

# Dynamic Programming

## ▼ TOC

[Introduction](#)

[Difference between DP, Greedy, Divide & Conquer](#)

[Floyd-Warshall Algorithm](#)

[Advantages](#)

[Disadvantages](#)

[Algorithm](#)

[Rule](#)

[Example](#)

[Complexities](#)

[0/1 Knapsack](#)

[Advantages](#)

[Disadvantages](#)

[Recurrence relation](#)

[Algorithm](#)

[Example](#)

[Complexities](#)

[Travelling Salesperson](#)

[Advantages](#)

[Disadvantages](#)

[Algorithm](#)

[Example](#)

[Complexities](#)

[Bellman Ford Algorithm](#)

[Advantages](#)

[Disadvantages](#)

[Algorithm](#)

[Example](#)

[Complexities](#)

## Introduction

Dynamic Programming is a technique for solving complex optimization problems by breaking them down into smaller subproblems and solving each subproblem only once, storing the solution in a table for future reference. This approach is called **memoization**.

It is used in situations where a problem can be divided into overlapping subproblems, making it possible to avoid solving the same subproblems multiple

times.

Dynamic Programming is based on the principle of optimal substructure, which means that the optimal solution to a problem can be found by combining the optimal solutions to its subproblems.

Dynamic Programming can be applied to two types of problems: **top-down and bottom-up**. The top-down approach involves starting with the original problem and breaking it down into smaller subproblems, while the bottom-up approach involves starting with the smallest subproblems and working up to the original problem.

## Difference between DP, Greedy, Divide & Conquer

DP involves breaking down a problem into smaller subproblems and solving them individually, storing the results in a table or array so that they can be easily accessed in subsequent computations. DP is used for problems with overlapping subproblems and optimal substructure.

Greedy algorithms make the locally optimal choice at each step in hopes of finding a global optimum. This means that the algorithm selects the best option at each step without considering the future consequences. Greedy algorithms are used for problems with the greedy choice property.

Divide and Conquer involves breaking down a problem into smaller subproblems, solving them independently, and then combining the solutions to solve the original problem. This approach is useful when the problem can be divided into smaller independent subproblems that can be solved recursively.

## Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm is an algorithm used to find the shortest path between all pairs of vertices in a weighted graph. The algorithm builds a table of the shortest path distances between all pairs of vertices by considering all possible paths through intermediate vertices.

### Advantages

- The Floyd-Warshall Algorithm is a well-known and widely used algorithm for finding the shortest

### Disadvantages

- The time complexity of the algorithm is  $O(V^3)$ , which can be slow for large graphs.

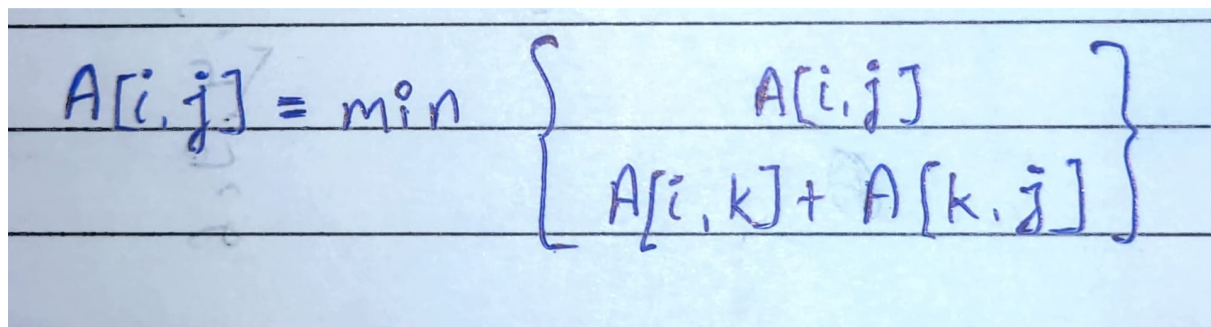
path between all pairs of vertices in a graph.

- The algorithm can handle negative edge weights, unlike Dijkstra's Algorithm.
- The algorithm requires storing a table of the shortest path distances between all pairs of vertices, which can be memory-intensive for large graphs.

