**Dynamic Programming**

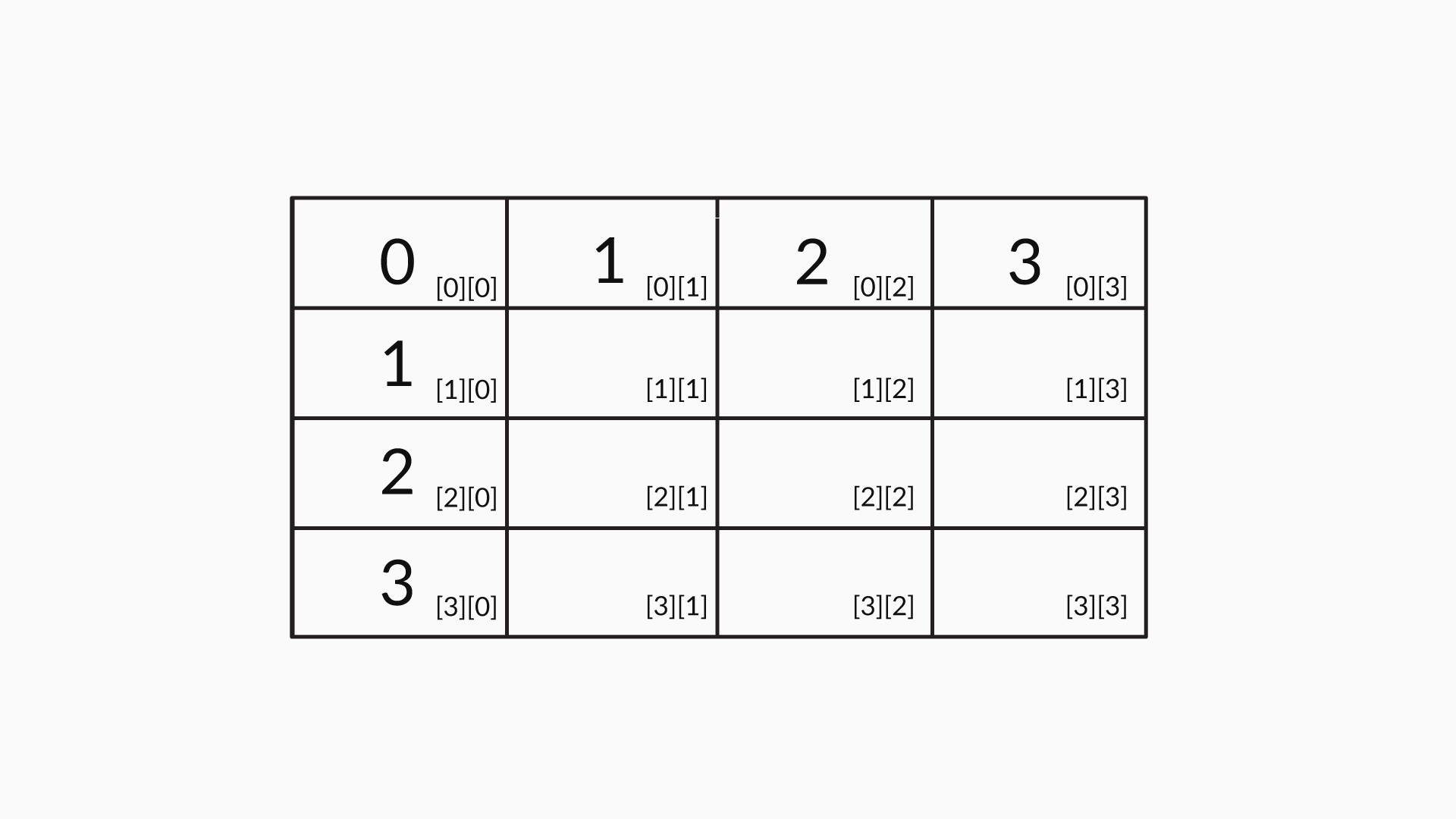
**Revision**

Before you proceed to learn about dynamic programming, let us recall some concepts that we discussed in the previous modules by attempting some questions based on them.

Suppose you are given this two-dimensional (2D) array.

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | 3 |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |

Here is another representation of the same array to help you visualise it better.



2D Array

Now, you need to fill the matrix such that each element is the maximum of its neighbours, i.e., the cell above it and the cell to its left. So, array[1][1] will be the maximum of array[1][0] and array[0][1], both of which happen to be 1 in this case. Similarly, array[1][2] will be equal to the maximum of array[1][1] and array[0][2].

Now, attempt this question for a better understanding of this logic.

#### Q1: 2D Array

The indices of the array given above are represented by i and j, which stand for a row and a column, respectively. What will be the recursive equation to fill this array?

