This problem set is due **Wednesday, December 13 at 2:00pm, KST**. Note that that the due date that you see on Blackboard is not accurate since it shows the time in EST. You should go by the due date in this handout.

Add your name and email address as a comment at the top of each file you submit.

**What to submit**

Hand in your work on Blackboard. Your submission should include the following. Please do not submit other files that I did not ask for.

```
ps8.py
```

Multiple submissions are allowed before the due date and time. Late submissions will **not** be graded.

**Assignment objectives**

In this problem set you will practice with more dictionaries and will be exposed to some of binary numbers, regular expressions, and cryptography.

**Incomplete work**

Please read what I said about this in PS 2.

**Naming conventions in Python and programming style in general**

Please read what I said about this in PS 2.

**Part 1. Autocorrection (10 points)**

In `ps8.py` complete the function `autocorrect` that takes two arguments: a sentence of only lowercase letters called `message` and a dictionary called `mappings` where the key is the correct spelling of a word and the value is a list of common misspellings of the word. Your function should replace all misspellings with the correct word.

**For example:** if `message` is `'hye thsi is mi'` your return value should be: `'hey this is me'`.

**Examples:**

```python
mappings = {
    'the': ['hte','teh'],
    ```
'this': ['thsi', 'tis', 'htis', 'tshi'],
'hey': ['hye', 'ehy', 'yhe'],
'you': ['yuo', 'ouy', 'uyo', 'u'],
'how': ['haw', 'hwo'],
'are': ['r', 'aer'],
'is': ['si'],
'test': ['tset', 'tets', 'etts'],
'am': ['ma', 'm'],
'best': ['bset', 'bets', 'btes'],
'me': ['em', 'mi'],
'hello': ['hallo', 'heello', 'helio', 'hell']
}

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>autocorrect('hye thsi si mi', mappings)</td>
<td>'hey this is me'</td>
</tr>
<tr>
<td>autocorrect('you are the best', mappings)</td>
<td>'you are the best'</td>
</tr>
<tr>
<td>autocorrect('', mappings)</td>
<td>''</td>
</tr>
</tbody>
</table>

**Part 2. Extracting Bits (10 points)**

In `ps8.py` complete the function `extract_bits_5_13` that takes in one parameter: an integer called `num`. Your function should extract bits 5 through 13 inclusive and return them as an integer. Remember that the rightmost bit is the 0th bit.

**Note:** Integers are 32-bit values, meaning that your code must be able to accommodate numbers with 32 bits.

As an example, suppose `num` is 252,645,135.

Written in binary, this number is 00001111000011110000111100001111.

Let's highlight bits 5 through 13: 00001111000011110000111100001111. The number 001111000 in decimal is 120, which is what your function would return for the argument 252,645,135.

You will find a [Base Conversion Website](#) quite helpful while completing this assignment.

**Additional Examples:**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract_bits_5_13(20000)</td>
<td>113</td>
</tr>
<tr>
<td>extract_bits_5_13(0)</td>
<td>0</td>
</tr>
<tr>
<td>extract_bits_5_13(16383)</td>
<td>511</td>
</tr>
</tbody>
</table>

**Part 3. Setting Bits (10 points)**

In `ps8.py` complete the function `set_bits_3_7` that takes in two parameters: an integer called `num` and an integer called `bit`. Your function should set all the bits from 3 to 7 inclusive to `bit` and return the result. Remember that the rightmost bit is the 0th bit.

**Note:** Integers are 32-bit values, meaning that your code must be able to accommodate numbers with 32 bits.

**Example 1:**

Suppose `bit` is 0 and the binary value of `num` is 0000111100001111000011110000111100001111. After masking, `num` becomes 0000111100001111000011110000111100000111, which is the return value.

**Example 2:**

Suppose `bit` is 1 and the binary value of `num` is 0000111100001111000011110000111100001111. After masking, `num` becomes 000011111000011111111111111111, which is the return value.
### Part 4. 16-bit Sums (10 points)

In `ps8.py` complete the function `add_halves` that takes in one parameter: an integer called `num`. Your function should take the rightmost 16 bits and add it to the leftmost 16 bits, returning the sum.

**Note:** Integers are 32-bit values, meaning that your code must be able to accommodate numbers with 32 bits.

**Example:**

Suppose the binary representation of `num` is 00110101001101010100101101010101.

- The rightmost 16 bits are 0100101101010101, which is 19,285 in decimal.
- The leftmost 16 bits are 0011010100110101, which is 13,261 in decimal.

The sum of these two values is 32,906, which is the return value.

**Additional Examples:**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add_halves(20000)</code></td>
<td>20000</td>
</tr>
<tr>
<td><code>add_halves(0)</code></td>
<td>0</td>
</tr>
<tr>
<td><code>add_halves(123456)</code></td>
<td>57921</td>
</tr>
</tbody>
</table>

### Part 5. Mathematical Expressions (20 points)

**Preliminaries:** For this problem you will be working with regular expressions in Python. Various functions for working with regular expressions are available in the `re` module. Fortunately, Python makes it pretty easy to see if a string matches a particular pattern.

At the top of the file we must import the `re` module:

```
import re
```

Then, we can use the `search()` function to test whether a string matches a pattern. In the example below, the regular expression has been saved in a string called `pattern` for convenience:

