Daily Coding Problem #250

Problem

This problem was asked by Google.

A cryptarithmetic puzzle is a mathematical game where the digits of some numbers are represented by letters. Each letter represents a unique digit.

For example, a puzzle of the form:

```
  SEND
+ MORE
--------
 MONEY
```

may have the solution:

```
{'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O', 0, 'R': 8, 'Y': 2}
```

Given a three-word puzzle like the one above, create an algorithm that finds a solution.

Solution

One way of solving this would be to check every numerical value between `0` and `9` for each

character, and return the first character-to-number mapping that works. Assuming it takes $O(N)$ to validate a mapping, where N is the number of digits in the sum, this would take $O(N * 10^C)$, where C is the number of distinct characters.

Instead, we can use backtracking to cut down on the number of possible mappings to check. For backtracking to be effective, we should be able to construct a partial solution, verify if that partial solution is invalid, and check if the solution is complete. Let's answer each of these in turn.

Can we construct a partial solution?

Yes, we can assign digits to a subset of our distinct characters. In the problem above, for example, a partial solution might just assign `S = 5`.

Can we verify if a partial solution is invalid?

Even though we may not know the value of each letter, there are cases where we can disqualify certain solutions. Once we have substitued all the known numbers for letters, we can start with the rightmost column and try to add digits. If we find that column addition results in an incorrect sum, we know the solution will not work. For example, if we had assigned `D = 3`, `E = 2`, and `Y = 8`, our partial solution above would look like the following:

```
  S2N3
+ MOR2
--------
 MON28
```

No matter what the other characters represent, this cannot work. If the ones column works, we can continue this process, moving leftward across our columns until we find an incorrect sum, or a character whose value is unknown, at which point we cannot check any further.

```
def is_valid(letters, words):
    a, b, c = words
    n = len(c)

    carry = 0
    for i in range(n - 1, -1, -1):
        if any(letters[word[i]] is None for word in words):
            return True
        elif letters[a[i]] + letters[b[i]] + carry == letters[c[i]]:
```

```
            carry = 0
        elif letters[a[i]] + letters[b[i]] + carry == 10 + letters[c[i]]:
            carry = 1
        else:
            return False

    return True
```

Can we check if a solution is complete?

Yes, if `is_valid` returns `True`, and we have assigned all letters to numbers, then we have a complete solution.

Therefore, we can implement the solver as a depth-first search, where at each level we take an unassigned letter, assign a digit to it, and check whether the resulting letter map is valid. If it is, we go one step deeper. Otherwise, we try a different digit. If we reach a point where all letters have been assigned and the solution is valid, we return this result.

```
def solve(letters, unassigned, nums, words):
    if not unassigned:
        if is_valid(letters, words):
            return letters
        else:
            return None

    char = unassigned[0]

    for num in nums:
        letters[char] = num
        nums.remove(num)

        if is_valid(letters, words):
            solution = solve(letters, unassigned[1:], nums, words)
            if solution:
                return solution

        nums.add(num)
        letters[char] = None

    return False
```

If we chose characters to assign at random, this still would be fairly inefficient. For example, it is not useful to assign `S = 5` first, because we do not have enough other letters in place to check whether this could be valid. To fix this, we can order our letters by when they appear when carrying out our validity check. We can find this order by going down each column, from right to left, and appending new characters to an ordered dictionary of letters.

```
def order_letters(words):
    n = len(words[2])

    letters = OrderedDict()
    for i in range(n - 1, -1, -1):
        for word in words:
            if word[i] not in letters:
                letters[word[i]] = None

    return letters
```

In order for this to work, we must guarantee that each word is the same length. To do so, we can prepend a character that represents `0`, such as #, to each of the first two words, until their length is the same as that of the sum.

```
def normalize(word, n):
    diff = n - len(word)
    return ['#'] * diff + word
```

Finally, the code that invokes these functions would be as follows:

```
def cryptanalyze(problem):
    words = list(map(list, problem))

    n = len(words[2])
    words[0] = normalize(words[0], n)
    words[1] = normalize(words[1], n)

    letters = order_letters(words)
    unassigned = [letter for letter in letters if letter != '#']
    nums = set(range(0, 10))

    return solve(letters, unassigned, nums, words)
```

To analyze the time complexity, we can look at each function. For `is_valid`, we check up to three words for each column, so this will be `O(N)`, where `N` is the number of letters in the sum. If we let `C` be the number of distinct characters, then we will call `solve` `O(C!)` times, since each call will reduce the number of characters to solve by one. Since each solve attempt requires a validity check, the overall complexity will be `O(N * C!)`.