6 Splitting a string without spaces into words

Certain languages, such as Thai or classical Latin, do not use spaces to separate words within sentences. This creates some challenges to text-processing software that has to handle words.

Given a sentence recorded as a string without spaces and a dictionary (a list of words), split the sentence into words. In case multiple results are possible, return any of them.

Example 1: Split sentence `"helloworld"`, given the dictionary `["hello", "goodbye", "world"]`. Result: `["hello", "world"]`.

Example 2: Split sentence `"catseatmice"`, given the dictionary `["cat", "cats", "eat", "mice", "seat"]`. Result: `["cats", "eat", "mice"]` or `["cat", "seat", "mice"]`.

Clarification questions

Q: What result should be returned for the empty sentence?
A: The empty list `[]`.

Q: Is it possible that the sentence cannot be split into words?
A: Yes. In such a situation, return `None`.

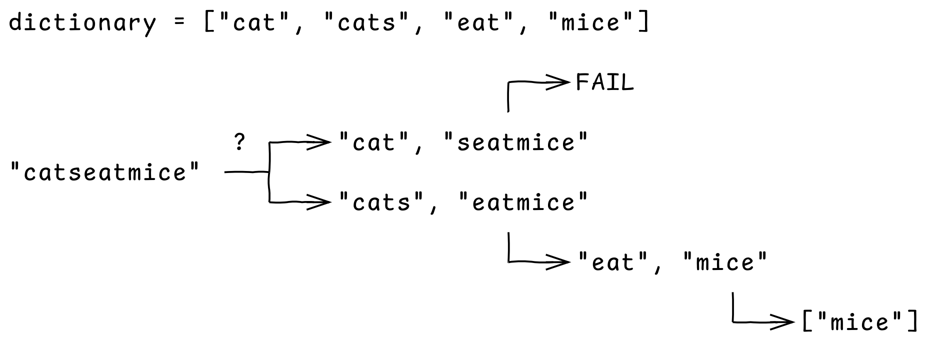
Solution 1: dynamic programming, top-down, $O(nw)$ time

This problem is similar to the change-making problem, except that we use strings instead of numbers, and concatenation instead of addition. Instead of paying a target amount, we have to split a sentence into words.

We can start from a side of the sentence, either the start or the end. Let's choose the start. We look at all the words in the dictionary, and check which ones are prefixes of the sentence.

For example, for the sentence `"catseatmice"`, we find two possible prefixes in the dictionary: `"cat"` and `"cats"`. Since we do not know which one may lead to a correct split, we test both of them. In each case, we remove the prefix, then check if the suffix may be split into valid words. Splitting the suffix is a subproblem of the original problem, so we can solve it with a recursive call of the same algorithm.

Let's sketch how the algorithm works on the example `"catseatmice"`, if the dictionary is `["cat", "cats", "eat", "mice"]`:



We first find that the word “cat” is a prefix of "catseatmice". We split the string into "cat" and the suffix "seatmice". Now we try to find a way to split the suffix. We do not find any words in the dictionary that are a prefix of "seatmice", so we abandon the search.

Next, we look at the other prefix of "catseatmice": "cats". We split the string into "cats" and the suffix "eatmice".

Now we try to find a way to split the string "eatmice". The only word that is a prefix of this string is "eat". We split the string into "eat" and the suffix "mice", then try to split the suffix.

