CHAPTER FOUR

Dynamic Programming

Dynamic Programming

- **Dynamic programming is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.**
- **It is used to solve optimization problems.**
- **The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.**

Dynamic Programming

- **Dynamic Programming helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.**

- **Overlapping Subproblems:**
  - **When the solutions to the same subproblems are needed repetitively for solving the actual problem.**

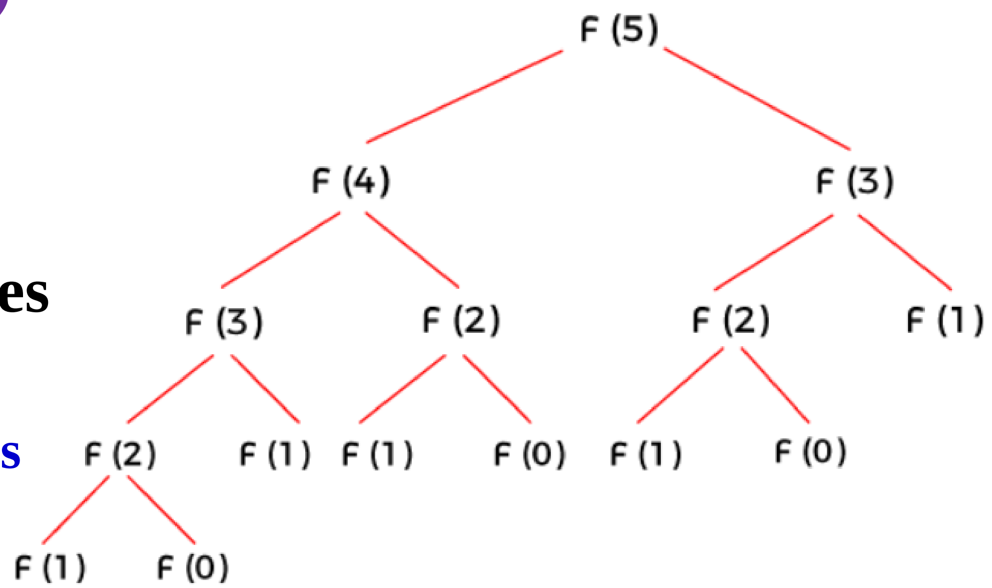
- **Optimal Substructure:**
  - **If the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.**

Dynamic Programming

- Recursive program of Fibonacci series:

```
int fib(int n)
{
   if(n<=1)
    return n;
  return fib(n-1) + fib(n-2)
}
```

- If we want to calculate fib(5), then the diagrammatic representation of f(5) is shown as below:



- F(3) is calculated twice
- F(2) is calculated three times

- This is **Overlapping subproblems** property

How it works?

- The following are the steps that the dynamic programming follows:

1. It breaks down the complex problem into simpler subproblems.
2. It finds the optimal solution to these sub-problems.
3. It stores the results of subproblems (memorization).
4. It reuses them so that same sub-problem is calculated more than once.
5. Finally, calculate the result of the complex problem.

Approaches of Dynamic Programming

- There are two approaches to dynamic programming: **Top-down** and **Bottom-up** approaches.

1. **Top-down approach:**
   - It uses the **memorization** technique,
   - Memorization is equal to the sum of recursion and caching.
   - **Recursion** means calling the function itself, while **caching** means storing the intermediate results.
   - It uses the recursion technique that occupies more memory in the call stack. (disadvantage)

Approaches of Dynamic Programming

1. **Top-down approach**: **Algorithm for Fibonacci sequence**

```
if (n < 2)
 then return n
 if (F[n] is undefined)
 then F[n] = MEMOFIB (n - 1) + MEMOFIB (n - 2)
 return F[n]
```

- **Fibonacci sequence** is the sequence of numbers in which every next item is the total of the previous two items.

- **We have used the memorization technique in which we store the results in an array to reuse the values.**
- **This is also known as a top-down approach in which we move from the top and break the problem into sub-problems.**

Approaches of Dynamic Programming

2. Bottom-up approach:

- It uses the tabulation technique
- It used to avoid the recursion, thus saving the memory space.
- It starts from the beginning, whereas the recursive algorithm starts from the end and works backward.
- If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions.
- In this tabulation technique, we solve the problems and store the results in a matrix.

