Solution Review: The Edit Distance Problem

In this lesson, we will look at some solutions to the edit distance problem.

We'll cover the following

Solution 1: Simple recursion

```
def editDistanceRecurse(str1, str2, i, j):
    if i == len(str1):  # base case of reaching the end of str1
        return len(str2) - j

    if j == len(str2):  # base case of reaching the end of str2
        return len(str1) - i

    if str1[i] == str2[j]:  # if the characters match, we move ahead
        return editDistanceRecurse(str1, str2, i+1, j+1)
    # if characters don't match
    return 1 + min(editDistanceRecurse(str1, str2, i+1, j+1),   # we can change characters
                   editDistanceRecurse(str1, str2, i, j+1),     # we can have an insertion in str1
                   editDistanceRecurse(str1, str2, i+1, j))     # we can have a deletion in str1 (

def editDistance(str1, str2):
    return editDistanceRecurse(str1, str2, 0, 0)
```

```
    return editDistanceRecurse(str1, str2, 0, 0)

print(editDistance("teh", "the"))
```

Explanation

The idea behind the algorithm is to align both the sequences so that the fewest number of operations is used. Let's first see some examples of how alignment helps us convert `str1` into `str2`.

- `dog` and `dodge` can be aligned as `do-g-` and `dodge`. And then we can insert `d` and `e` in both the blanks.
- `read` and `red` can be aligned as `read` and `re-d`. A dash in the second string means we may remove the corresponding character in the first string thus converting `read` to `red`.
- `thr` and `the` can be aligned as `thr` and `the`. Here, aligning mismatching characters means that we replace the mismatching character in the first string with the corresponding character in the second string, i.e., replace `r` with `e`.

Now let's look at the algorithm we wrote. First, we have the base cases: if we run out of `str1`, we will have to insert all the remaining characters of `str2`(*lines 2-3*). Similarly, if we run out of `str2`, we will need to delete all the remaining characters from `str1` (*lines 5-6*).

We have two cases: if the current characters match, we are all good and can move forward in both strings (*lines 8-9*). However, if they do not match we need to check each of the three possibilities and return the one with the minimum cost. The three recursive calls correspond to the change of character (*line 11*), insertion of the character (*line 12*), or removal of the character (*line 13*).

Let's look at a visualization of this algorithm below.

editDistance("teh", "the")

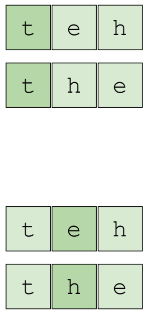
editDistance("teh", "the")

t	e	h
t	h	e

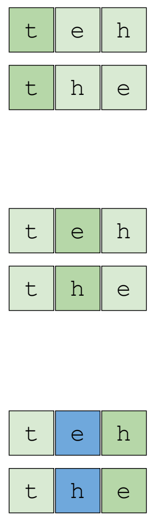
editDistance("teh", "the")

t	e	h
t	h	e

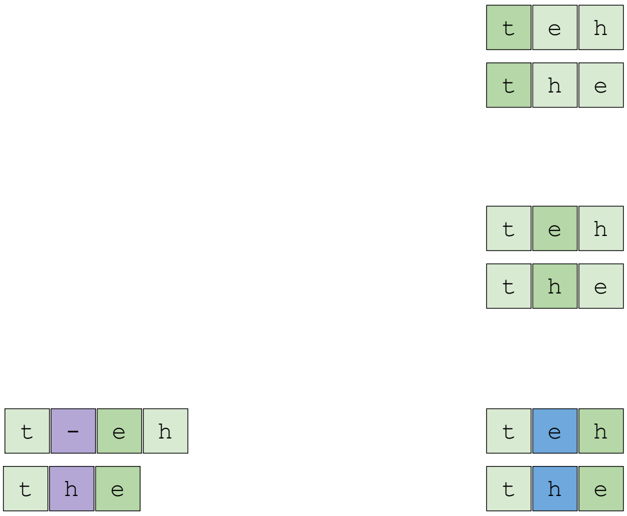
We will begin with the first characters of both the string



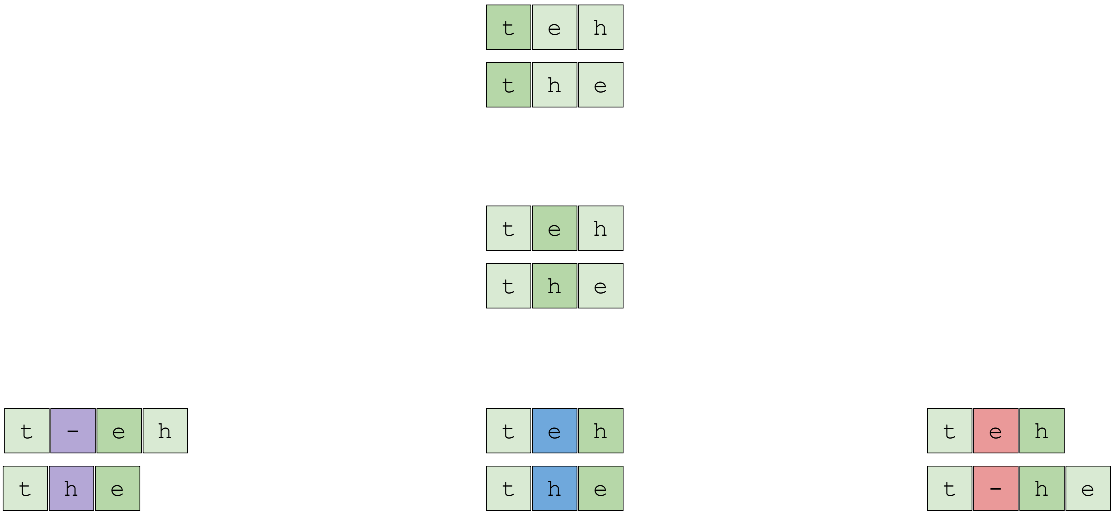
Since the characters matched, we will only have one call where we move one step ahead in both the strings