**Hint:** Array(i, j) represents the value at the ith row and the jth column, i.e., array[i][j].

Ans: array(i, j) = max(array(i, j-1), array(i-1, j))

**✓ Correct**

**Feedback:**

If the value in a cell is array[i][j], then it means that cell is in the ith row and jth column. Now, the cell above this should be in the i-1th row, whereas the cell to its left should be in the j-1th column. Try to visualise this using a pen and paper. The value in the cell above will be array[i-1][j] and the value to the left of the cell will be array[i][j-1]. Since the value of array[i][j] should be the maximum of its two neighbours, array[i][j] should equal max(array(i, j-1), array(i-1, j)).

#### Q2: Assumptions

Suppose you are given coins of denominations d1,d2,d3,d4,d5,d6 for a coin exchange problem. Which of these assumptions can be made in this case? (**Note:** More than one option may be correct.)

Recall the concepts discussed in the video to answer the question.

Ans: d1d1 = 1

**✓ Correct**You missed this!

**Feedback:**

The denomination d1d1 will always be 1. This is essential because you can pay any amount using 1. So, suppose you have coins of values 2, 4 and 5. You cannot use these denominations to pay an amount of 1 or 3. If you had a denomination of 1, then you could have made these payments.



d1<d2<d3<d4<d5<d6d1<d2<d3<d4<d5<d6

**✓ Correct**

**Feedback:**

The denominations are in increasing order.



There are infinite coins of each denomination.

**✓ Correct**You missed this!

**Feedback:**

In the coin exchange problem, it is assumed that there are infinite coins of each denomination.

#### Q3: Minimum Number of Coins Required

Suppose you are given coins of denominations d1=1,d2=2,d3=6,d4=8,d5=10,d6=15, and you have to make a payment of 29. Considering a greedy algorithm picks the highest possible denominations first, what will be the coins picked by the algorithm?

Ans: d6,d5,d2,d2

**✓ Correct**

**Feedback:**

The greedy algorithm will pick 15 first and then pick 10, and then it will pay the remaining amount using two coins of denomination 2.

#### Q4: Optimal Way

Suppose you are given coins of denominations d1=1,d2=2,d3=6,d4=8,d5=10,d6=15. What will be the optimal way to make a payment of 29?

Ans: d6,d4,d3

**✓ Correct**

**Feedback:**

The optimal way to pay 29 will be using coins of denominations 15, 8 and 6.

#### Q5: Understanding the Subproblem Definition

Suppose you are given these six denominations: d1=1,d2=2,d3=6,d4=8,d5=10,d6=15. Considering you have to pay i = 17 using only the first three denominations, i.e., j = 3, how will you represent the minimum number of coins required to do this?

Ans: V(17, 3)

**✓ Correct**

**Feedback:**

V(i, j) represents the minimum number of coins required to pay i using the first j denominations. Therefore, V(17, 3) represents the minimum number of coins required to pay 17 using the first three denominations.

#### Q6: Defining the Subproblem

Suppose you are given these six denominations: d1=1,d2=2,d3=6,d4=8,d5=10,d6=15.  
What does V(14, 4) represent?

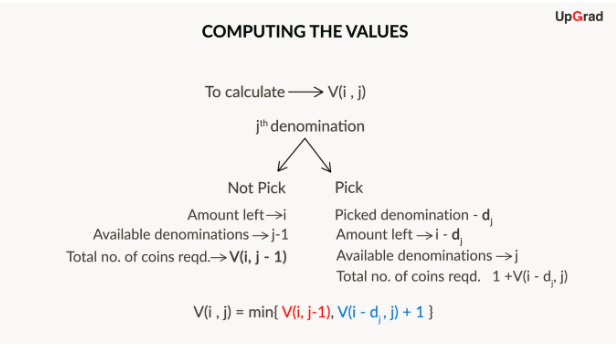
Ans: The minimum number of coins required to pay 14 using the first four denominations, i.e., d1,d2,d3,d4

**✓ Correct**

**Feedback:**

V(i, j) represents the minimum number of coins required to pay i using the first j denominations. Therefore, V(14, 4) represents the minimum number of coins required to pay 14 using the denominations d1,d2,d3,d4.

we derived a recursive equation for V(i, j), which can be used to fill the table.



Recursive Equation

#### Q7: Understanding the Recursive Logic

Fill in the blank.

In the coin exchange problem, V(i, j) represents the minimum number of coins required to pay i using the first j denominations.

Now, suppose you are given coins of denominations d1=1,d2=7,d3=10, and you are required to pay an amount of 9. If you are allowed to use only the first two coins, and you pick d2, i.e., the coin of value 7, then the minimum number of coins required to pay 9 can be represented as \_\_\_\_\_\_\_\_\_.

**Hint:** Refer to the formula for V(i, j) discussed in the video.

Ans: V(9, 2) = 1 + V(i - dj, j) = 1 + V(9 - 7, 2) = 1 + V(2, 2)

**✓ Correct**

**Feedback:**

V(9, 2) represents the minimum number of coins required to pay an amount of 9 using the first two denominations. This means V(9, 2) represents the minimum number of coins required to pay 9 using the first two denominations: d1= 1 and d2= 7. Since d2, which has a denomination of 7, has been picked, you are left with the amount 9 - 7 = 2, and you are still allowed to use the two denominations. So, the number of coins required to pay 2 using the two denominations is V(2, 2). However, since you have already picked a coin at this point, V(9, 2) can be written as 1 + V(2, 2). Or, it can be interpreted as 1 Coin + The minimum number of coins used for V(2, 2).