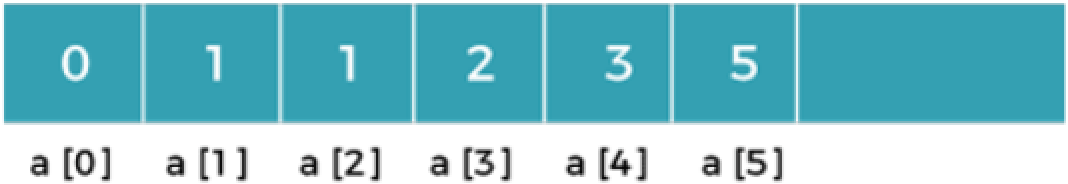
Approaches of Dynamic Programming

2. **Bottom-up approach:** Algorithm for Fibonacci sequence

```
Fib [0] = 0
Fib [1] = 1
for i = 2 to n
do
Fib[i] =  Fib [i - 1] + Fib [i - 2]
return F[n]
```

- We can replace recursion with a simple for-loop that just fills up the array Fib [] to iterate over the sub-problems.

- For example, the value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:



Dynamic Programming Application

- **The following computer problems can be solved using dynamic programming approach:**
  - **Fibonacci number series**
  - **All pair shortest path by Floyd-Warshall**
  - **Knapsack problem**
  - **Project scheduling**
  - **Matrix Chain Multiplication**

All Pairs Shortest Path Problem

- **The problem is to find the shortest path between all the pairs of vertices in a weighted graph.**
  - **It computes the shortest path between every pair of vertices of the given weighted graph.**
  - **A weighted graph is a graph in which each edge has a numerical value associated with it.**
- **Floyd-Warshall algorithm is used to solve All Pairs Shortest Path Problem.**
- **As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any vertex to all other vertices in the graph.**

How Floyd-Warshall Algorithm Works?

- **Consider a graph, G = {V, E} where**
  - **V is the set of all vertices present in the graph and**
  - **E is the set of all the edges in the graph.**
- **The graph, G, is represented in the form of an adjacency matrix, A, that contains all the weights of every edge connecting two vertices.**

✓ **Step 1: Construct an adjacency matrix A with all the costs of edges present in the graph.**
  - **If there is no path between two vertices, mark the value as ∞.**

How Floyd-Warshall Algorithm Works?

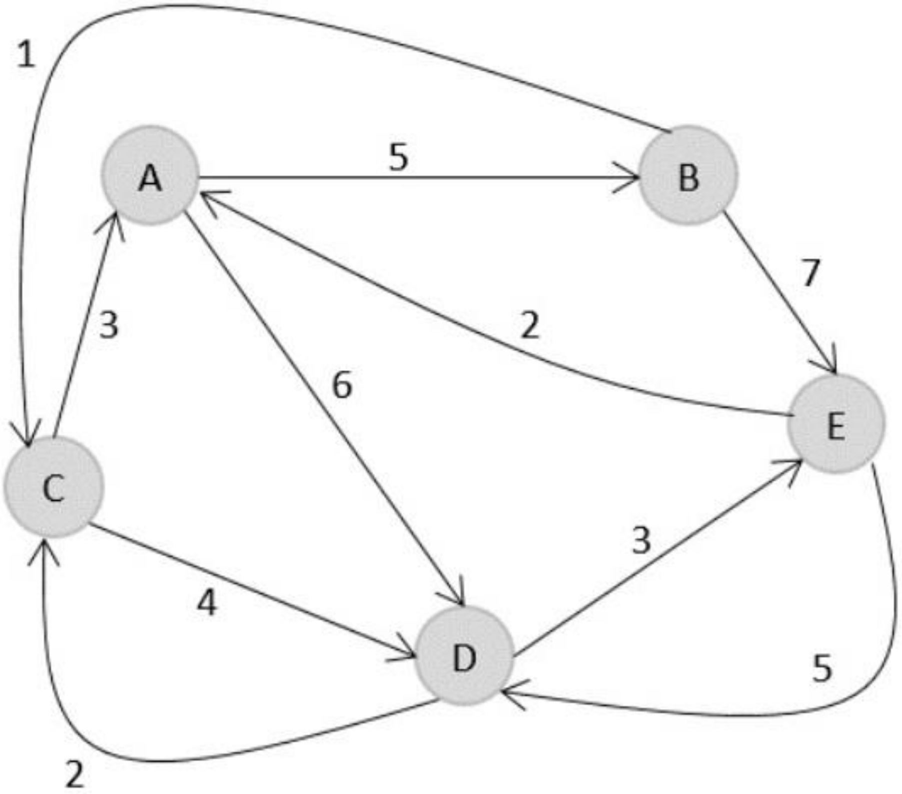
- ✓ **Step 2:** **Derive another adjacency matrix A_1 from A keeping the first row and first column of the original adjacency matrix intact in A_1.**
  - **And for the remaining values, say $A_1[i,j]$, if $A[i,j]>A[i,k]+A[k,j]$ then replace $A_1[i,j]$ with $A[i,k]+A[k,j]$. Otherwise, do not change the values.**
  - **Here, in this step, $k = 1$ (first vertex acting as pivot).**
- ✓ **Step 3:** **Repeat Step 2 for all the vertices in the graph by changing the k value for every pivot vertex until the final matrix is achieved.**
- ✓ **Step 4:** **The final adjacency matrix obtained is the final solution with all the shortest paths.**

