The two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:

1) Overlapping Subproblems
2) Optimal Substructure

Let us discuss Optimal Substructure property here.

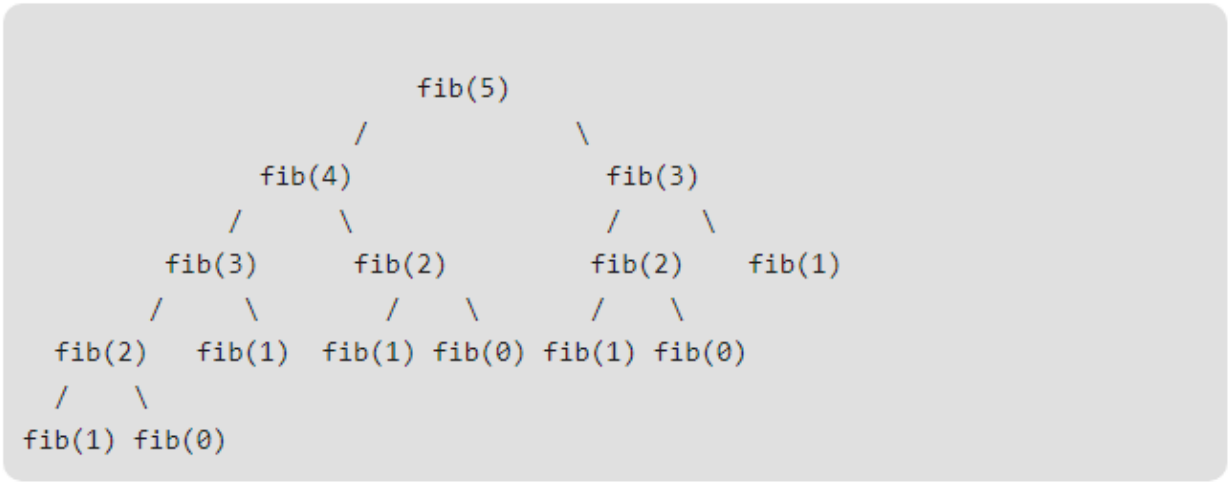
2) Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
For example, the Shortest Path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

2) Overlapping Subproblems:

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take an example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
   if ( n <= 1 )
      return n;
   return fib(n-1) + fib(n-2);
}
```

Recursion tree for fib(5)



We can see that the function fib(3) is being called 2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value.

There are following two different ways to store the values so that these values can be reused:

a) Memoization (Top Down)

b) Tabulation (Bottom Up)

a) Memoization (Top Down): The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need the solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise, we calculate the value and put the result in the lookup table so that it can be reused later.

b) Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up.

Multistage graph

Multistage graph problem can be solved by applying dynamic programming strategy. We will solve it using forward method. A multi-stage graph is a directed weighted graph and the vertices are divided into stages such that the edges connecting vertices from one stage to another stage.

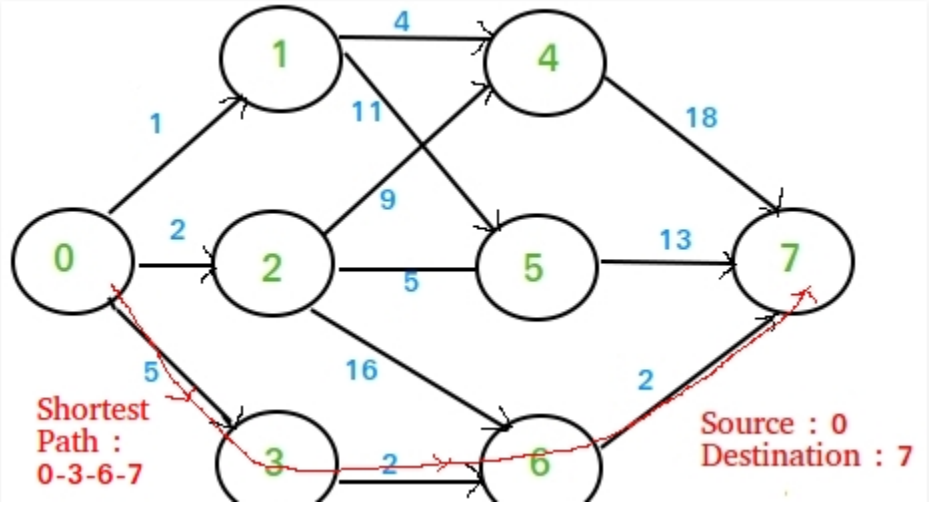
Only first stage and last stage will have just single vertex to represent the starting point that is source or ending point or sink of a graph.