#### Q8: Understanding the Recursive Logic

In the coin exchange problem, you are given coins of denominations d1,d2, ...,dj, ...,dk.  
You know that V(i, j) represents the minimum number of coins required to pay i using the first j denominations. Considering you pick the denomination dj, what will be the minimum number of coins required to pay i if i is greater than the value of dj?

Ans: 1 + V(i - dj, j)

**✓ Correct**

**Feedback:**

V(i, j) represents the minimum number of coins required to pay i using the first j denominations, i.e., d1,d2,....,dj. Since dj has been picked, you can subtract dj from i. This leaves you with the amount i - dj to be paid, and you are still allowed to use the j denominations. So, the number of coins required to pay i - dj using these j denominations is V( i - dj, j). Since a coin of denomination dj has already been picked, V(i, j) can be written as 1 + V( i - dj, j).

#### Q9: Understanding the Recursive Logic

In the coin exchange problem, you are given coins of denominations d1,d2,...,dj,  
. ...,dk.   
You know that V(i, j) represents the minimum number of coins required to pay i using the first j denominations. Considering you choose not to pick the denomination dj, what will be the minimum number of coins required to pay i?

Ans: V(i, j - 1)

**✓ Correct**

**Feedback:**

Even if you choose to not pick the denomination dj, you still have to pay the amount i, and now you can use only j - 1 denominations to pay this amount. Therefore, V(i, j) will be written as V(i, j - 1).

you now know how to use dynamic programming logic to obtain the solution to the coin exchange problem.

The steps that you followed can be summarised as follows:

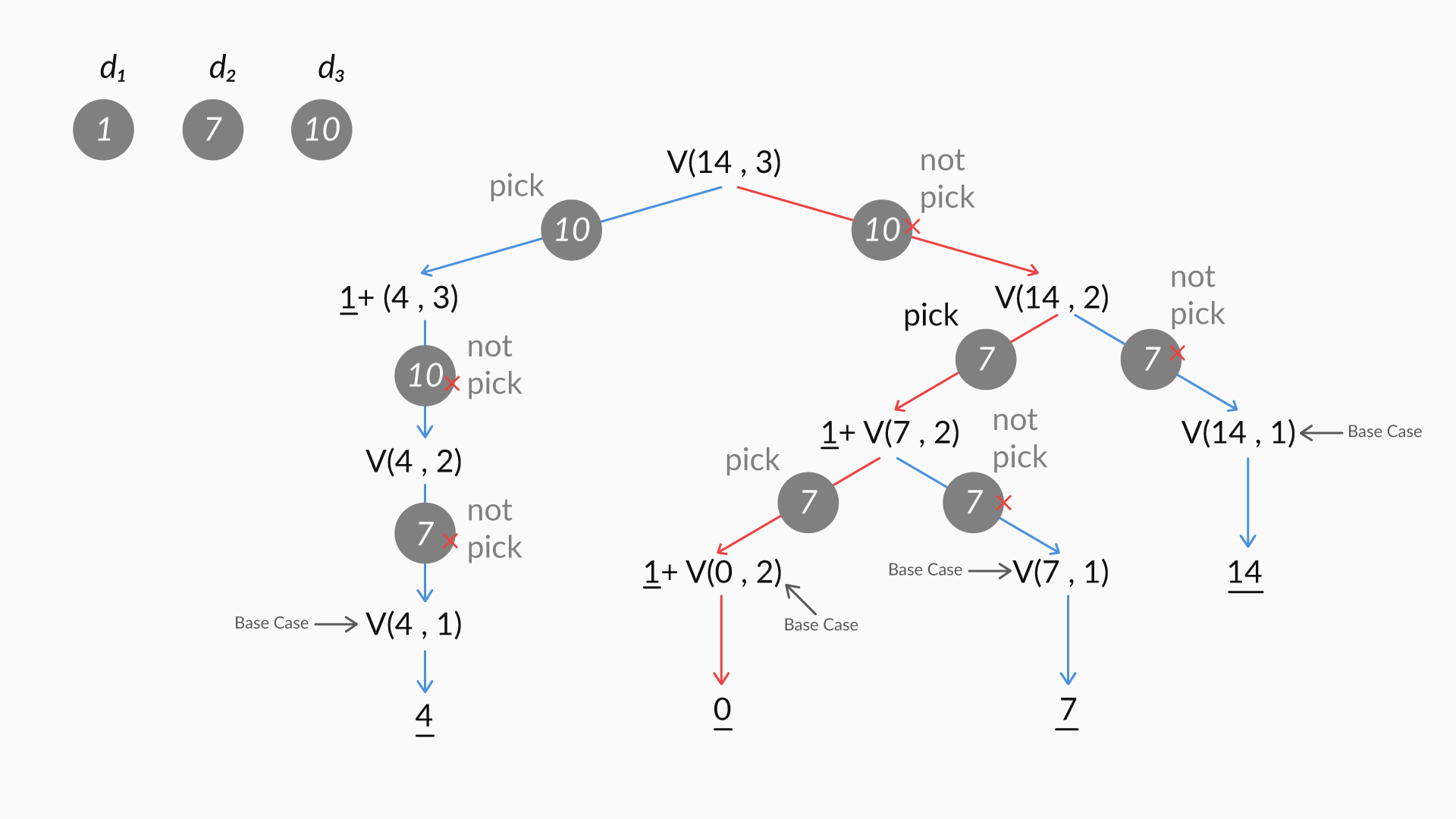
1. First, you defined the subproblem V(i, j), which represents the minimum number of coins required to pay i using j denominations.

