Lecture Notes

Dynamic Programming

Welcome to the session on 'Dynamic Programming'.

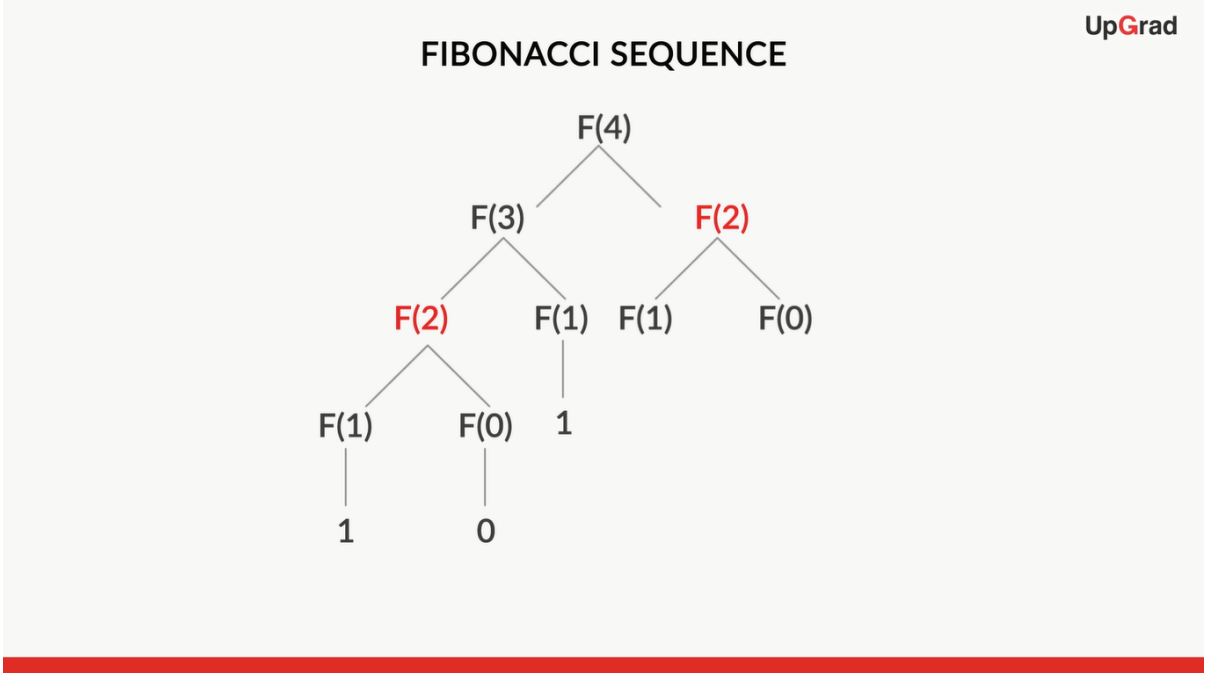
In the last module, you learnt about algorithmic analysis, and 'divide and conquer' algorithms. You also solved some problems using recursions. The dynamic programming technique is applicable to optimisation problems, where you need to find the best solution out of many possible solutions.

In general, the characteristics of a problem where you could apply dynamic programming can be summarised into the steps—

- A problem should be broken down into overlapping subproblems.
- Each subproblem must have a recursive solution.
- Each subproblem must be solved exactly once.
- The solutions to the subproblems should be combined to get the overall solution for the given problem.

Let's start with the Fibonacci Sequence example. You'd notice that this problem satisfies all the above-mentioned properties of dynamic programming.

You are familiar with the recursive solution for the Fibonacci Sequence — F(i) = F(i-1) + F(i-2). In this solution, if you were to find the 4^{th} number in the sequence, your recursive call would look like this —



The subproblem F(2) is calculated twice, which takes up extra computational time. A better way to do this would be to store the value of F(2) when it is calculated, and then use the same stored value when it is required the next time.

So, here is where dynamic programming comes in.

In the dynamic programming solution, you would calculate F(i) and store its value in an array.

0	1	1	2	3
F[0]	F[1]	F[2]	F[3]	F[4]

You'd call the stored value only when required. For example, in the array above, once F[2] is calculated, the value is stored in the array. Now, it can be called when required. Since this avoids repetitive computations of subproblems (e.g. F(2)), it saves time.

A dynamic solution for the Fibonacci Sequence requires the following steps:

- **Defining the subproblem** (or identifying the subproblem in words): F[i] is the subproblem in this case, and the value of F[i] represents the i^{th} Fibonacci number.
- **Writing down the recurrence that relates to subproblems**: The recurrence relation in this problem is — F[i] = F[i-1] + F[i-2].
- **Recognising and solving the base cases**: The base cases in Fibonacci Sequence are — F[0] = 0 and F[1] = 1.
- **Solving the subproblems and storing the solutions in a table**: The solutions to the subproblems are stored in an array. F[n] would store the number at the n^{th} index in the Fibonacci Sequence.

Remember that each dynamic programming problem can be solved using the steps above.