Floyd-Warshall Algorithm: Pseudocode

```
Floyd-Warshall(w, n) // w: weights, n: number of vertices
  for i = 1 to n do // initialize,
    for j = 1 to n do
        if (i = = j)
            A[i, j] = 0  // For all diagonal elements, value = 0
        if (i , j) is an edge in E
            A[i, j] = w[i, j];  // If there exists a direct edge between the vertices,
                                   value = weight of edge
        else
            A[i, j] = infinity // If there is no direct edge between the vertices, value = ∞
    for k = 1 to n do // Compute A (k) from A (k-1)
      for i = 1 to n do
        for j = 1 to n do
          if (A[i, k] + A[k, j] < A[i, j])
            A[i, j] = A[i, k] + A[k, j];
    return A[1..n, 1..n];
```

Floyd-Warshall Algorithm: Example

- Consider the following directed weighted graph G = {V, E}.
- Find the shortest paths between all the vertices of the graphs using the Floyd-Warshall algorithm.



Floyd-Warshall Algorithm: Example

Solution:

- **Step 1:** Construct an adjacency matrix A with all the distances as values.

$$A = \begin{matrix} 0 & 5 & \infty & 6 & \infty \\ \infty & 0 & 1 & \infty & 7 \\ 3 & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 2 & \infty & \infty & 5 & 0 \end{matrix}$$

- Create a matrix A of dimension n*n where n is the number of vertices.
- The row and the column are indexed as i and j respectively.
- i and j are the vertices of the graph.

- Each cell A[i][j] is filled with the distance from the i^{th} vertex to the j^{th} vertex.
- If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

Floyd-Warshall Algorithm: Example

Solution:

- **Step 2:** **Considering the above adjacency matrix as the input, derive another matrix A_1 by keeping only first rows and columns intact. Take k = 1, and replace all the other values by A[i, k] + A[k, j].**

- **While considering k^{th} vertex as intermediate vertex, there are two possibilities :**
  1. **If k is not part of shortest path from i to j, we keep the distance A[i, j] as it is.**
  2. **If k is part of shortest path from i to j, update distance A[i, j] as A[i, k] + A[k, j].**

Floyd-Warshall Algorithm: Example

Solution:

- **We can use the following optimal substructure formula for Floyd's algorithm,**

$$A^k[i, j] = \min\{ A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j] \}$$

  - A^k = Distance matrix after k^{th} iteration

$$A = \begin{array}{ccccc} 0 & 5 & \infty & 6 & \infty \\ \infty & & & & \\ 3 & & & & \\ \infty & & & & \\ 2 & & & & \end{array}$$

$$A_1 = \begin{array}{ccccc} 0 & 5 & \infty & 6 & \infty \\ \infty & 0 & 1 & \infty & 7 \\ 3 & 8 & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 2 & 7 & \infty & 5 & 0 \end{array}$$

Floyd-Warshall Algorithm: Example

Solution:

Iteration 1 (k = 1)

for i = 1

- **A1[1, 2] = min{A[1, 2], A[1, 1] + A[1, 2]} = min { 5, 0 + 5} = 5**
- **A1[1, 3] = min{A[1, 3], A[1, 1] + A[1, 3]} = min {∞, 0 + ∞} = ∞**
- **A1[1, 4] = min{A[1, 4], A[1, 1] + A[1, 4]} = min {6, 0 + 6} = 6**
- **A1[1, 5] = min{A[1, 5], A[1, 1] + A[1, 5]} = min {∞, 0 + ∞} = ∞**

for i = 2

- **A1[2, 1] = min{A[2, 1], A[2, 1] + A[1, 1]} = min {∞, ∞ + 0} = ∞**
- **A1[2, 2] = min{A[2, 2], A[2, 1] + A[1, 2]} = min {0 , ∞ + 0} = 0**
- **A1[2, 3] = min{A[2, 4], A[2, 1] + A[1, 3]} = min {1, ∞ + ∞} = 1**
- **A1[2, 4] = min{A[2, 5], A[2, 1] + A[1, 4]} = min {∞, ∞ + 6} = ∞**
- **A1[2, 5] = min{A[2, 5], A[2, 1] + A[1, 5]} = min {7, ∞ + ∞} = 7**

