDS TO HANDLE LISTS OF BINARY DIGITS**, that is, this function can and should assume that `L` will always be a list containing only `0`s and `1`s.

You may not call Python's `sort` to solve this problem! Also, you should not use your own sort (asked in a question below), but you may use any other technique to implement`blsort`. In particular, you might want to think about how to take advantage of the constraint that the input will be a binary list -- this is a considerable restriction!

One function that some have found helpful is `count(e,L)`, one of the helper functions we used in an earlier class. Grab it from there (or try rewriting it, perhaps... here's the crucial piece: `LC = [ 1 for x in L if x==e ]` !)

You would need to include `count(e,L)` in your file, and then you could use it to return the number of times that `e` appears in `L`... .

Here are some examples:

```
In [1]: blsort( [1, 0, 1] )
Out[1]: [0, 1, 1]
```

```
In [2]: L = [1, 0, 1, 0, 1, 0, 1]

In [3]: blsort(L)
Out[3]: [0, 0, 0, 1, 1, 1, 1]
```

**Hint**: in the end, this problem is much *easier* than ordinary sorting!

## Function to write #4: `gensort( L )`: General-purpose sorting

Use recursion to write a general-purpose sorting function `gensort( L )` which takes in a list `L` and should output a list with the same elements as `L`, but in ascending order. Feel free to use the `max` function built-in to Python (or `min` if you prefer) and the `remOne` function we discussed in class. Recursion -- that is, sorting the *rest* of the list -- will help, too.

Here are some examples:

```
In [1]: gensort( [42, 1, 3.14] )
Out[1]: [1, 3.14, 42]

In [2]: L = [7, 9, 4, 3, 0, 5, 2, 6, 1, 8]

In [3]: gensort(L)
Out[3]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For this problem, you should ***not*** use any of Python's built-in implementations of sorting for this problem -- for example, `sorted(L) or L.sort()`. Rather, you're designing and implementing your own approach from scratch!

Note that `gensort( L )` should work for *lists* `L`. It does **not** have to work for string inputs.

## Function to write #5: `jscore( S, T )`: Jotto scoring

Write a function named `jscore( S, T )`, which will take in two strings, `S` and `T`. Then, `jscore` outputs the "jotto score" of `S` compared with `T`.

This jotto score is the number of characters in `S` that are shared by `T`. Repeated letters are counted multiple times, as long as they appear multiple times in both strings. The examples below will make this clear. Note that, in contrast to the traditional game of 5-letter jotto, we are not constraining the lengths of the input strings here!

There are several ways to accomplish this, many of which use small helper-functions - feel free to add any such helper functions you might like. (We may have written what you need in class... .)

Note that if either `S` or `T` is the empty string, the jotto score should be zero!

**Hint** This line turns out to be a useful test: `if S[0] in T:`

Some examples:

```
     In [1]: jscore( 'diner', 'syrup' )  # just the 'r'
     Out[1]: 1

     In [2]: jscore( 'geese', 'elate' )  # two 'e's are shared
     Out[2]: 2

     In [3]: jscore( 'gattaca', 'aggtccaggcgc' ) # 2 'a's, 1
't', 1 'c', 1 'g'
     Out[3]: 5

     In [4]: jscore( 'gattaca', '' ) # if empty, return 0
     Out[4]: 0
```

## Function to write #6: `exact_change( target_amount, L )`

**Making change!** Use recursion to write a Python function `exact_change` with the following signature:

```
def exact_change( target_amount, L ):
```

where the input `target_amount` is a single non-negative integer value and the input `L` is a list of positive integer values. Then, `exact_change` should return either `True` or `False`: it should return `True` if it's possible to create `target_amount` by adding up some-or-all of the values in `L`. It should return `False` if it's ***not*** possible to create `target_amount` by adding up some-or-all of the values in `L`.

For example, `L` could represent the coins you have in your pocket and `target_amount` could represent the price of an item -- in this case, `exact_change` would tell you whether or not you can pay for the item *exactly*.

Here are a few examples of `exact_change` in action. Notice that you can *always* make change for the target value of `0`, and you can *never* make change for a negative target value: these are two, but not all, of the base cases!

```
In [1]: exact_change( 42, [25, 1, 25, 10, 5, 1] )
Out[1]: True

In [2]: exact_change( 42, [25, 1, 25, 10, 5] )
Out[2]: False

In [3]: exact_change( 42, [23, 1, 23, 100] )
Out[3]: False

In [4]: exact_change( 42, [23, 17, 2, 100] )
Out[4]: True

In [5]: exact_change( 42, [25, 16, 2, 15] )
Out[5]: True  # needs to be able to "skip" the 16...