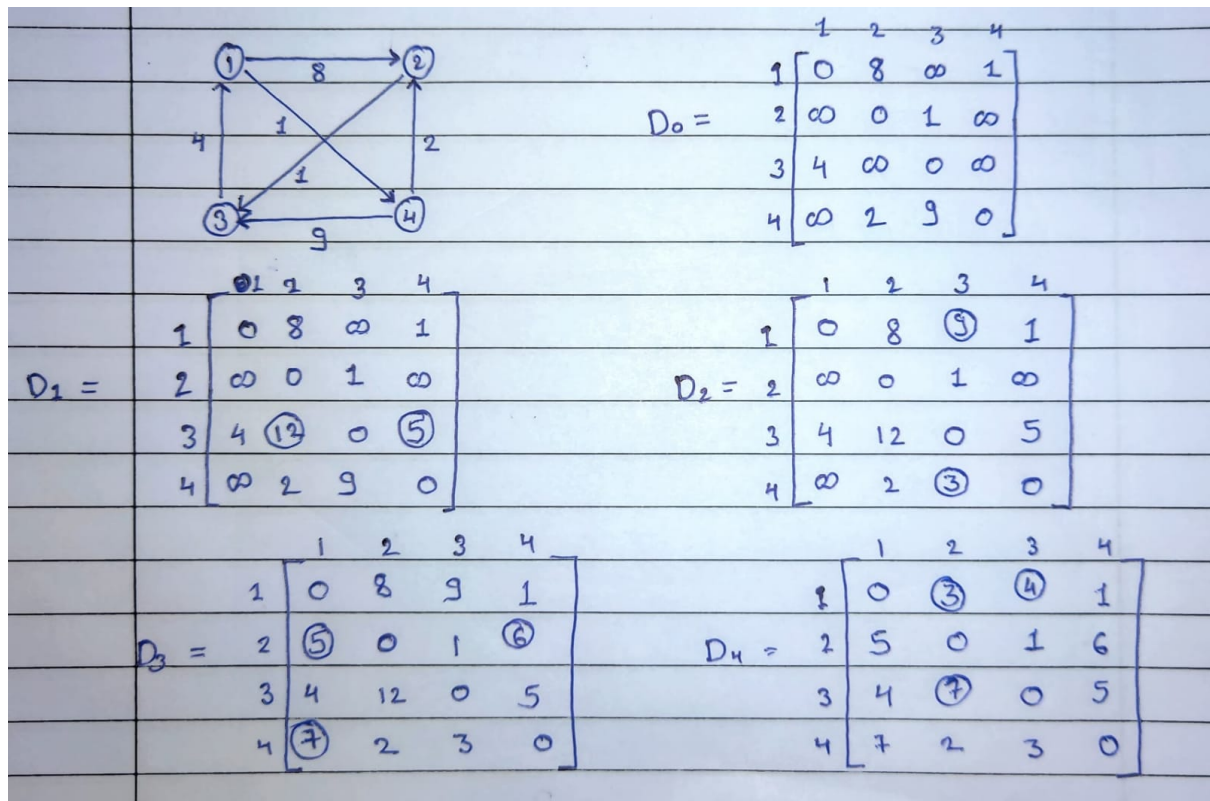
## Algorithm

1. Initialize a table of the shortest path distances between all pairs of vertices with the edge weights of the graph.
2. For each intermediate vertex, update the table by considering all possible paths through the intermediate vertex and choosing the shortest path distance.
3. Repeat until all intermediate vertices have been considered.

## Rule


$$A[i, j] = \min \left\{ \begin{array}{l} A[i, j] \\ A[i, k] + A[k, j] \end{array} \right\}$$

## Example



## Complexities

The time complexity is  $O(V^3)$ , where  $V$  is the number of vertices in the graph.

The space complexity is also  $O(V^3)$ , as it requires storing a table of the shortest path distances between all pairs of vertices. However, the space complexity can be reduced to  $O(V^2)$  by only storing the current and previous rows of the table instead of the entire table.

## 0/1 Knapsack

0/1 Knapsack is a problem in which we have a knapsack with a fixed capacity and a set of items with weights and values. The goal is to maximize the total value of the items that can be placed in the knapsack, without exceeding its capacity. Each item can only be either fully included or fully excluded from the knapsack.

### Advantages

- 0/1 Knapsack is a well-known algorithm that provides an optimal solution to the problem.

### Disadvantages

- 0/1 Knapsack assumes that each item can only be fully included or

- The algorithm can handle items with different weights and values.

fully excluded from the knapsack, which may not always be practical.

- The algorithm may not be able to handle items with complex dependencies or interrelationships.

## Recurrence relation

0/1 Knapsack( $n, m$ ) where,

$n$  is total number of items

$m$  is total weight of the knapsack

$P_n$  is profit by  $n^{th}$  item

$W_n$  is weight of the  $n^{th}$  item

Handwritten recurrence relation for 0/1 Knapsack:

$$0/1 \text{ Knapsack}(n, m) = \begin{cases} \max \begin{cases} 0/1 \text{ KS}(n-1, (m - W_n) + P_n \\ 0/1 \text{ KS}(n-1, m) \end{cases} \\ 0/1 \text{ KS}(n-1, m) \text{ if } W_n > m \\ 0 \text{ if } n=0 \text{ or } m=0 \end{cases}$$

## Algorithm

1. Initialize a two-dimensional array with size  $(n+1) \times (W+1)$ , where  $n$  is the number of items and  $W$  is the maximum capacity of the knapsack.
2. For each item  $i$  from 1 to  $n$  and for each weight  $w$  from 1 to  $W$ , if the weight of item  $i$  is less than or equal to  $w$ , then compare the maximum value that can be obtained by either including item  $i$  or excluding it from the knapsack and take the maximum value.
3. The final value in the bottom-right cell of the array is the maximum value that can be obtained.