Floyd-Warshall Algorithm: Example

Solution:

- **Step 3:** **Considering the above adjacency matrix as the input, derive another matrix A_2 by keeping only first rows and columns intact. Take k = 2, and replace all the other values by $A_1[i, k]+A_1[k, j]$.**

$$A_2 = \begin{array}{ccccc} & 5 & & & \\ \infty & 0 & 1 & \infty & 7 \\ & 8 & & & \\ & \infty & & & \\ & 7 & & & \end{array}$$

$$A_2 = \begin{array}{ccccc} 0 & 5 & 6 & 6 & 12 \\ \infty & 0 & 1 & \infty & 7 \\ 3 & 8 & 0 & 4 & 15 \\ \infty & \infty & 2 & 0 & 3 \\ 2 & 7 & 8 & 5 & 0 \end{array}$$

Floyd-Warshall Algorithm: Example

Solution:

- **Step 4:** **Considering the above adjacency matrix as the input, derive another matrix A_3 by keeping only first rows and columns intact. Take k = 3, and replace all the other values by $A_2[i, k]+A_2[k, j]$.**

$$A_3 = \begin{array}{ccccc} & & 6 & & \\ & & 1 & & \\ 3 & 8 & 0 & 4 & 15 \\ & & 2 & & \\ & & 8 & & \end{array}$$

$$A_3 = \begin{array}{ccccc} 0 & 5 & 6 & 6 & 12 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 15 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 8 & 5 & 0 \end{array}$$

Floyd-Warshall Algorithm: Example

Solution:

- **Step 5:** **Considering the above adjacency matrix as the input, derive another matrix A_4 by keeping only first rows and columns intact. Take k = 4, and replace all the other values by $A_3[i, k] + A_3[k, j]$.**

$$A_4 = \begin{matrix} & & & 6 & \\ & & & 5 & \\ & & & 4 & \\ 5 & 10 & 2 & 0 & 3 \\ & & & 5 & \end{matrix}$$

$$A_4 = \begin{matrix} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix}$$

Floyd-Warshall Algorithm: Example

Solution:

- **Step 6:** **Considering the above adjacency matrix as the input, derive another matrix A_5 by keeping only first rows and columns intact. Take k = 5, and replace all the other values by $A_4[i, k] + A_4[k, j]$.**

$$A_5 = \begin{array}{ccccc} & & & & 9 \\ & & & & 7 \\ & & & & 7 \\ & & & & 3 \\ 2 & 7 & 7 & 5 & 0 \end{array} \qquad A_5 = \begin{array}{ccccc} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{array}$$

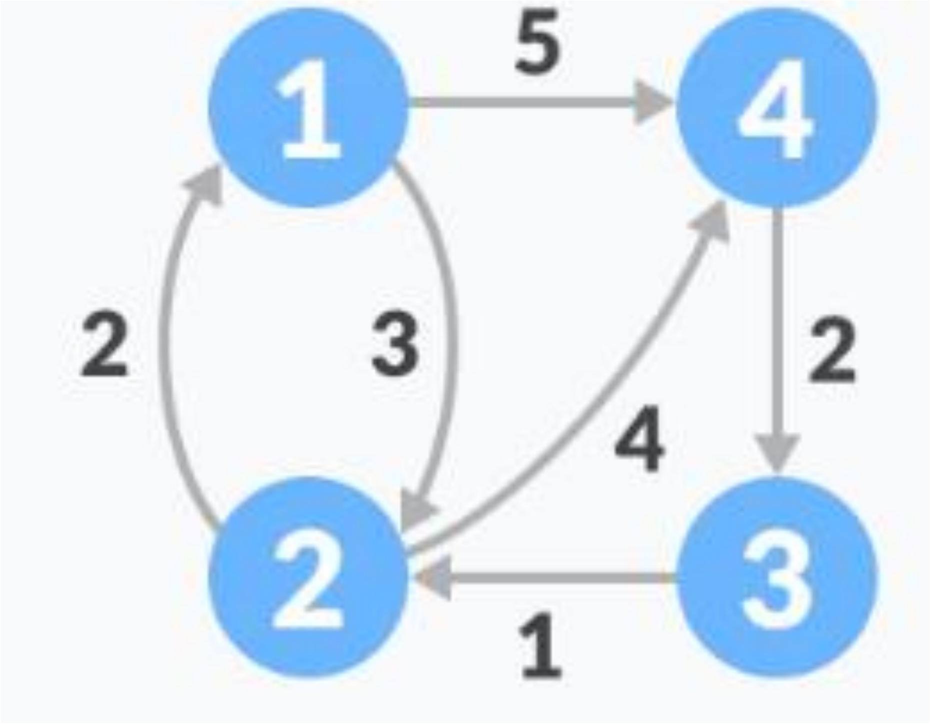
- **The last matrix A_5 represents the shortest path distance between every pair of vertices.**

