10-dynamic-programming

Contents

    - Can construct
    - All words
    - Grid counter
- Iterative problems
    - Fibonacci
    - Can sum
    - Best sequence
    - Can construct
    - Count construct
    - Grid counter

Introduction

Dynamic programming problems usually are all about finding a non analytical solutions to problems. What does that mean ? Well with most other algorithms we find at most one single solution to a specific problem - sorting an array, finding an element in an array, swapping array elements, finding tree nodes by value, on and on. We can see how many of those problems often have one analytical solution and that is it. However dynamic programming problems are all about finding `all` possible solutions to a given problem. Or in other words, dynamic programming is involved when a solution does not have one analytical solution, or one solution only is not possible

Principles

The major principles here are as follows

Approach

- recursion - very often the most dynamic programming problems are solved by leveraging recursion, since it is an easy way to explore many permutations of a given set of inputs, it is very useful if we want to solve the problem in a top down approach, where the recursion calls will drill down from top to the bottom, base cases first, and then ascend the recursive call stack.
- iterative - very similar to the recursive approach, however it often uses a table to store results, and build future results off of this table, usually the table is initialized with one positive case, the base case, and everything else is initialized with the negative case, the size of the array is usually based on the argument which will be mutated, and that same argument is used to `key` into the table/array

Types

- bottom up - this is when we start the problem solving from the base case, for example to solve the Fibonacci sequence we will start from the two base cases, in this case the first two numbers of the Fibonacci sequence are 1 and 1, from this we built the solution from `the bottom to the top`
- top down - this is the inverse of the bottom up approach where we build the solution by starting with some final `end paramter`, and is mostly used with recursive approach, where the recursive calls drill down to the base cases first, then ascend the call stack. Using Fibonacci as an example, we start with the target number N from the sequence and drill down to N - 1 and N - 2 and so on, until we reach the base cases for `N = 1 and N = 2`

Optimizations

- memoization - very often when using the recursive approach, the same input might be visited in subsequent branches of the recursion, it is quite normal to store the result of this specific input and when / if it is visited again, to reuse the already computed value. When computing Fibonacci sequence, many of the numbers are going to be re-computed and overlapping in different branches of the recursive descent, we can remember them and re-use them
- minimization - reducing the number of input arguments to our function, can greatly help reduce the space complexity of the call stack frames themselves.

Steps

Recursive approach

- identify the input arguments

```
// the primary input arguments to the function
int target; int[] sequence
```

- identify which input arguments will mutate, and which will not, remain constant

```
// will be mutated on each call to the function, target is reduced on
   each call
// the main sequence of numbers remains `unchanged`, across all function
   calls
target
```

- use a simple example to find the base cases, the negative and positive ones

```
// negative base case, when target reaches negative value
if (target < 0) {
    return false;
}

// positive, affirmation base case when target is equal/reaches to 0
if (target == 0) {
    return true;
}
```

- write the main body of the function which will recurse and reduce the input

```
// the main body which decides to recurse or stop, based on the return
   value of the `sub` problem
for (int i : sequence) {
    // mutate the target, reduce the current target by the current number
        `i` from the sequence
    int newTarget = target - i;
    // call with mutated target sum, and see if way to generate newTarget
        from the sequence exists
    if (recurse(newTarget, sequence)) {
        // stop on the first positive result
        return true;
    }
}
```

- make sure to propagate the return value, correctly accordingly to the task

```
// stop on the first positive result
if (recurse(newTarget, sequence)) {
    return true;
}
```

- add default return value if needed at the end of the main function body

```
// the base cases
// main loop body
// if no positive return value is returned inside and after the loop, we
// return `false`, at the end of the function body
return false;
```

- memoize it, to optimize the solution, select a key, usually the one we mutate, locate return statements, but not the base cases, and store the results of the calls to the recursive calls

```
// first somewhere at the top of the recursive function check if we have
   the key in the memo table
if (MEMO.containsKey(key)) {
    return MEMO.get(key);
}

.................

// in the actual implementation, not the base cases, memo the results,
   including the negative/false ones too
if (condition) {
    MEMO.put(key, true);
    return true;
}

.................

// at the end of the recursive function after the main code body, as a
   fallback default value memoize the negative case too
MEMO.put(key, false);
```

- go through the problem and execution flow by drawing a stack call tree

Iterative approach

- identify the input arguments

```
// the primary input arguments to the function
int target; int[] sequence
```

- identify which are the main input arguments that are going to 'mutate'

```
// the main input argument which will be used to develop the main
   algorithm body
// based on this input argument, we will key into the tabulation table
target
```

- create the tabulation table, size is usually based on the mutating argument + 1

```
int[] table = new int[target + 1];
```

- initialize the table with correct positive and negative cases

```
// init the base positive case which we use to build off off
table[0] = 1;

// all other cases default to the negative value, to start with
Arrays.fill(table, -1);
```

- implementation of the main algorithm body accumulating data into the tabulation table

```
// loop through all states in the tabulation table
for (int i = 0; i < table.length; i++) {
    // when we meet a positive case, we can build off of it
    if (table[i] == true) {
        // we are in a positive case, meaning the sum to target[i] can be
            generated
        for (int num : sequence) {
            // sums `i + num` can be generated if the array has enough
               space to fit the number, remember that the tabulation
               table
            // is created with at most target + 1 elements, meaning that
               we have to check our bounds first
            if (i + num < table.length) {
                // i + num is in range and we mark this sum target as
                   possible
                table[i + num] = true;
            }
        }
    }
}
```

- go through the problem and execution flow by drawing out the states of the tabulation table

Recursive problems

Fibonacci sequence

One of the most prolific and common examples, to demonstrate what dynamic programming is about, is the Fibonacci sequence. We want to generate the first N numbers of the Fibonacci sequence, and we are given only the number N, the final position / index of the number we are looking for of the sequence

The most common solution is to use dynamic programming, the approach is based on two approaches

- recursively find the current Fibonacci number based on the previous ones.
- memorization of the already computed numbers, so we do not recompute them again.

```
static final class Fibonacci {

    final int[] MEMO = new int[100];
```

```
    Fibonacci() {
        // initialize the memoization array to be some invalid number that
           can not exist in a valid Fibonacci sequence, we are going to
        // use this as a flag to check if the number stored at this position
           or index is already computed
        Arrays.setAll(MEMO, operand -> -1);
    }

    int fibonacci(int n) {
        // the first two numbers of the Fibonacci sequence are both 1, so for
            N less than or equal to 2 we return 1
        if (n <= 2) {
            return 1;
        }

        // when the number is already computed, we can simply return whatever
            value is stored at this position in the memo array
        if (MEMO[n] != -1) {
            return MEMO[n];
        }

        // compute the current Fibonacci number n, by computing the previous
           two numbers of the sequence, remember the current number in
        // the memo array, for future reuse
        MEMO[n] = fibonacci(n - 1) + fibonacci(n - 2);

        // return the stored value at the position N, it contains the current
            value of the number at position N in the Fibonacci
        // sequence
        return MEMO[n];
    }
}
```