2. Then you identified and solved the base cases:**V(0, j) = 0 and V(i, 1) = i**.

3. Next, you defined a recursive equation:**V(i, j) = min (1 + V(i - dj, j),  V(i, j - 1))**.

4. Finally, using the information above, you started filling the cells from V(1, 1) up to V(n, k) column-wise. You obtained V(n, k) as the final solution.

So, each V(i, j), or each subproblem, is computed and stored in the table. This value is then called when required for computing further values. If you still have doubts regarding why you calculated and stored the values for the subproblems when you only had to obtain the final solution, then you can take a look at the diagram given below. It indicates how the solution to the larger problem is dependent on the solutions to the subproblems.



Coin Exchange Problem

Also, dynamic programming is a much better approach than recursion when run-time efficiency is of paramount importance. This is because in dynamic programming, computed values can be stored in a table, and recalled and reused whenever required. However, recursion will discard computed values after they are used, and, therefore, the same value may need to be computed and re-computed multiple times.

If you had to summarise the solution, then you would need to do the following in dynamic programming:   
1. Define subproblems.  
2. Write down the recurrence relationship that relates to the subproblems.  
3. Recognise and solve the base cases.  
4. Store the results of the subproblems in a table. (Start with the base case and then fill the rest of the table using the recurrence relation that is defined.)

#### Q10: Coin Exchange Problem

In the coin exchange problem, you are given coins of denominations d1,d2, ...,dj, ...,dk. You know that V(i, j) represents the minimum number of coins required to pay i using the first j denominations. Which of these options is correct in this case?

(Recall the concept discussed in the video to answer the question)

Ans:   
IF dj <= i THEN V(i, j) = min(V(i, j - 1), 1 + V(i - dj, j))

**✓ Correct**

**Feedback:**

IF dj <= i, then V(i, j) is the minimum of two cases: one where dj is picked and the other where it is not picked.

#### Q11: Coin Exchange Problem

In the coin exchange problem, you are given coins of denominations d1,d2, ...,dj, ..., dk.   
You know that V(i, j) represents the minimum number of coins required to pay i using the first j denominations. Which of these options is correct in this case?

(Recall the concept discussed in the video to answer the question)

Ans: IF dj > i THEN V(i, j) = V(i, j - 1)

**✓ Correct**

**Feedback:**

If dj > i, then you cannot pick the coin dj. Therefore, V(i, j) will be V(i, j - 1).

# Space Optimisation

For this, you performed these steps:

1. You created a one-dimensional (1D) array, T[i], to store your results. T[i] indicates the minimum number of coins required to make a change for i.
2. In the first case, you could use only one coin, i.e., d1=1. Here, you could pay i using i coins of value 1.  Thus, T[i] = i.
3. Next, you increased the number of denominations to two; so, you could use d1 and d2. Then you checked whether you could also use fewer coins to pay the amount using two denominations.

* 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, then you will not be able to pick a coin, and the value of T[i] will remain as it is.
* Also, it makes sense to pick a new coin if picking it will reduce the total number of coins to be used. Therefore, you used the formula T[i - dj] + 1 < T[i]. If this condition was true, then it would make sense to pick a coin such as dj as it would reduce the number of coins required to pay i.

      4. In the next iteration, you increased the number of denominations that could be used by 1 and updated the value of T[i], as you did in Step 3.

# Recap

In general, the characteristics of a problem to which you can apply dynamic programming can be summarised as follows:

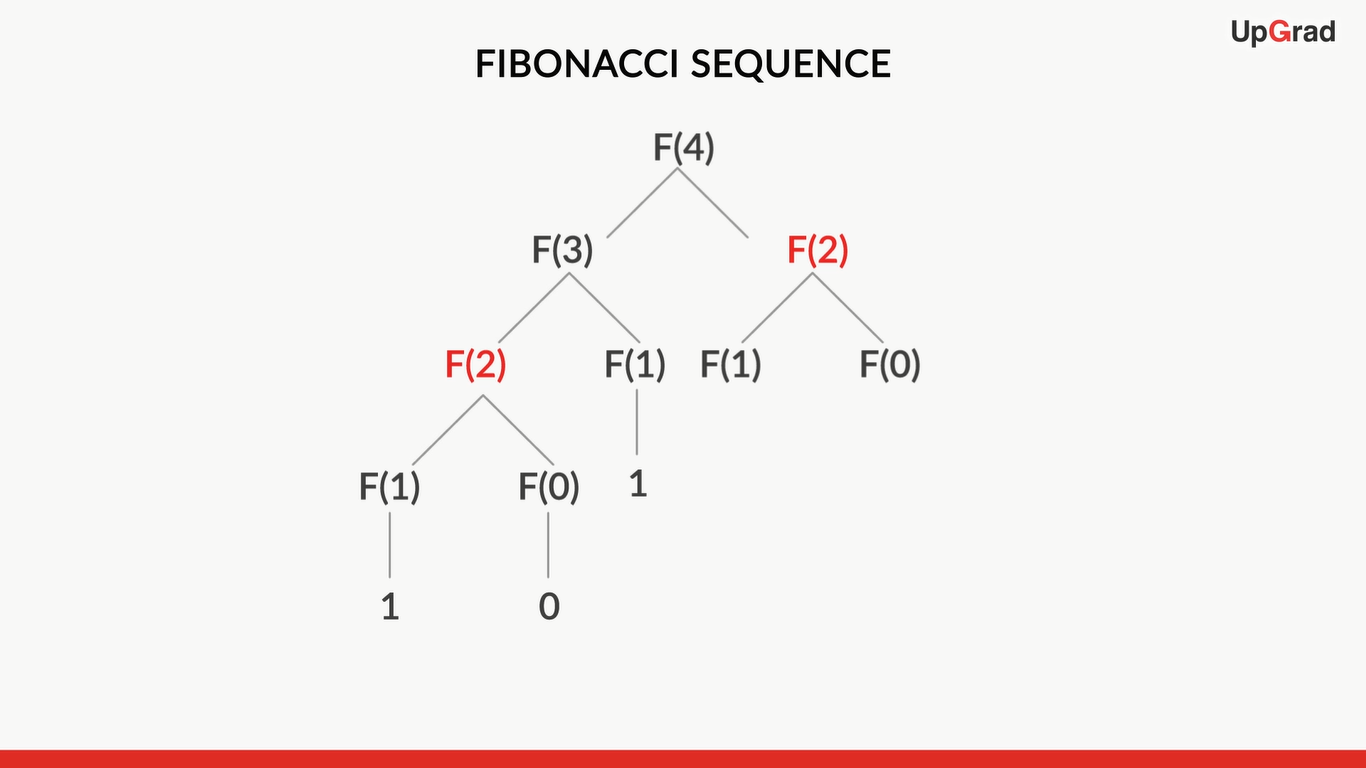
* You should be able to break down the problem into overlapping subproblems.
* Each subproblem must have a recursive solution.
* Each subproblem must be solved exactly once.
* The solutions to the subproblems should be combined in order to arrive at a solution to the given problem.

And this is what you need to do to solve the problem:

* Define the subproblems - This may seem easy but it is a significant step. You need to define the subproblems in words before you can start solving 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 that could be used. Had you defined the problem differently, the solution would have been different.
* Write down the recurrence that relates to the subproblems - In the coin exchange problem, for instance, you defined the recurrence relation where V(i, j) = min(V(i,j−1),1+V(i−dj,j)).
* Identify and solve base cases - In the coin exchange problem, for instance, you defined base cases where V(i, 1) = i and V(0, j) = 0. You could use these base cases to solve the rest of the subproblems.
* Store the results of the subproblems in a table - In the coin exchange problem, for example, you created a 2D array, in which you stored the solutions to all the subproblems.

# Revisiting the Fibonacci Sequence

When you were using a recursive solution, you ended up solving the same subproblem again and again. As you can see in this image, F(2) is being calculated twice in a recursive solution.



Recursive solution

On the other hand, if you used a dynamic programming solution, then you would have had to compute F(2) once only and store it in a table, and call it whenever required. This helps you avoid recomputation and, in turn, saves time.

A dynamic solution for the Fibonacci sequence involves these steps:

* Defining the subproblem (identifying the subproblem in words) - In this case, F[i] is the subproblem, where F[i] is the ith Fibonacci number.
* Writing down the recurrence that relates to the subproblems - The recurrence relation in this problem is F[i] = F[i-1] + F[i-2].
* Identifying and solving base cases - The base cases in the 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] stores the number at the nth index in the Fibonacci sequence.

#### Q12: Base Case

Instead of the Fibonacci sequence, suppose you are given this sequence: 0, 1, 2, 3, 6, 11, 20, 37, 68, …, n. Here, each number is the sum of the three preceding numbers. Also, it is given that the value at index i in this sequence is being represented with S(i).

What should be the base case to solve this problem using dynamic programming?

Ans: S(0) = 0, S(1) = 1 and S(2) = 2

**✓ Correct**

**Feedback:**

A base case helps the algorithm to take off. Thus, this option is correct, as you can calculate S(3) and then the other terms in the sequence.

#### Q13: Recursive Relation

Instead of the Fibonacci sequence, suppose you are given this sequence: 0, 1, 2, 3, 6, 11, 20, 37, 68, …, n. Here, each number is the sum of the three preceding numbers. Also, it is given that the value at index i in this sequence is being represented with S(i).