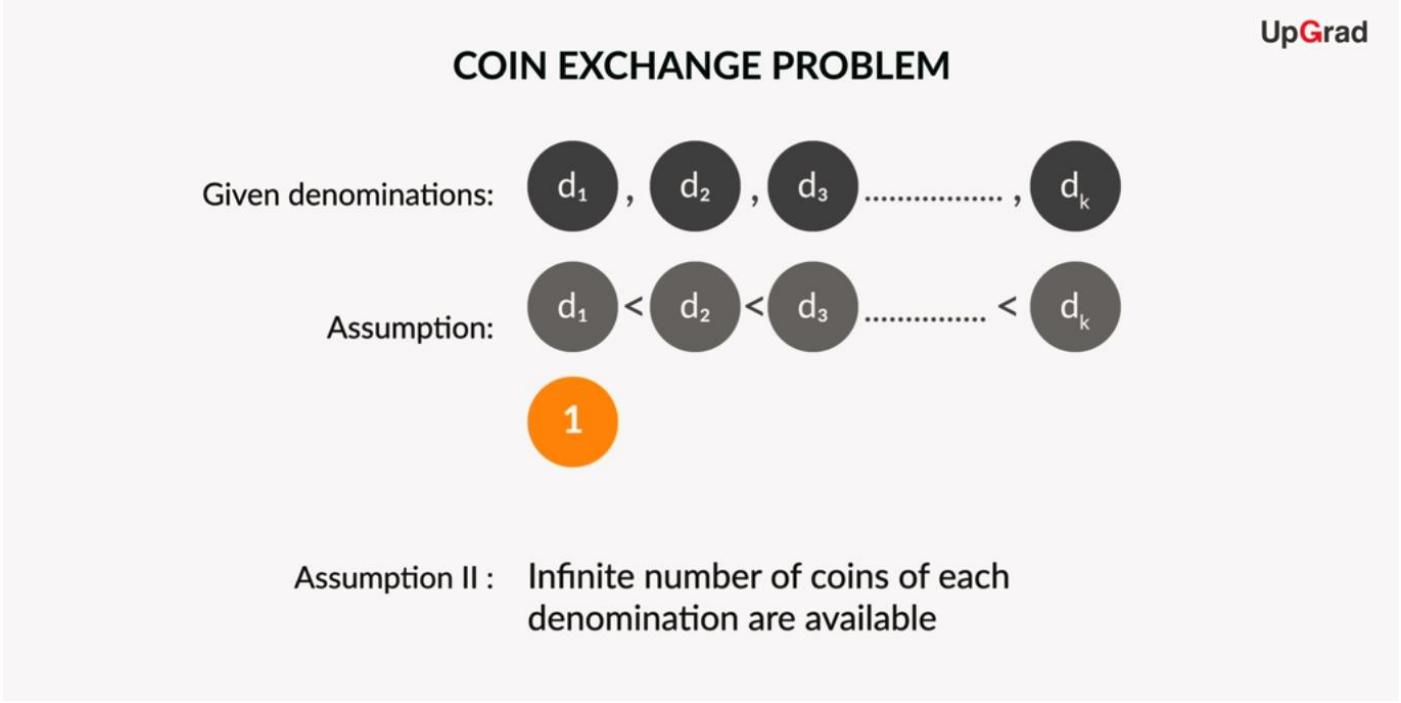
The Coin Exchange Problem

In this lesson, you will revise the Coin Exchange problem. The challenge involved in this problem is that given an amount, and a set of choices for denominations, you have to find the minimum number of coins required to add up to that amount.

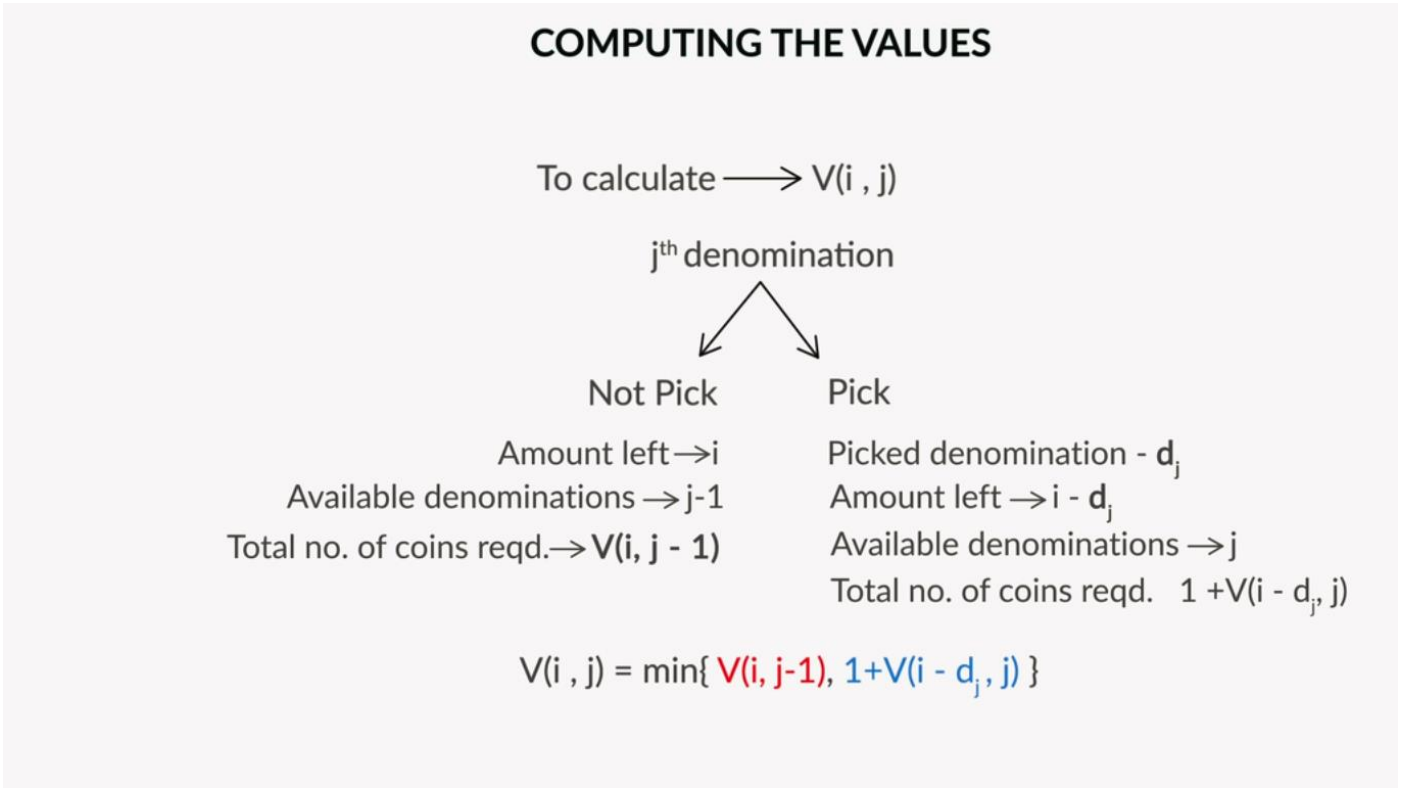
So, you defined a state as V(i, j), where V(i, j) represents the minimum number of coins required to pay i amount using the first j coins.

So, let's say that you have some coins of values 1, 2, 5, and 10 and you have to find out the minimum number of coins required to pay INR 14. You will represent the problem as V(14, 4).

At this stage, you made some assumptions as shown here:



You know that when you have to pay an amount, you have two options — one where you can pick the coins, and another where you can choose to reject the coins.



Now, you have seen the base cases. To fill the table, you need some values. These values are called base cases. First, you start with the problem where you have only one coin, and you have to pay ‘i’ amount. Since there is only one way to pay ‘i' using a coin of value 1, i.e. paying i coins of value 1, you must also know that there is no way to pay INR 0. Thus, no matter what coin value you have, the value will remain 0.

Thus, you defined that V(0, j) = 0 and V(i, 1) = i.

Then, using the information above, you started filling the cells column-wise from V(1, 1) up to V(n, k). Here, V(n, k) was your final solution.