The complexity of this algorithm is defined for both time and space. For the non optimized version without memoization is as follows

- Without memoization table, each number of the sequence has to be evaluated every time

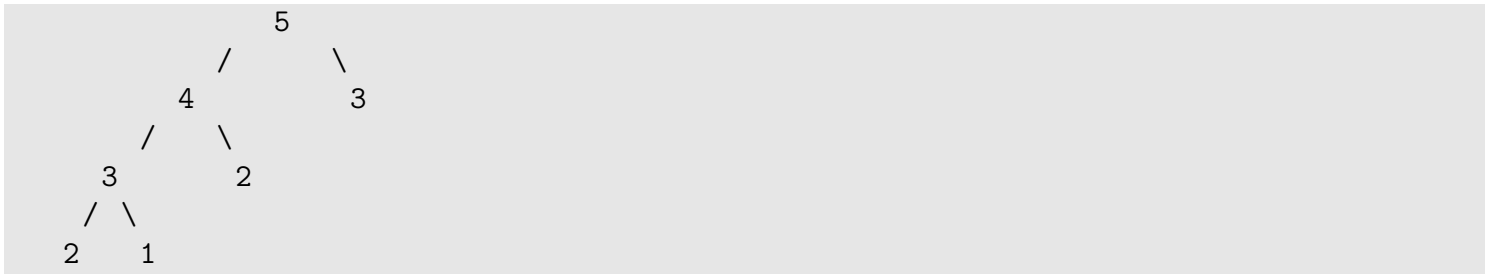
```
              5
            /     \
          4         3
        /   \     /   \
      3       2  2      1
    / \
   2   1
```

```
-   _time complexity_ - O(2^d) - this is defined as the branching factor to
    the power of the depth of the tree formed by the recursive calls,
       technically we
    have exactly 1.6^d recursive calls but for simplicity we can say - 2^d.
    Therefore for the example above, the number of calls for N = 5, assuming
       2^d,
```

```
    will be 2^5 = 32, however the exact calls are actually less, as mentioned
       , it is
    closer to 1.6, in this casethe calls are ~10.

-   _space complexity_ - O(d) - the space complexity is based on the number
    of call stack frames occupied during execution, remember that we do not
       evaluate
    all numbers of the sequence at once, the recursive call stack is at most
       N deep,
    where N is the target number position from the fib sequence. Meaning we
       make at
    most N number of recursive calls for each given number from the sequence,
        where
    N is the position or the index of the number from the Fibonacci sequence.
```

- With memoization table, which stores already computed numbers from the sequence, they are reused



- *time complexity* - O(d) - with memoization each number of the sequence is calculated exactly once, since we have a top down approach, the recursive calls will drill down to N = 1 and N = 2, and calculate upwards, to the target N, meaning that older, smaller numbers will be cached in the memo table, and this is how we get O(n) complexity in run time
- *space complexity* - O(d) - the space complexity is based on the number of call stack frames occupied during execution, this will not improve even with the memo table, since the recursive calls will still reach the very bottom, base case and then ascend, unwind the call stack, the depth of the stack will be N here too. We can be pedantic and say that the comp

We can clearly see from the call stack representation that a lot of the numbers from the sequence are getting re-computed constantly, we it is easy to see that they can be cached. For example given our implementation the first branch that will be computed immediately is `5 -> 4 -> 3 -> ( 2 + 1 )`. Meaning that the first number of the sequence that is going to be computed is `3` the next is going to be `4`, the next `5` Since the recursive approach is top down, we first drill down to the base case and then unwind the recursion, meaning we can easily cache each value we find to re-use later on when the recursion stack unwinds.

To see why the space complexity is O(d) - we can see that once we reach the bottom of the recursion in the following call path - `5 -> 4 -> 3 -> ( 2 + 1)`, stack has at this point consists of 5 calls to the fib function. Then the recursion stack unwinds, and it will never be deeper than N.

Can sum

Another very common problem is to check if a sequence or array of `positive` numbers sums up to a given target sum. Given a target array of numbers, allowing number reuse from the array, find if there is at least one combination of the given numbers that sums up to the given target sum, note that the same number from the array can be repeated multiple times. To solve this task we have to come up with the basic base cases, where

- a target sum of 0, would always produce a result of `true`, this is because we can always reach a target sum of 0 with any sequence of numbers, by not taking any numbers
- a negative target sum is invalid, meaning that it will always return `false`, when the target sum is negative, there is no way to `sum` a sequence of positive numbers to reach the target negative sum

Let us take a few examples first

```
target - 5
numbers - [2, 3, 1, 4, 5]
sequences - [2,3], [5, 1], [4, 3], [1, 1, 1, 1, 1] ... etc.
```

As we can see from this example above, there are many ways to sum these numbers to `5`, note that as mentioned above the numbers can be repeated, take a good note of the last example - `1,1,1,1,1` - this is a worse case scenario and will help us later to correctly compute the complexity of space and time for this algorithm

To evaluate the runtime complexity we have to consider two things

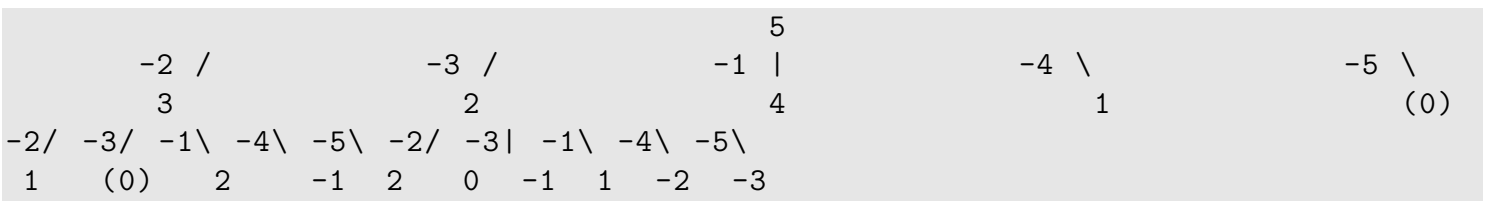
- the height of the recursive call stack - if we imagine that our sequence is composed of only `ones` that means that we will have exactly target number of function calls, this is because we can imagine taking `target - 1`, `target` number of times, if we subtract `1` from the target, `target` number of times we will get 0, meaning that at worst we will make `target` number of function calls
- the branching factor - this is how many recursive calls do we make inside each singular recursive call, or in other words, how many times we branch out on each recursive call, and in our case this is exactly expressed by the `sequence.length`, we loop, and for each item in the sequence we call the function again