Floyd-Warshall Algorithm: Analysis

- The Floyd-Warshall algorithm operates using three for loops to find the shortest distance between all pairs of vertices within a graph.
- Therefore, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$, where ‘n’ is the number of vertices in the graph.
- The space complexity of the algorithm is $O(n^2)$.

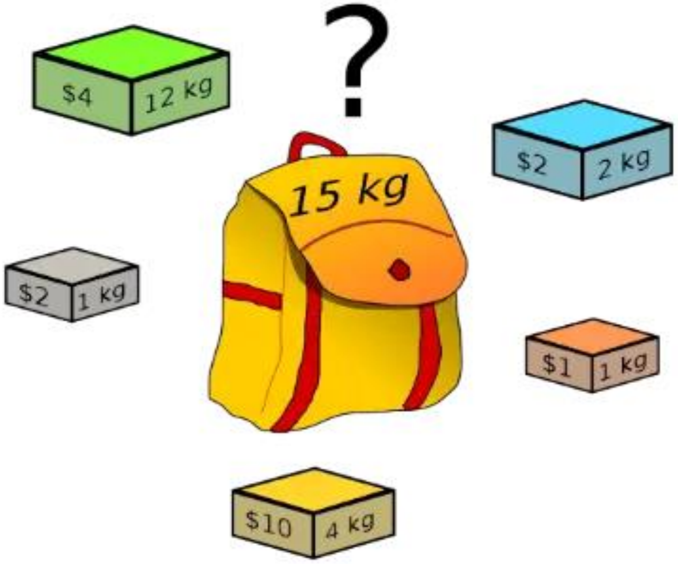
Floyd-Warshall Algorithm: Exercise

- Consider the following directed weighted graph G = {V, E}.
- Find the shortest paths between all the vertices of the graphs using the Floyd-Warshall algorithm.



Knapsack problem

- The knapsack problem states that − given a set of items (n), holding weights (wi) and profit values (vi), one must determine the subset of the items to be added in a knapsack (a kind of bag) such that:
  - the total weight of the items must not exceed the limit of the knapsack and
  - its total profit value is maximum.



- Types of Knapsack problem:
  1. Fractional Knapsack Problem
  2. 0/1 Knapsack Problem

Knapsack problem

1. Fractional Knapsack Problem

- In this problem, items are divisible.
- We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
- It is solved using Greedy Method.

2. 0/1 Knapsack Problem

- In this problem, items are indivisible.
- We can not take the fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using Dynamic Programming approach.