So, let's say you were to pay INR 14 using three denominations – 1, 7, and 10. The table containing the solution would look like this:

j →

d_j	1	7	10
i/j	1	2	3
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	1	1
8	8	2	2
9	9	3	3
10	10	4	1
11	11	5	2
12	12	6	3
13	13	7	4
14	14	2	2

i ↓

Here are the steps you followed to get the solution to the problem above using dynamic programming:

- **Define subproblems**: This appears easy, but is a significantly important step. You need to define the subproblems in words before you can begin to solve the problem. For example, in the coin exchange problem, you clearly defined a subproblem as V(i, j), where i was the amount to be paid, and j was the number of denominations used. If you had defined the problem using any other approach, the solution could have been different.
- **Write down the recurrence that relates to subproblems**: For example, in the coin exchange problem, you defined a recurrence relation where V(i, j) = 1 + **V**(i - d_j , j), V(i, j-1).
- **Recognise and solve the base cases**: For example, in the coin exchange problem, you defined base cases where V(i, 1) = i and V(0, j) = 0. You could use this base case to solve the rest of the subproblems.
- **Store the results of the subproblems in a table**: For example, in the coin exchange problem, you created a 2D array where you could store the solutions to all the subproblems.

The pseudocode for the coin exchange problem looked like this:

```
Set V(i, 1) = i
Set V(0, j) = 0

FOR i = 1 -> n
 FOR j = 2 -> k
   IF ( i >= d_j AND V(i - d_j , j) + 1 < V(i , j - 1)) THEN
     V(i, j) = V(i - d_j , j) + 1
   ELSE
     V(i, j) = V(i , j - 1)
   END IF
 END FOR
END FOR
```

Now that you know how to solve the problem using the dynamic programming approach, you are also required to find out which coins were used. For this exercise, you can get the desired results using backtracking.

The algorithm for backtracking is simple, but it relies on the recursive logic that you created to solve the problem.

$$V(i, j) = \min\{ V(i, j-1)\ \ V(i - d_j, j) + 1 \}$$

IF $V(i, j) == V(i, j-1)$
$\Longrightarrow$ Coin j not used

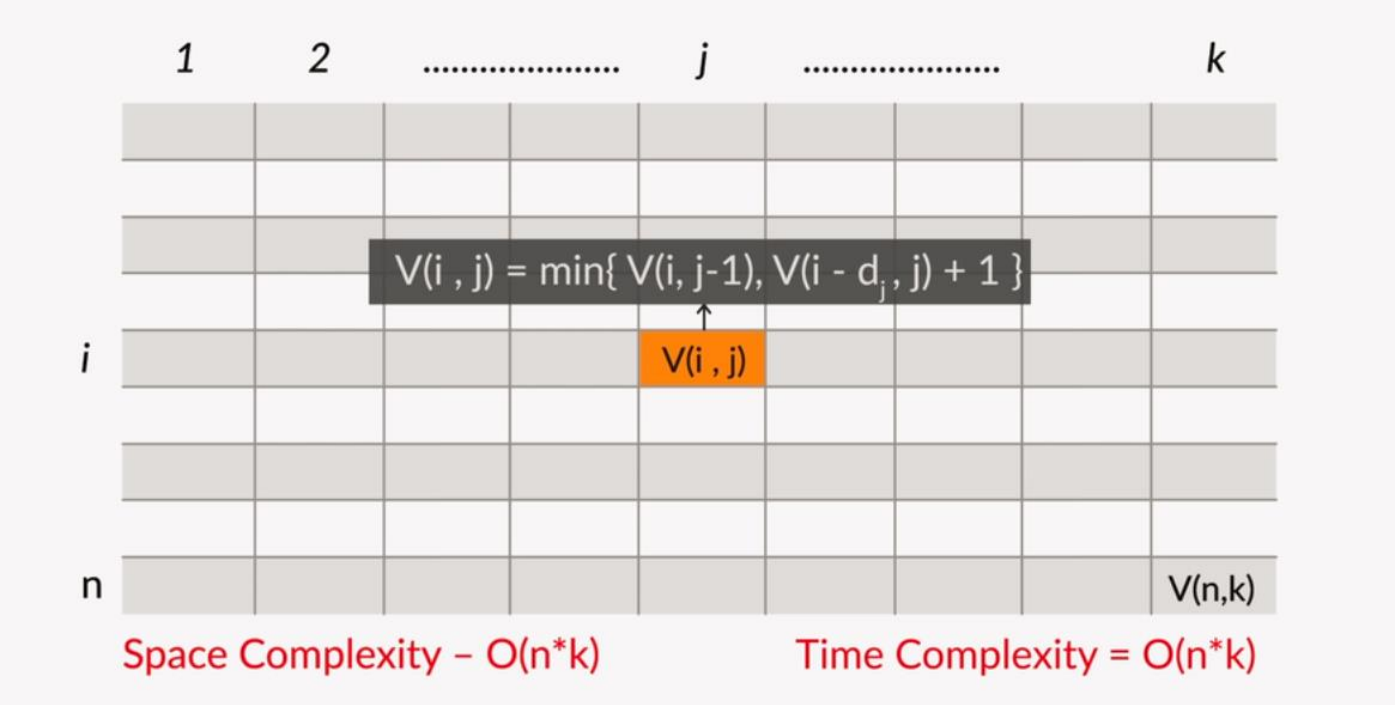
IF $V(i, j) \neq V(i, j-1)$
$\rightarrow V(i, j) == 1 + V(i - d_j, j)$
$\Longrightarrow$ Coin j used

Now, to implement the backtracking algorithm, you used the following pseudocode:

```
FOR i = n -> 1
 FOR j = k -> 1
   WHILE ( j > 0 )
     IF V(i, j) == V(i, j - 1) THEN  // have not used jth coin
       j = j - 1
     ELSE   // have used jth coin
       Print j
       i = i - d_j
     END IF
   END WHILE
 END FOR
END FOR
```

You must have noticed that the solution above utilised a 2D array.

So, the space and time complexity of the solution above can be visualised using this table:



Once you have seen this solution, you'd realise that not always would you require backtracking. If you only wanted to know the minimum number of coins that are required and not the values of the coins used, you can save some space. In this case, you'd only need a 1D array to solve the coin problem.

Here are the steps you need follow to reach the solution:

1. Create a 1D array T[i] to store your results. T[i] indicates the minimum number of coins required to pay amount i.
2. The first case was when you could use only one coin, i.e. d_1. In this case, you could pay i amount using i number of coins of value 1. Thus, T[i] = i. So, if you were considering a problem where you were to find the minimum number of coins required to pay INR 8 using coins of 1, 2, and 3, then here is how the solutions array would look like. Here, T[i] stores the minimum number of coins required to pay INR **i** using only d_1 denomination.

0	1	2	3	4	5	6	7	8
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]

3. Then, you increased the number of denominations to two. So, now you can use coins of d_1 and d_2. You can then check if using two denominations will bring down the number of coins to pay the amount.

- It is established that you can pick a coin only when its value is less than the amount to be paid, i.e. dj <= i. Therefore, if dj > i, you won't pick a coin, and the value of T[i] will remain as it is.
- Also, it makes sense to pick a new coin if choosing this coin reduces the total number of coins that need to be used. Thus, you used the formula T[i - dj] + 1 < T[i]. If this condition was true, then it made sense to pick a coin of dj value, since it would reduce the number of coins required to pay amount i.