To evaluate the space complexity we have to consider that

- without memoization, the complexity is exactly the depth of the tree, meaning that we will not make more than M number of calls, meaning that it is therefore `O(m)`
- with memoization, the complexity is also modified, we still make at most `O(m)` number of calls however, we have to add the extra space occupied by the items in the memo table, now worse case scenario if we have a sequence of all ones, meaning that we will also store at most `target` number of keys and values in the memo table, what does that mean for our complexity is that it is actually `O(m^2)`

To put everything together we now can conclude that the complexity is the branching factor N to the power of function calls M or in other words `O(n^m)`

To approach this solution let us draw a tree first, which will represent the call tree of our dynamic solution approach



Each node in the tree is the current target sum, each edge / connection to the child, is by how much we reduce that target when we go down the recursive calls, we can see that for the top-down approach we will drill down into the left most branching of the tree. When we reach a node that is negative we know that base case is `negative` we return `false`, when we reach a target node that is exactly 0 we know that base case is the `positive` we return `true`, anything greater than zero, we keep recursing. From this example we see that we first hit the sequence of `2 3`, that is correct, `2 + 3` does indeed sum up to `5`, another sequence is `5` itself,

another one is `3 2` or `1 4` or `4 1`, if we do not early return we will infect find all of them. But for this target task we are asked to find `if there is at least one sequence, if yes return true and bail`

Below is a simple explanation and implementation of the problem, we can see that again we employ a top-down approach. Where the recursive calls first drill down until the base cases and then up the stack we unwind. The main take away here is to see the base cases, we can have the target become negative, which is no good, since our sequence by design is of only positive numbers, meaning we have overshot with the subtraction, or we can have the target be exactly zero, which means we have found the sequence of numbers that sums up to the initial target with the `summable` was initially called. We return and unwind the stack, propagating the result as long as we have at least one positive hit meaning a base case which hit `true` we can return that one.

```
boolean summable(int target, int[] sequence) {
    // optimization case, first check if the target sum is already in the
       memo, this means that there was at least one call to
    // canSum, that reached this target sum, and computed the result, before
       we reach this again, we do not need to re-compute it
    // since the sequence is always the same, if we reach this target sum
       again, we can simply return the cached result value
    // instead.
    if (MEMO.containsKey(target)) {
        return MEMO.get(target);
    }
    // the first base case where the target is negative, meaning that there
       is no way to sum to the target, since the sequence is
    // only composed of positive numbers.
    if (target < 0) {
        return false;
    }
    // when the target is 0, that means that at this point the sum was
       reached, therefore we can early return true for this target
    // sum.
    if (target == 0) {
        return true;
    }

    // go through each number in the sequence, and find the new target sum
    for (int i : sequence) {
        // the new target sum is the original - the current number, meaning
           that with each call our target sum will reduce by each
        // number in the sequence, eventually it will reach 0, or be negative
           , hitting the 2 base cases, when that happens, we will
        // receive a return of false or true if the sum can be generated from
            the target sequence
        int newTarget = target - i;

        // the first call which returns true, can short circuit this loop and
            return early, we care only about if there is at least
        // one combination in the sequence which sums up to the target, we do
            not care about the actual sequence of numbers, the
        // first one that was found will suffice
        if (summable(newTarget, sequence)) {
            // before we early return, we also cache the current newTarget,
```

```
                into the memo object, meaning that we know that for this
            // target sum there is a sequence of numbers, that will sum/reach
                this newTarget sum
            MEMO.put(target, true);
            return true;
        }
    }

    // we have gone through all numbers in the sequence for the current
       target sum, and none of them returned true, this is the
    // final base case where we can say that there is no number in the
       sequence that sums up to the target number, on top of that
    // we also mark that target sum as 'unreachable' for this given input of
       sequence numbers
    MEMO.put(target, false);
    return false;
}
```

Best sequence

An extension of the can sum task, could be to find the best sequence which means, the sequence with the least amount of numbers that makes up the target sum. This task is not much different from the `cansum` problem, with a few minor changes. We still keep reducing the target sum, then drill down, top - down. The call stack will first reach bottom, then start unwinding, during the unwinding process each call to `best` will either return an `null or empty list` from the base cases, or the shortest list, produced in the body of the `for loop`. Note that we employ a clone, this is purely language specific, ideally we actually copy all elements of the current result to `best` and append the current number from the for loop iteration. The clone is done to avoid messing up with references (potentially).