Fractional Knapsack Problem

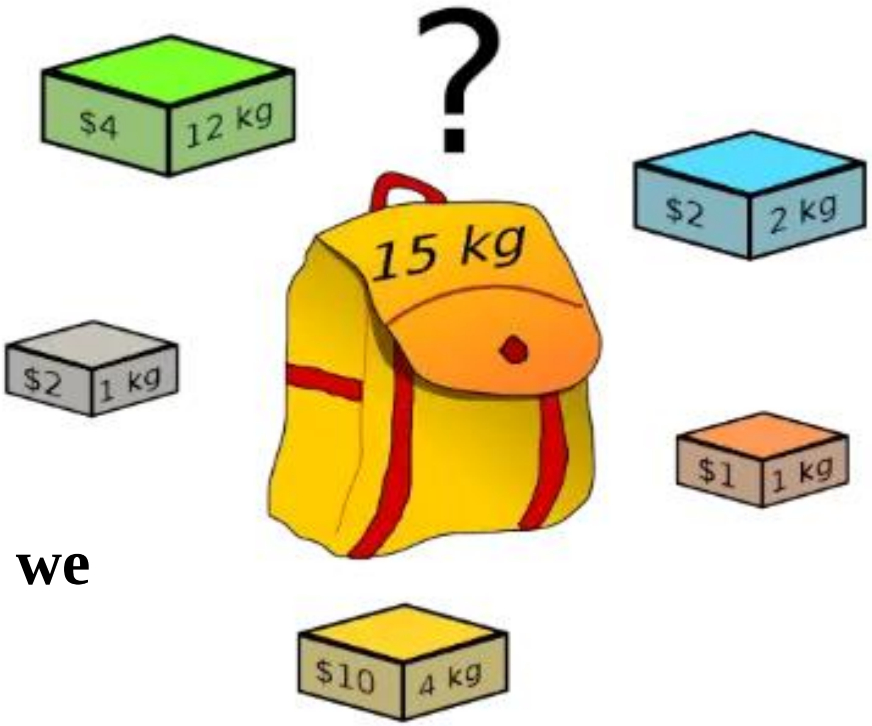
- **Algorithm:**

1. **Compute value / weight ratio of all the items .**
2. **Sort the items in descending order based on their value / weight ratio.**
3. **Start putting the items into the knapsack beginning from the item with the highest ratio.**
   - **Put as many items as you can into the knapsack.**
4. **If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next item can be added.**

Fractional Knapsack problem: Example

- Suppose we have a knapsack of 15 kg. The total weight of the items to be included in the knapsack should have the weight less than or equal to the weight of knapsack (15 kg).

Item	A	B	C	D	E
Weight (in kg)	12	2	1	4	1
Value (Profit)	4	2	1	10	2



- Here, we have to decide whether we have to include the item or not.

Fractional Knapsack problem: Example

Solution:

1. First calculate the profit (pi)/weight (wi) ratio of the items.

Item	A	B	C	D	E
Weight (in kg)	12	2	1	4	1
Value (Profit)	4	2	1	10	2
Pi/Wi	0.33	1	1	2.5	2

2. Sort the items in descending order based on their pi/wi ratio.

Item	D	E	B	C	A
Weight (in kg)	4	1	2	1	12
Value (Profit)	10	2	2	1	4
Pi/Wi	2.5	2	1	1	0.33

Fractional Knapsack problem: Example

Solution:

3. Start putting the items into the knapsack beginning from the item with the highest ratio.

Item	Weight	Profit	Knapsack	Remaining Weight
D	4	10	1	15 – 4 = 11
E	1	2	1	11 – 1 = 10
B	2	2	1	10 – 2 = 8
C	1	1	1	8 – 1 = 7
A	7	4	7/12	7 – 7 = 0

- Therefore, the knapsack holds the weights = [(1 * 4) + (1 * 1) + (1 * 2) + (1 * 1) + (7/12 * 4)] = 15, with maximum profit of [(1 * 10) + (1 * 2) + (1 * 2) + (1 * 1) + (7/12 * 4)] = 17.33.

0/1 Knapsack problem: Algorithm

- **Step 1: Create a table (T) with maximum weight of knapsack (W) as columns and items (N) with respective weights (wi) and profits (pi) as rows: T[N][W].**
- **Step 2: add zeroes to the 0^{th} row and 0^{th} column because:**
  - **if the weight of item is 0, then it weighs nothing;**
  - **if the maximum weight of knapsack is 0, then no item can be added into the knapsack.**

0/1 Knapsack problem: Algorithm

- **Step 3: Start filling the table row wise top to bottom from left to right by using following formula:**

T (i , j) = max { T (i-1 , j) , valuei + T(i-1 , j – weighti) }

  - **Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.**
  - **This step leads to completely filling the table.**
  - **Then, value of the last box represents the maximum possible value that can be put into the knapsack.**

0/1 Knapsack problem: Algorithm

```
for j = 0 to W
        T[0, j] = 0
for i = 0 to n
        T[i, 0] = 0
for i = 1 to n
        for j = 1 to W
                if wi <= j // item i can be part of the solution
                        if vi + T[i-1, j-wi] > T[i-1, j]
                                T[i, j] = vi + T[i-1, j- wi]
                        else
                                T[i, j] = T[i-1, j]
                else T[i, j] = T[i-1, j] // wi > j
```

Fill the 0^{th} row and 0^{th} column of the table to Zeros

0/1 Knapsack problem: Example

- **Find an optimal solution for following 0/1 Knapsack problem using dynamic programming:**
  - **Number of Items n = 4,**
  - **Knapsack Capacity W = 5,**
  - **Weights (w1, w2, w3, w4) = (2, 3, 4, 5) and**
  - **profits/values (v1, v2, v3, v4) = (3, 4, 5, 6).**

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

0/1 Knapsack problem: Example

Solution:

- Solution of the knapsack problem is defined as:

$$T[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ T[i-1, j] & \text{if } j < wi \\ \max\{T[i-1, j], vi + T[i-1, j - wi]\} & \text{if } j \geq wi \end{cases}$$