OR

A Multistage graph is a directed graph in which the nodes can be divided into a set of stages such that all edges are from a stage to next stage only (In other words there is no edge between vertices of same stage and from a vertex of current stage to previous stage).

We are give a multistage graph, a source and a destination, we need to find shortest path from source to destination. By convention, we consider source at stage 1 and destination as last stage.

Following is an example of multi-graph:



This is usually useful for representing resource allocation. so here **there are various paths from source to sink we have to select the path which gives minimum cost, this is the objective of the problem,** so it's a minimization problem that is optimization problem and it can be solved using dynamic programming.

Now there are various strategies we can apply :-

- The **Brute force** method of finding all possible paths between Source and Destination and then finding the minimum. That's the WORST possible strategy.
- **Dijkstra's Algorithm** of Single Source shortest paths. This method will find shortest paths from source to all other nodes which is not required in this case. So it will take a lot of time and it doesn't even use the SPECIAL feature that this MULTI-STAGE graph has.
- **Simple Greedy Method** – At each node, choose the shortest outgoing path. If we apply this approach to the example graph give above we get the solution as 1 + 4 + 18 = 23. But a quick look at the graph will show much shorter paths available than 23. So the greedy method fails !
- The best option is Dynamic Programming. So we need to find **Optimal Sub-structure, Recursive Equations and Overlapping Sub-problems.**

Optimal Substructure and Recursive Equation :-

We define the notation :- M(x, y) as the minimum cost to T(target node) from Stage x, Node y.

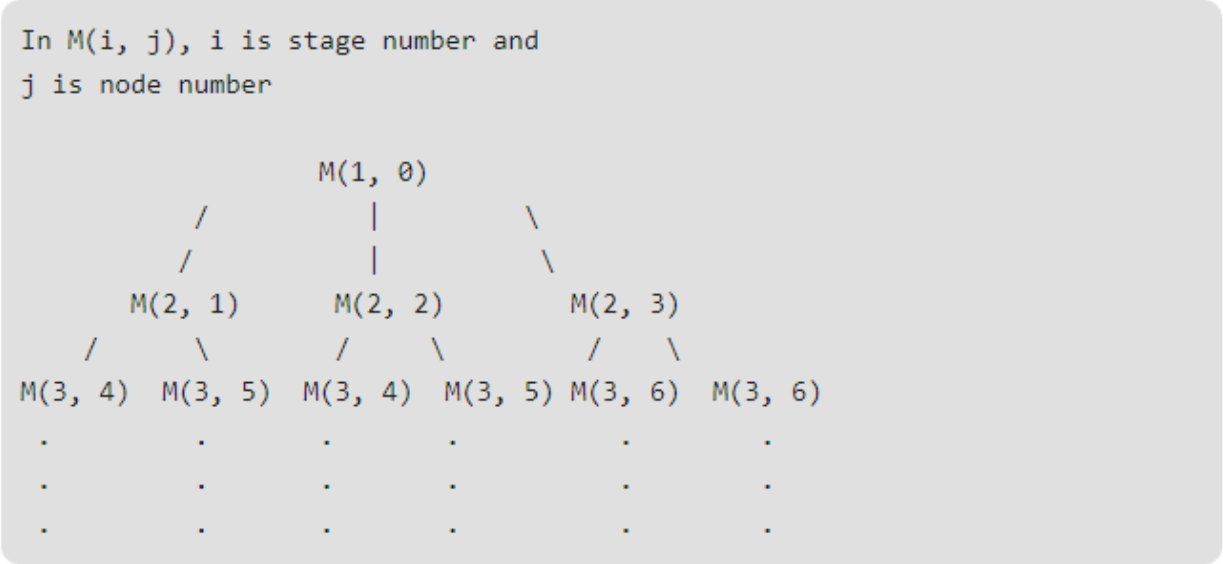
```
Shortest distance from stage 1, node 0 to
destination, i.e., 7 is M(1, 0).