- we adjust the base case returns - instead of `boolean` we return empty list or `null`. Empty list is returned when we reach a positive base case, meaning the sum has reached `zero`, a `null` is returned when the target sub overshot, and is negative.
- when we call the recursive function we have to now check if the return of the call to `best` returned a valid value, if it is not `null`, we clone the return value (the sequence thus far) append the current value `i` to the end of the sequence
- we have to find the min length, meaning we have to go through all numbers in the sequence, and find the one call stack that produced the smallest list of numbers which sums up to the target
- `bestSequence` is initially `null`, and can remain null in case there is actually no sequence that sums up to the target, however if we have found the at least one or the one with `min` length that is what the value of `bestSequence` will be after the for loop.

```
List<Integer> best(int target, int[] sequence) {
    // optimization case, first check if the target sum is already in the
       memo, this means that there was at least one call to
    // canSum, that reached this target sum, and computed the result, before
       we reach this again, we do not need to re-compute it
    // since the sequence is always the same, if we reach this target sum
       again, we can simply return the cached result value
    // instead.
    if (MEMO.containsKey(target)) {
```

```
        return MEMO.get(target);
    }
    // the first base case where the target is negative, meaning that there
       is no way to sum to the target, since the sequence is
    // only composed of positive numbers. Note that we here return a null,
       which means this is a negative base case, similar to
    // `false`,
    // however we can not return `false` here, so null will suffice.
    if (target < 0) {
        return null;
    }
    // when the target is 0, that means that at this point the sum was
       reached, therefore we can early return true for this target
    // sum. Note we return an empty array which can be later used to easily
       identify the positive base case, versus the negative
    if (target == 0) {
        return new ArrayList<>();
    }

    List<Integer> bestSequence = null;

    // go through each number in the sequence, and find the new target sum
    for (int i : sequence) {
        // the new target sum is the original minus the current number,
           meaning that with each call our target sum will reduce by
        // each
        // number in the sequence, eventually it will reach 0, or be negative
           , hitting the 2 base cases, when that happens, we will
        // receive a return of false or true if the sum can be generated from
            the target sequence
        int newTarget = target - i;

        // the first call which returns a non-null list, can short circuit
           this loop and return early, we care only about if there
        // is at least one combination in the sequence which sums up to the
           target, the first found current sequence returned below,
        // is for newTarget
        List<Integer> currentSequence = best(newTarget, sequence);
        if (!Objects.isNull(currentSequence)) {
            // the actual result here of currentSequence is for newTarget,
               however to find the final result sequence for the input
            // argument of `target`, we have to add the current number `i`
               too. We extend the currentSequence with the current
            // number `i`. What that means is newTarget can be reached with `
               currentSequence`, therefore target can be reached with
            // a result sequence of (...currentSequence, i). We clone the
               currentSequence to avoid reference assignment, this is
            // purely implementation detail, but good to know
            List<Integer> resultSequence = (List<Integer>) ((ArrayList<
               Integer>) currentSequence).clone();
            resultSequence.add(i);
```

```
            // make sure to remember the shortest sequence in the
               bestSequence variable, this way at the end of the function we
               will
            // have either the shortest sequence that can sum up to the
               target, or nil if no sequence exists that sums up to target
            if (Objects.isNull(bestSequence) || resultSequence.size() <
               bestSequence.size()) {
                bestSequence = resultSequence;
            }
            // memo the extended resultSequence, remember we memo here for `
               target` as key, since the resultSequence is for
            // newTarget + i or in other words = `target`
            MEMO.put(target, resultSequence);
        }
    }

    // we have gone through all numbers in the sequence for the current
       target sum, and none of them returned true, this is the
    // final base case where we can say that there is no number in the
       sequence that sums up to the target number, on top of that
    // we also mark that target sum as 'unreacheable' for this given input of
        sequence numbers by storing `null` for the sequence
    MEMO.put(target, bestSequence);
    return bestSequence;
}
```

Can construct

A variation of the `cansum` problem where we have to now find if a `target word` can be constructed by taking `words from a dictionary` passed in as input arguments. You must immediately notice that this is practically the same problem, we have the `target` and `sequence`. The dictionary of words is not mutable, we can take the same word from it as many times as we want, there is no limit, the target word is going to be the one to be mutated.

The base cases for this problem, are basically only one. If the target word is empty i.e. the emtpy string `""` we return `true`, any dictionary of any words, even the empty dictionary can construct an empty target word.

The way we solve this issue, is by taking away from the target word, until it becomes empty, if it does, that means we can construct from the given dictionary of words, otherwise if we end up with the target word being shorter than all words in the dictionary, we know that this case must return `false`.

```
target - abcdef
dictionary - ["ab", "abc", "cd", "ef", "def", "abcd"]
combinations - ["abc", "def"], ["ab", "cd", "ef"], ["abcd", "ef"].
```

From the example above we can see that at most we have two different combinations from the dictionary which can make up the word, they need not be in order of appearance in the dictionary array, however, when we take an item from the dictionary and compare against a sub-string from the target word, we must `always` start from the beginning of the string / target word, we can not simply take any `substring`, which actually makes the task somewhat easier too.

```
boolean constructible(String target, String[] dictionary) {
```

```
    // the main base case here, every empty word, can be "built" from an
       array of any words, by simply not picking anything from the
    // array in the first place, this base case is a bit weird, but it helps
       us work with the string, for example below we leverage
    // a `substring` property, where if the `substring` method is called with
        beginIndex equal to the length of the string, the
    // returned string is empty "", it is not an exception, or out of bounds
       error.
    if (target.length() == 0) {
        return true;
    }
    // check if the target string is already in the map, if it is we already
       have gone through it once, during the top-down recursion, since we
       drill down first, and then unwind, it is possible to meet the same
       suffx/target word multiple times, depending on the initial target word
        and dictionary of words, if it is in the map we will have the result,
        either we can (true) or we can not (false) construct that particular
       key/suffix from the dictionary of words.
    if (MEMO.containsKey(target)) {
        return true;
    }

    for (String word : dictionary) {
        // in case the main word is smaller than the current item from words,
            then simply skip this one, there is no way the build
        // the shorter main word from a longer word from the words dictionary
        if (target.length() < word.length()) {
            continue;
        }

        // pull the same number of characters from the main word, as our
           prefix, when this prefix matches
        String prefix = target.substring(0, word.length());

        // compare the prefix characters with the actual current item word,
           if they match, call recursively the function with the
        // left of suffix, the rest of the string note, there is a trick here
           , String.substring when invoked with a value for
        // beginIndex of exactly the string.length, will return emtpy string
           "", our base case this is how we keep pulling prefixes
        // from the main word, until the main word becomes empty string - "".
            We can also check if the current item.length is equal
        // to the main word.length, and if yes simply stop there, but we use
           this special property if String.substring instead
        if (prefix.equalsIgnoreCase(word) || constructible(target.substring(
           word.length()), dictionary)) {
            // if we hit this point that means the prefix was exactly equal
               to the word, and the recursive call also returned `true`
            // for the `suffix` part of the main word
            MEMO.put(target, true);
            return true;
```

```
        }
    }

    // in case we do not enter the true branch above, this means we could not
        build anytning from the words dictionary, therefore
    // return false
    MEMO.put(target, false);
    return false;
}
```

Things to note from the implementation above.

- we do not have a `false` base case exactly, we can add one where if the target word is `null` but for the sake of this task let us assume the user will not pipe in a `null` into the function call
- we have a guard case in the for loop body, which makes sure the current word from the dictionary is not bigger, than the actual target word, since we reduce the target word, on each recursive call.
- we pull a `prefix` i.e. always from the start of the target word, from the target word, which is the same length as the current word from the dictionary, and we then compare that prefix, to the current word from the dictionary, to make sure the current word actually can 'construct' the word from the start.
- we use a special case of the substring function, which when invoked exactly with the length of the string as first argument (which should be an index), returns an empty string
- in case we do not enter the positive branch in the `if` inside the for loop, we simply return `false` at the end of the function, meaning none of the words in the dictionary actually matched up during the `for` loop.

Evaluating the complexity of the solution

- without memoization, the complexity is exactly the branching factor to the power of the depth of the tree, meaning that we will not make more than M number of calls and we branch at most N times, meaning that it is therefore `O(n^m)`
- with memoization, the complexity is also modified, we now make at most `O(m)` number of calls however, we have to add the extra space occupied by the items in the memo table, now worse case scenario if we have a sequence of all ones, meaning that we will also store at most `target` number of keys and values in the memo table, there is something more to add to this complexity, and that is the actual memory occupied by the `prefix` string we construct which at worse can be of length `m`, so the final complexity has to account for that and actually it ends up being `O(m^2)`.