## Example

no.	1	2	3					
P	10	12	28	Capacity = 6				
W	1	2	4					
				m →				
		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
n ↓	1	0	10	10	10	10	10	10
	2	0	10	12	22	22	22	22
	3	0	10	12	22	28	38	40

## Complexities

The time complexity is  $O(nW)$ , where  $n$  is the number of items and  $W$  is the maximum capacity of the knapsack.

The space complexity is also  $O(nW)$ .

## Travelling Salesperson

The Travelling Salesperson Problem (TSP) is an optimization problem that involves finding the shortest possible tour that visits all cities in a given list exactly once and returns to the starting city. The TSP is an NP-hard problem, which means that finding an optimal solution is computationally infeasible for large instances of the problem.

### Advantages

- The TSP has many real-world applications, such as route optimization for delivery trucks or planning the order of visits for a salesperson.
- Solving the TSP can be a challenging and interesting problem

### Disadvantages

- Because it is NP-hard problem finding an optimal solution to the TSP is computationally infeasible for large instances of the problem.
- The TSP can be sensitive to the initial conditions and input parameters, which can lead to



different results for the same problem instance.

- Brute force: Try all possible permutations of cities to find the shortest tour. This approach is only feasible for small instances of the problem.
- Nearest Neighbor: Start with a random city and visit the nearest unvisited city until all cities have been visited.
- Christofides Algorithm: Construct a minimum spanning tree of the cities and find a minimum weight perfect matching in the tree. Then, construct a tour that visits each city in the minimum weight perfect matching exactly once and completes the tour by visiting the remaining cities in the order given by a depth-first search of the minimum spanning tree.
- Lin-Kernighan Algorithm: This is a heuristic algorithm that iteratively improves an initial tour by exchanging pairs of edges that result in a shorter tour.

Hamiltonian Cycle

	1	2	3	4
1	0	10	15	20
2	5	0	25	10
3	15	20	0	5
4	15	10	20	0

35 55 75 35 75 75

35 55 75 35 75 75

The time complexity is  **$O(n!)$  or  $O(n^n)$** , where n is the number of cities.

The space complexity of TSP algorithms varies, with some algorithms requiring only **constant space**, while others require  $O(n^2)$  or more space, depending on the data structures used.

## Bellman Ford Algorithm

---

The Bellman-Ford Algorithm is an algorithm used to find the shortest path between a source vertex and all other vertices in a weighted graph. The algorithm works by relaxing each edge in the graph repeatedly until it finds the shortest path distances between the source vertex and all other vertices.

### Advantages

- The Bellman-Ford Algorithm can handle negative edge weights, unlike Dijkstra's Algorithm.
- The algorithm is useful in many applications such as network routing and distance vector protocols.

### Disadvantages

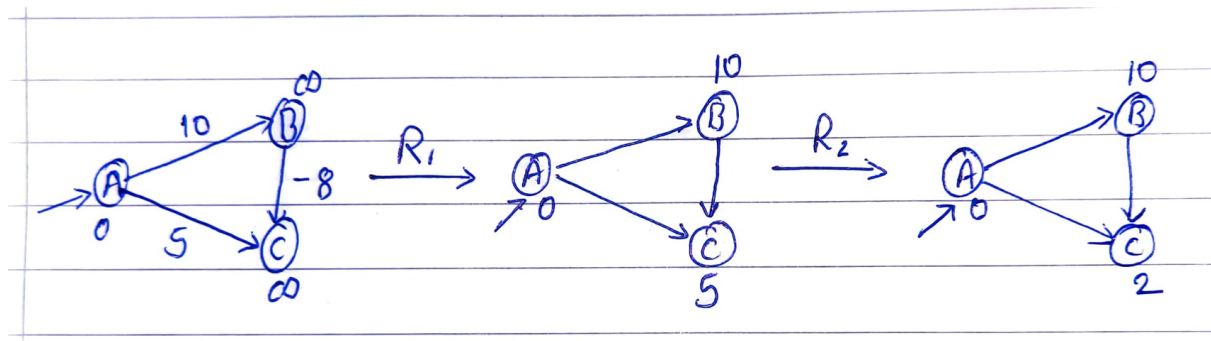
- The time complexity of the algorithm is  $O(VE)$ , which can be slow for large graphs.
- The algorithm can be prone to running into negative weight cycles, which can cause it to enter an infinite loop.

### Algorithm

1. Initialize the shortest path distances between the source vertex and all other vertices as infinity, except for the source vertex, which is initialized as 0.
2. For each edge in the graph, relax the edge by updating the shortest path distance to the destination vertex if it can be reached with a shorter path through the source vertex.
3. Repeat the above step  $V-1$  times, where  $V$  is the number of vertices in the graph.

### Example





## Complexities

The time complexity is  $O(VE)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.

The space complexity is  $O(V)$ .