- Create a Table T with N+1 rows and W+1 columns

i\j	0	1	2	3	4	5
0						
1						
2						
3						
4						

0/1 Knapsack problem: Example

Solution:

for j = 0 to W

T[0, j] = 0

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for i = 0 to n

T[i, 0] = 0

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

0/1 Knapsack problem: Example

Solution:

Filling first row, i = 1

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

T[1, 1] ⇒ i = 1, j = 1, wi = w1 = 2, vi = 3

As, j < wi, T[i, j] = T[i – 1, j]

T[1, 1] = T[0, 1] = 0

T[1, 2] ⇒ i = 1, j = 2, wi = w1 = 2, vi = 3

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max {T[0, 2], 3 + T[0, 0]}

T[1, 2] = max(0, 3 + 0) = 3

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	-	-	-	-	-
2	0					
3	0					
4	0					

T[1, 3] ⇒ i = 1, j = 3, wi = w1 = 2, vi = 3

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max {T[0, 3], 3 + T[0, 1]}

T[1, 3] = max(0, 3 + 0) = 3

0/1 Knapsack problem: Example

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

Solution:

Filling first row, i = 1

T[1, 4] ⇒ i = 1, j = 4, wi = w1 = 2, vi = 3

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max{T[0, 4], 3 + T[0, 2]}

T[1, 4] = max(0, 3 + 0) = 3

T[1, 5] ⇒ i = 1, j = 5, wi = w1 = 2, vi = 3

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max{T[0, 5], 3 + T[0, 3]}

T[1, 5] = max(0, 3 + 0) = 3

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	-	-	-	-	-
2	0					
3	0					
4	0					

0/1 Knapsack problem: Example

Solution:

Filling first row, i = 1

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

0/1 Knapsack problem: Example

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

Solution:

Filling second row, i = 2

T[2, 1] ⇒ i = 2, j = 1, wi = 3, vi = 4

As, j < wi, T[i, j] = T[i – 1, j]

T[2, 1] = T[1, 1] = 0

T[2, 2] ⇒ i = 2, j = 2, wi = 3, vi = 4

As, j < wi, T[i, j] = T[i – 1, j]

T[2, 2] = T[1, 2] = 3

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	-	-	-	-	-
3	0					
4	0					

T[2, 3] ⇒ i = 2, j = 3, wi = 3, vi = 4

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max{T[1, 3], 4 + T[1, 0]}

T[2, 3] = max(3, 4) = 4

0/1 Knapsack problem: Example

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

Solution:

Filling second row, i = 2

T[2, 4] ⇒ i = 2, j = 4, wi = 3, vi = 4

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max{T[1, 4], 4 + T[1, 1]}

T[2, 4] = max(3, 4) = 4

T[2, 5] ⇒ i = 2, j = 5, wi = 3, vi = 4

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max{T[1, 5], 4 + T[1, 2]}

T[1, 5] = max(3, 4+3) = 7

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	-	-	-	-	-
3	0					
4	0					

0/1 Knapsack problem: Example

Solution:

Filling second row, i = 2

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

0/1 Knapsack problem: Example

Solution:

Filling third row, i = 3

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

T[3, 1] ⇒ i = 3, j = 1, wi = 4, vi = 5

As, j < wi, T[i, j] = T[i – 1, j]

T[3, 1] = T[2, 1] = 0

T[3, 2] ⇒ i = 3, j = 2, wi = 4, vi = 5

As, j < wi, T[i, j] = T[i – 1, j]

T[3, 2] = T[2, 2] = 3

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	-	-	-	-	-
4	0					

T[3 3] ⇒ i = 3, j = 3, wi = 4, vi = 5

As, j < wi, T[i, j] = T[i – 1, j]

T[3, 3] = T[2, 3] = 4

0/1 Knapsack problem: Example

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

Solution:

Filling third row, i = 3

T[3, 4] ⇒ i = 3, j = 4, wi = 4, vi = 5

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max{T[2, 4], 5 + T[2, 0]}

T[3, 4] = max(4, 5 + 0) = 5

T[3, 5] ⇒ i = 3, j = 5, wi = 3, vi = 5

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max{T[2, 5], 5 + T[2, 1]}

T[3, 5] = max(7, 5 + 0) = 7

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	-	-	-	-	-
4	0					

0/1 Knapsack problem: Example

Solution:

Filling third row, i = 3

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

0/1 Knapsack problem: Example

Solution:

Filling fourth row, i = 4

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

T[4, 1] ⇒ i = 4, j = 1, wi = 5, vi = 6

As, j < wi, T[i, j] = T[i – 1, j]