Therefore we can conclude that

- runtime complexity is O(n^m) without memo or O(m) with memo
- space complexity is O(m) without memo or O(m^2) with memo

All words

A variation of the previous problem is to collect all combinations from the dictionary that make up the target word, if any exist, we will try to collect the sequences in a list of strings, where each item in the list will be one of the words from the dictionary which makes up the final target word, they will be separated by coma, this is the same problem which basically needs a few small modifications, to make it work, and here is the implementation

```
List<String> allwords(String target, String[] dictionary) {
```

```
// this is the positive base case, reaching this point means that we have
    truncated the target string or word, to be empty, if
// it is empty that means we have found a sequence to exist that
   constructs the intial word, kind of
if (target.length() == 0) {
    return new ArrayList<>();
}
if (MEMO.containsKey(target)) {
    return MEMO.get(target);
}

// this is in a way the negative case, if none of the branches below
   suffice, we will return nil, which means there is no way to
// build the target string / word, from any words in the dictionary,
   provided above.
List<String> result = null;
for (String word : dictionary) {
    // in case the main word is smaller than the current item from words,
        then simply skip this one, there is no way the build
    // the shorter main word from a longer word from the words dictionary
    if (target.length() < word.length()) {
        continue;
    }

    // pull the prefix, the size of the prefix is exactly the size of the
        current word from the dictionary
    String prefix = target.substring(0, word.length());

    // only in case the prefix matches, can we continue with the rest of
       the main target word, or to be more precise, the suffix
    // is what we use to drill down
    if (prefix.equalsIgnoreCase(word)) {
        // construct the rest of the word, calling the function
           recursively, in case this returns a non nil result we know
           that
        // for the current prefix and suffix we can construct the target
           word, note we do this again top-down, we drill down to
        // the base case, we unwind, and only then can we make the
           decision if we can construct the word, this is very important
        // observation
        List<String> current = allwords(target.substring(word.length()),
           dictionary);
        if (!Objects.isNull(current)) {
            // from the result of the recursive call construct the string
                of words from the dictionary that can be combined to
            // make the word
            StringBuilder builder = new StringBuilder(word);

            // simply construct the string with some delimiter easily
               identify the words from the dictionary which were used to
            // construct the target word.
```

```
                for (String string : current) {
                    builder.append(",");
                    builder.append(string);
                }

                // the first time we see the result, it might be nil, meaning
                    we have to initialize it first, with an empty array
                if (Objects.isNull(result)) {
                    result = new ArrayList<>();
                }

                // add the built string of combinations to the final result
                   which we then return eventually, after the body of the
                // for loop
                result.add(builder.toString());
            }
        }
    }

    // at this point, result will either be initialized, or nil, if it was
       initialized, that means we have found at least one, but
    // maybe more, combinations from word dictionary that make up the target
       word
    MEMO.put(target, result);

    // this return serves as both a positive and negative case, if we never
       entered any true branch in the for loop above, it will
    // be nil, otherwise initialized
    return result;
}
```

The implementation is very similar to previous approaches, and here is where we can see how a very familiar pattern emerges, the general difference is how we mutate and mold the result, the way we collect and return it. In the implementation above we have again two base cases

- positive case - when the target word length is 0, we return a valid empty list, after all we already know that a target word that is empty, we can create the target empty word, by simply not taking anything from the dictionary in the first place
- negative case - here we have to make sure we can distinguish between the positive case and another case, we have to choose a return value for our list, that makes sense in an invalid case, the invalid case here, signifies that there is no sequence in the dictionary that makes up the target word, the most obvious solution is to make sure in a negative case we return null for the list

In the example above, the challenging step is to realize that we actually flatten the results of the calls to the recursive calls before we add them to the final result, this is important since the final result returned from the recursive function has to always be a one dimensional array with entries, however the call to the function itself also returns one dimensional array of entries too, we can not simply concatenate those two results, we have to flatten the return value of the call then add it to the final result. This is the main difference as far as this task goes, the rest of the task and solution follows the already well established pattern, of

- creating the base cases - negative and positive
- building the actual body of the function

- adding to the final result of the function
- returning the result of the computation
- memoization of the results

Grid counter

Another common task is, given a grid of NxM cells. Where N is the number of rows, and M is the number of cols, find all possible ways to get from the top left corner (start) to the bot right one (end).

This one, similarly to Fibonacci, is also leveraging recursion and memoization, in this task we are required to basically compute all possible ways to go from one start position to another, while we can move only in two directions, which are right and down, to reach the bottom right (end) coordinates.

In the example below the grid coordinates start from 1,1 and the end coordinates and size of the grid is defined by the input arguments to the `count` function R,C

```
static final class GridPathCounter {

    // hold a string key which represents the count for the specific
       coordinates of the current row and col, [r,c]. The reason we can
    // use memo here is that once a given coordinate position of row,col is
       visited, we do not have to visit it again, all paths
    private final Map<String, Integer> MEMO = new HashMap<>();

    public int count(int r, int c) {
        // when either of the coordinates points at an invalid position, then
            there are no valid paths to reach the end goal, there is
        // no grid where the row or columns can be 0, remember the rows and
           cols here are defined positions, therefore they must be
        // greater or equal to 1
        if (r == 0 || c == 0) {
            return 0;
        }

        // in a grid of 1x1, there is only 1 path between the start, top left
           , and the end, bot right, and both are exactly the same
        // grid cell
        if (r == 1 && c == 1) {
            return 1;
        }

        // build a unique key, from the current pair of coordinates, this
           will ensure we can key the map, correctly and have a unique
        // key - value relation between the coordinates and the count of
           paths to these specific coordinates
        String key = String.format("%d,%d", r, c);
        if (MEMO.containsKey(key)) {
            return MEMO.get(key);
        }

        // calculate the count paths, from the current position to the
           previous, possible positions, from the current position of
```

```
        // [row,col] we can either go up a row, or to the left in the col
           coordinates.
        int count = count(r - 1, c) + count(r, c - 1);

        // remember the count for this specific combination of rows and cols
           using the unique key, note the key is delimiting the
        // coordinates with a coma, which is a good idea, to ensure
           uniqueness of pairs. Why does remembering the key work, well since
        // we go in both directions, in this case first in the row the col,
           it is possible for one branch of these two, to each a
        // specific path first, meaning that when the second branch goes
           through, and sees the same path i.e coordinates they have been
        // already visisted and computed, this ONLY works, because we are
           doing a top down approach, where the recursive calls drill
        // down to the bottom, to the base cases and then ascends up,
           accumulating the count for specific paths.
        MEMO.put(key, count);
        return count;
    }
}
```