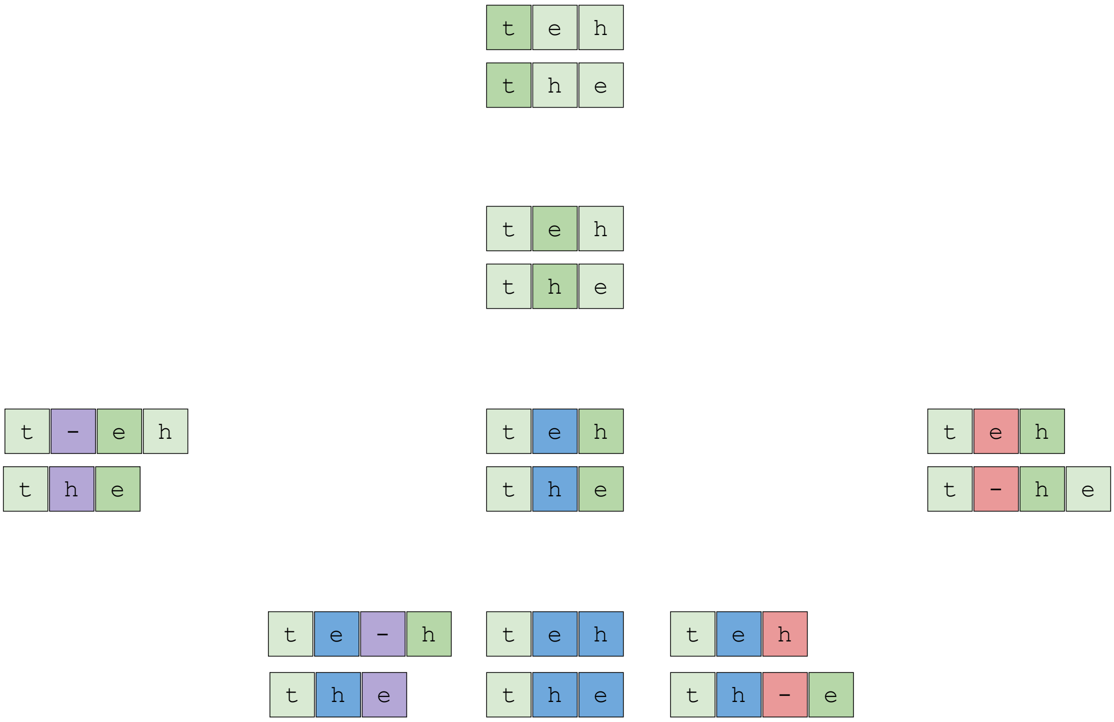
Since these characters do not match, we will have three recursive calls: one where we progress in both strings i.e. denoting change of the character



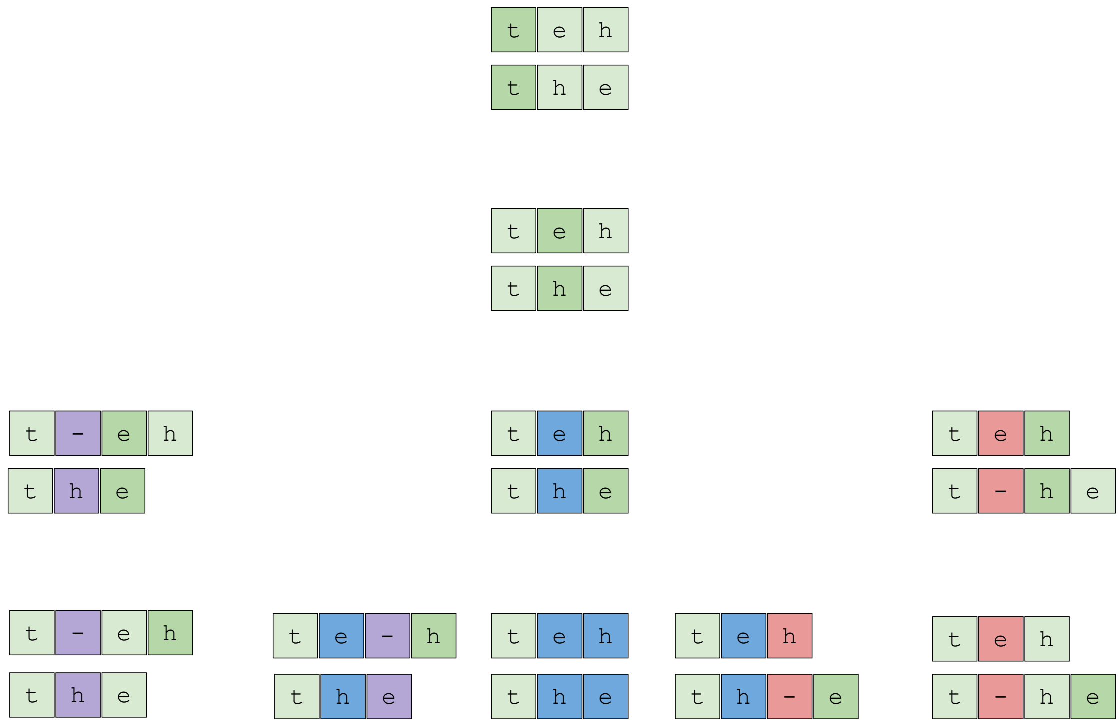
another call where we progress in the second string (insert _ in first string): denoting the insertion



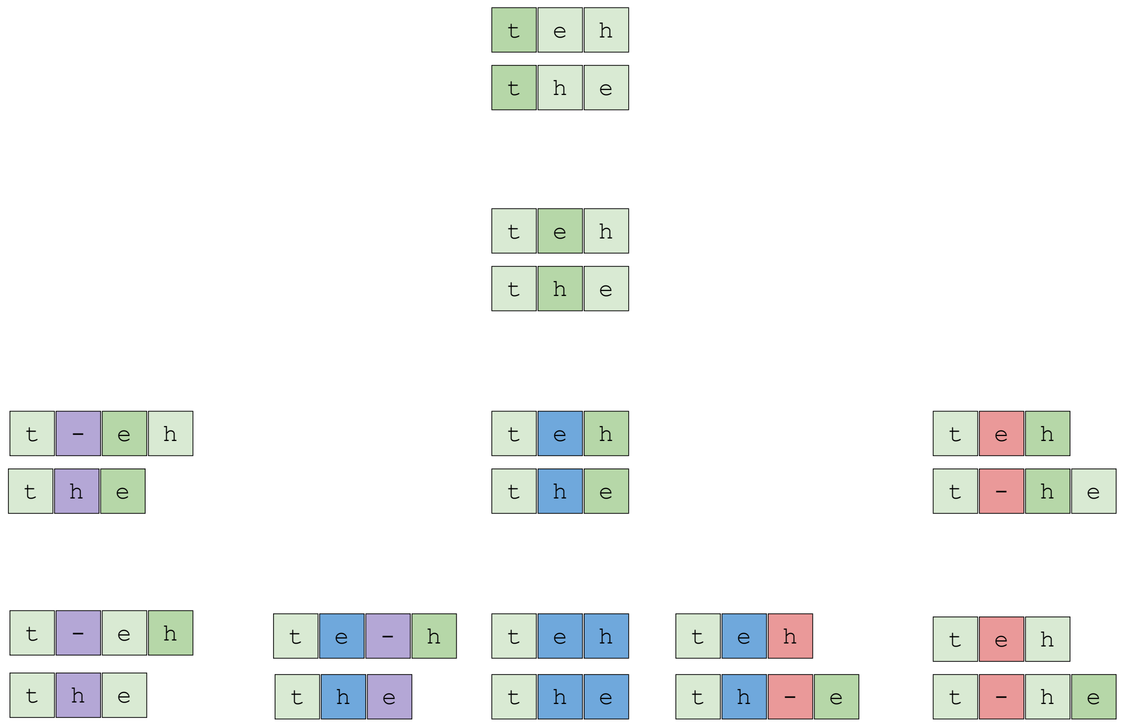
and third call where we progress in first string (insert _ in second string): denoting deletion