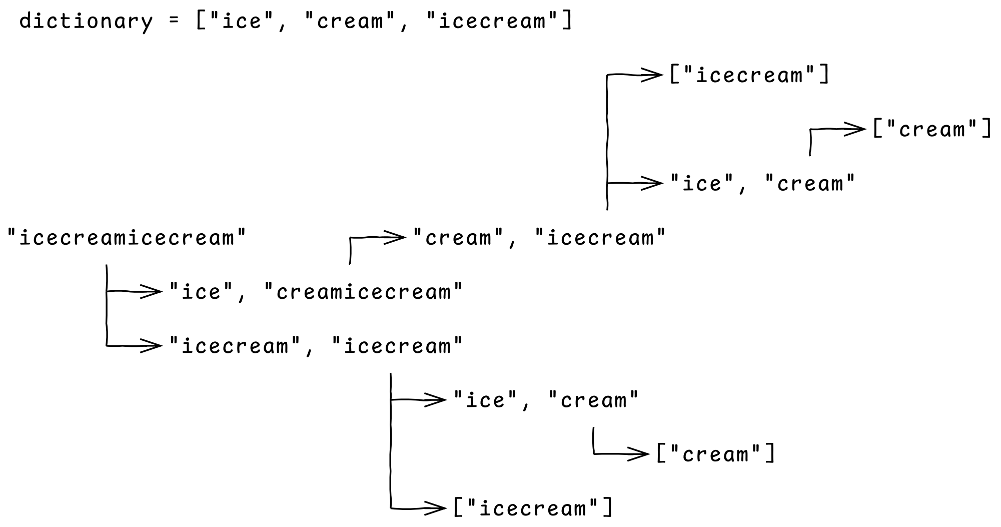
We find that the suffix "mice" is in the dictionary, so we complete the split as ["mice"].

Walking in reverse over the splits we made, we prepend the prefixes to obtain the solution ["cats", "eat", "mice"].

We can implement this algorithm as:

```python
def split_sentence(dictionary, sentence):
    if not sentence:
        return []
    for word in dictionary:
        if sentence.startswith(word):
            suffix = sentence[len(word):]
            split = split_sentence(dictionary, suffix)
            if split is not None:
                # Any solution works: return the first found.
                return [word] + split
    return None
```

This algorithm has exponential complexity, since it performs redundant computations. This is not obvious from the given example, but noticeable when splitting a sentence like: "icecreamicecreamicecream" when the dictionary is ["ice", "cream", "icecream"]:



We can see that the suffix `"icecream"` has been split multiple times, so redundant work has been done. This number keeps doubling for each `"icecream"` we add to the input.

We can avoid redundant work by caching the results of the splits:

```python
from functools import lru_cache

def split_sentence(dictionary, sentence):
    @lru_cache(maxsize=None)
    def helper(sentence):
        if not sentence:
            return []
        for word in dictionary:
            if sentence.startswith(word):
                suffix = sentence[len(word):]
                split = helper(suffix)
                if split is not None:
                    # Any solution works: return the first found.
                    return [word] + split
        return None
    return helper(sentence)
```

Just as in the change-making problem solution, we had to write a helper function to avoid caching the `dictionary` parameter, which is not supported by `lru_cache`, since it is mutable.

To analyze the complexity, we need to think about how deep the recursive calls may be, and how much work is done in each call. The depth of the call stack is limited by the length of the string n, since in the worst case we split into short words of 1 character each. In each call we must iterate over the dictionary, in w steps. Overall, the time complexity is $O(nw)$.

Solution 2: dynamic programming + BFS/DFS, bottom-up, $O(nw)$ time

Once we have implemented the top-down solution, we can rewrite it as bottom-up: we start from the empty string, and keep adding words as suffixes in all possible ways that match the sentence:

```python
def split_sentence(dictionary, sentence):
    # splits[prefix] = list of words that form the prefix
    splits = {"": []}
    # prefixes of the sentence that we have not processed yet
    prefixes_to_process = collections.deque([""])
    while prefixes_to_process:
        prefix = prefixes_to_process.popleft()
        if prefix == sentence:
            # Any split works. Return the first found.
            return splits[prefix]
        # Try to extend the prefix with a word matching the sentence.
        for word in dictionary:
            if sentence[len(prefix):].startswith(word):
                next_prefix = prefix + word
                # Only add the new prefix to be processed if it is not
                # already known, to avoid doing redundant work.
                if next_prefix not in splits:
                    splits[next_prefix] = splits[prefix] + [word]
                    prefixes_to_process.append(next_prefix)
    return None
```

This method is treating the subproblem space as a graph, which is explored in breadth-first order (BFS). We start from the subproblem of extending the prefix `""`. For each such prefix, we keep expanding using all the possible word suffixes that match the contents of the sentence. The BFS order guarantees that we choose the split with the fewest number of words, although this is not a requirement of the problem; we could have used as well any other search method.