The memo approach in this task is rather important, first the key has to be unique, to make sure we can correctly distinguish between the combination of row x col pairs, secondly we can not swap those since those coordinates represent specific unique position pairs.

```
    // call the path grid counter with a grid of 2 rows and 3 columns
    gridCounter.count(2, 3)
```

```
          (2, 3)
        /       \
    (1, 3)       (2, 2)
   /     \       /     \
(0, 3)  (1, 2) (1, 2) (2, 1)
        /    \        /     \
     (0, 2) (1, 1)  (1, 1) (2, 0)
```

If we have a look at an example of 2x3 grid below, we can notice a few nice properties and make several observations by just looking at the tree representing the call stack

- the `count` function is called with the max coordinates (NxM), the recursive approach is top down
- going to the left in the tree means we go up a row (row - 1)
- going to the right in the tree means we go left a col (col - 1)
- the pair branch root pair of `(1, 2)`, is contained twice, this is where memo will store the first time (coming from the row sub-path) it was visisted and simply reference it from the memo table the second time (coming from the col sub-path)
- nodes that contain `0` in one of their pairs are invalid, and indeed grids with 0 dimensions are not valid according to our task

Iterative problems

In this chapter we are going to solve the same issue as above, however taking the iterative approach, or the tabulation approach in dynamic programming, where instead of starting from the top, drilling down to

the bottom, we take the inverse approach and we start from the bottom and build up to the top to find the solution. Usually these implementations involve us using iterative approach, where the values are being accumuated into a table or array of some sort, be it a one or n-dimensional array

Fibonacci

In the solution provided below, we can make a few observations, mainly that the table that is used to create the sequence is of length `n + 1`, why is that, this is because we have to account for the positive base case, in our example that is `n[0] and n[1]`, usually in dynamic programming solutions with iterative approach, the `0-th` case is used to initialize the table with the positive base case, while the rest of the table is initialized with the negative case values

```
int fibonacci(int n) {
    // this is a good idea overall, we can add a basic guard case first, to
       make sure we do not receive an invalid input, and we
    // know that there is no 0-th Fibonacci sequence number
    if (n <= 0) {
        return 0;
    }
    // we first create our tabulation table, which is going to store all
       values of the Fibonacci sequence, this is a common approach
    // when solving these issues, with iterative dynamic programming
    int[] table = new int[n + 1];

    // by default we have to fill it up with some invalid, negative case
       values, in this case -1 is a good candidate since we know
    // that the Fibonacci sequence has no negative numbers
    Arrays.fill(table, -1);

    // we have to initialize our base case, we know that the first two
       numbers of the sequence are 0 and 1, and every other number
    // in the sequence can be built on top of those two initial values
    table[0] = 0;
    table[1] = 1;

    // start form the 2-nd fib number and calculate up until we reach n, note
        that the size of the array here is actually n + 1,
    // this is because the base case of n[0]
    for (int i = 2; i < table.length; i++) {
        table[i] = table[i - 1] + table[i - 2];
    }

    // return the value at position n in the array, our array is holding n +
       1 elements, and the last element is at index n, since
    // we already have a base case value at index 0
    return table[n];
}
```

Can sum

Given a target array of numbers, allowing number reuse from the array, find if there is at least one combination of the given numbers that sums up to the given target sum, note that the same number from the array can be repeated multiple times. To solve this task we have to come up with the basic base cases, where

```
boolean summable(int target, int[] sequence) {
    // we can put a small base case boundary on our input, to make sure that
       the function would operate correctly, after all if we
    // have a target sum of 0, we can easily assume that it can be generated
       by simply not taking any element from the sequence
    if (target == 0) {
        return true;
    }

    // start with a pre-defined array of boolean statuses, note that we again
        generate an array which is filled with all negative
    // cases, by default
    boolean[] table = new boolean[target + 1];
    Arrays.fill(table, false);

    // set the base postive case as we know, for a target sum of 0, we can
       say that there can be a sequence of numbers, by simply
    // taking no numbers, we start off from this initial value and build from
        it
    table[0] = true;

    // loop through all states in the tabulation table
    for (int i = 0; i < table.length; i++) {
        // when we meet a positive case, we can build off of it
        if (table[i] == true) {
            // we are in a positive case, meaning the sum to target[i] can be
                generated, below we check starting from `i` which
            // other sums can be generated by adding `num` to `i`
            for (int num : sequence) {
                // sums `i + num` can be generated if the array has enough
                   space to fit the number, remember that the array / table
                // is created with at most target + 1 elements, meaning that
                   we have to check our bounds first
                if (i + num < table.length) {
                    // i + num is in range and we mark this sum target as
                       possible
                    table[i + num] = true;
                }
            }
        }
    }

    // return the state at the target sum position, remember that we will
       have one of three cases for this position
    // - nothing ever summed up to target - meaning it will remain `false`
       from the initialization above
```

```
    // - there was no pair of `i + num` that summed up to target, we either
       overshot or undershot
    // - there was a pair of `i + num` that exactly summed up to target and
       the cell was marked true
    return table[target];
}
```

Best sequence

An extension of the can sum task, could be to find the best sequence which means, the sequence with the least amount of numbers that makes up the target sum.

```
List<Integer> best(int target, int[] sequence) {
    // we can put a small base case boundary on our input, to make sure that
       the function would operate correctly, after all if we
    // have a target sum of 0, we can easily assume that it can be generated
       by simply not taking any element from the sequence
    if (target == 0) {
        return new ArrayList<>();
    }

    // start with a pre-defined array of negative states, note that we again
       generate an array which is filled with all negative
    // values, by default
    List[] table = new List[target + 1];
    Arrays.fill(table, null);

    // set the base positive case as we know, for a target sum of 0, we can
       say that there can be a sequence of numbers, by simply
    // taking no numbers, we start off from this initial value and build from
        it
    table[0] = new ArrayList<>();

    // loop through all states in the tabulation table
    for (int i = 0; i < table.length; i++) {
        // when we meet a positive case, we can build off of it
        if (!Objects.isNull(table[i])) {
            // we are in a positive case, meaning the sum to target[i] can be
                generated, below we check starting from `i` which
            // other sums can be generated by adding `num` to `i`
            for (int num : sequence) {
                // sums `i + num` can be generated if the array has enough
                   space to fit the number, remember that the array / table
                // is created with at most target + 1 elements, meaning that
                   we have to check our bounds first
                if (i + num < table.length) {
                    // i + num is in range what we do is clone the sequence
                       already at table[i] and append the new number that we
                    // have used to generate `i + num`
                    List<Integer> seq = new ArrayList<>(table[i]);
                    seq.add(num);
```

```
                    // note that this will override sequences that are
                       already in place, we can optimize this a bit by first
                    // checking if there is already something in table[i +
                       sum] and skip it, to find the first, instead of the
                       last
                    // sequence, but that is implementation detail
                    List<Integer> current = (List<Integer>) table[i + num];

                    // before we can replace the sequence with the new one,
                       we have to check if there is already something there,
                       or
                    // if the new sequence is 'better', or in other words
                       have less elements than the one currently being there
                    if (Objects.isNull(current) || current.size() > seq.size
                       ()) {
                        table[i + num] = seq;
                    }
                }
            }
        }
    }

    // return the value at the position for target, remember again we can
       either have valid list here, or nil, if we never managed
    // to generate the target sum whatsoever
    return (List<Integer>) table[target];
}
```