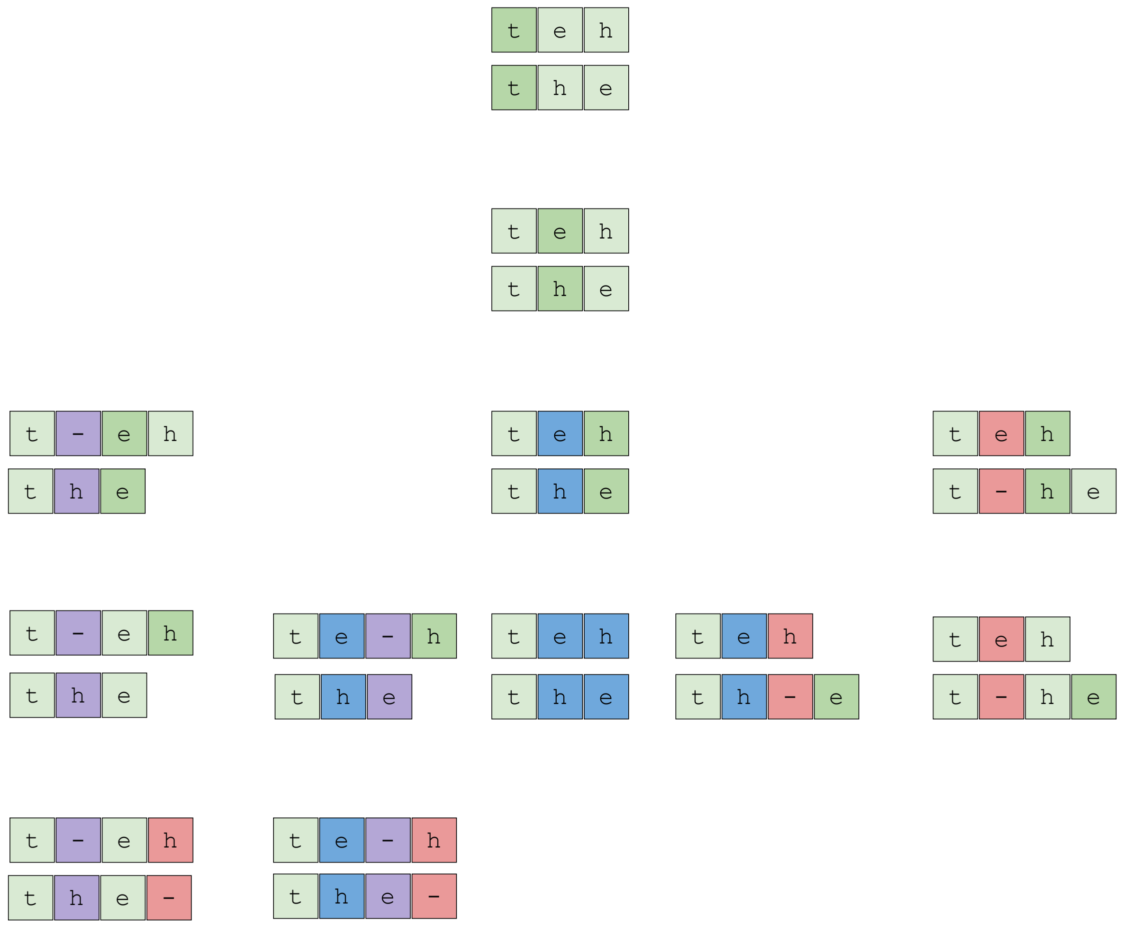
For the second call, we have a mismatch again so we will have three calls again