Here is how the above array would look like at this stage. Now, T[i] represents the minimum number of coins required to pay amount i using the denominations d_1 and d_2.

0	1	1	2	2	3	3	4	4
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]

4. In the next step, you increased the number of denominations by one and updated the values of T[i], as was done in step 3.

After successfully executing the above step, you can use all the three denominations, i.e. d_1, d_2, and d_3. Thus, the array now looks like the following:

0	1	1	1	2	2	2	3	3
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]

The pseudocode for the code is as follows:

```
FOR j = 1 AND i = 0 -> n
  T[i] = i
END FOR

FOR j = 2 -> k
  FOR i = 1 -> n
     IF ( (i >=  d_j)  AND (T[i - d_j] + 1  < T[i]) THEN
       T[i]= 1 + T[i - d_j]
     ELSE
       Do Nothing // older value of T[i] retained
                  // denomination d_j not used
     END IF
 END FOR
END FOR
```

The 0-1 Knapsack Problem

The 0-1 knapsack challenge states that given a knapsack and some items, each with its own weight and value, you must fill the knapsack with the items that maximise the total value in the knapsack. The constraint is that the knapsack can hold only up to a certain weight.

Assume that you were given a knapsack that can hold up to 8 kg, and you were given this table that contains the details about the items:

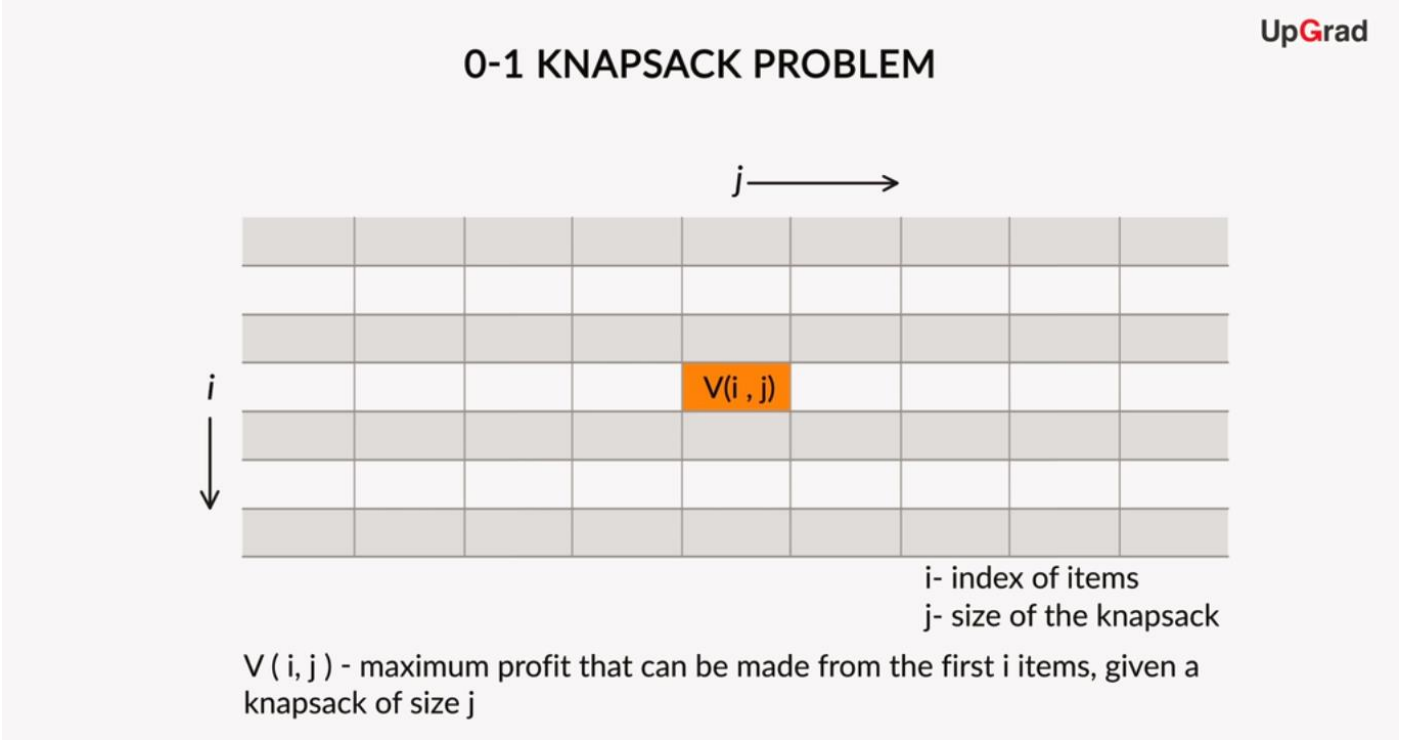
Item number	Weight	Value
1	2 kg	$12
2	1 kg	$10
3	3 kg	$21
4	2 kg	$15

You can only have one quantity of each item. So, what you basically have to do is either pick an item or skip an item.

As in every DP (dynamic programming) problem, you were required to carry out the following steps:

- Define the subproblems
- Write down the recurrence that relates to the subproblems
- Recognise and solve the base cases
- Decide what you wish to store in the table and then fill the table with results.