// From 0, we can go to 1 or 2 or 3 to
// reach 7.
M(1, 0) = min(1 + M(2, 1),
              2 + M(2, 2),
              5 + M(2, 3))
```

This means that our problem of 0 —> 7 is now sub-divided into 3 sub-problems :-

```
So if we have total 'n' stages and target
as T, then the stopping condition  will be :-
M(n-1, i) = i ---> T + M(n, T) = i ---> T
```

Recursion Tree and Overlapping Sub-Problems:-

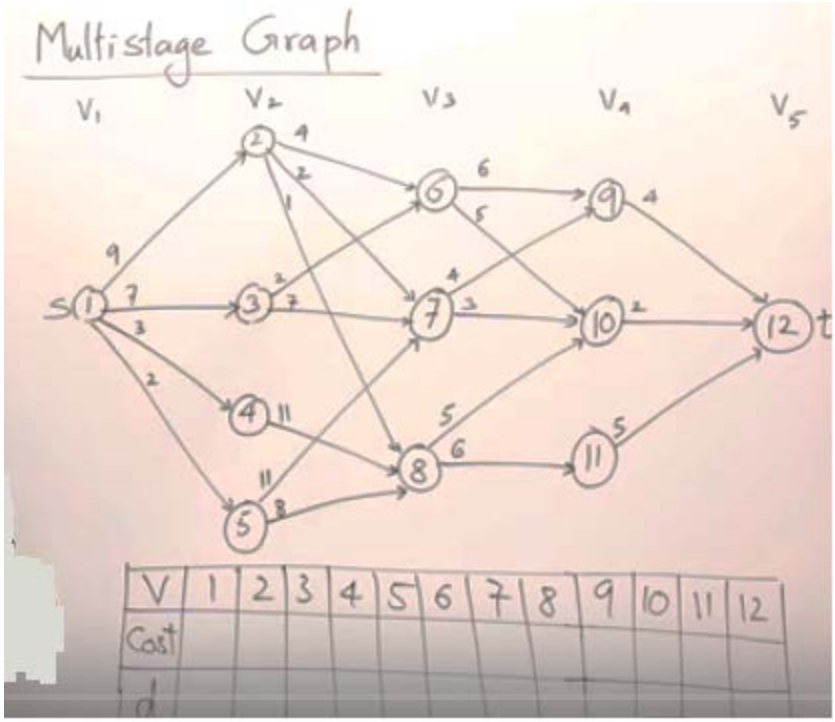
So, the hierarchy of M(x, y) evaluations will look something like this :-

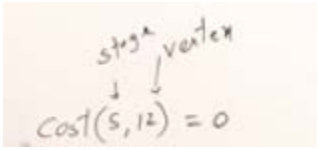


So, here we have drawn a very small part of the Recursion Tree and we can already see Overlapping Sub-Problems. We can largely reduce the number of M(x, y) evaluations using Dynamic Programming.

Let us understand whether the dynamic programming can be applied on this problem or not so dynamic programming works on the principle of optimality and it says that a problem must be solved in sequence of decisions. Let us see how we can solve this by taking sequence of decisions. For example from stage1, I select vertex 5 and should make sure what I am selecting is optimal and will give me optimal result. Again two options, either I can select 7 or 8. So in each stage, I m taking decisions, here the principle of optimality holds.

Now we apply dynamic programming procedure that is a tabulation method to get the data filled in the table. For each vertex we will find the cost and will start from last sink vertex





In d, write vertex that gives us the shortest path

Cost(5 ,12)=0

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost												0
d												12

4th Stage

Cost(4,9) =4 cost of reaching sink from vertex 9 is 4

Cost(4,10) =2

Cost(4,11) =5

From 4the stage vertices, where are we reaching? sink

V	1	2	3	4	5	6	7	8	9	10	11	12
Cost						7	5	7	4	2	5	0
d						10	10	10	12	12	12	12

3rd Stage: find cost of vertices 6,7 and 8

Cost/weight of edge+ cost of vertex 9 at 4th stage

Cost (3,6) = min { cost(6,9)+ cost(4,9), cost(6,10) + cost(4,10)}

= min { 6+4, 5+2} = 7

Vertex 10 has given smaller value/minimum cost, so put 10 in d against 6.