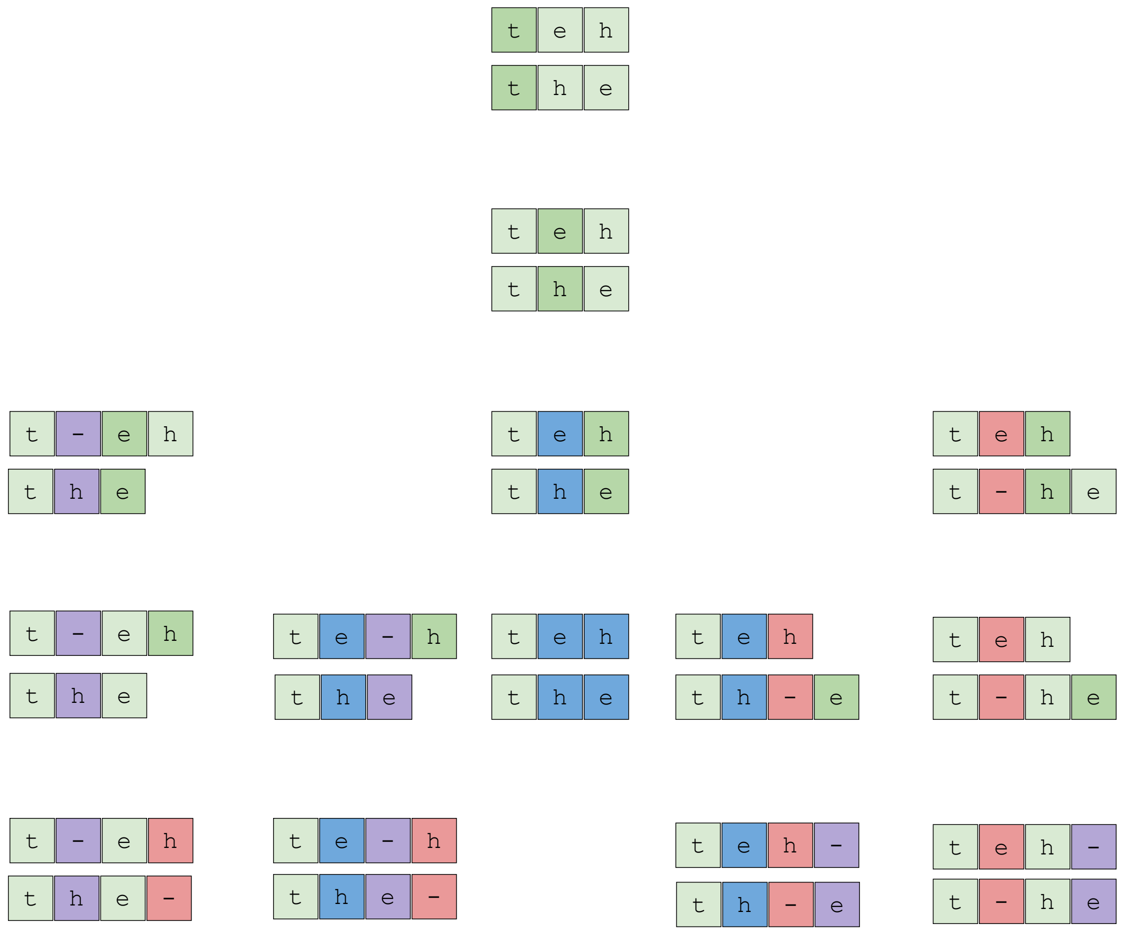
Whereas first and third call have a match of characters so we will have one call for them



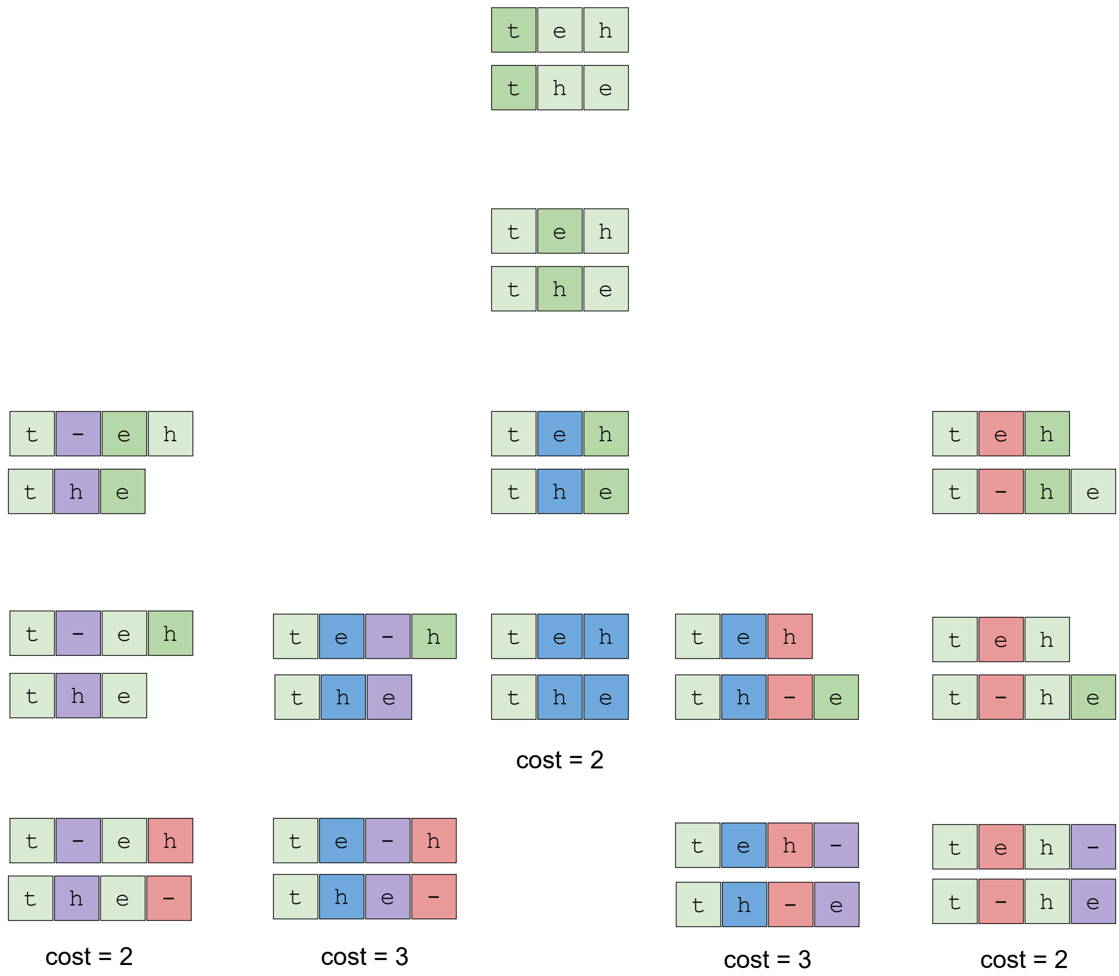
Now we have run out of at least one string in all the calls so we will now hit the base case