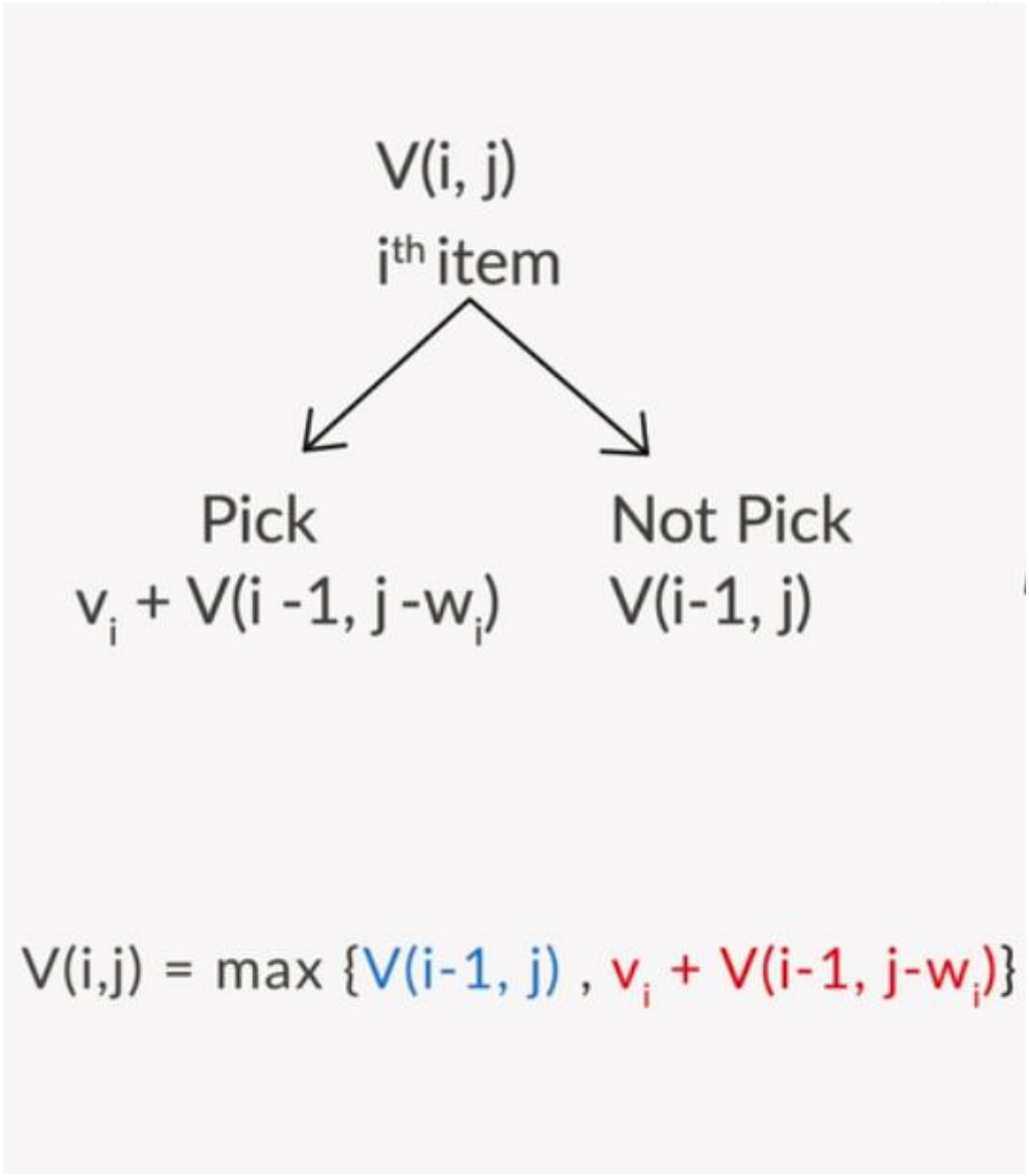
So, you defined a subproblem V(i, j) and came to the conclusion that you'd need a 2D array to store the results of your subproblems.



The next step was to define the base case, which was defined as in the following:

$$V(1, j) = \begin{cases} v_1 & j \geq w_1 \\ 0 & j < w_1 \end{cases}$$

Then, you defined the recursive relation:


V(i, j)
ith item
Pick
vi + V(i -1, j -wi)
Not Pick
V(i-1, j)
V(i,j) = max {V(i-1, j) , vi + V(i-1, j-wi)}

Then, using the recursive logic, you wrote a pseudocode that could solve this problem.

UpGrad

PSEUDOCODE

```
FOR i = 1 -> n
 FOR j = 1 -> W
   IF (w_i <= j AND v_i + V(i-1, j-w_i) > V(i-1, j)) THEN
     V(i, j) = v_i + V(i-1, j-w_i)
   ELSE
     V(i, j) = V(i-1, j)
   END IF
 END FOR
END FOR
```

$V(i , j) = \max\{ V(i-1, j), v_i+V(i - 1 , j-w_i) \}$

IF $w_i <= j$
→Can pick an item
→$V(i , j) = \max\{ V(i-1, j), v_i+V(i - 1 , j-w_i) \}$

Item Picked when:

$v_i + V(i-1, j-w_i) > V(i-1,j)$

IF $w_i > j$
→Can't pick an item
→$V(i , j) = V(i-1, j)$

i - index of item being considered
w_i - weight of the item being considered
v_i - value of the item being considered
j - Size of the knapsack being considered

Using the pseudocode above, you filled the table.

So, in a case where you were given the four items and you had to fill a knapsack that can carry a weight of 8 kg, you filled the following table.

Then, you framed the logic for backtracking based on the recursive logic you had used to fill the table.

size of knapsack j →

no. of items ↓ i

	1	2	3	4	5	6	7	8
1	0	12	12	12	12	12	12	12
2	10	12	22	22	22	22	22	22
3	10	12	22	31	33	43	43	43
4	10	15	25	31	37	46	48	58

UpGrad

UpGrad

BACKTRACKING

To check ⟶ **IF** V(i, j) == V(i-1, j)

V(i, j)

i^{th} item

Pick

$v_i + V(i-1, j-w_i)$

Not Pick

V(i-1, j)

IF V(i, j) ≠ V(i-1, j)

i = i - 1

$j = j - w_i$

IF V(i, j) == V(i-1, j)

i = i - 1

Then, you wrote the following pseudocode for backtracking and further used it to write the code:

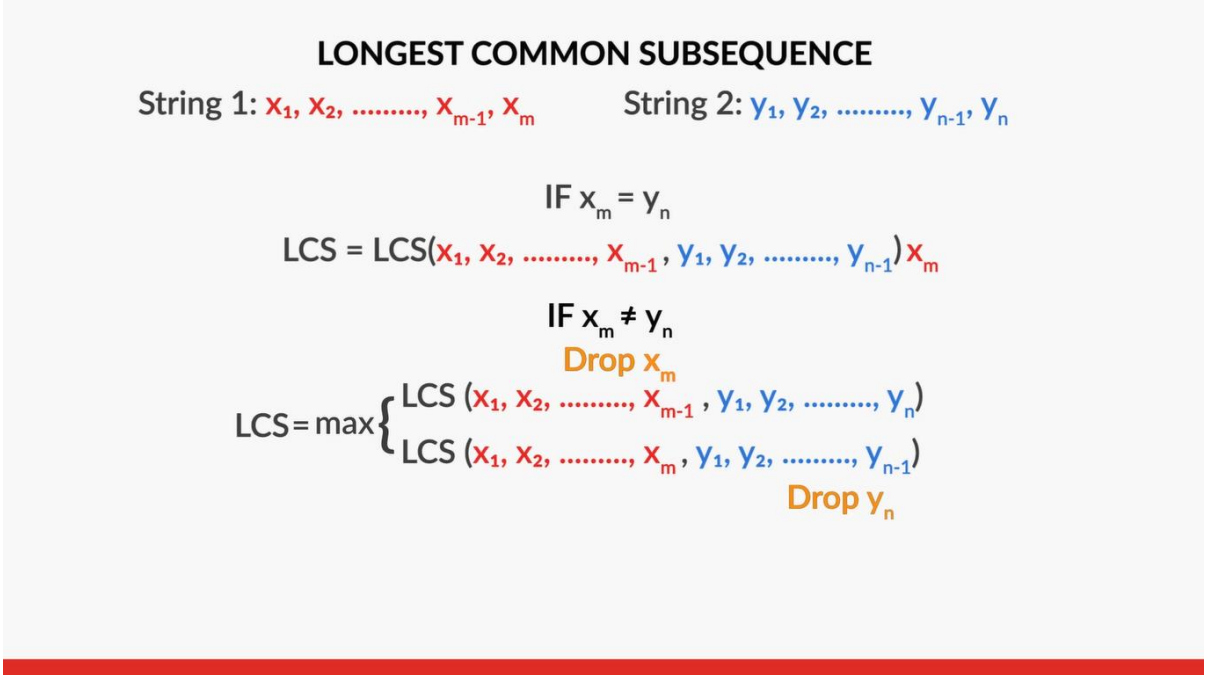
```
Set i = n
Set j = W
WHILE (i > 0 && j > 0)
  IF V(i, j) = V(i-1, j) THEN
    i = i-1
  ELSE
    Print i
    i = i-1
    j = j-w_i
  END IF
END WHILE
```