T[4, 1] = T[3, 1] = 0

T[4, 2] ⇒ i = 4, j = 2, wi = 5, vi = 6

As, j < wi, T[i, j] = T[i – 1, j]

T[4, 2] = T[3, 2] = 3

T[4, 3] ⇒ i = 4, j = 3, wi = 5, vi = 6

As, j < wi, T[i, j] = T[i – 1, j]

T[4, 3] = T[3, 3] = 4

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	-	-	-	-	-

0/1 Knapsack problem: Example

Solution:

Filling fourth row, i = 4

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

T[4, 4] ⇒ i = 4, j = 4, wi = 5, vi = 6

As, j < wi, T[i, j] = T[i – 1, j]

T[4, 4] = T[3, 4] = 5

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	-	-	-	-	-

T[4, 5] ⇒ i = 4, j = 5, wi = 5, vi = 6

As, j ≥ wi, T[i, j] = max{T[i – 1, j], vi + T[i – 1, j – wi]}

= max{T[3, 5], 6 + T[3, 0]}

T[3, 5] = max(7, 6 + 0) = 7

0/1 Knapsack problem: Example

Solution:

Filling fourth row, i = 4

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

0/1 Knapsack problem: Example

Solution:

- **This algorithm only finds the maximum possible value that can be carried in the knapsack**
  - **i.e., the value in T[n,W]**
- **To know the items that make this maximum value, an addition to this algorithm is necessary.**

- **So, how to find the actual Knapsack items?**

0/1 Knapsack problem: Example

Solution:

- All of the information we need is in the table.
- T[n,W] is the maximal value of items that can be placed in the Knapsack.
- Let i=n and j=W

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

if $T[i, j] \neq T[i - 1, j]$ then

$i = i - 1$ and $j = j - w_i$

//mark the i^{th} item as in the knapsack

else

$i = i - 1$ //assume the i^{th} item is not in the knapsack

0/1 Knapsack problem: Example

Solution:

- Therefore, find the selected items for W = 5.

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

Step 1: Initially, set i = n = 4 and j = W = 5

- T[i, j] = T[4, 5] = 7
- T[i – 1, j] = T[3, 5] = 7
- T[i, j] = T[i – 1, j], so don't select i^{th} item and check for the previous item.
- so i = i – 1 = 4 – 1 = 3

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Solution Set S = { }

0/1 Knapsack problem: Example

Solution:

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

Step 2: i = 3 and j = 5

- **T[i, j] = T[3, 5] = 7**
- **T[i – 1, j] = T[2, 5] = 7**
- **T[i, j] = T[i – 1, j], so don't select i^{th} item and check for the previous item.**
- **so i = i – 1 = 3 – 1 = 2**

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Solution Set S = { }

0/1 Knapsack problem: Example

Solution:

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

Step 3: i = 2 and j = 5

- T[i, j] = T[2, 5] = 7
- T[i – 1, j] = T[1, 5] = 3
- T[i, j] ≠ T[i – 1, j], so add item $I_i = I_2$ in solution set.
- Reduce problem size j by wi
- j = j – wi = j – w2 = 5 – 3 = 2
- i = i – 1 = 2 – 1 = 1

Solution Set S = { I_2 }

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

0/1 Knapsack problem: Example

Solution:

Items	Weight	Value
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

Step 4: i = 1 and j = 2

- T[i, j] = T[1, 2] = 3
- T[i – 1, j] = T[0, 2] = 0
- T[i, j] ≠ T[i – 1, j], so add item $I_i = I_1$ in solution set.
- Reduce problem size j by wi
- j = j – wi = j – w1 = 2 – 2 = 0
- i = i – 1 = 1 – 1 = 0

Solution Set S = { I_1, I_2}

i\j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

- Problem size has reached to 0, so final solution is S = {I_1, I_2} with a maximum value (profit) = v1 + v2 = 7.

0/1 Knapsack problem: Exercise

- **Find an optimal solution for following 0/1 Knapsack problem using dynamic programming:**
  - **Number of Items n = 3,**
  - **Knapsack Capacity W = 6,**
  - **Weights (w1, w2, w3) = (3, 4, 2) and**
  - **profits/values (v1, v2, v3) = (9, 8, 10).**

Items	Weight	Value
I_1	3	9
I_2	4	8
I_3	2	10

Conclusion

- **Dynamic programming is a useful technique of solving certain kind of problems.**
- **When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and reuse them as necessary (memorization).**
- **Running time of dynamic programming algorithm vs. naïve algorithm, for 0-1 Knapsack problem: $O(W*n)$ vs. $O(2^n)$.**

THANK YOU!