In [6]: exact_change( 0, [4, 5, 6] )
Out[6]: True

In [7]: exact_change( -47, [4, 5, 6] )
Out[7]: False

In [8]: exact_change( 0, [] )
Out[8]: True

In [9]: exact_change( 42, [] )
Out[9]: False
```

**Hint**: Similar to `LCS`, below, this problem can be handled by recursing *twice* and giving a name to each of the two results.

- For the first, try solving the problem *without* the first coin. (This is the *loseit* case!)

    - You might even use the variable name `loseit`, as in `loseit = exact_change( ... )`
- For the second, try solving it *with* the first coin. (This is the *useit* case!)
    - You might continue by using the variable name `useit`, as in `useit = exact_change( ... )`
- Then, have your code figure out what the appropriate boolean value to return, depending on the results it gets!
    - **Hint on this last part of the hint**: This problem puts the `or` into *useit or loseit* - literally!

## Function to write #7: `LCS( S, T )`: DNA matching

This week's final algorithmic challenge is to write a function named `LCS( S, T )`, which will take in two strings, `S` and `T`. Then, `LCS` should output the longest common subsequence (LCS) that `S` and `T` share. The LCS will be a string whose letters are a subsequence of `S` and a subsequence of `T` (they must appear in the same order, though not necessarily consecutively, in those input strings).

Note that if either `S` or `T` are the empty string, then the result should be the empty string!

Some examples:

```
      In [1]: LCS( 'human', 'chimp' )
      Out[1]: 'hm'

      In [2]: LCS( 'gattaca', 'tacgaacta' )
      Out[2]: 'gaaca'

      In [3]: LCS( 'wow', 'whew' )
      Out[3]: 'ww'

      In [4]: LCS( '', 'whew' )      # first input is the empty
string
      Out[4]: ''

      In [5]: LCS( 'abcdefgh', 'efghabcd' )
      Out[5]: 'abcd'
```

Note that if there are ties, any one of the ties is OK: in the last example above, `'efgh'` would be an equally acceptable result.

**Hint**: Consider the following strategy:

- if the first two characters match, use them!
- If the first two characters don't match, recurse *twice*: you could call this *use it or lose it or lose it*!
- For the first "lose it," recurse to toss out one input's initial letter:
- `            result1 = LCS(S[1:],T)`
- For the second "lose it," recurse to toss out the other input's initial letter:
- `            result2 = LCS( _ , ____ )`
- here, a couple details need to be filled in... .
- Finally, return the *better* of those two results -- you'll have to remind yourself what "better" means for this problem!
- good luck!

## Extra!

Are you saying to yourself, *Never enough algorithms!* ?

Here is an optional extra-credit algorithm-design challenge that builds from `exact_change`. It's more difficult because

- it returns the actual coins for making change, and
- it can also return `False`, so there are several cases to handle *after* the recursion...

### Extra-credit option #1: `make_change( target_amount, L )`   (up to +6ec pts)

For up to +6 e.c. points, write a second change-handling function named `make_change( target_amount, L )`.

This function should actually determine which values (from `L`) could be returned to total the `target_amount`.

That is, instead of simply returning `True` or `False`, your `make_change` function should return ***a list*** of coins taken from `L` that sum up to `target_amount`. If there is no such list, then `make_change` should simply return `False`. If there are more than one possible lists of values from `L`, then your function may return any one of the valid answers.

The *order* of the values returned does not matter, though it's natural to have them in the same order as they appear in the original list (our tests will do this...).

You do not have to, but you are welcome to use `exact_change` as a subroutine here!

The examples below show how `make_change` should work; these are the same inputs as in the `exact_change` function above.

In addition, `sorted` has been called, at least on the non-empty feasible cases, so that the results have a well-defined order:

```
In [1]: sorted( make_change( 42, [25, 1, 25, 10, 5, 1] ) )
Out[1]: [1, 1, 5, 10, 25]

In [2]: make_change( 42, [25, 1, 25, 10, 5] )
Out[2]: False

In [3]: make_change( 42, [23, 1, 23, 100] )
Out[3]: False

In [4]: sorted( make_change( 42, [23, 17, 2, 100] ) )
Out[4]: [2, 17, 23]

In [5]: sorted( make_change( 42, [25, 16, 2, 15] ) )
Out[5]: [2, 15, 25]

In [6]: make_change( 0, [4, 5, 6] )
Out[6]: []

In [7]: make_change( -47, [4, 5, 6] )
Out[7]: False

In [8]: make_change( 0, [] )
Out[8]: []
```

```
In [9]: make_change( 42, [] )
Out[9]: False
```

## Submission

Be sure to submit your `hw3pr2.py` file in the usual way at the submission site!