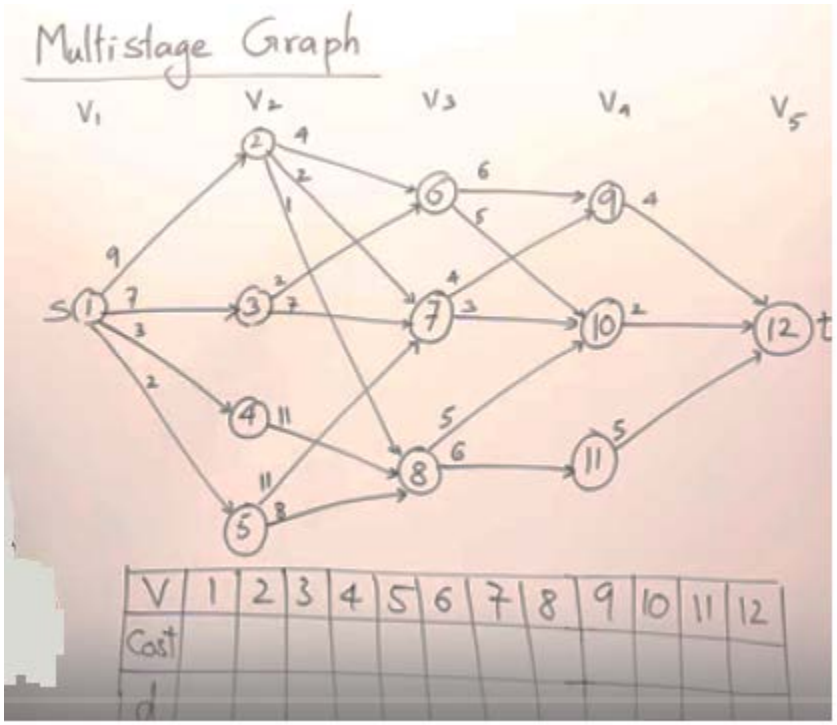
Cost of 9 is already found, so it is useful for vertex 6 and 7, so this what overlapping subproblems are there in dynamic programming

Cost (3,7) = min { cost(7,9)+ cost(4,9), cost(7,10) + cost(4,10)}

= min { 4+4, 3+2} = 5

Cost (3,8) = min { cost(8,10)+ cost(4,10), cost(8,11) + cost(4,11)}

= min { 5+2, 6+5} = 7



V	1	2	3	4	5	6	7	8	9	10	11	12
Cost		7	9	18	15	7	5	7	4	2	5	0
d		7	6	8	8	10	10	10	12	12	12	12

2nd Stage: find cost of vertices 2,3 and 4 and 5

Cost (2,2) = min { cost(2,6)+ cost(3,6), cost(2,7) + cost(3,7), cost(2,8) + cost(3,8)}

= min { 4+7, **2+5**, 1+7} = 7

Cost (2,3) = min { cost(3,6)+ cost(3,6), cost(3,7) + cost(3,7)}

= min { **2+7**, 7+10} = 9

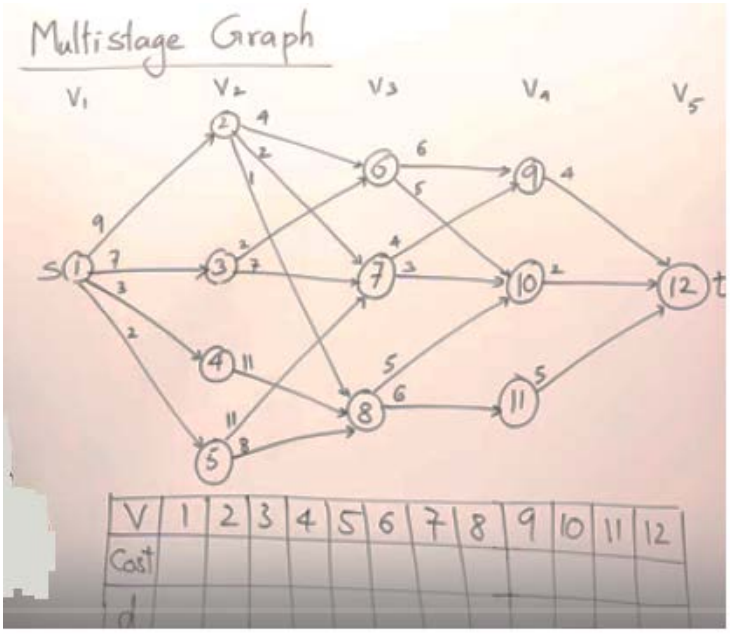
Cost (2,4) = min { cost(4,8)+ cost(3,8)}

= min { **11+7**} = 18

Cost (2,5) = min { cost(5,7)+ cost(3,7), cost(5,8) + cost(3,8)}

= min { **11+5, 8+7**} = 15

Now we are at stage 1 to find the minimum cost of vertex 1(starting vertex) to reach sink, i.e the min cost from source to sink.



V	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	5	0
D	2/3	7	6	8	8	10	10	10	12	12	12	12

Cost (1,1) = min { cost(1,2)+ cost(2,2), cost(1,3) + cost(2,3), cost(1,4) + cost(2,4),
cost(1,5) + cost(2,5) }

= min { 9+7, **7+9**, 3+18, 2+15} = 16

First filling their smaller values then getting the answer for the larger value.
We started from the backside of our multistage graph i.e from last stage onwards where we started but still this is called as forward method

Formula for Multi-stage graph

Cost(i, j)= min { cost (j , l) + cost(i+1, l) } where j, l belongs to set of integers
L belongs to next stage i.e $l \subseteq v_{i+1}$

Now actual dynamic programming starts, we have the formula and filled up the table based on the dynamic programming or the dynamic programming approach to get the result. But the method of dynamic programming solves a problem by taking sequence of decisions, we have not taken in a decision so far but now we are going to take the sequence of decision based on the data available.