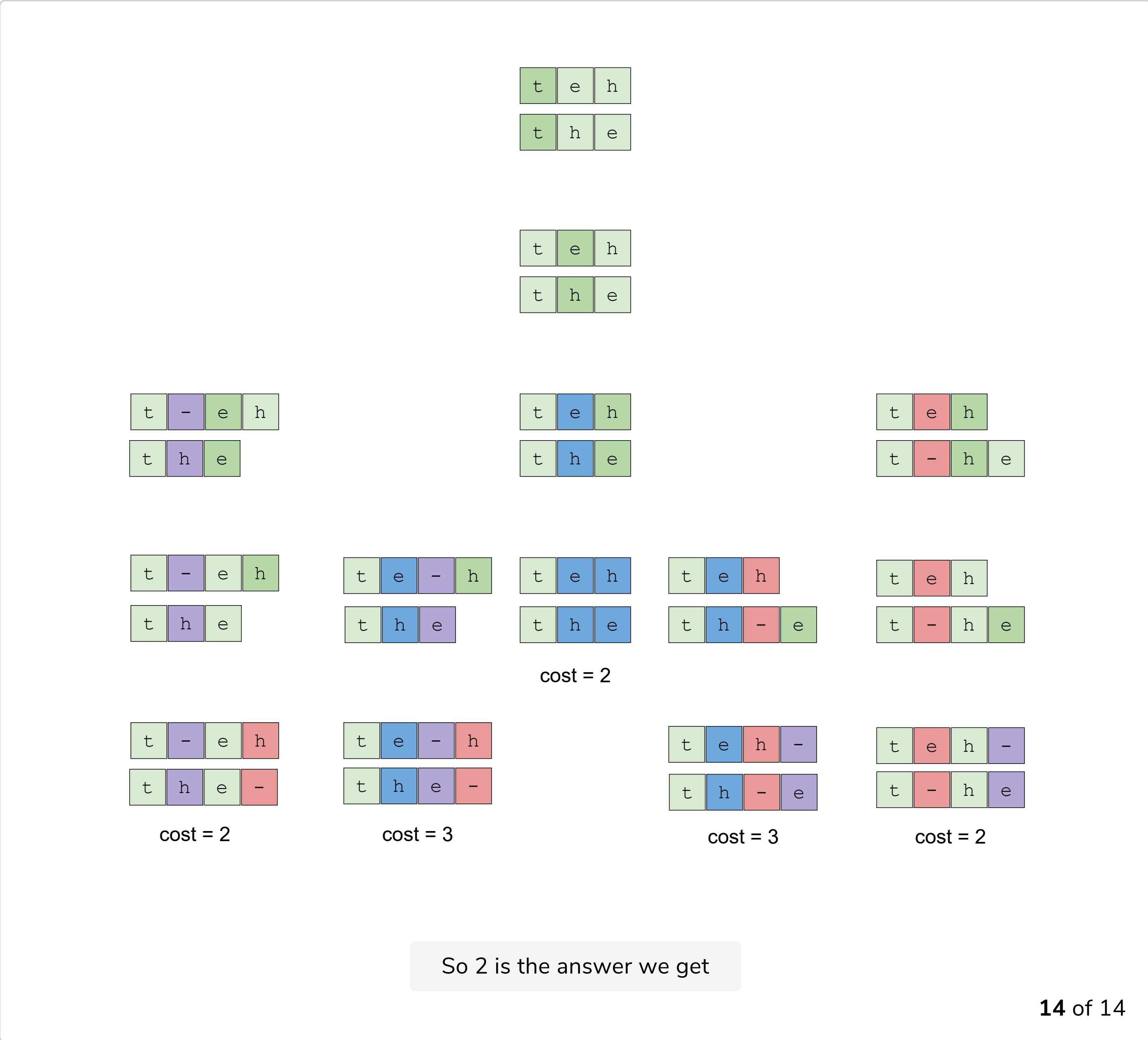
The two left most calls will have deletion of the character since its the second string we ran out of



While the right most two calls will have insertions since second string characters were left



Let's look at the cost for each of these leave calls



Time complexity

At each point, we are faced with three options in the worst case. Thus, if we keep in mind the worst case of two strings of sizes n and m, containing no common character, we will have a time complexity of $\mathbf{O(3^{n+m})}$.

Solution 2: Top-down dynamic programming

Let's first see how this problem satisfies both prerequisites for applying dynamic programming.

Optimal substructure

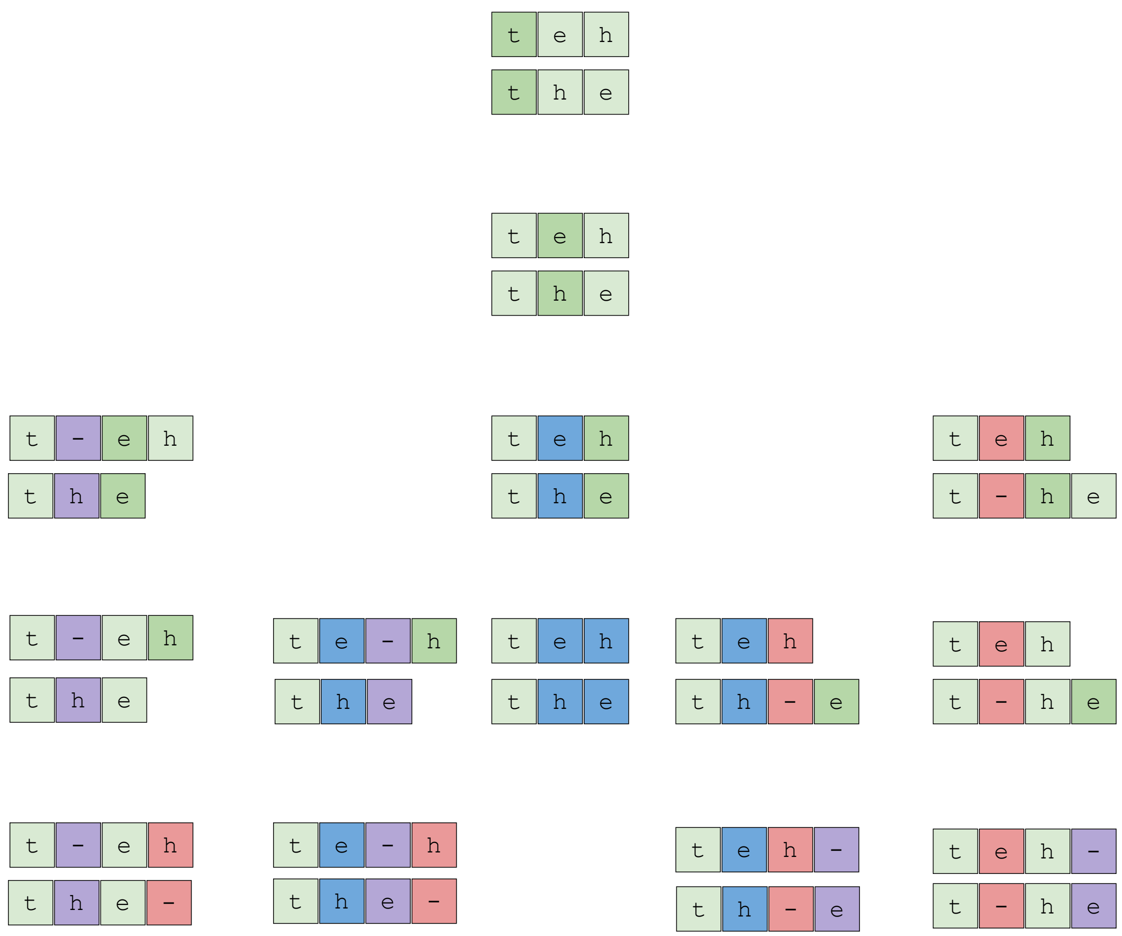
The optimal answer for a pair of strings of size n and m can be found by using the following:

