Recall back in the first chapter, we said that a good way to check if a problem should be solved with DP or greedy is to first assume that it can be solved greedily, then try to think of a counterexample.

Let's say that we had s= "abcdef" and wordDict = [ "abcde", "ef", "abc", "a", "d"]. A greedy algorithm (picking the longest substring available) will not be able to determine that picking "abcde" here is the wrong decision. Likewise, a greedy algorithm (picking the shortest substring available) will not be able to determine that picking "a" first is the wrong decision.

With that being said, let's develop a DP algorithm using our framework:

For this problem, we'll look at bottom-up first.

1 . An **array** that answers the problem for a given state

Despite this problem being unlike the ones we have seen so far, we should still stick to the ideas of the framework. In the article where we learned about multi-dimensional dynamic programming, we talked about how an index variable, usually denoted i is typically used in DP problems where the input is an array or string. All the problems that we have looked at up to this point reflect this.

* With this in mind, let's use a state variable i, which keeps track of which index we are currently at in s.
* Do we need any other state variables? The other input is wordDict - however, it says in the problem that we can reuse words from wordDict as much as we want. Therefore, a state variable isn't necessary because wordDict and what we can do with it never changes. If the problem was changed so that we can only use a word once, or say k times, then we would need extra state variables to know what words we are allowed to use at each state.

In all the past problems, we had a function dp return the answer to the original problem for some state. We should try to do the same thing here. The problem is asking, is it possible to create s by combining words in wordDict. So, let's have an array dp where dp[i] represents if it is possible to build the string s up to index i from wordDict. To answer the original problem, we can return dp[s.length - 1] after populating dp.

2. A **recurrence relation** to transition between states

At each index i, what criteria determines if dp[i] is true? First, a word from wordDict needs to be able to **end** at index i. In terms of code, this means that there is some word from wordDict that matches the substring of s that starts at index i - word.length + 1 and ends at index i.

We can iterate through all states of i from 0 up to but not including s.length, and at each state, check all the words in wordDict for this criteria. For each word in wordDict, if s from index i - word.length + 1 to i is equal to word, that means word **ends** at i. However, this is not the sole criteria.

Remember, we are forming s by adding words together. That means, if a word meets the first criteria and we want to use it in a solution, we would add it on top of another string. We need to make sure that the string before it is also formable. If word meets the first criteria, it starts at index i - word.length + 1. The index before that is i - word.length, and the second criteria is that s up to this index is also formable from wordDict. This gives us our recurrence relation:

dp(i) = true if s.substring(i - word.length + 1, i + 1) == word and dp[i - word.length] == true for any word in wordDict, otherwise false

Chart

Description automatically generated

Text, application, email

Description automatically generated

In summary, the criteria is:

1. A word from wordDict can **end** at the current index i.
2. If that word is to end at index i, then it starts at index i - word.length + 1. The index before that i - word.length should also be formable from wordDict.

3. **Base cases**

The base case for this problem is another simple one. The first word used from wordDict starts at index 0, which means we would need to check dp[-1] for the second criteria, which is out of bounds. To fix this, we say that the second criteria can also be satisfied by i == word.length - 1.

Bottom-up Implementation

Text

Description automatically generated

Text

Description automatically generated

Top-down Implementation

In the top-down approach, we can check for the base case by returning true if i < 0. In Java, we will memoize by using a -1 to indicate that the state is unvisited, 0 to indicate false, and 1 to indicate true.

Text, timeline

Description automatically generated

Let's say that n = s.length, k = wordDict.length, and L is the average length of the words in wordDict. While the space complexity for this problem is the same as the number of states n, the time complexity is much worse. At each state i, we iterate through wordDict and splice s to a new string with average length L. This gives us a time complexity of *O*(*n*⋅*k*⋅*L*).

**300. Longest Increasing Subsequence**

Let dp[i] represent the length of the longest increasing subsequence that ends at index i.

Let's say you have an index j, where j < i. If nums[i] > nums[j], that means we can add onto whatever subsequence ends at index j using nums[i].

By default, every number on its own is an increasing subsequence of length 1.