Longest Common Subsequence

Before moving forward, recall the difference between a substring and a subsequence. You will remember that a substring is part of a string. E.g. in “Developer”, “Dev” and “lop” are substrings. Notice that the characters in the substrings have the same sequence as the string.

A subsequence on the other hand is not necessarily characters in consecutive sequence, but they follow some order. For example, E L R would be a subsequence — like E, L, R — and it would occur in the same sequence as in the string “Developer”.

Let’s quickly revise what you learnt. So, you had two strings — string 1 and string 2 — and you wanted to derive the longest common subsequence. Therefore, you found the logic for finding LCS, which is as follows:



You defined the subproblem as L(i, j), which stands for the length of the LCS between two strings of length i and j.

Here's how you defined the base case:

$$L(1, j) = \begin{cases} 1 & y_k = x_1 \quad k \le j \\ 0 & \text{Other cases} \end{cases}$$

$$L(i, 1) = \begin{cases} 1 & x_k = y_1 \quad k \le i \\ 0 & \text{Other cases} \end{cases}$$

Then, you defined the recursive logic:

$$\text{IF } x_i = y_j$$

$$L(i, j) = 1 + L(i-1, j-1)$$

$$\text{IF } x_i \neq y_j$$

$$L(i, j) = \max \begin{cases} L(i-1, j) \\ L(i, j-1) \end{cases}$$

Then, based on this logic, you filled the table and that gave you the desired result. Let's see how you filled the table in the following:

UpGrad

TABULATION

$x_1, x_2, \ldots\ldots x_i$

$y_1, y_2, \ldots\ldots y_j$

IF $x_i = y_j$

$L(i, j) = 1 + L(i-1, j-1)$

IF $x_i \neq y_j$

$$L(i, j) = \max \begin{cases} L(i-1, j) \\ L(i, j-1) \end{cases}$$

i \ j		C	B	A	B	A	C	A
	A	0	0	1	1	1	1	1
	B	0	1	1	2	2	2	2
	C	1	1	1	2	2	3	3
	A	1	1	2	2	3	3	4
	B	1	2	2	3	3	3	4
	B	1	2	2	3	3	3	4
	A	1	2	3	3	4	4	4

The Edit Distance Problem

According to Wikipedia, **edit distance** is a way of quantifying how dissimilar two strings are to one another. And this is done by counting the minimum number of operations required to transform one string into the other.

Now, if you have to transform one string into another, you generally need to consider three operations:

1. Insertion
2. Deletion
3. Replacement

If you want to transform “ishan” to “nishant”, the brute force approach would include —
i -> n, s -> i, h -> s, a -> h, n -> a. Then, you should insert n and t at the end of the string.
But, this approach would require seven operations, and that is clearly not the minimum.

The minimum number of operations required would be two. It is possible by inserting ‘n’ at the beginning, and ‘t’ at the end.

So, the challenge states that given two strings — *source* and *target* — you need to calculate the edit distance considering only three operations:

1. Insertion
2. Deletion
3. Replacement

The notations that you used were:

- source(i): This is a substring of source containing characters from index 0 to i
- Source[i]: This is the ith character of source

You learnt how you can transform source to target using either one of these four operations:

1. Insert last character of target, that is target[n-1], in the last position of source.
   editDistance(m, n) = 1 + editDistance(source(m), target(n-1))
   ➔ 1 is added because of the insertion operation.

2. Delete the last character, that is source[m-1], from source.
   editDistance(m, n) = 1 + editDistance(source(m-1), target(n))
   ➔ 1 is added because of the deletion operation.

3. Replace the last character of source, that is source[m-1], with the last character of target, that is target[n-1].
   editDistance(m, n) = 1 + editDistance(source(m-1), target(n-1))
   ➔ 1 is added because of the replace operation.

4. Do nothing since the last character of source is the same as the last character of target.
   editDistance(m, n) =0 + editDistance(source(m-1), target(n-1))
   ➔ No cost, because no operation was performed.

In either of these operations, the cost is one, except for the last one, where we do nothing and simply carry over the edit distance that we've seen so far.

In each of these cases, editDistance can be derived from the subproblems that are the editDistance of the substrings (of the source string) and the target string.

There are four possible operations, but essentially there are only three choices to choose from at every step — insert, delete, and replace/do nothing. We can choose either replace or do nothing depending on whether or not the last characters are the same.

Each of these choices has 1 cost, except for the last one in which the cost could be 1 or 0, depending on whether we'd need to replace the last character or not. We can also define a cost function for it:

cost = if (source[m-1] == target[n-1]), then 0 else 1;

Now, we have three choices at every step, so we choose the best among those, i.e. the one with the minimum cost (the minimum number of operations required to transform the source string to the target string).

Based on this information, you wrote the following recursive logic:

UpGrad

EDIT DISTANCE EQUATION

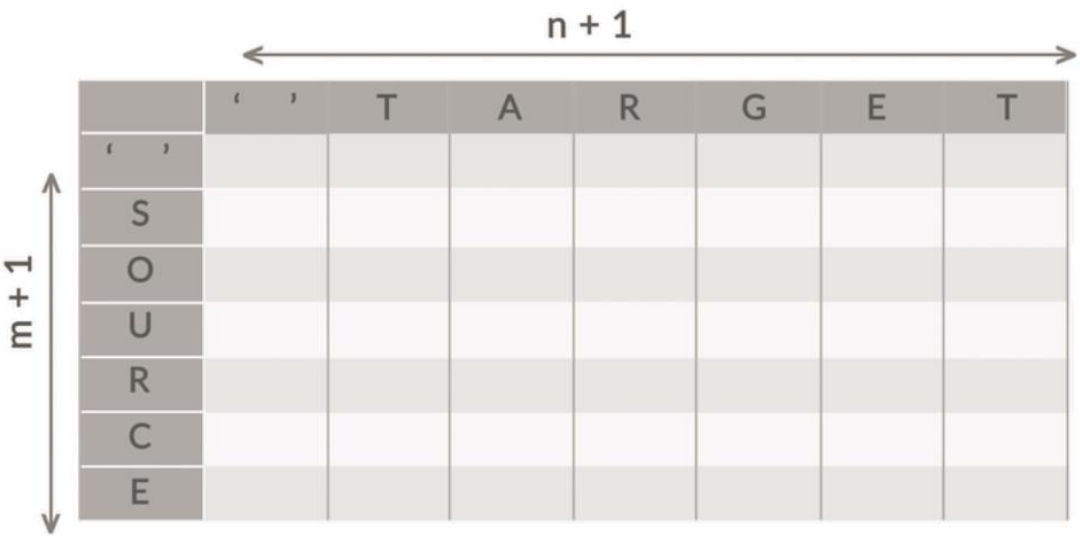
```
cost = (source[m-1] = target[n-1]) ? 0 : 1

                    //Insertion          1 + editDistance (m, n-1)
editDistance (m,n) = min  //Deletion           1 + editDistance (m-1, n)
                    //Replace/Do Nothing cost + editDistance (m-1, n-1)
```