Can construct

A variation of the `cansum` problem where we have to now find if a `target word` can be constructed by taking `words from a dictionary` passed in as input arguments.

The way this is solved by using iteration approach is a bit different, the key thing to note here is how to "encode" the characters of the target word, into the `boolean` table array we use. What we do here is first to make the boolean array of size `target.length + 1`

```
boolean constructible(String target, String[] dictionary) {
    // this is a generally good idea, overall, we can add this basic fallback
        case since we know that if the target word is empty,
    // we can simply return true, we can build the empty string by not taking
        anything from the dictionary in the first place
    if (target.isEmpty()) {
        return true;
    }

    // we create the table here, which has to be of at least length + 1 size,
        this is explained below, on how the indices are mapped
    // to represent substrings from the main target word
    boolean[] table = new boolean[target.length() + 1];
```

```
// fill the array by default with false, we assume that we can not build
   the target word from the words in the dictionary first
Arrays.fill(table, false);

table[0] = true;
for (int i = 0; i < target.length(); i++) {
    // we loop over the main target word, and update the table of boolean
        values, whenever we find a prefix taken from the
    // current position `i` up until `i + word.length` from the string,
       that matches up with any of the words in the dictionary.
    for (String word : dictionary) {
        // for each word from the dictionary we check first if we can
           pull a prefix from the target word.
        if (i + word.length() > word.length()) {
            continue;
        }
        // we pull a prefix, based on the current position we are at `i`
           and the length of the current word from the dictionary.
        String prefix = target.substring(i, i + word.length());
        // if the prefix we pulled from the target word matches with the
           word from the dictionary, we can mark this position in
        // the boolean table array as true
        if (prefix.equalsIgnoreCase(word)) {
            // now note what is going on here, the index we mark is
               actually not an index, it is a length or position, this is
            // why we create the array to be target.length + 1, take the
               following example here, which follows the table updates
            // target word - abcdef, dictionary - ["abc", "ab", "ef", "
               def"], table - [(a)t, (b)f, (c)f, (d)f, (e)f, (f)f, (-)f]

            // i = 0 prefix is "abc", word "abc" from the dictionary
               table of booleans will become - [(a)t, (b)f, (c)f, (d)t, (
               e)f, (f)f, (-)f]
            // i = 0 prefix is "ab", word "ab", from the dictionary table
                of booleans will become - [(a)t, (b)f, (c)t, (d)t, (e)f,
               (f)f, (-)f]
            // i = 0 prefix is "ef", word "ef" from the dictionary table
               of booleans unchanged, there is no "ef" prefix starting
               from i = 0
            // i = 1 prefix is "bcd" word is "abc" from the dictionary
               table of booleans unchanged, there is no "abc" prefix
               starting from i = 1
            // i - keeps increasing table will change when i becomes
               equal to 3, where we find another prefix match, against
               word in the dictionary
            // i = 3 prefix is "def" word is "abc" from the dictionary
               table of booleans unchanged, there is no "abc" prefix
               starting from i = 3
            // i = 3 prefix is "def" word is "ab" from the dictionary
               table of booleans unchanged, there is no "ab" prefix
               starting from i = 3
```

```
                // i = 3 prefix is "def" word is "ef" from the dictionary
                   table of booleans unchanged, there is no "ef" prefix
                   starting from i = 3
                // i = 3 prefix is "def" word is "def" from the dictionary
                   table of booleans will become - [(a)t, (b)f, (c)f, (d)t, (
                   e)f, (f)f, (-)t]

                // we can see that for the first prefix match of "abc", we
                   marked at index / position i = 0, "abc".length = 3, target
                   [3] = true
                // we can see that for the second prefix match of "def", we
                   marked at index / position i = 3, "def".length = 6, target
                   [6] = true
                table[i + word.length()] = true;
            }
        }
    }

    // return whatever value is stored at the last position available,
       remember that the table array is of target.length + 1,
    // meaning that at index target.length we will hold the actual answer, if
        we can build the target word from the dictionary of
    // words.
    return table[target.length()];
}
```

If we go through the implementation above, we can track how the `boolean` table will change, remember we go through the length of the target word, and for each `char` from the target word, we go through each word of the dictionary. We then take a prefix starting from the current character we are at `i` and the prefix is from `i` to `i + word.length`. If the prefix from the target word matches the current word from the dictionary we can set that position in the `boolean` table to true, `table[i + word.length] = true`

Take a good note at the index that is being set above, it is actually taken from the current index `i` position we are at in the target word plus the length of the current word from the dictionary, this means that if we find that there are words that can build up the target word, the final position in the `boolean` table that will be set to true and that we need to check is actually `table[target.length]`, if we have a `true` value at that position in the `boolean` table we can conclude that there were words from the dictionary which make up the target word

Taking the following example for the given words, we can see which ones can up the target word immediately, there are a few options

```
target - abcdef
dictionary - ["ab", "abc", "cd", "ef", "def", "abcd"]
combinations - ["abc", "def"], ["ab", "cd", "ef"], ["abcd", "ef"].
```

```
i = 0 prefix is "ab", word "ab" from the dictionary table of booleans - [2] =
    true <> [(a)t, (b)f, (c)t, (d)f, (e)f, (f)f, (-)f]
i = 0 prefix is "abc", word "abc" from the dictionary table of booleans - [3]
    = true <> [(a)t, (b)f, (c)t, (d)t, (e)f, (f)f, (-)f]
i = 0 prefix is "ab", word "cd" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
```

```
i = 0 prefix is "ab", word "ef" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 0 prefix is "abcd", word "abcd" from the dictionary table of booleans -
   [4] = true <> [(a)t, (b)f, (c)t, (d)t, (e)t, (f)f, (-)f]

i = 1 prefix is "bc", word "ab" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 1 prefix is "bcd", word "abc" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 1 prefix is "bc", word "cd" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 1 prefix is "bc", word "ef" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 1 prefix is "bcde", word "abcd" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)

i = 2 prefix is "cd", word "ab" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 2 prefix is "cde", word "abc" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 2 prefix is "cd", word "cd" from the dictionary table of booleans - [4] =
    true <> [(a)t, (b)f, (c)t, (d)t, (e)t, (f)f, (-)f]
i = 2 prefix is "cd", word "ef" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 2 prefix is "cdef", word "abcd" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)

i = 3 prefix is "de", word "ab" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 3 prefix is "def", word "abc" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 3 prefix is "de", word "cd" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 3 prefix is "de", word "ef" from the dictionary table of booleans -
   unchanged, prefix != current word (string.equals is false)
i = 3 prefix is "def", word "def" from the dictionary table of booleans - [6]
    = true <> [(a)t, (b)f, (c)t, (d)t, (e)t, (f)f, (-)t]
```

We can already see that by the time we go through `i = 3`, the position `tbl[6]` in the `boolean` table, has already become true, which means that there are certainly words in the dictionary which build up to the target word, we can keep going here, but we can also early bail from the iterations, since we have found what we need to know.

Count construct

A variation of the task above which aims at counting the total number of times the target string can be constructed from a set of words in a dictionary, needs a few small changes, but follows the same recipe.

```
int count(String target, String[] dictionary) {
    // this is a generally good idea, overall, we can add this basic fallback
        case since we know that if the target word is empty,
```

```
    // we can simply return 1, we can build the empty string by not taking
       anything from the dictionary in the first place
    if (target.isEmpty()) {
        return 1;
    }

    // we create the table here, which has to be of at least length + 1 size,
        this is explained below, on how the indices are mapped
    // to represent substrings from the main target word
    int[] table = new int[target.length() + 1];

    // fill the array by default with 0, we assume that we can not build the
       target word from the words in the dictionary first
    Arrays.fill(table, 0);

    // default value, which will be used to start the accumulation of the
       total count
    table[0] = 1;
    for (int i = 0; i < target.length(); i++) {
        // we loop over the main target word, and update the table of boolean
            values, whenever we find a prefix taken from the
        // current position `i` up until `i + word.length` from the string,
           that matches up with any of the words in the dictionary.
        for (String word : dictionary) {
            // for each word from the dictionary we check first if we can
               pull a prefix from the target word.
            if (i + word.length() > target.length()) {
                continue;
            }
            // we pull a prefix, based on the current position we are at `i`
               and the length of the current word from the dictionary.
            String prefix = target.substring(i, i + word.length());
            if (prefix.equalsIgnoreCase(word)) {
                // each time we reach this index, e will increment the count
                   here, this position can be reached multiple times from
                // multiple dictionary words, in the end we will have the
                   table containing all possible combinations, remember that
                // we have a default value of 1 for i = 0, each time we hit
                   this branch we will accumulate the current count at [i]
                // at the [i + w.len].
                table[i + word.length()] += table[i];
            }
        }
    }

    // return whatever value is stored at the last position available,
       remember that the table array is of target.length + 1,
    // meaning that at index target.length we will hold the actual answer, if
        we can build the target count
    return table[target.length()];
}
```

You will note that the general algorithm follows the same idea, however here we are accumulating the current count to the next, each time we find a new prefix match at `table[i + w.len]` we accumulate to that value the current count at `table[i]`. Why is that ? Well if we can construct at [i + w.len] that means that we have already constructed at [i], therefore we take the current count at [i] and accumulate it. The idea is very much the same as the example above, each time we reach a certain string state, for example

```
target - abcdef
dictionary - ["ab", "abc", "cd", "ef", "def", "abcd"]
combinations - ["abc", "def"], ["ab", "cd", "ef"], ["abcd", "ef"].
```

Here is a very brief flow of how the accumulation happens when we go through the string and the words, below we are only looking for the positive `true` branch cases where the `prefix` matches the current word from the dictionary, the most important things to note is the current value if `i` as well as the current value stored at `table[i]`, which will be used to accumulate at `table[i + w.len]`. We accumulate only when the current prefix taken from the main target word matches the current word from the dictionary, and we also loop through each character in the target word

We reach the word from dictionary `ab` only once, so the count in the table at `ab` position (which is table[2] += table[0]) will be one, `i = 0`

We reach the word from dictionary `abc` only once, so the count in the table at `abc` position (which is table[3] += table[0]) will be one, `i = 0`

We reach the word from dictionary `abcd` only once, so the count in the table at `abcd` position (which is table[4] += table[0]) will be one, `i = 0`

Eventually here is where the interesting part comes in, when `i > 0` and we find a sub-prefix, like the ones presented below, is how the accumulation works

We reach the word from dictionary `cd` twice, so at this point after we have processed over `cd` the count at that index will be 2, (which is `table[2 + 2]`, since `cd` starts at index 2, and is of length 2, therefore it ends at position 4, due to the formula of `table[i + word.length]`. The first time we go through it is actually when `[i] = 0`, and the word is `abcd`, then we will set `table[0 + 4] += table[0]` (table[0] is actually the default, base value 1, always). The second time is when we are at `[i] = 2`, and the current words is `cd`, then we will set `table[2 + 2] += table[2]` (table[2] is actually the count for `ab`). At the end we therefore have a value of `2` at the `table[4]` position, and that is indeed true, we can reach `abcd` from two routes.

We reach the word from dictionary `def` only once, `i = 3`. This is because to reach `def` the current `i = 3`, with the length of `def = 3`, therefore the value at `table[3 + 3] += table[3]`, remember that at `table[3]` we have a count of 1 already, that is `abc`

We reach the word from dictionary `ef` only once, `i = 4`. This is because to reach `ef` the current `i = 4`, with the length of `ef = 2`, therefore the value at `table[4 + 2] += table[4]`, remember that at `table[4]` we have a count of 2 already, that is `abcd` and `ab cd`. Now since at table[6] we already had `1` from the step above, when we reached `def`, we add to that the value at `table[4]` which is 2 at this point, meaning at now at `table[6] = 3`. And that is indeed the actual total number of ways to build `abcdef` from the given dictionary of words.

Grid counter

Another common task is, given a grid of NxM cells. Where N is the number of rows, and M is the number of cols, find all possible ways to get from the top left corner (start) to the bot right one (end).

To solve this problem iteratively we have to change the way we inspect the