```python
tel_str = '123-456-7890'
pattern = r'^\d{3}-\d{3}-\d{4}$'
if re.search(pattern, tel_str):
    print('The string matches the pattern.')
else:
    print('The string does not match the pattern.')
```

The `r` that precedes the pattern string is not a typo. Rather, the `r` indicates that the string is a "raw" string. In a raw string, as opposed to a "normal" string, any backslash character is interpreted as simply a backslash, as opposed to defining an escape sequence like \n or \t. Make sure you use raw strings in your Python code when defining regular expressions.

The `^` and `$` at the beginning and end of the regular expression indicate that the entire string must match the regular expression, and not just part of the string. Make sure you include these symbols in your regular expressions too!

**Now, the problem to solve:** In `ps8.py` complete the function `math_expr` that takes one parameter, which is a list of strings, each being a potential mathematical expression. Your function should examine each string in the list and in the end return a list of indices of the strings which meet the following description/requirement:
A valid string starts with (1) a positive (+) or negative (−) symbol; directly followed by (2) an integer; followed by (3) one or more spaces; followed by (4) one of the following operators: +, −, *, /, or ^; followed by (5) one or more spaces; followed by (6) a positive (+) or negative (−) symbol; directly followed by (7) an integer.

An example string that meets this requirement is: ‘+6 * −2’

Note: The integer in the string will contain at least one digit. In cases where no string in the list meets the conditions above, or that the passed in argument is an empty list, your function should return an empty list. The order of the indices in your returned list does not matter, but it must not contain duplicate indices.

Examples:

expressions1 = [‘+6 * -2’, ‘+1 / -12’, ‘-11 - -11’]
expressions2 = [‘32 ^ 2’, ‘+2 ^ +4’, ‘+123 + -123’]
expressions3 = [‘one + three’, ‘’, ‘+s * -3’]

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>math Expr(expressions1)</td>
<td>[0, 1, 2]</td>
</tr>
<tr>
<td>math Expr(expressions2)</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>math Expr(expressions3)</td>
<td>[]</td>
</tr>
</tbody>
</table>

Part 6. The Snake Cipher (20 points)

In ps8.py complete the function `snake_encrypt` that takes two parameters, in this order:

- **plaintext**: a plaintext string containing letters, digit and spaces that will be encrypted
- **num_cols**: the number of columns the will be used in the encryption process

Your function should take the plaintext, encrypt it using the transposition cipher described below, and return a single encrypted string as the result.

Here's how the snake cipher works. First, take a plaintext string and start drawing characters from it, placing the first `num_cols` of characters into the first row of a grid from left to right. We assume here that the grid has `num_cols` columns in it. Then, take the next `num_cols` characters, placing them into the second row of the grid in reverse order, from right to left. Continue in this fashion, drawing characters from the plaintext string until you run out of characters, alternating between placing them from left-to-right and right-to-left. With some arithmetic you should be able to figure out exactly how many rows your grid will need to store all the characters from plaintext.

As an example, suppose we want to encrypt the string 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' with num_cols = 5. The characters would be placed into a grid as follows (the arrows indicate the direction that the letters are written to the grid):

```
ABCDE -->
JIHGF <--
KLMNO -->
TSRQP <--
UVW -->
```

The characters are thereby written in a S-shaped, "snake-like" manner into the grid. If all characters from the plaintext are placed into the grid, but the last row is incomplete, we fill the last row with @ symbols:

```
ABCDE
JIHGF
KLMNO
TSRQP
UVW@@
```
Next, take each column of characters, from left to right, and concatenate them to generate the final, encrypted string. For the example above, the final encrypted string would be: AJKTU + BILSV + CHMRW + DGNQ@ + EFOP@ = AJKTUBILSVCHMRWGDGNQ@EFOP@.

Here's another example:

plaintext = 'STONYBROOKUNIV'
num_cols = 4

Insert the characters into the grid:

STON
ORBY
OKUN
   VI

Insert @ symbols into the last row, as necessary:

STON
ORBY
OKUN
@@VI

Concatenate the columns of characters together to generate the final, encrypted string: SOO@TRK@OBUVNYNI.

**Note:** num_cols must be greater than zero. In the case that num_cols ≤ 0, your function should return plaintext. In the case that plaintext is an empty string, your function should return a string of num_cols @ symbols.

**Examples:**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>snake_encrypt('AMERICA', 7)</td>
<td>'AMERICA'</td>
</tr>
<tr>
<td>snake_encrypt('STONYBROOKUNIV', 4)</td>
<td>'SOO@TRK@OBUVNYNI'</td>
</tr>
<tr>
<td>snake_encrypt('ILOVEDORITOS', -2)</td>
<td>'ILOVEDORITOS'</td>
</tr>
</tbody>
</table>

**Part 7. The Snake Cipher: Decryption (20 points)**

In ps8.py complete the function `snake_decrypt` that takes two parameters, in this order:

- encrypted: ciphertext that was generated from plaintext by using the snake cipher described in Part 6 above
- num_cols: the number of columns that was used to encrypt the original plaintext string

Your function should take the encrypted string, decrypt it by applying snake encryption backwards, then return the (decrypted) original plaintext.

**Note:** num_cols must be greater than zero. In the case that num_cols ≤ 0, your function should return encrypted. In the case that encrypted is an empty string, your function should return the value None.

**Examples:**

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>snake_decrypt('AMERICA', 7)</td>
<td>'AMERICA'</td>
</tr>
<tr>
<td>snake_decrypt('SOO@TRK@OBUVNYNI', 4)</td>
<td>'STONYBROOKUNIV'</td>
</tr>
<tr>
<td>snake_decrypt('ILOVEDORITOS', -2)</td>
<td>'ILOVEDORITOS'</td>
</tr>
</tbody>
</table>