What should be the recurrence relation to solve this problem using dynamic programming?

Ans: S(i) = S(i - 1) + S(i - 2) + S(i - 3)

**✓ Correct**

**Feedback:**

Since S(i) is the sum of the three preceding values, i.e., S(i - 1), S(i - 2) and S(i - 3), your recurrence relation should be S(i) = S(i - 1) + S(i - 2) + S(i - 3).

#### Q15: Recursive Solution

Instead of the Fibonacci sequence, suppose you are given this sequence: 0, 1, 2, 3, 6, 11, 20, 37, 68, …, n. Here, each number is the sum of the three preceding numbers. Also, it is given that the value at index i in this sequence is being represented with S(i).

You have defined the recurrence relation as S(i) = S(i - 1) + S(i - 2) + S(i - 3). Considering you are solving this problem using recursion, what will be the space and time complexity?

Ans: Space complexity: O(n), time complexity: O(3n)

**✓ Correct**

**Feedback:**

The space complexity is O(n). The time complexity is 3n as there are three nodes, i.e., the value of S(i) is dependent upon three values, and each of these needs to be solved to get the answer.

#### Q16: Applying Dynamic Programming

Instead of the Fibonacci sequence, suppose you are given this sequence: 0, 1, 2, 3, 6, 11, 20, 37, 68, …, n. Here, each number is the sum of the three preceding numbers. Also, it is given that the value at index i in this sequence is being represented with S(i).

You have defined the recurrence relation as S(i) = S(i - 1) + S(i - 2) + S(i - 3). Considering you are solving this problem using **dynamic programming**, what will be the space and time complexity?

Ans:   
Space complexity: O(n), time complexity: O(n)

**✓ Correct**

**Feedback:**

Since you are only maintaining a 1D array of size n, the space complexity is O(n). To calculate each value, you need to use the sum of the three previously stored values, which is the operation O(1). Therefore, to calculate S(n), you need an O(n) time period.

# Practice Exercise I

**Comprehension I**

You are given a map to go from one place to another. The map is in the form of a cost matrix named cost[][], where each cell represents the amount of time required to pass through that cell. The robot starts from position (0, 0) and has to reach the location (m, n) on the map. You have to find the minimum amount of time required for the robot to reach the destination, (m, n). Each cell of the matrix represents the cost required to pass through that cell. The total cost of a path for the robot to reach (m, n) is the sum of all the costs on that path (including the source as well as the destination). From a given cell, you can only traverse downward, rightward and to diagonally lower cells to the right of that cell, i.e., you can traverse the given cell (i, j), and the cells (i + 1, j), (i, j + 1) and (i + 1, j + 1). You may assume that all costs are positive integers.

Now, suppose you are given this table.

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 8 | 2 |
| 1 | 5 | 3 |

So, the minimum cost path for [2][2] is 1 + 2 + 2 + 3 = 8.

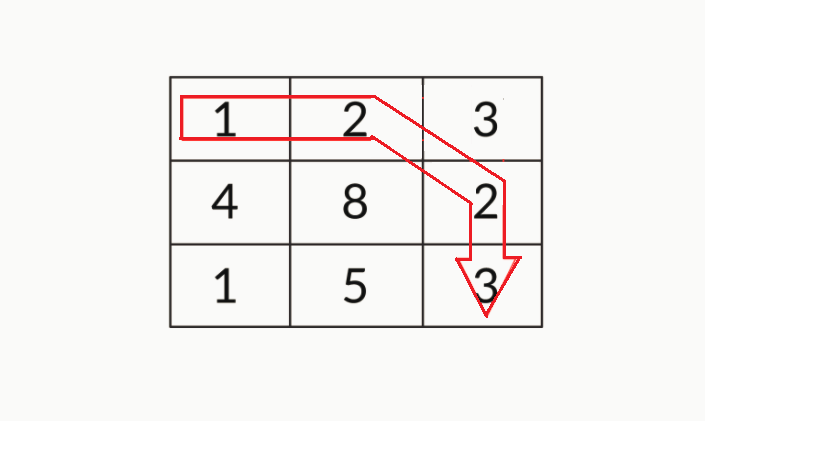


Image-MinimumCost

The problem can be framed as follows: Given a cost matrix cost[][] and a position (m, n) in cost[][], write a function that returns the cost of the minimum cost path to reach (m, n) from (0, 0).

#### Q17: Applying Dynamic Programming

According to you, what is the reason dynamic programming can provide an optimal solution to the problem given above?

Ans: Dynamic programming provides optimal solutions for any problems that consist of an optimal substructure property and an overlapping substructure property. The given problem consists of several overlapping problems. For example, to reach [3][3], you have to go through [1][3], for which you also need to pass to reach [2][3]. Also, the problem consists of an optimal substructure as a solution to reach any point, depending upon the minimum cost to reach its neighbouring points.

#### Q18: Minimum Cost Problem

Given that the subproblem is defined as cost[i][j], what is the recursive relation?