To solve the problem using dynamic programming, you first need to construct a 2D matrix comprising edit distances of size (m+1)x(n+1), where m and n are the lengths of the source and target strings, respectively. You would call this matrix E.

UpGrad

TABULATION



E[i][j] = edit distance of converting source(i-1) to target(j-1).

m - length of source
n - length of target

In our matrix, E[i][j] will represent the edit distance of converting source(i-1) to target(j-1).

So, if source = “ankit” and target = “amit”, then E[2][3] will be the edit distance for converting the substring “ankit” with the characters at index 0 to 1(that is “an”) to the substring “amit” with the characters 0 to 2 (that is “ami”).

And the value of E[2][3] will be 2, because we will need a minimum of two operations to convert “an” to “ami”. Specifically, we will need —

1. One operation to convert “n” in “an” to “m” and
2. One operation to add “i” to “am”

Similarly, E[0][4] will be the edit distance for converting the base case of an empty string “” to “amit”, which will be 4.

As shown in the previous examples, you can follow a bottom-up approach to fill the matrix — from the very first cell to the final destination cell. Similarly, here you can start from E[0][0] and compute the edit distance value for each cell of the matrix until you reach E[m][n].

By filling out the value of each cell in the matrix, you would get the edit distance values for all the substrings (or more generally, the optimal substructure) in the source and target strings. Thus, you can find the edit distance between the two strings.

Then you identified the base cases:

- The first complete column indicates the base case, which shows that target is an empty string. So, in that case, the transformation will take place after deleting all the characters from the source string, and edit distance will be the length of the source string. Hence, FOR i = 0 to m, E[i][0] = i // all deletions.

- Similarly, the first complete row is also the base case and indicates that source is an empty string. So, in that case, the transformation will happen by inserting all the characters of target, and edit distance will be the target string length. For j = 0 to n, E[0][j] = j (for all insertions).

Based on this, you wrote the pseudocode below, which was used to fill the table.

```
FOR i = 0 -> m
   E[i][0] = i

FOR j = 0 -> n
 E[0][j] = j

FOR i = 1 -> m
   FOR i = j -> n
     IF source[i-1] == target[j-1] THEN
       cost = 0
     ELSE
       cost = 1
     END IF

   E[i][j] = min( 1+  E[i-1][j] ,1+ E[i][j-1], cost + E[i-1][j-1])
   END FOR
END FOR
Return E[m][n]
```

Then, you filled the table as shown here:

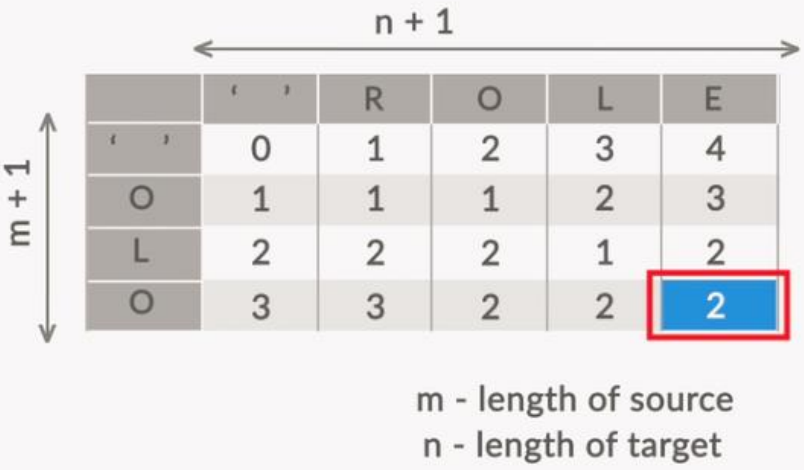
```
FOR i = 0 -> m
   E[i][0] = i

FOR j = 0 -> n
 E[0][j] = j

FOR i = 1 -> m
   FOR i = j -> n
     IF source(i-1) == target(j-1) THEN
       cost = 0
     ELSE
       cost = 1
     END IF

   E[i][j] = min( 1+  E[i-1][j] ,
            1+ E[i][j-1], cost + E[i-1][j-1])

   END FOR
```



Spell-Checker

You also learnt how to leverage the edit distance to create a spellchecker.

In the edit distance problem, you were allowed to consider three operations:

A. Insertion
B. Deletion
C. Replacement

In the spellchecker, you also introduced another operation called 'Transpose'. Let's recall what it was.

When a person types a message or a response, one of the most common typos he/she makes is the transposition error, where he/she swaps two consecutive characters. For example, many times you mistype "length" as "lenght". To rectify this mistake, you need to transpose the same characters again. In this case, to rectify "lenght", you need to swap the second last character 'h' with the last character 't'; then, you will get the correct word — "length".

If you have used the previous edit distance algorithm to calculate the edit distance between "length" and "lenght", the result would be 2. However, with the new transpose operation, the edit distance is reduced to 1.

So, now you have four operations:

A. Insertion
B. Deletion
C. Substitution
D. Transposition

To build a spellchecker, you need to have a collection of all the valid words, which you will use to check the validity of the input word. You will derive this collection from a text corpus, which serves as the basis of all valid words. So, if a word exists in this collection, then it is a valid word, and if a word doesn't appear in this collection, then it must be a misspelt word.

Google and Microsoft use a huge corpus of words in their databases to fill their spellcheckers, but in your spellchecker, you used the entire text from the book "Adventures of Sherlock Holmes", which was taken from the open source digital library — Project Gutenberg.

To understand what a dictionary is, you need to understand hashmaps.

A hashmap is a data structure that stores data (key, value) in pairs, and all the keys are unique, i.e. no key can be repeated inside the hashmap.

For example, consider this sentence:

"I went to the supermarket to buy some apples, but when I reached the supermarket, I found no apples."

If we build a dictionary (hashmap) from this text, it will look like this:

"i"	3
"apples"	2
"supermarket"	2
"the"	2
"found"	1
"went"	1
"to"	2
"buy"	1

“some”	1
“but”	1
“when”	1
“reached”	1
no	1

This was a small example, but the book you took to build the dictionary had many more words. When your program encounters an input word typed by the user, it checks if that word exists in the dictionary by constantly looking up on the hashmap. If the word does exist in the dictionary, it is assumed that the user typed the correct word; otherwise, the typed word is incorrect or misspelt. In this case, you need to pass the misspelt word to the spellchecker to get a list of all possible corrections. For this, a list to hold all the possible corrections is initialised, and then the program matches the misspelt word against all the suitable candidate words in the dictionary and computes their edit distances.

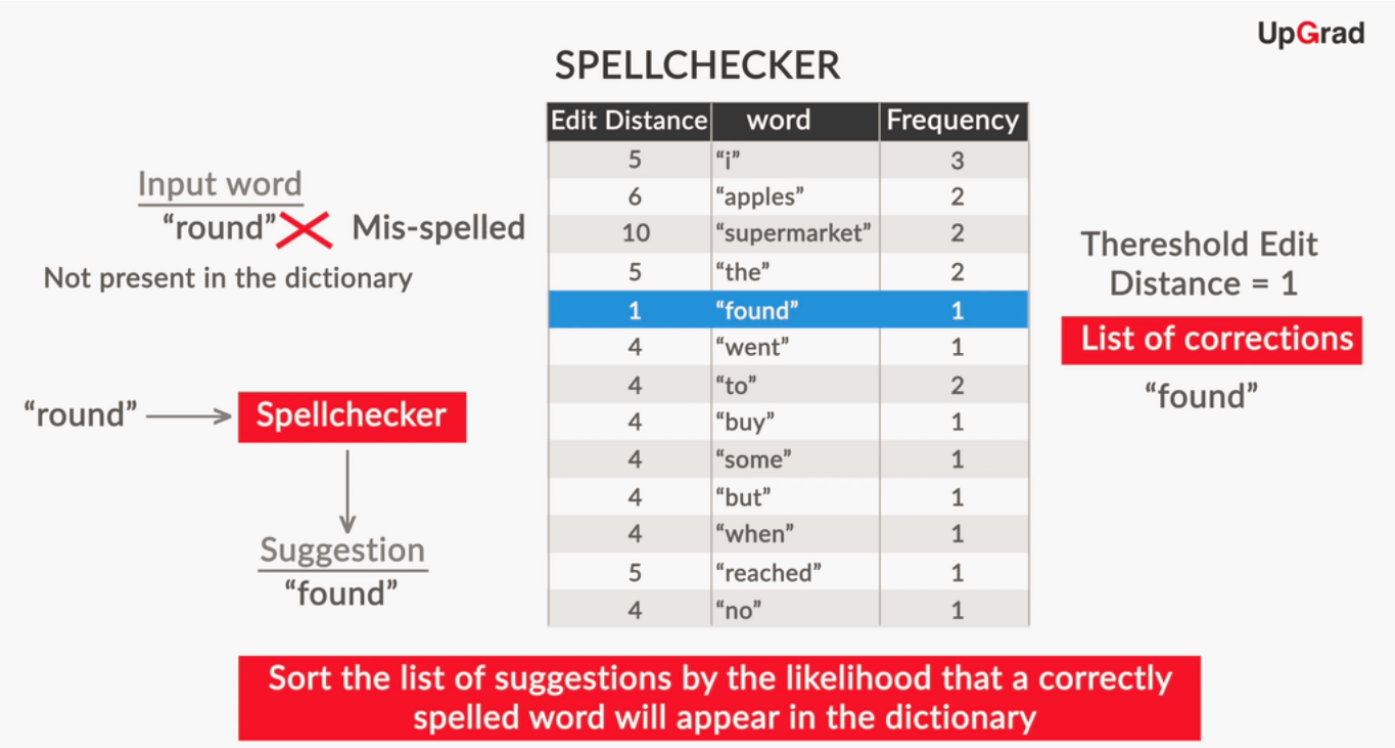
If the edit distance is less than or equal to your threshold value (say 1), then that dictionary word is considered as a probable candidate for the correct word and is put in the list of possible corrections.

The list of corrections might potentially have many suggestions. In the instance where it has more than five suggestions, it will be cumbersome for the user to find the most relevant word through the list. So, the list of possible corrections is sorted by the likelihood that a correctly spelt word will appear in the dictionary, the likelihood of a word is basically its frequency in the dictionary. The more the word appears in the dictionary, the more its probability becomes of being the correct word.

Thus, in the program, you took two valid assumptions:

1. The **difference** in the lengths between the correct word and the misspelt word cannot be more than your defined threshold. This will help you to do a quick first-level filter for candidate words.
2. The **editDistance** between the correct and misspelt words cannot be more than your defined threshold. This will pick your final correction candidates.

Finally, your spellchecker looks like this diagram:



Now, in your edit distance algorithm, you added one more operation, i.e. Transpose.

Remember that, in every iteration, you had a precondition for the 'do nothing' operation, where you checked if the last characters of both the strings were the same. Similarly, you had a precondition for the transpose operation too, which can only be done when there are at least two characters in both the source and the target strings. This operation also requires that the last character of source matches the second last character of target, and the last character of target matches the second last character of source.

If (i > 1 and j > 1 and source[i-1] == target[j-2] and source[i-2] == target[j-1]) is true —

➔ Only then then transpose operation becomes active.

You also wrote the pseudocode that includes the transpose operation.

```
FOR i = 0 -> m
   E[i][0] = i

FOR j = 0 -> n
 E[0][j] = j

FOR i = 1 -> m
   FOR i = j -> n
     cost = (source[i-1] == target[j-1]): 0 ? 1
     E[i][j] = min( 1+  E[i-1][j] ,1+ E[i][j-1], cost + E[i-1][j-1 )

     IF (i > 1  && j > 1
              && source[i - 1] == target[j - 2]
                   && source[i - 2] == target[j - 1]) THEN
       E[i][j] = Math.min(E[i][j], E[i - 2][j - 2] + 1); // for transpose
     END IF
  END FOR
END FOR
Return E[m][n]
```

Summary

In general, the characteristics of a problem where you can apply dynamic programming can be summarised into the following points:

- A problem should be broken down into overlapping subproblems.
- Each subproblem must have a recursive solution.
- Each subproblem must be solved exactly once.
- The solutions to the subproblems should be combined to get the solution to the given problem.

And to solve the problem, you need to carry out the following steps:

- Define the subproblem (or, identify the subproblem in words).
- Write down the recurrence that relates to the subproblems.
- Recognise and solve the base cases.
- Solve the subproblems and store the solutions in a table.