- Optimal answer to the subproblem of substrings of sizes $n - 1$ and $m - 1$ formed by removing the first characters.
- Optimal answer to the subproblem of keeping the first string as it is (size of n) and removing the first character of the second string (size of $m - 1$).
- Optimal answer to the subproblem of keeping the second string as it is (size of m) and removing the first character of the first string (size of $n - 1$).

Since we can break down the main problem in terms of specific subproblems, this problem has an optimal substructure.

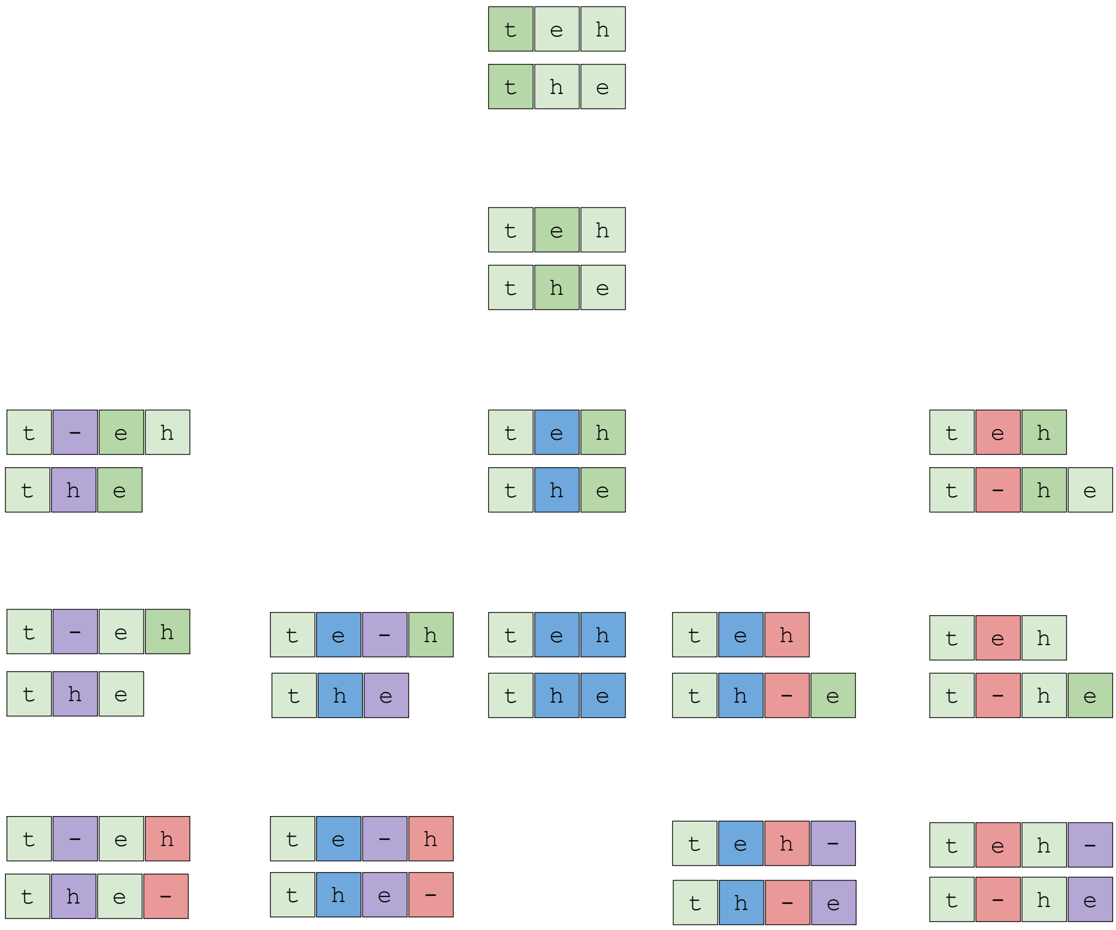
Overlapping subproblems

Let's revisit the visualization above to see some repeating subproblems.