Recall the constraints on traversal in this problem and then compute the minimum cost.

Ans:   
minCost[i][j] = min(minCost[i-1][j-1] , minCost[i-1][j] , minCost[i][j-1] ) + cost[i][j]

**✓ Correct**

**Feedback:**

The cost to reach a point on the map is the minimum of the cost to reach its neighbouring points + the cost to reach the current location from the neighbouring points.

**Comprehension II**

Marvel is coming up with a new superhero named Jumping Jack. The co-creator of this superhero is a mathematician and he decides to add a mathematical touch to the character’s superpowers.

One of Jumping Jack’s most prominent superpowers is that he can jump long distances. Nevertheless, this superpower comes with these restrictions:  
A. Jumping Jack can only jump a distance 1 km less than the current distance. For example, if he is 12 km away from the destination, then he will not be able to jump directly to the destination as he can only jump to a location 11 km away in this case.  
B. He can only jump a distance half the current distance. For example, if Jumping Jack is 12 km away from the destination, then he will not be able to jump directly to the destination as he can only jump to a location 6 km away in this case.  
C. He can only jump a distance that is one-third the current distance. For example, if Jumping Jack is 12 km away from the destination, then he will not be able to jump directly to the destination as he can only jump to a location 4 km away in this case.

You need to create an algorithm that can help the superhero reach any destination using minimum number of jumps. Destination is defined as the place where the distance becomes 1. Jumping Jack can cover the last 1 km running. Also, he can only jump to a destination that is an integer distance away from the main destination. For example, if he is at a distance of 10 km, then by jumping one-third the distance, he cannot reach ten-thirds the distance; he has to either jump to 5 or 9.

So, you have to find the minimum number of jumps required to reach a destination. For instance, if the destination is 10 km away, then there are two ways to reach it:

* 10 -> 5 -> 4 -> 2 -> 1 (four jumps)
* 10 -> 9 -> 3 -> 1 (three jumps)

The minimum number of jumps that these two methods would require is three; so, the superhero can make a minimum of three jumps to reach the destination.

**Note:**Assume that the distance entered will always be a positive integer.

#### Q19: Defining the Subproblem

How will you define the subproblem for the dynamic programming problem given above? [**Note:** You may use the F() notation as used for the Fibonacci segment.]

Ans: You can define the subproblem as F(n), where n represents the distance from the destination and F(n) represents the minimum number of jumps required from distance n to distance 1.

#### Q20: Base Case

What is the base condition for this problem? [Note that F() denotes the number of jumps required to reach the destination, wherein a number as the destination is passed as an argument.]

Ans:   
F(1) = 0

**✓ Correct**

**Feedback:**

The aim is to reach the position that is 1 km away from the destination. Thus, the number of jumps required to reach the destination from 1 km away is F(1) = 0.

#### Q21: Recursive Relation

Which of these is the correct recursion relation for the given problem? Recall all conditional cases discussed in the problem statement.

Ans: F(n) = 1 + min{F(n-1), F(n/2), F(n/3)}

**✓ Correct**

**Feedback:**

The total number of jumps is one jump more than the number of places that Jumping Jack can jump to. Hence, F(n) = 1 + min{F(n-1), F(n/2), F(n/3)}.

**Practice Exercise II**

By now, you must have understood the coin exchange problem, wherein you needed to determine the minimum number of coins required such that they would add up to pay a given amount. Let us go ahead and make a small change to this problem. Instead of determining the minimum number of coins required to pay an amount, let us determine the total number of ways in which a change can be made.

So, suppose you have an unlimited supply of coins of these four denominations: d1=1,d2=5,d3=7,d4=10. You can use these combinations if you are asked to pay an amount of 11:

* 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
* 10 + 1
* 7 + 1 + 1 + 1 + 1
* 5 + 1 + 1 + 1 + 1 + 1 + 1
* 5 + 5 + 1

Thus, you can pay an amount of 11 in five different ways, provided you have coins of values 1, 5, 7 and 10. Nevertheless, if you have coins of values 1, 5 and 7 only, then you can pay 11 in four ways (the combination 10 + 1 cannot be used).

Let **V(i, j)**be the number of ways in which you can pay an amount i using the first j coins.

#### Q22: Identifying the Base Case

What should be the base case for this problem? [Note that **V(i, j)**is the number of ways in which you can pay an amount i using the first j coins.)

Ans: V(0, j) = 1 and V (i, 1) =1

**✓ Correct**

**Feedback:**

When you are finding the ways in which to pay an amount, using 0 coins of all denominations will also be counted as one of the methods. Hence, when the amount to make a change of is 0, the total number of ways in which you can make a change will be 1 (0 coins of all denominations). Also, if you have to pay an amount of i using the denomination 1 only, then there is only one way to pay this amount. Note that this is different from the number of coins required to pay the amount of i using a coin of value 1.

#### Q23: Coin Exchange Problem

What is the recursive relation that you can use to obtain the solution to this problem?