We are taking d values to make decisions, we will start from vertex1 onwards to sink. Decisions will be taken in forward direction

D (stage, vertex)
D(1,1)= 2

Second stage vertex 2
D(2,2)=7 , 7 has given min result for vertex 2

Third stage vertex 7
D(3,7)=10

Fourth stage vertex 10

D(4,10)=12

Path is 1---2-----7----10-----12

D(1,1)=3 , d(2,3)= 6, d(3, 6)=10, d(4, 10)= 12

Path2: 1-----3----------6----10-------12

Multi-Stage Graph Problem

It's a special graph where nodes are arranged in stages, and you must find the shortest or cheapest path from a start node to an end node stage by stage.

Real-World Applications:

1. Shortest Path Problems (e.g., GPS Navigation)

Think of a trip planner that divides your journey into stages:

Stage 1: Start city → Midway city

Stage 2: Midway city → End city

Multi-stage graph helps find the fastest or cheapest path by evaluating options at each stage.

2. Project Scheduling (like PERT/CPM)

Imagine managing a software project. Tasks are arranged in stages:

Stage 1: Requirement gathering

Stage 2: Design

Stage 3: Development

Stage 4: Testing

Each task has a cost/time. The multi-stage graph helps you find the minimum time path from start to end.

3. Dynamic Programming Examples

The multi-stage graph is used in problems like matrix chain multiplication or shortest path in layered graphs, where a problem is broken into steps and solved step by step.

4. Speech Recognition / Natural Language Processing

In speech-to-text systems, the spoken sentence is analyzed stage-by-stage (word-by-word).

At each stage, the system chooses the most likely interpretation of that part based on the best path.

5. AI Decision Making

In robotics or games, agents often make decisions in stages (e.g., move → aim → shoot).

Each stage has multiple options. Multi-stage graph helps choose the sequence with best outcome.