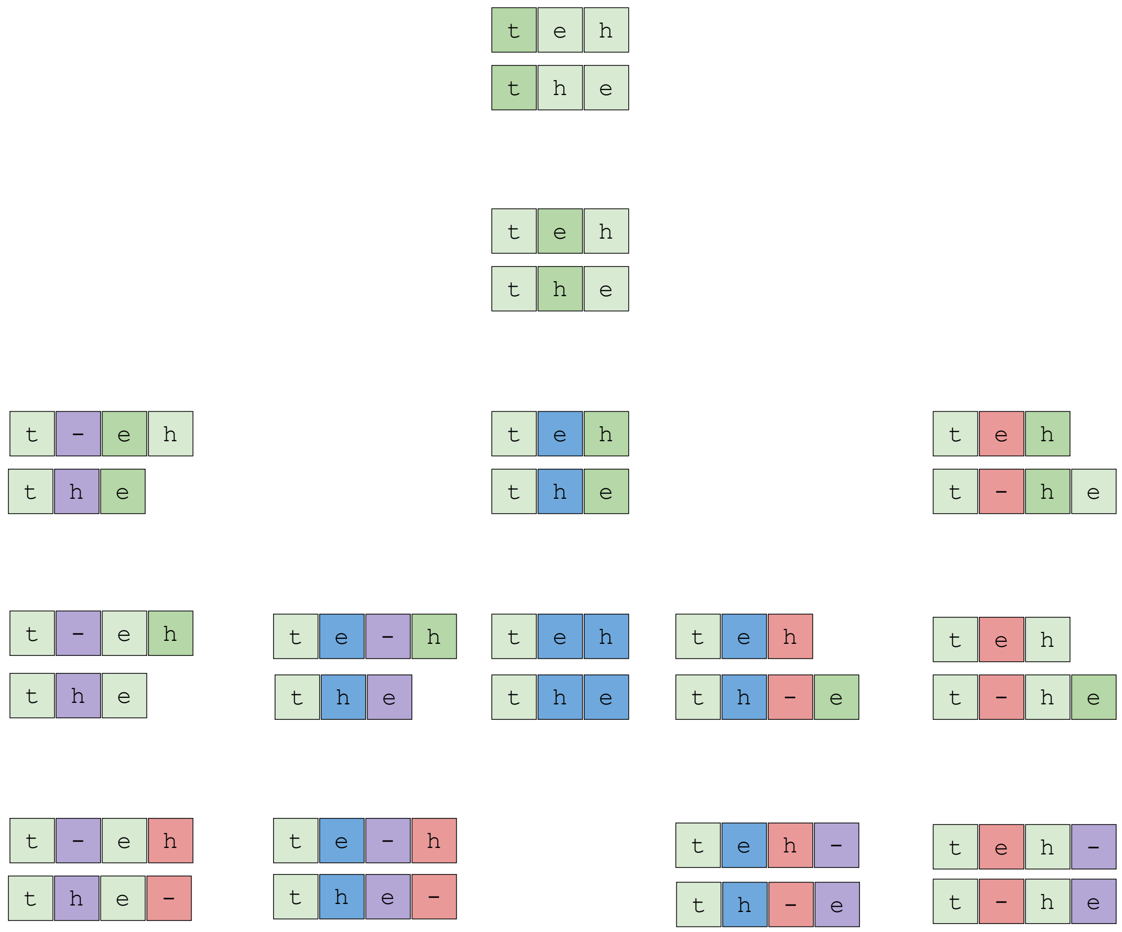


Let's look at some repeating subproblems

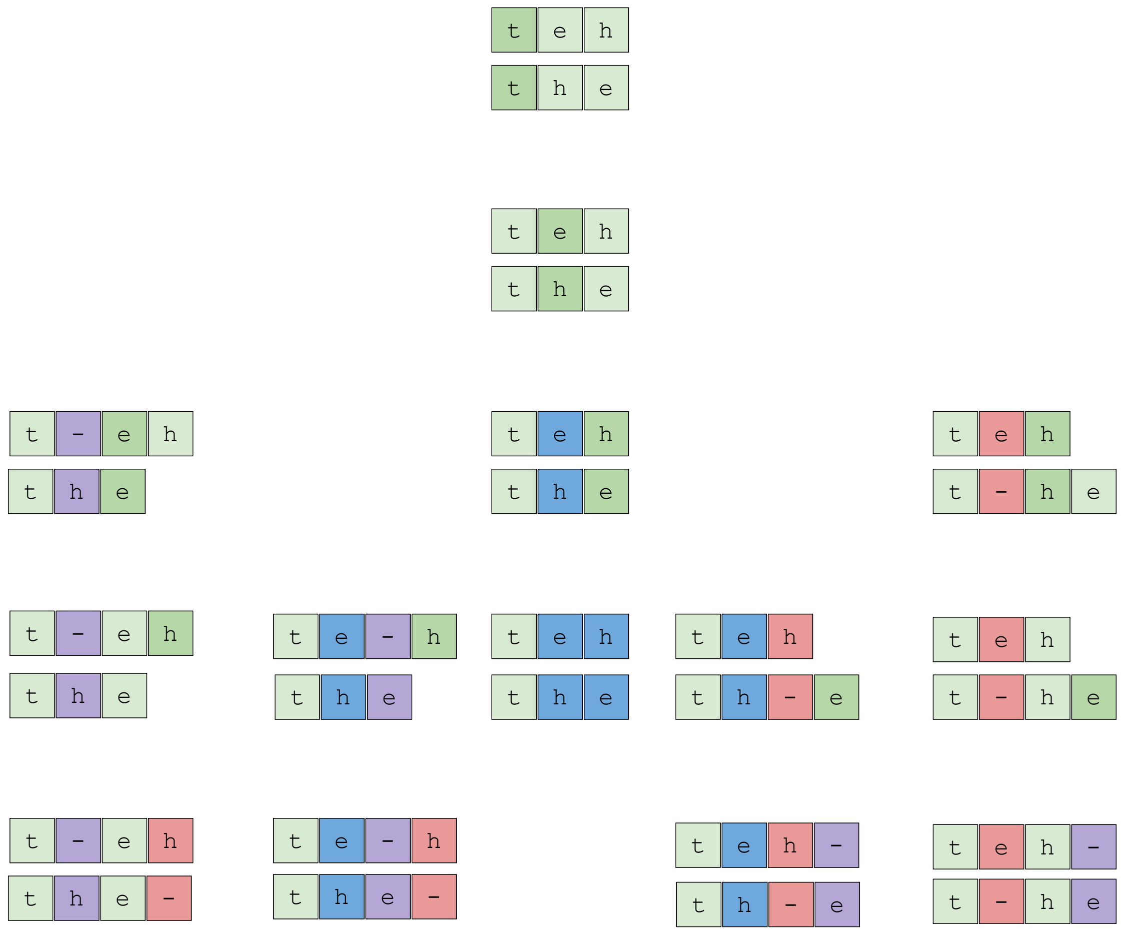
1 of 4



The subproblem of 'h' remaining in the first string and nothing in second, occurs twice.



The subproblem of 'e' remaining in the second string and nothing in first, occurs twice.



Let's look at recomputation due to repeating subproblems

4 of 4

So, this shows that we can benefit from memoization. Let’s look at the memoized version of this algorithm.

```
def editDistanceRecurse(str1, str2, i, j, memo):
    if i == len(str1):  # base case of reaching the end of str1
        return len(str2) - j

    if j == len(str2):  # base case of reaching the end of str2
        return len(str1) - i

    if (i,j) in memo:
        return memo[(i,j)]

    if str1[i] == str2[j]:  # if the characters match, we move ahead
        memo[(i,j)] = editDistanceRecurse(str1, str2, i+1, j+1, memo)
        return memo[(i,j)]
    # if characters don't match
```

```
    # if characters don't match
    memo[(i,j)] = 1 + min(editDistanceRecurse(str1, str2, i+1, j+1, memo),   # we can change chara
                          editDistanceRecurse(str1, str2, i, j+1, memo),     # we can have an inse
                          editDistanceRecurse(str1, str2, i+1, j, memo))     # we can have a delet

    return memo[(i,j)]

def editDistance(str1, str2):
    memo = {}
    return editDistanceRecurse(str1, str2, 0, 0, memo)

print(editDistance("teh", "the"))
```

Explanation

We use the tuple of `i` and `j` for indexing `memo` since this tuple can uniquely identify a subproblem. The rest of the idea is quite similar to what we have already seen in this course. We look in the `memo` for the result before evaluating a subproblem and store the result in the `memo` after evaluation.

Time and space complexity

Think in terms of how many unique subproblems we need to evaluate when our keyspace is mapped in terms of a tuple of `i` and `j`. Where `i` goes from 0 to n, whereas `j` goes from 0 to m. Thus, they keyspace mapped by their tuple would be of size $n \times m$, where n and m are sizes of the strings. Hence our time complexity would be **O(nm)**. Similarly, all the results will need space in order of **O(nm)** as well.

Solution 3: Bottom-up dynamic programming

```
def editDistance(str1, str2):
    n = len(str1)
    m = len(str2)
    # dp table of size nxm
    dp = [[0 for j in range(m+1)] for i in range(n+1)]

    # filling up dp
    for i in range(n+1):
        for j in range(m+1):
            if i == 0:          # base case of running out of str1
                dp[i][j] = j
            elif j == 0:         # base case of running out of str2
                dp[i][j] = i
            elif str1[i-1] == str2[j-1]:    # case when both characters match
                dp[i][j] = dp[i-1][j-1]
            else:               # case of mismatch
                dp[i][j] = 1 + min(dp[i-1][j-1],    # change character
                                   dp[i][j-1],      # insert i-th character
```

```
                                      dp[i-1][j])       # delete i-th character
    return dp[n][m]


print(editDistance("teh", "the"))
```

Explanation

We have already seen how we can construct the optimal answer to this problem using just three subproblems. In this solution, we start building up to our solution by solving subproblems first. We have `dp` table of size `(n+1)x(m+1)`, and we iterate over it to fill it. For the base case, when we have one of the strings empty, we know the cost for such a subproblem is equal to the length of the non-empty string (*lines 10-13*). For the general case, if our characters match, our answer is equal to the answer of the subproblem in the last diagonal (`dp[i-1][j-1]`), i.e., the substrings without the last character for both the string (*lines 14-15*). If the character's match, we have to pick the minimum from the three subproblems we discussed earlier, i.e., `dp[i-1][j-1]`, `dp[i][j-1]` and `dp[i-1][j]` (*lines 17-19*).

Let's look at a dry run of this algorithm.

editDistance("teh", "the")

editDistance("teh", "the")

1 of 22

			t	e	h
		0	1	2	3
	0				
t	1				
h	2				
e	3				

Let's make dp table of size 4x4

2 of 22

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1			
h	2	2			
e	3	3			

For base case, we will have first row and column filled with he corresponding values of j and i respectively

3 of 22

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1			
h	2	2			
e	3	3			

Let's start filling the dp table, starting with 1,1

4 of 22

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0		
h	2	2			
e	3	3			

Since the characters match we will replicate value from previous diagonal

5 of 22

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0		
h	2	2			
e	3	3			

moving on

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	
h	2	2			
e	3	3			

Since the characters do not match, we take the minimum from the highlighted three positions, and add 1 to it

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	
h	2	2			
e	3	3			

moving on

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2			
e	3	3			

Since the characters do not match, we take the minimum from the highlighted three positions, and add 1 to it

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2			
e	3	3			

moving on

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1		
e	3	3			

Since the characters do not match, we take the minimum from the highlighted three positions, and add 1 to it

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1		
e	3	3			

moving on

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	
e	3	3			

Since the characters do not match, we take the minimum from the highlighted three positions, and add 1 to it

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	
e	3	3			

moving on

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	1
e	3	3			

Since the characters match we will replicate value from previous diagonal

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	1
e	3	3			

moving on

16 of 22

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	1
e	3	3	2		

Since the characters do not match, we take the minimum from the highlighted three positions, and add 1 to it

17 of 22

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	1
e	3	3	2		

moving on

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	1
e	3	3	2	1	

Since the characters match we will replicate value from previous diagonal

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	1
e	3	3	2	1	

moving on

20 of 22

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	1
e	3	3	2	1	2

Since the characters do not match, we take the minimum from the highlighted three positions, and add 1 to it

21 of 22

			t	e	h
		0	1	2	3
	0	0	1	2	3
t	1	1	0	1	2
h	2	2	1	1	1
e	3	3	2	1	2

Thus our answer is 2

22 of 22

Time and space complexity

As apparent, we only need to fill a `dp` table of size $n \times m$ where n and m are the sizes of strings respectively. This entails a time and space complexity of **O(nm)**.

Solution 4: Space optimized bottom-up dynamic programming

As we can see in the above visualization, for filling any row, we only require the row before it. This means instead of saving a complete 2-d `dp` table we can just keep a list for the previous row.

```
def editDistance(str1, str2):
    n = len(str1)
    m = len(str2)
    # dp table of size n, stores a row at a time, for base case filled as [0,1,2..]
    dp = [i for i in range(n+1)]

    # filling up dp
    for j in range(1,m+1):
        thisrow = [0 for i in range(n+1)]
```

```
        thisrow = [0 for i in range(n+1)]
        for i in range(n+1):
            if i == 0:                      # base case of running out of str1
                thisrow[i] = j
            elif str1[i-1] == str2[j-1]:    # case when both characters match
                thisrow[i] = dp[i-1]
            else:                                # case of mismatch
                thisrow[i] = 1 + min(dp[i-1],    # change character
                                     dp[i],      # insert i-th character
                                     thisrow[i-1])      # delete i-th character
        dp = thisrow
    return dp[n]

print(editDistance("teh", "the"))
```

Explanation

The idea is that we use the last row stored in `dp` to fill the current row, i.e., `thisrow`. And at the end of each iteration, we update `dp` to be equal to `thisrow`, to be used in the next iteration.

Time and space complexity

The time complexity stays the same, i.e., **O(nm)** because we are still evaluating all results as before, but the space complexity would become **O(n)**.

That was it! In the next lesson, we will conclude this course.