Ans: V(i, j) = V(i, j-1) + V(i - dj , j)

**✓ Correct**

**Feedback:**

There are two ways in which you could have paid the amount: when you pick a coin, in which case you need to find - V(i - dj, j), and when you do not pick a coin, in which case you need to find V(i, j-1). Now, each of these methods stores the total number of ways in which a change can be made for these amounts. Thus, V(i, j), i.e., total number of ways in which you can pay i using the first j coins, should be the sum of V(i - dj, j) and V(i, j-1).

#### Q24: Dynamic Programming

Could you think of some reasons why dynamic programming would not be the optimal way to solve problems you saw in the Divide & Conquer module, such as the Binary Search?

Ans: Divide and Conquer works by dividing the problem into sub-problems, conquer each sub-problem recursively and combine these solutions. This technique works best when all the sub-problems are independent(not overlapping). So, you pick partition that makes algorithm most efficient & simply combine solutions to solve entire problem. Dynamic programming is needed when subproblems are dependent(overlapping); you don’t know where to partition the problem. In dynamic programming algorithms each sub-problem is solved only once and is stored in a table, thus consuming more space. In cases such as Binary search, this space could have been reduced, as each problem would be solved only once, and hence no re-computations would be required, and thus, no need to store the results of all sub-problems.

‘**Solving Problems Using Dynamic Programming**’

# Knapsack Problem

#### **Q25:** 0-1 Knapsack Problem

Let us consider the 0-1 knapsack problem with these profit and weight values: Item 1 = ($12; 4 kg), Item 2 = ($10; 6 kg), Item 3 = ($16; 5 kg), Item 4 = ($11; 7 kg), Item 5 = ($14; 3 kg), Item 6 = ($7; 1 kg) and Item 7 = ($9; 6 kg). In the 0-1 knapsack problem, you can either pick an item or leave it. Considering you are given a knapsack of size 5 kg, what will be the maximum profit that you can make? Remember, you can pick each item once only.

**Ans:** $21

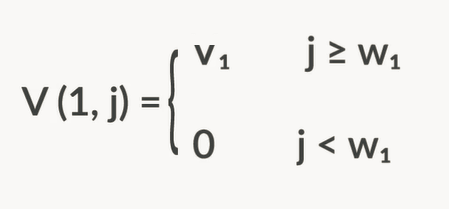
**✓ Correct**

**Feedback:**

The value in the knapsack will be maximised if you pick item 5 with value $14 and weight 3 kg and item 6 with value $7 and weight 1 kg. Although there is still some space left in the knapsack, you cannot add any more items to it, as the size of the remaining items is greater than the space left in the knapsack. Therefore, this is the maximum value that you can get with the items available.

**Applying Dynamic Programming**

Before you start solving the problem, remember the steps involved in dynamic programming:

* Define subproblems.
* Write down the recurrence that relates to the subproblems.
* Identify and solve base cases.
* Decide what you wish to store in a table and then fill the table with the results.
* So, you have now defined the base case for the 0-1 knapsack problem, as shown in this image.
* 
* Base Case
* The next step in dynamic programming is to find the recursive logic

#### Q26: Defining the Recursive Relation

Suppose you are given a knapsack of size 8, which has five items, whose profit and weight values are in this order: ($12, 4 kg), ($10, 6 kg), ($8, 8 kg), ($41, 4 kg) and ($14, 3 kg).

To arrive at the final dynamic programming solution, you need to solve many subproblems, represented with V(i, j), which gives the maximum profit that can be made from the first i items given a knapsack of size j.

You are required to calculate V(3, 7), i.e., the maximum profit that can be made from the first 3 items given a knapsack of size 7. What will V(3, 7) be? Recall the formula discussed in the video above.

Ans:   
V(3, 7) = V(i-1, j) = V(2, 7)

**✓ Correct**

**Feedback:**

V(3, 7) represents the maximum profit that can be made from the first 3 items given a knapsack of size 7. Now, the weight of item 3 is 8 kg. Since you cannot fit this item into a knapsack of size 7, you cannot pick it. So, you now have a knapsack of size 7 and items 1 and 2 to choose from; therefore, you need to find V(2, 7).

#### Q27: Defining the Recursive Relation

Let us consider the 0-1 knapsack problem with these profit and weight values: ($15, 4 kg), ($13, 6 kg), ($8, 5 kg), ($11, 7 kg), ($14, 3 kg), ($7, 1 kg) and ($9, 6 kg). Considering we know that V(3, 7)  = 15 and V(3, 14) = 28, what will be V(4, 14)?

Ans: V(4, 14) = V(i-1, j) = V(4-1, 14) = V(3, 14) = 28

**✓ Correct**

**Feedback:**

Although the weight of the knapsack is more than the weight of item 4, this does not mean picking item 4 will provide the optimal solution. Recall that V(i, j) = max(vi+V(i−1,j−wi), V(i-1, j)).

So, (V(3, 14) = 28) > (11 + V(3, 7) = 26).

Therefore, V(4, 14) = max(V(3, 14), (11 + V(3, 7)) = 28.

# Tabulation

Q28: