Design Technique: Dynamic Programming

Dynamic Programming

- *Dynamic programming (DP)* is an algorithm design technique (*like divide and conquer*) used to solve a wide variety of *optimization problems* such as scheduling, travelling salesman, packaging, and inventory management, etc.

- In *divide and conquer* approach, *problem* is *divided* into *independent sub-problems* which are recursively solved, while in *DP, sub-problems* are not *independent*, e.g., they *share* the *same sub-problems/ repeating subproblems*.

- DP is a useful technique for solving *optimization problems* that can be divided into *smaller subproblems* with *optimal substructure* and *overlapping subproblems*.

- DP solves *each sub-problems just once* and *stores the result in a table* so that it can be *repeatedly* retrieved if needed again.

- It is called *dynamic* since it *decide dynamically* whether to *call function* or *use table*.

Development of Dynamic Programming Algorithm

Dynamic Programming works when a problem has the following features:-

Optimal Substructure: If an optimal solution contains optimal sub solutions, then a problem exhibits optimal substructure.

Overlapping sub-problems: When a recursive algorithm would visit the same sub-problems repeatedly, then a problem has overlapping sub-problems.

Elements of Dynamic Programming-

1. **Substructure:** Decompose the given *problem* into *smaller sub-problems*. Express the *solution* of the *original problem* in terms of the *solution for smaller problems.*
2. **Table Structure:** After *solving* the *sub-problems*, store the *results* of the *sub problems* in a *table/ array*. This is done because *sub-problem solutions* are *reused many times*, and we *do not want* to *repeatedly solve* the *same problem over and over again.*

There are two ways to attain the above properties-

- Memoization
- Tabulation

DP will only be applicable to any problem, if we can represent problem recursively. Let us understand the concept of DP using Fibonacci series problem.

Fibonacci sequence using Recursion

The Fibonacci numbers F_n are defined as follows:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{(n-1)} + F_{(n-2)}$$

```
FIB (n)
    If (n < 2)
    then return n
    else return FIB (n – 1) + FIB (n – 2)
```

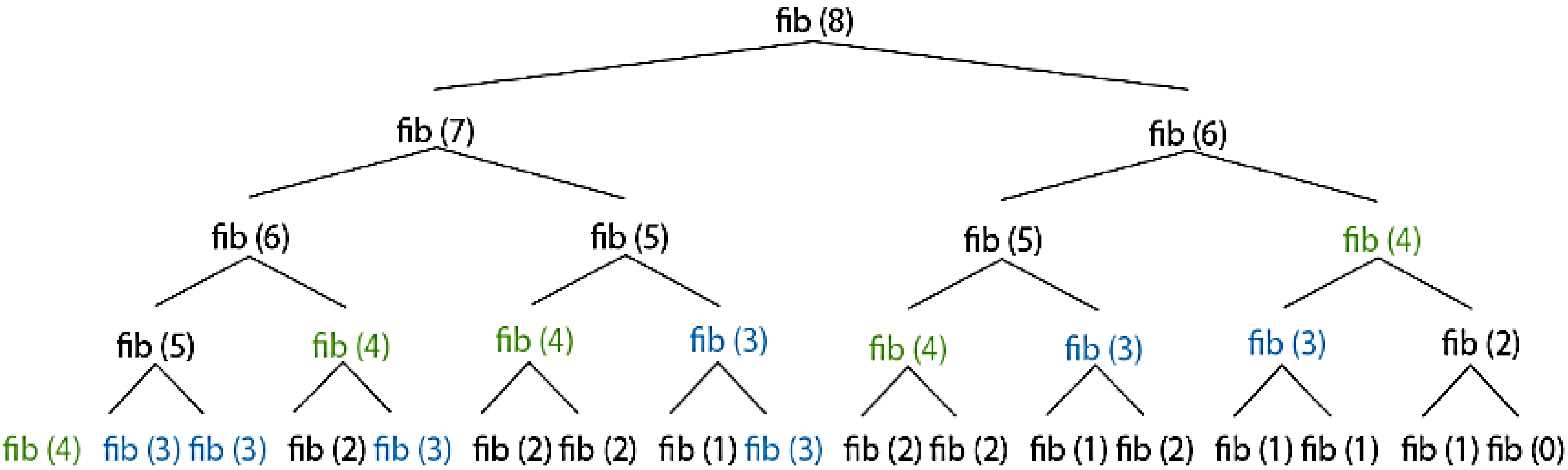


Figure: Recursive calls during computation of Fibonacci number

Time complexity =?

Fibonacci sequence using Recursion

$F_0 = 0$

$F_1 = 1$

$F_n = F_{(n-1)} + F_{(n-2)}$

```
FIB (n)
  1.If (n < 2)
  2.then return n
  3.else return FIB (n - 1) + FIB (n - 2)
```

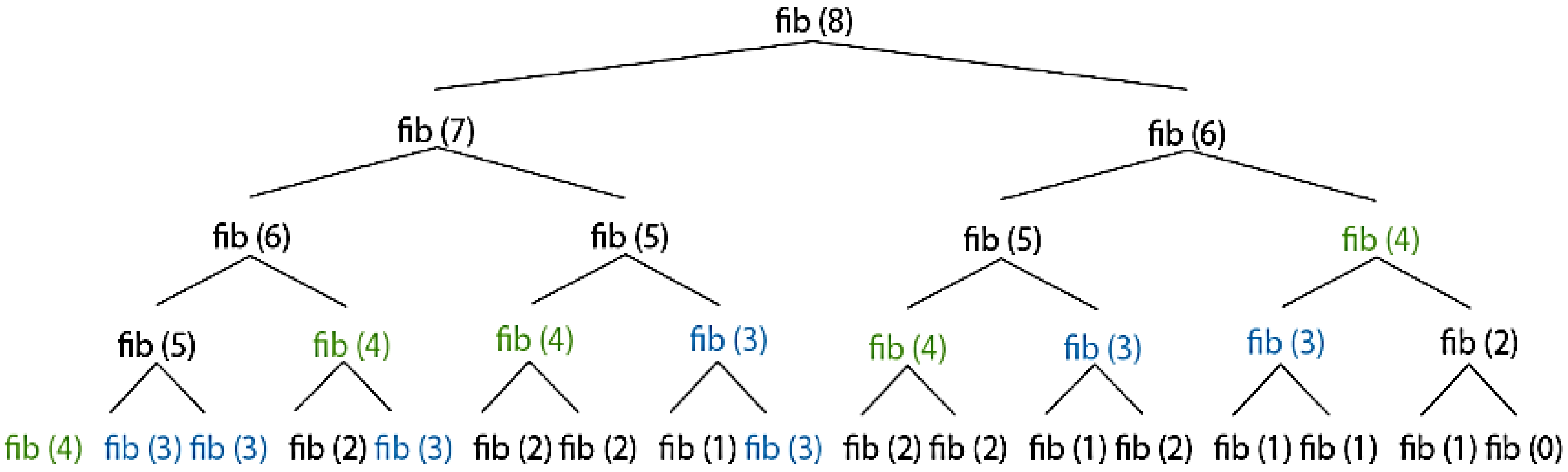


Figure: Recursive calls during computation of Fibonacci number

Recurrence relation: $T(n) = 2T(n-1) +1$,

So, the number of function calls made to compute **Fib(n) = O(2^n /2) = O(2^n)** using the master method. There are also many repeating/ overlapping subproblems.

Fibonacci sequence using Recursion

How to reduce number of function call and time complexity?

Fibonacci sequence using DP

How to reduce number of function call and time complexity?

DP can be used to reduce the time complexity. DP will only be applicable, if we can find optimal substructure and overlapping subproblems.

It has been observed from the Fibonacci example that larger problem can only be solved if we have solution of smaller similar subproblems. It means, optimal substructure has been found.

Besides, there are many overlapping subproblems. So, We can use DP. In DP, solution of subproblems are stored in *table/array*.

Two ways the solution of sub-problem can be stored so that it can be reused-

- Memoization Method
- Tabulation Method

Fibonacci sequence : Memoization

Memorization:

In memorization, a global array of size *n+1 (for unique sub-problems)* is defined to store results of subproblems. The array is initialized with *-1*. In this approach, recursion tree is traversed in top-down manner to fill the array *A[n+1]*. The steps of memorization approach is given below-

If we trace through the recursive calls to MEMOFIB, we find that array A [n] gets filled from top down.

Algorithm with memorization

```
MEMOFIB (n)
    if (n < 2)
          return n
       if (A[n] !=-1)
          return A[n]
     else
          return A[n] = MEMOFIB (n - 1) + MEMOFIB (n - 2)
```

Fibonacci sequence : Memoization

Memoization method

This algorithm clearly takes only O(n) time to compute Fib (n).

Memoization approach reduces the time complexity from 2^n to n.

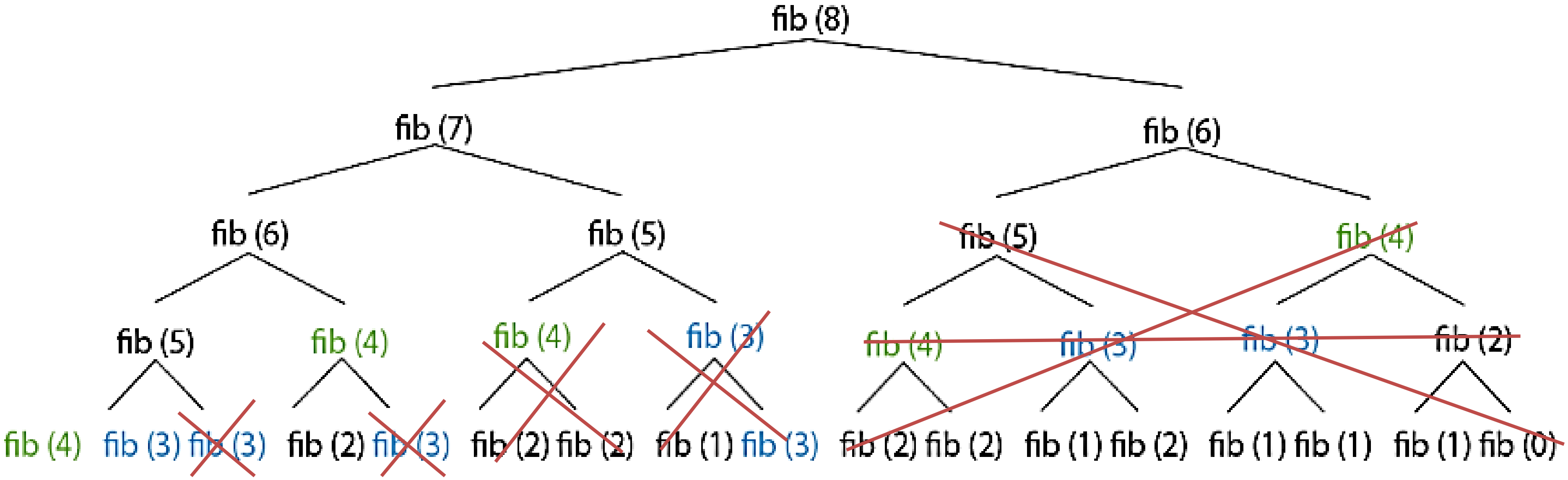


Figure: Recursive calls during computation of Fibonacci number

Fibonacci sequence : Tabulation Method

- This approach defines a *table/array* of *size n+1* and *fills* the *table/array* from *its starting index* i.e., *0^{th} index* to the *last index*. It means the *base cases* are *first filled* and then the *next higher indexes* are filled.
- This approach is also known as *bottom-up method* for solving DP problems, since it first fills the values of function calls present at the last level (Base cases).
- In the tabulation method, the problem is iteratively defined to compute the overlapping subproblems only once.

$$
\begin{array}{ll}
F_0 = 0, & \text{if } n=0 \\
F_1=1 & \text{if } n=1 \\
F_n=F_{(n-1)}+ F_{(n-2)}, & \text{if } n>=1
\end{array}
$$

Fibonacci sequence: Tabulation Method

Tabulation Method:

```
int A[n+1] ={-1};
int FIB_tab (int n)
     {
    A[0] = 0;
    A[1] = 1;
    for (int i=2; i<=n; i++)
      {
       A[i] = A[i-1]+A[i-2];
      }
    return A[n]
      }
```

There exist only *n+1* unique subproblems for the Fibonacci series. So, the number of subproblems called will be *O(n+1)*. Thus, this algorithm takes only O(n) time to compute Fib (n).

Memoization or Tabulation : Which one is better

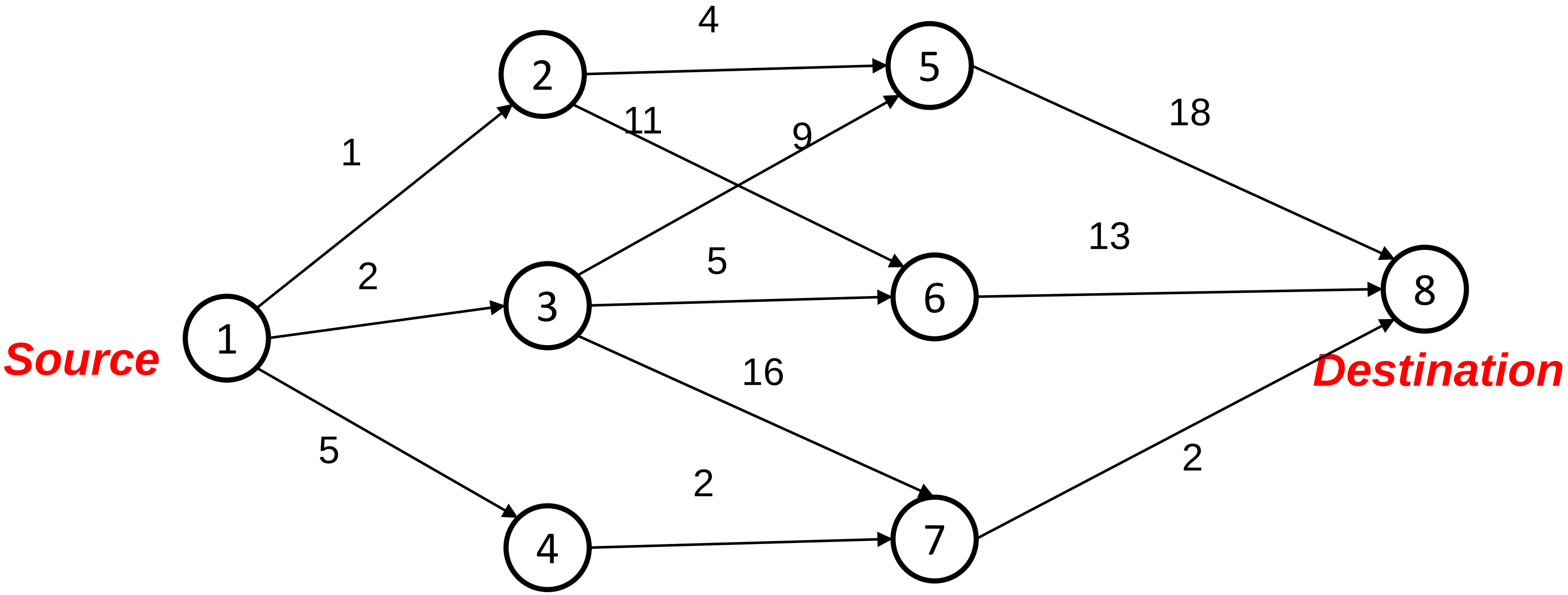
- Tabulation is often faster than memoization because it is iterative and solving subproblems requires no overhead.

- Memoization algorithms are easier to understand and implement but they can cause the stack overflow (SO) error.

- Therefore, Tabulation method is normally used in DP approach.

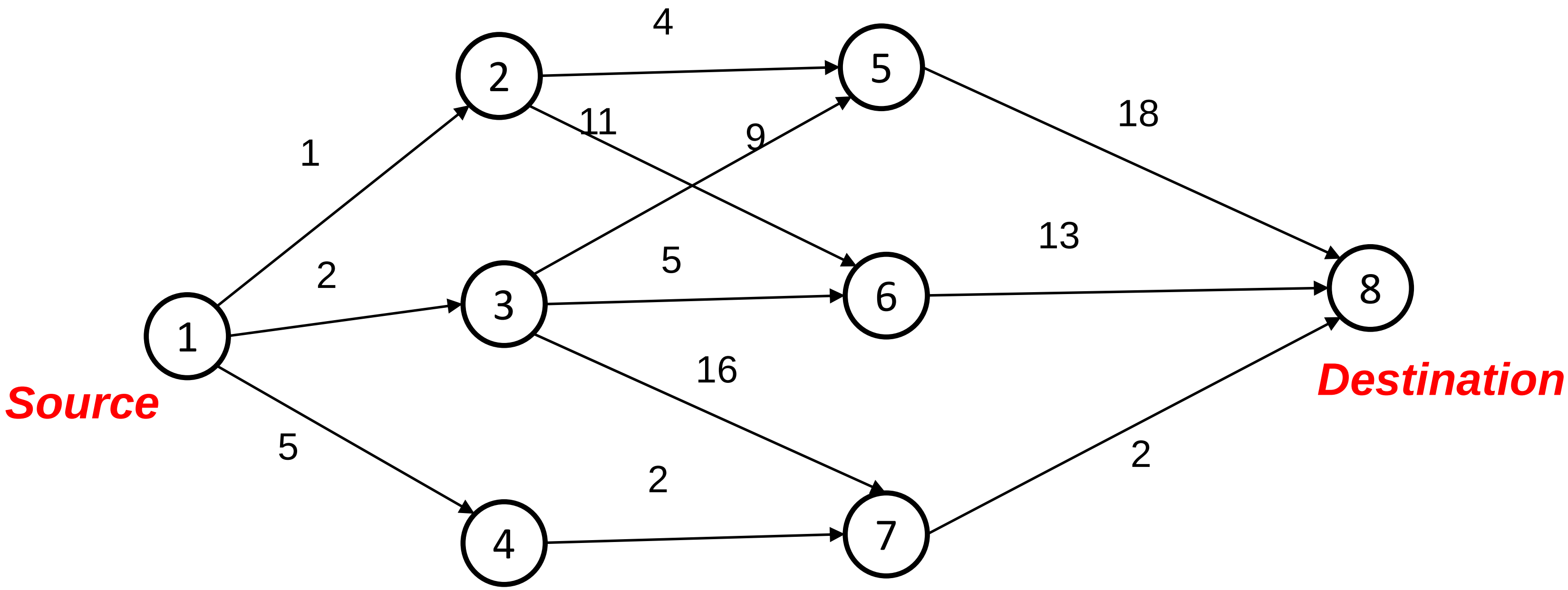
Multistage Graph (Shortest Path)

A multistage graph *G=(V,E)* is a directed graph, in which all the vertices are partitioned into the *k stages* where *k>=2*. There is *no edge* between vertices of within same stage and from a vertex of *current stage* to *previous stage*.



Multistage Graph (Shortest Path)

Find the ***shortest path*** from ***source to sink/ destination*** vertex for a multistage **graph G=(V,E).**



Multistage Graph (Shortest Path)

Problem: Find the ***shortest path*** from ***source to sink/ destination*** vertex for a multistage **graph G=(V,E).**

- In the *brute force* approach, *all paths* from *source to destination* are *identified* and their *costs are computed* to find the *shortest path*. This approach would be *inefficient* because there will be *2^n possibilities* for a graph having *n vertices*.
- *Dijkstra algorithm* can also be *one way* to find the shortest path, but Dijkstra algorithm will *generate shortest path* for *all vertices,* which is not our *objective.*
- **Is there any other efficient approach that can be used to find the optimal solution?**
    - DP may be used, however, DP will only applicable if we should-
        - Able to find the optimal substructure / Able to write recursive equation
        - Able to find overlapping subproblems
- So, Let us define a recursive equation to find the optimal substructure and repeating/ overlapping subproblems.

Multistage Graph -Shortest Path

- If we want to find the shortest path from *1 to 8* then, problem can be broken into subproblems as follows-
  - We can go from *1 to 2* and find the *shortest path* from *2 to 8*, or
  - We can go from 1 to 3 and find the *shortest path* from *3 to 8*, or
  - We can go from *1 to 4* and find the shortest path from *4 to 8*
- The above statements can be formulated as follows-

$$\text{min(1, 1/S)} = \text{min} \begin{cases} \text{1/S-> 2 + min (2, 2)} \\ \text{1/S ->3 + min (2, 3)} \\ \text{1/S->4 + min (2, 4)} \end{cases}$$

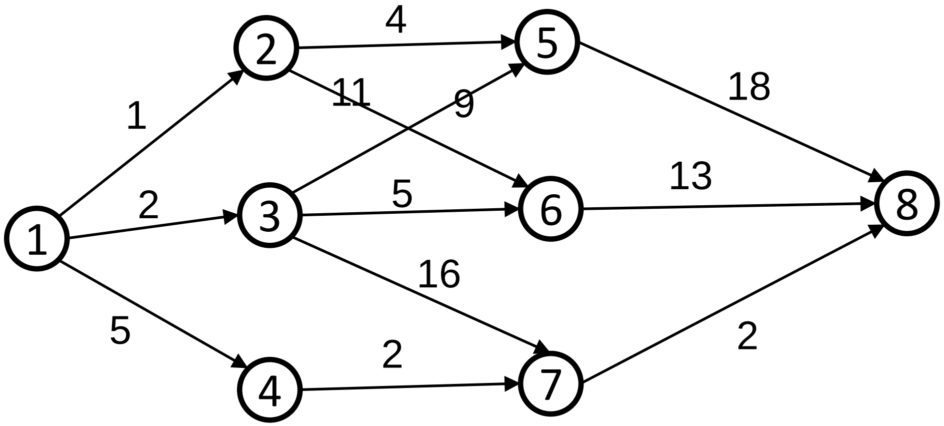
min(n-1, i) = cost(i->T), where *T* is sink/target, *n* is no. of stages

Base case

Min. cost to go from stage 1 and node 1 to sink

Recursive step

Using the formula given above, the *stage1 problem* can be converted into *stage 2* and *stage 2* to *stage 3* and ... *recursively*.



Multistage Graph -Shortest Path

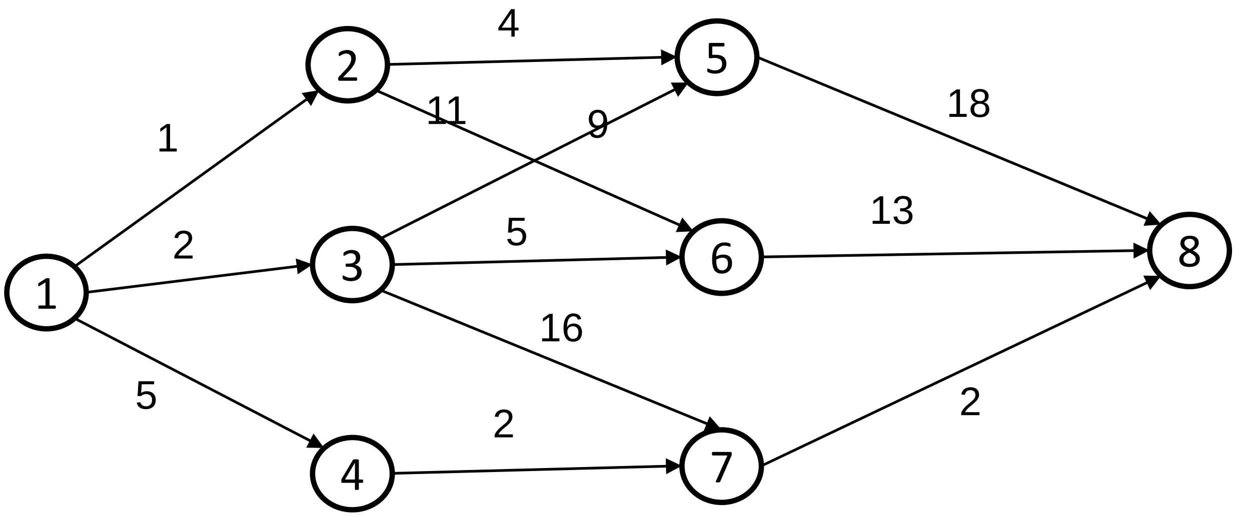
- We know that DP will only be beneficial if problem has optimal sub-structure and overlapping sub-problem.
- So, lets use the following formula and draw the recursion tree to identify the overlapping sub-problems.

$$\min(1, 1/S) = \min \begin{cases} 1/S\text{->}\ 2 + \min(2, 2) \\ 1/S\ \text{->}3 + \min(2, 3) \\ 1/S\text{->}4 + \min(2, 4) \end{cases}$$

Recursive step

min(n-1, i) = cost(i->T), where T is sink/target, n is no. of stage

Base case



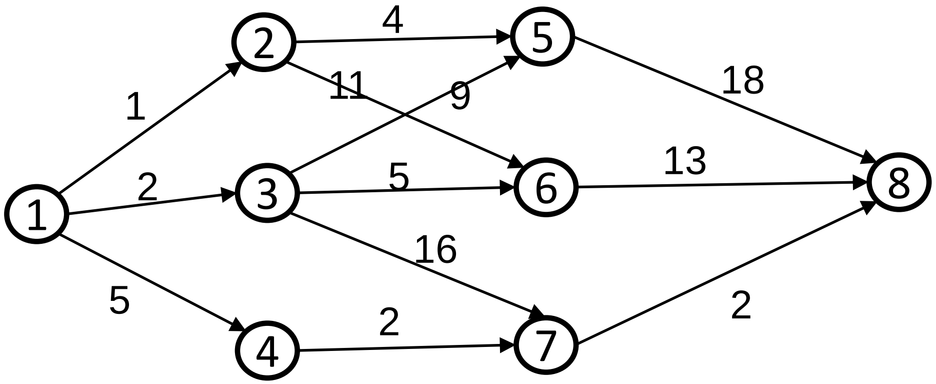
Multistage Graph -Shortest Path

min(1, 1/S) = min
- 1/S-> 2 + min (2, 2)
- 1/S ->3 + min (2, 3)
- 1/S->4 + min (2, 4)

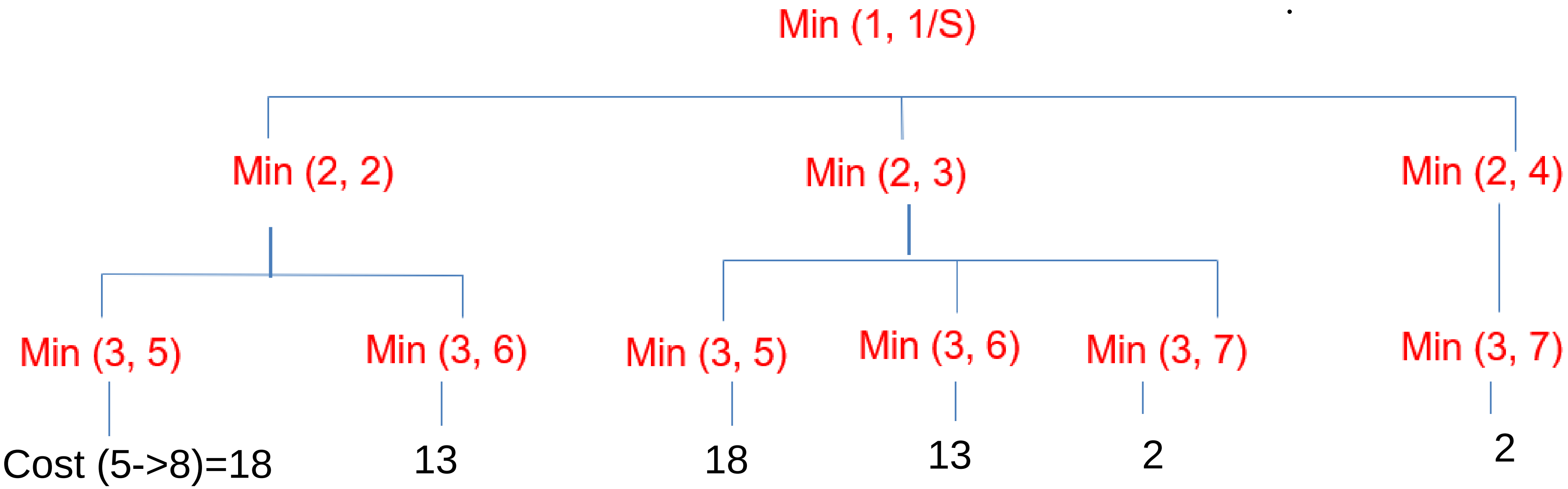
Recursive step

min(n-1, i) = cost(i->T), where T is sink/target, n is no. of stage

Base case



Recursion tree

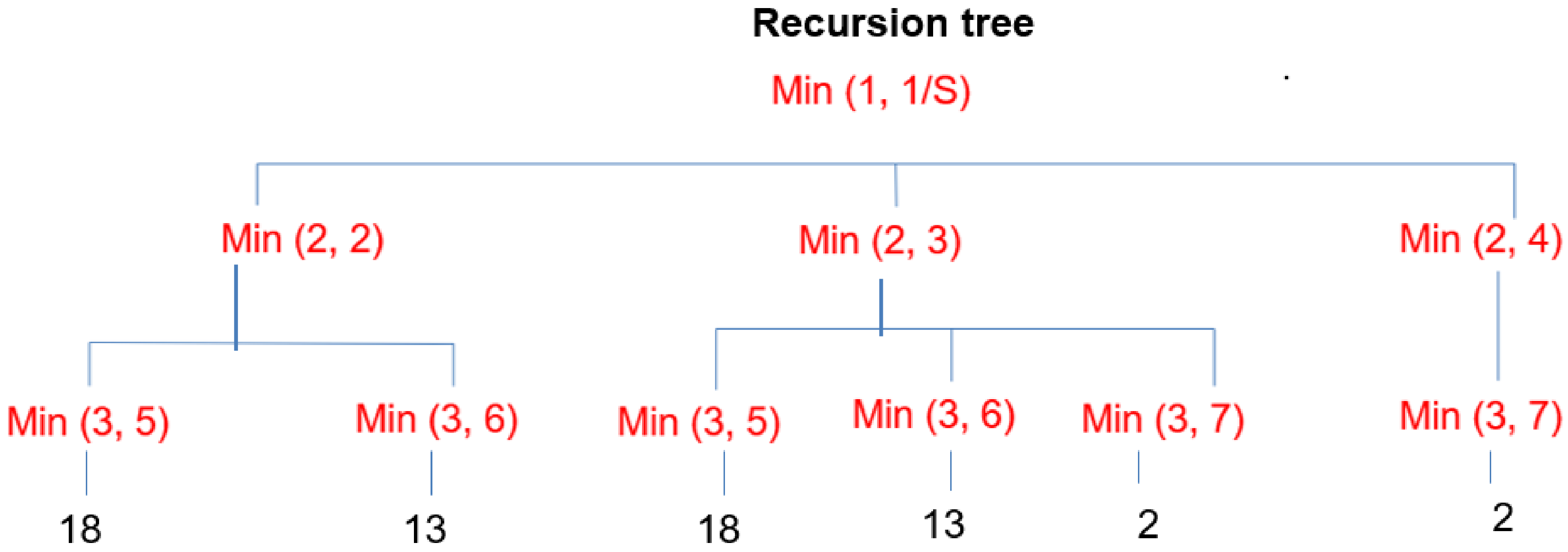

Min (1, 1/S)

Min (2, 2)
Min (2, 3)
Min (2, 4)

Min (3, 5)
Min (3, 6)
Min (3, 5)
Min (3, 6)
Min (3, 7)
Min (3, 7)

Cost (5->8)=18
13
18
13
2
2

Multistage Graph -Shortest Path



- The depth of the tree is equal to the number of stages, i.e., *k*. So, space complexity will *O(K),* but at each level there may be *n* vertices and total there could be k^n vertices in worst case. So, time complexity will be exponential.
- If we explore the recursion tree then we could notice, there are *many overlapping subproblems*. So, we can use *DP to save the results in a table* so that overlapping subproblems cannot be computed in each call and it reduces the time complexity.
- In DP, only *distinct subproblems* are *computed* and their *result are stored in a table*. For this *problems*, number of *distinct sub-problems* are equal to *number of vertices*. So, we will *define an array* whose *size will be equal to number of vertices* to *store cost from a source vertex to the target.*

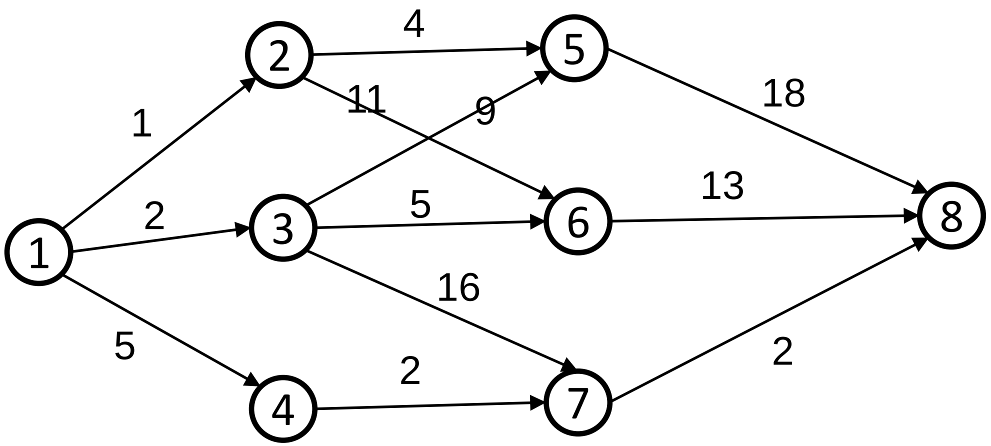
Multistage Graph -Shortest Path

$$\text{min(1, 1/S) = min} \begin{cases} \text{1/S-> 2 + min (2, 2)} \\ \text{1/S ->3 + min (2, 3)} \\ \text{1/S->4 + min (2, 4)} \end{cases}$$

Recursive step

min(n-1, i) = cost(i->T), where T is sink/target, n is no. of stage

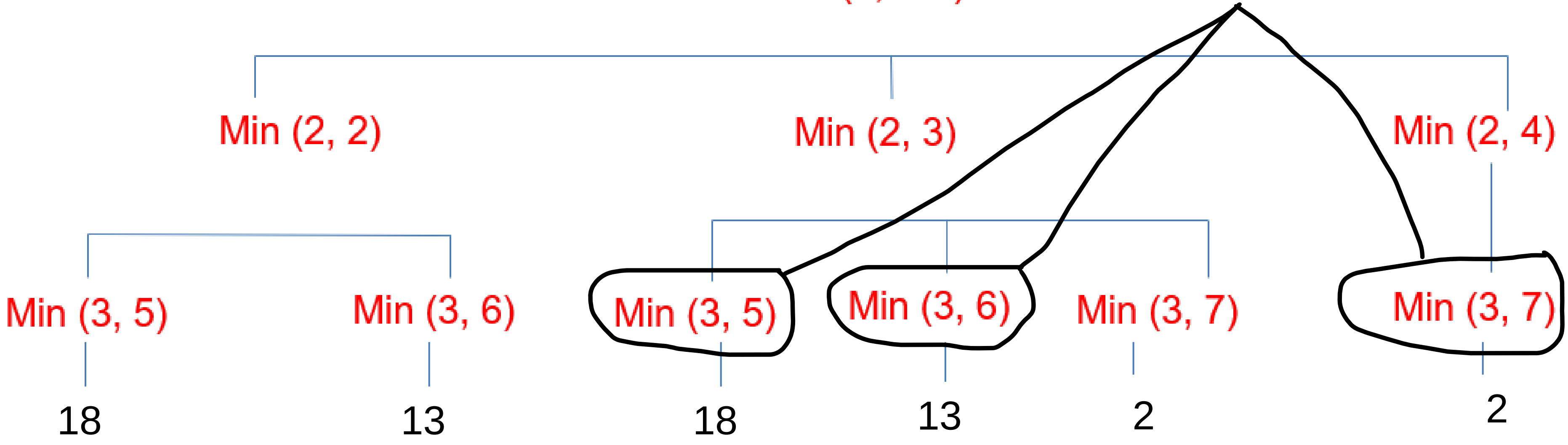
Base case



Recursion tree

Overlapping sub-problems

Min (1, 1/S)



Min (2, 2)

Min (2, 3)

Min (2, 4)

Min (3, 5)

Min (3, 6)

Min (3, 5)

Min (3, 6)

Min (3, 7)

Min (3, 7)

18

13

18

13

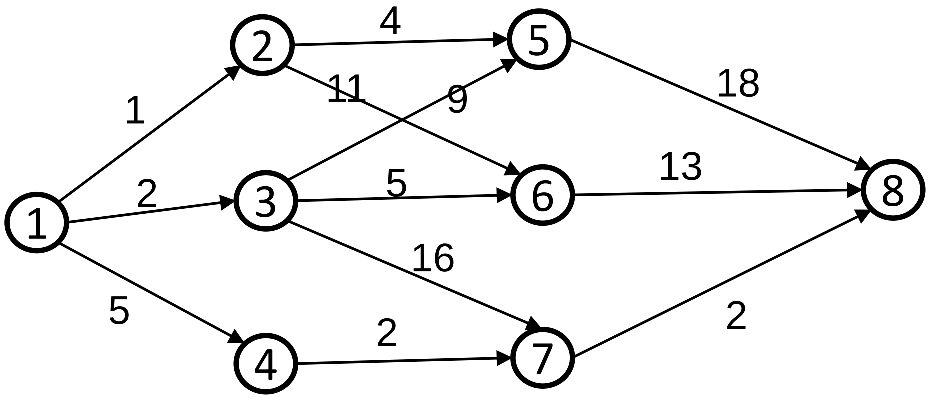
2

2

Multistage Graph -Shortest Path-bottom up Approach

Problem: Find the shortest path from source to sink/ destination vertex for a multistage graph G=(V,E).

$$T[i] = \underset{j=(i+1) \text{ to } n}{\text{min of all}} \left\{ \text{Edge cost } (i, j) + T[j] \right.$$



Define an array T of size |v| and initialize T[8] =0, since cost (8->8) =0

							0
1	2	3	4	5	6	7	8

Compute T[7]

T[7] = min{(Edge cost (7, 8) + T[8])}

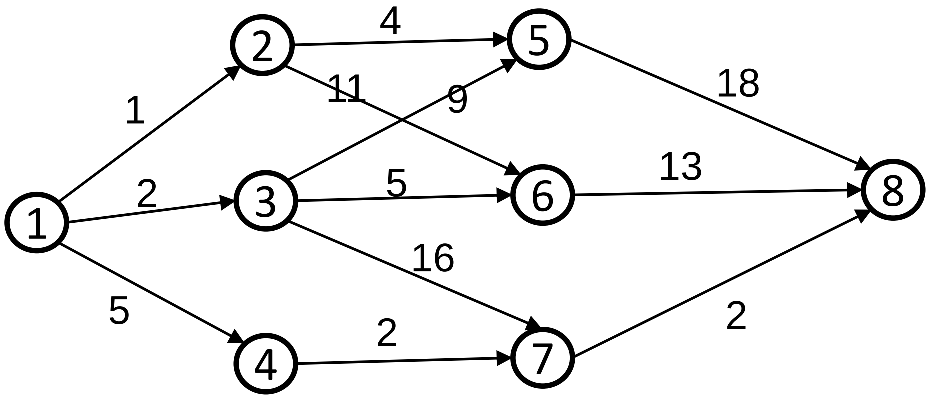
T[7] = min{(2+0)} =2

						2	**0**
1	2	3	4	5	6	7	8

Multistage Graph -Shortest Path-bottom up Approach

Problem: Find the shortest path from source to sink/ destination vertex for a multistage graph G=(V,E).

$$T[i] = \min_{j=(i+1) \text{ to } n} \left\{ \text{Edge cost } (i, j) + T[j] \right.$$



Compute T[6]

T[6] = min{(Edge cost (6, 7) + T[7], Edge cost (6, 8) + T[8])}

T[6] = min{(∞+2, 13+0)} =13

					13	2	0
1	2	3	4	5	6	7	8

Compute T[5]

T[5] = min{(Edge cost (5, 6) + T[6]), Edge cost (5, 7) + T[7]), Edge cost (5, 8) + T[8])}

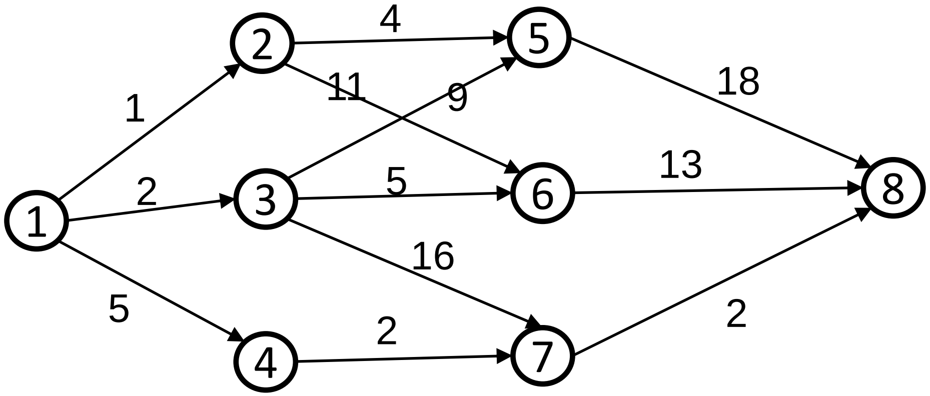
T[5] = min{(∞+13, ∞+2, 18+0)} =18

				18	13	2	0
1	2	3	4	5	6	7	8

Multistage Graph -Shortest Path-bottom up Approach

Problem: Find the shortest path from source to sink/ destination vertex for a multistage graph G=(V,E).

T[i] = min of all j=(i+1) to n { Edge cost (i, j) + T[j]



Compute T[4]

T[4] = min{(Edge cost (4, 7) + T[7]), (Edge cost (4, 6) + T[6])}

T[4] = min{(2+2), (∞+13) } =4

			4	**18**	**13**	**2**	**0**
1	2	3	4	5	6	7	8

Compute T[3]

T[3] = min{(Edge cost (3, 7) + T[7]), (Edge cost (3, 6) + T[6]), (Edge cost (3, 5) + T[5]) }

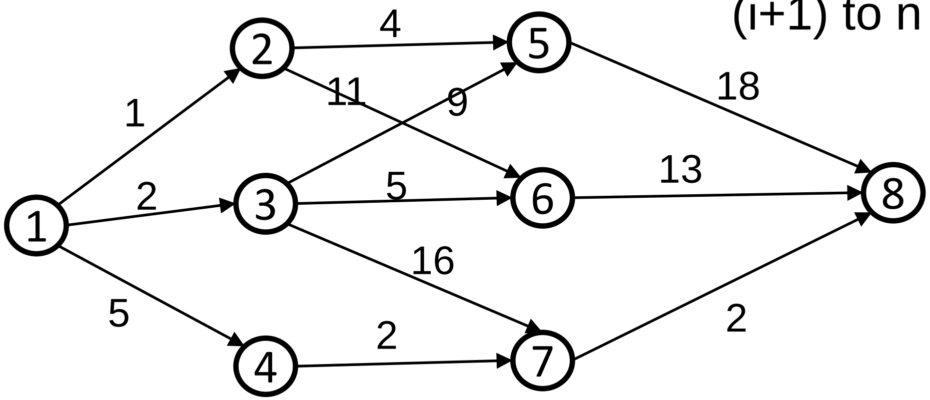
T[5] = min{(16+2), (5+13), (9+18) } =18

		18	**4**	**18**	**13**	**2**	**0**
1	2	3	4	5	6	7	8

Multistage Graph -Shortest Path-bottom up Approach

Problem: Find the shortest path from source to sink/ destination vertex for a multistage graph G=(V,E).

$$T[i] = \begin{matrix}\text{min of all j=}\\ \text{(i+1) to n}\end{matrix} \left\{ \text{Edge cost (i, j) + T[j]} \right.$$



Compute T[2]

T[2] = min{(Edge cost (2, 5) + T[5]), (Edge cost (2, 6) + T[6])}

T[2] = min{(4+18), (11+13) } =22

Compute T[1]

T[1] = min{(Edge cost (1, 2) + T[2]), (Edge cost (1, 3) + T[3]), (Edge cost (1, 4) + T[4])}

T[1] = min{(1+22), (2+18), (5+4)} =9

So, minimum cost to reach 8 from 1 will be 9.

9	**22**	**18**	**4**	**18**	**13**	**2**	**0**
1	2	3	4	5	6	7	8

Multistage Graph -Shortest Path-bottom up Approach

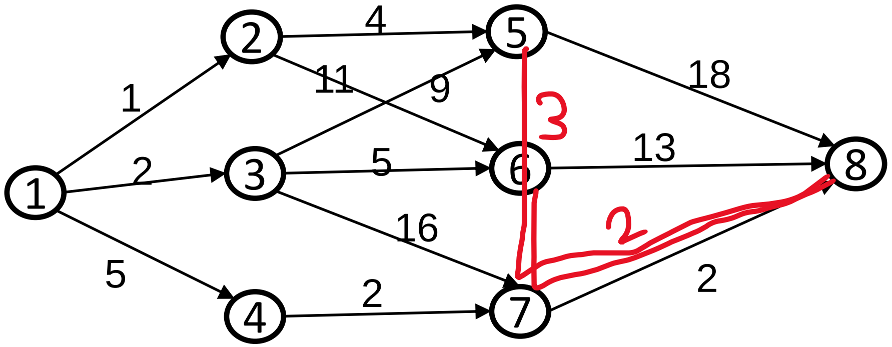
Time complexity:

Total time/ work done = Number of subproblems * time to solve each subproblems

Number of subproblems solved to get the shortest path *= n*

Overall work done = number of times functions will be evaluated at each stage

9	**22**	**18**	**4**	**18**	**13**	**2**	**0**
1	2	3	4	5	6	7	8
↓	↓	↓	↓	↓	↓	↓	↓
O(7)	O(6)	O(5)	O(4)	O(3)	O(2)	O(1)	O(1)



Overall work done=

1+1+2+3+…+n-1

n-1 (n-2)/2 = n^2

$O(n^2) = O(V^2) = O(E)$

0/1 Knapsack Problem

Given a set of items, each having different weight and value/ profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value/ profit earned is as large as possible.

0/1 Knapsack Problem

Given a set of items, each having different weight and value, or profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible.

Suppose we have **n** items:

1, 2, 3, 4, 5, …, n

Then there will **2** choices for each item whether it can be included or not included in the knapsack. So, total number of possible solutions would be **$= 2x2x2x2x \ldots x2 = 2^n$**

Total **2^n** solutions need to be explored to find the optimal solution using brute force approach.

Is there any other efficient approach that can be used to find the optimal solution?

- DP may be used, however, DP will only applicable if we should-
  - Able to find the optimal substructure / Able to write recursive equation
  - Able to find overlapping subproblems
- So, Let us define a recursive equation to find the optimal substructure and repeating/ overlapping subproblems.

0/1 Knapsack Problem

$$T(i, j) = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T(i-1, j) & wt[i] > j \\ \max[T(i-1, j-wt[i])+ P[i], T(i-1, j)] & \text{otherwise} \end{cases}$$

If there is no elements *i=0* or knapsack capacity *j= 0*, nothing can be included.

In above equation,

- *wt [i]* is the capacity of the object *i* and,
- j is the remaining/ available capacity. If none of the item is included in the knapsack, j will equal to the knapsack capacity.

There will be two possibility for each item, whether it is included in knapsack or not.

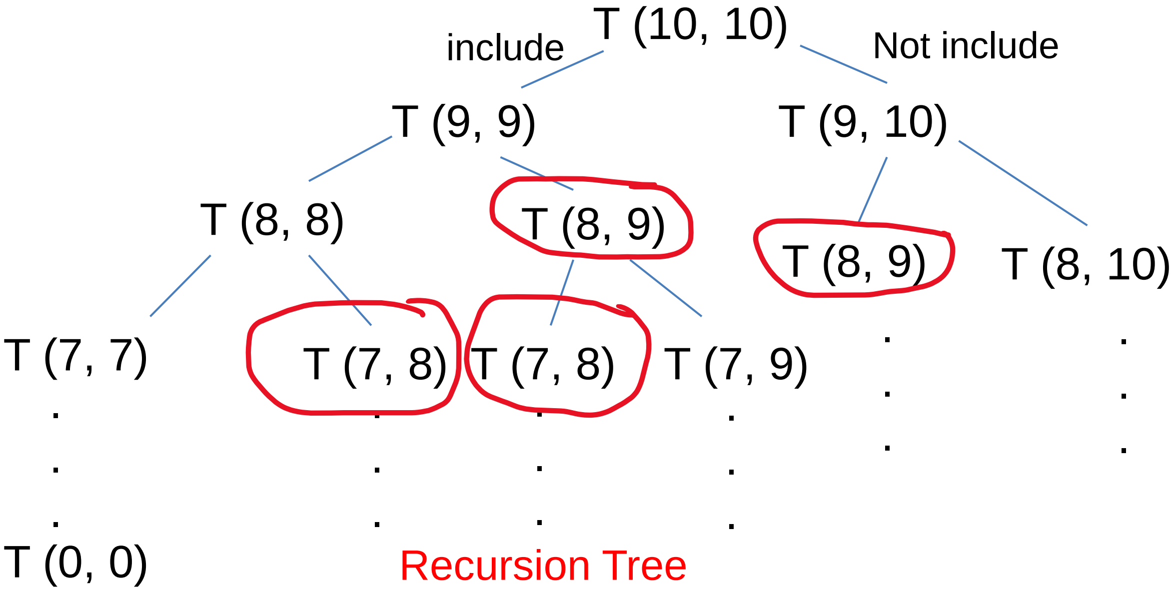
- If an object *i* is chosen, then profit *P[i]* is added and for the remaining *i-1* elements, max profit need to be identified. So, there will be a recursive call for remaining *i-1* elements with remaining knapsack capacity i.e., *j- wt [i].*

- If *i* is not chosen, then profit will not increase and object will not be included in knapsack, so there will be a recursive call for remaining *i-1* elements. The choice that gives the max profit will be selected.

0/1 Knapsack Problem

$$T(i, j) = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T(i-1, j) & wt[i] > j \\ \max[T(i-1, j-wt[i]) + P[i], T(i-1, j)] & \text{otherwise} \end{cases}$$

Consider there are 10 items, and the weight of each item is 1 kg. The knapsack capacity is 10 kg. There will two possibilities for each item, whether it will be included in the knapsack or not. Based on this, the recursion tree can be constructed as follows-



Recursion Tree

From the *recursion tree*, we can see that the *depth* of the *recursion tree* could be *n* in *worst case*. If the depth of a tree is *O(n)*, then there could be $2^n/2$ number of nodes (for half filled complete binary tree). So, *total number of function call* will be *$O(2^n)$.*

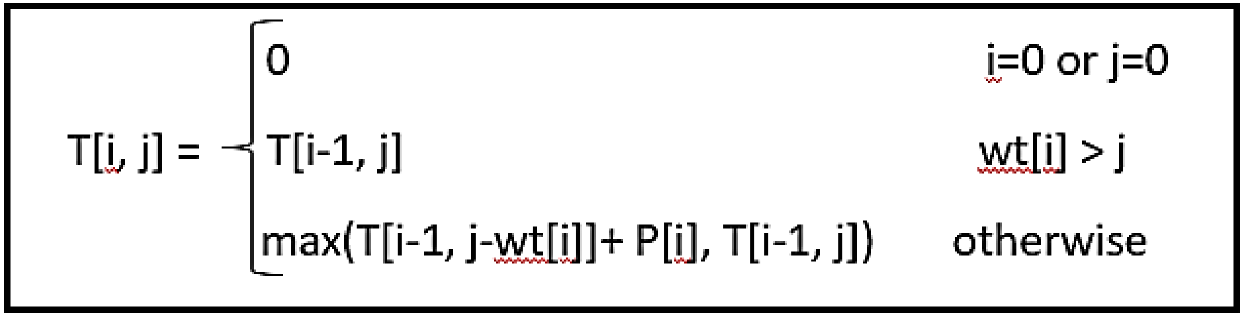
From the tree, it is identified that there are *overlapping sub-problems*, and it also has optimal substructure. So, *DP* can be used to solve this problem. For DP, unique sub-problems need to be identified.

Let us find unique subproblems for the *0/1 knapsack* with *n items* and knapsack *capacity w.*

0/1 Knapsack Problem

How many unique subproblems we have for the *0/1* knapsack with *n items* and *knapsack capacity w*?

- There will be *n x w* unique subproblems in worst case.
- So, to solve this problem using DP will need to create a table/matrix of size *n+1 x w+1, 1 extra row* and *column* will be created to store the base case information *(0 value).* After that, table is filled one by one using the following equation.



$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

0/1 Knapsack Problem

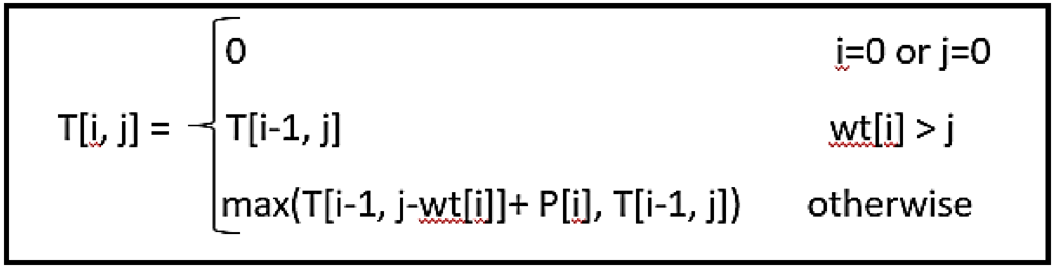
Problem : A thief enters a house for robbing it. He can carry a maximal weight of 7 kg into his bag. There are 4 items in the house with the following weights and values. What items should thief take if he either takes the item completely or leaves it completely?

Item	**Weight (kg)**	**Profit ($)**
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Given-

Knapsack capacity (W) = 7 kg

- Number of items (n) = 4



$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]] + P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

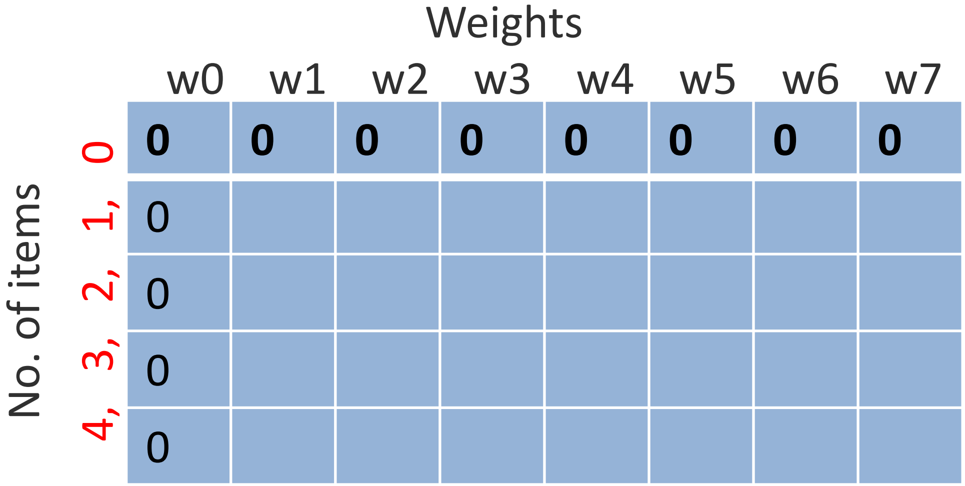
Step-01:

Draw a table say 'T' with *(N+1) = 4 + 1 = 5* number of rows and *(W+1) = 7 + 1 = 8* number of columns.

- Fill all the cells of *0^{th} row* and 0^{th} column with *0* [base case].

0/1 Knapsack Problem

Item	Weight (kg)	Profit ($)
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7



No. of items \ Weights	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1,	0							
2,	0							
3,	0							
4,	0							

Given-

Knapsack capacity (W) = 7 kg

Number of items (n) = 4

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

Where *i= 0, 1, …, N* and *j =0, 1, …, W.*
W is knapsack capacity and *N* = number of items

0/1 Knapsack Problem

Item	Weight (kg)	Profit ($)
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Weights (j)

No. of items (i)	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	1	1					
2	0							
3	0							
4	0							

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]] + P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

Finding T(1,1)

Since weight of item wt[i] =1 and the capacity of bag is j=1. so, we can place it in the bag.

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

Finding T[1,2]

weight of item *wt[i] =1*

the capacity of bag is *j=2*. so, we can place it in the bag.

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

0/1 Knapsack Problem

Item	Weight (kg)	Profit ($)
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Weights

No. of items

	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	1	1	1	1	1	1	1
2	0	1	1					
3	0							
4	0							

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

Similarly, *all columns* of *row 1* will be *1*, since we are placing only one item of *weight 1* in the bag of varying capacity (j= 1, 2, … 7).

Finding T[2,1] and T[2,2]

weight of item *wt[i] = 3*

the capacity of bag is *j=1 i.e.,* wt[2]>j,

T[2,1]= T[i-1, j] = T[1, 1] =1

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

0/1 Knapsack Problem

Item	**Weight (kg)**	**Profit ($)**
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Weights

No. of items	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5			
3	0							
4	0							

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

Finding T[2,3]

weight of item *wt[i] = 3*

the capacity of bag is *j=3.*

T[2,3]= max(T[1, 3-3]+ 4, T[1, 3])
=4

Finding T[2,4]

weight of item *wt[i] = 3*

the capacity of bag is *j=4*.

T[2,4]= max(T[1, 4-3]+ 4, T[1, 4])
=5 [Item 1 and 2 both can be kept in bag]

0/1 Knapsack Problem

Item	**Weight (kg)**	**Profit ($)**
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

Weights

No. of items	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5	5	5	5
3	0	1	1	4				
4	0							

All remaining values of this *row will be 5* since *we have two items*, and *both can be placed in bag.*

Finding T[3,1]

weight of item *wt[i] = 4,*

the capacity of bag is *j=1*, i.e., *wt[i] >j*

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

=1 [only Item 1]

All places before *T[3,3]* will be *1* since weight of item is larger than capacity.

Finding T[3,3)

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

=4 [only Item 2]

0/1 Knapsack Problem

Item	Weight (kg)	Profit ($)
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Weights

No. of items	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5	5	5	5
3	0	1	1	4	5	6		
4	0							

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

Finding T[3,4]

weight of item *$wt[i] = 4$*

the capacity of bag is *$j=4$.*

$T[3,4]= \max(T[2, 4-4]+ 5, T[2, 4])$

$=5$ [3rd object can be placed in bag]

Finding T[3,5]

weight of item *$wt[i] = 4$*

capacity of bag is *$j=4$.*

$T[3,5]= \max(T[2, 5-4]+ 5, T[2, 5])$

$=6$

[3rd and 1st object can be placed in bag]

0/1 Knapsack Problem

Item	Weight (kg)	Profit ($)
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

	Weights							
No. of items	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5	5	5	5
3	0	1	1	4	5	6	6	9
4	0							

Finding T[3,6]

weight of item *wt[i] = 4*

capacity of bag is *j=6*

T[3,6]= max(T[2, 6-4]+ 5, T[2, 6])
=6

Finding T[3,7]

weight of item *wt[i] = 4*

capacity of bag is *j=7*

T[3,7]= max(T[2, 7-4]+ 5, T[2, 7])
=9

[3rd and 2nd object can be placed in bag]

0/1 Knapsack Problem

Item	Weight (kg)	Profit ($)
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

	Weights							
No. of items	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5	5	5	5
3	0	1	1	4	5	6	6	9
4	0	1	1	4	5	7	8	

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

Finding T[4,5]

weight of item *wt[i] = 5*

capacity of bag is *j=5*

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

=7

Finding T[4,6]

weight of item *wt[i] = 5*

capacity of bag is *j=6*

T[4,6]= max(T[3, 6-5]+ 7, T[3, 6]

=8

[1st and 4th object can be placed in bag]

0/1 Knapsack Problem

Item	Weight (kg)	Profit ($)
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T[i, j] = \begin{cases} 0 & i=0 \text{ or } j=0 \\ T[i-1, j] & wt[i] > j \\ \max(T[i-1, j-wt[i]]+ P[i], T[i-1, j]) & \text{otherwise} \end{cases}$$

Weights

No. of items	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	0	1	1	1	1	1	1	1
2	0	1	1	4	5	5	5	5
3	0	1	1	4	5	6	6	9
4	0	1	1	4	5	7	8	9

Finding T[4,7]

weight of item *wt[i] = 5*

capacity of bag is *j=7*

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

=9

[3rd and 2nd object can be placed in bag]

0/1 Knapsack Problem

Weights

No. of items

	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1,	0	1	1	1	1	1	1	1
2,	0	1	1	4	5	5	5	5
3,	0	1	1	4	5	6	6	**9**
4,	0	1	1	4	5	7	8	9

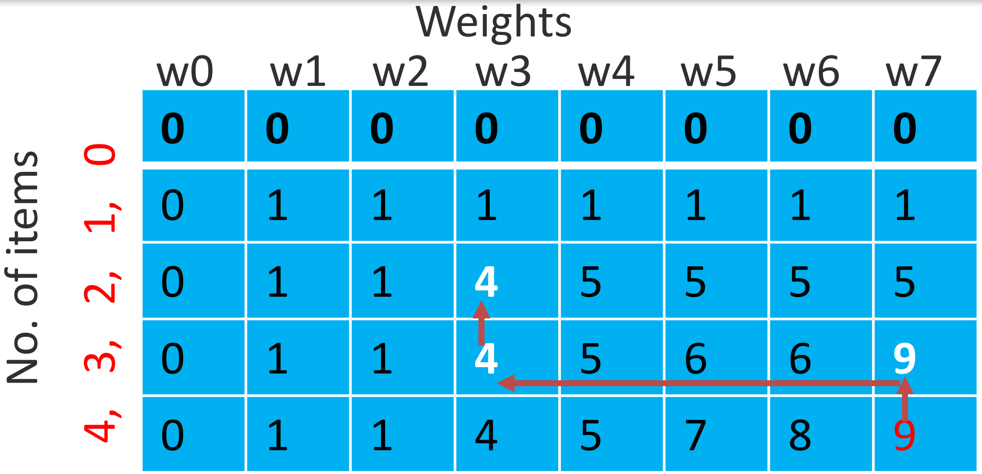
Item	**Weight (kg)**	**Profit ($)**
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Find the object that can be placed in the bag to maximize the profit-

- Go to last row last column i.e., **T[4][7] = 9** and check if the previous row with same column i.e., **T[3][7].** If the values of **T[3][7] = T[4][7]**, it means 4th item was not included in the knapsack. So, ignore **4th** item and include **3rd** item's weight and profit in the result set.

0/1 Knapsack Problem



Weights

No. of items

	w0	w1	w2	w3	w4	w5	w6	w7
0,	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1,	0	1	1	1	1	1	1	1
2,	0	1	1	**4**	5	5	5	5
3,	0	1	1	**4**	5	6	6	**9**
4,	0	1	1	4	5	7	8	9

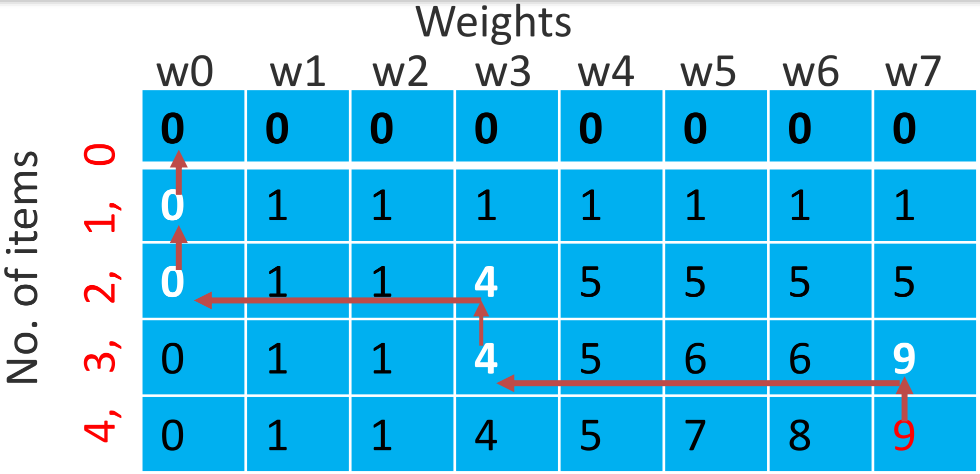
Item	Weight (kg)	Profit ($)
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Find the object that can be placed in the bag to maximize the profit-

- Go to last row last column i.e., **T[4][7] = 9** and check if the previous row with same column i.e., **T[3][7].** If the values of **T[3][7] = T[4][7]**, it means 4th item was not included in the knapsack. So, ignore **4th** item and include **3rd** item's weight and profit in the result set.
- Subtract its profit from total **(9), 9-5=4.** check for **4 in 2nd row**. if it is present include **object's weight and profit** in **result set** else go to its previous row, where **4** exists.
- As **4 is in 2nd row** include it in **result set** and **subtract its profit** from the remaining profit. **Profit =4-4=0**

0/1 Knapsack Problem



Weights

No. of items

	w0	w1	w2	w3	w4	w5	w6	w7
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1,	0	1	1	1	1	1	1	1
2,	0	1	1	4	5	5	5	5
3,	0	1	1	4	5	6	6	9
4,	0	1	1	4	5	7	8	9

Item	**Weight (kg)**	**Profit ($)**
Mirror	1	1
Silver nugget	3	4
Painting	4	5
Vase	5	7

Knapsack capacity (W) = 7 kg

Find the object that can be placed in the bag to maximize the profit-

- Go to last row last column i.e., **T[4][7] = 9** and check if the previous row with same column i.e., **T[3][7].** If the values of **T[3][7] = T[4][7]**, it means 4th item was not included in the knapsack. So, ignore **4th** item and include **3rd** item's weight and profit in the result set.
- Subtract its profit from total **(9), 9-5=4.** check for **4 in 2nd row**. if it is present include **object's weight and profit** in **result set** else go to its previous row, where **4** exists.
- As **4 is in 2nd row** include 2nd item in **result set** and **subtract its profit** from the remaining profit. **Profit =4-4=0**

Since, profit is now 0 means we reached to the final solution. The **bag/ result** set will include **item 2 and 3** and **profit will 9.**

0/1 Knapsack Problem

Time Complexity

Time complexity for 0/1 Knapsack problem solved using DP is **$O(N*W)$**

where N denotes number of items available, and W denotes the capacity of the knapsack

This approach may be inefficient and takes longer time than the brute forces approach, if **$W >= 2^N$** , since the T= **$O(N*2^N)$**

Longest Common Subsequence (LCS)

Given two strings, the task is to find the longest common subsequence (LCS) present in the given strings in the same order.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters. Please note, in substring, characters are selected in contiguous manner, while in subsequence characters may not be contiguous.

For example, "ace" is a subsequence of "abcde".

LCS :- cdgi

Suppose we have string s1= “a b c d e f g h i j” and

s2= “e c d g i”

The substring “ecdgi” is also present in s1 but its characters are not in the increasing order. Please note that matching character need not to be contiguous, but they must in the increasing order by their index.

s1= “a b c d e f g h i j”

s2= “e c d g i”

c is before e in string s1 so, it will not be considered as a subsequence.

Longest Common Subsequence (LCS)

string s1= “a b c d e f g h i j” (size m) → For generating a subsequence, a character may be either taken or not taken. So, for each character, there will be 2 choices. So, total no. of subsequence could be = 2^m

s1= “a b c d e f g h i j” (size m)
s2= “e c d g i” (size n)

To find the LCS, each subsequence of s1 must be checked in S2 by scanning the entire sequence. However, subsequence of s1 may be only checked in s2, when size of subsequence of s1 is smaller or equal to s2. The LCS is identified after finding all subsequences which are common in both strings.

So, **time complexity** of the above approach will include

- Time required to find all subsequences of s1 = **$O(2^m)$**
- For each, subsequence of s1, find all subsequences which are also present in s2= **$O(n* 2^m)$**
- Finding the longest common subsequence = **$O(2^m)$**

Time complexity= $O(2^m) + O(n* 2^m) + O(2^m)$ = **$O(n* 2^m)$**

Time complexity is exponential. Is there any other efficient approach that can be used?

Longest Common Subsequence (LCS)

- **Is there any other efficient approach that can be used to find the optimal solution?**

DP may be used, however, DP will only applicable if we should-

- Able to find the optimal substructure / Able to write recursive equation
- Able to find overlapping subproblems

So, Let us define a recursive equation to find the optimal substructure and repeating/ overlapping subproblems.

Longest Common Subsequence (LCS)

In optimal substructure, the solution of the smaller subproblems is going to be the solution of the main problems. The optimal substructure can easily be identified using recursive formula.

Suppose we have two strings, X= x1 x2 x3…xi and Y= y1 y2 y3…yj. Then to find the longest subsequence,

1. Define a recursive function that first compares xn and yn characters of X and Y, and if they are equal, call the same recursive function for x1 x2 x3…xi-1 and y1 y2 y3…yj-1.
2. If xn and yn are not equal, then, call two recursive function, one for x1x2x3…xi-1 and y1 y2 y3…yj and other for x1 x2 x3…xi and y1 y2 y3…yj-1 and return the max of their result.

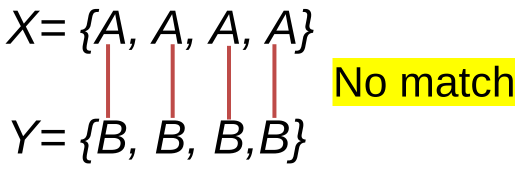
Let us assume *C(i, j)* is a LCS for string *X* and *Y*, where *X* = *1* to *i* and *Y* = *1* to *j*. Then, we can use the following formula to compute the LCS.

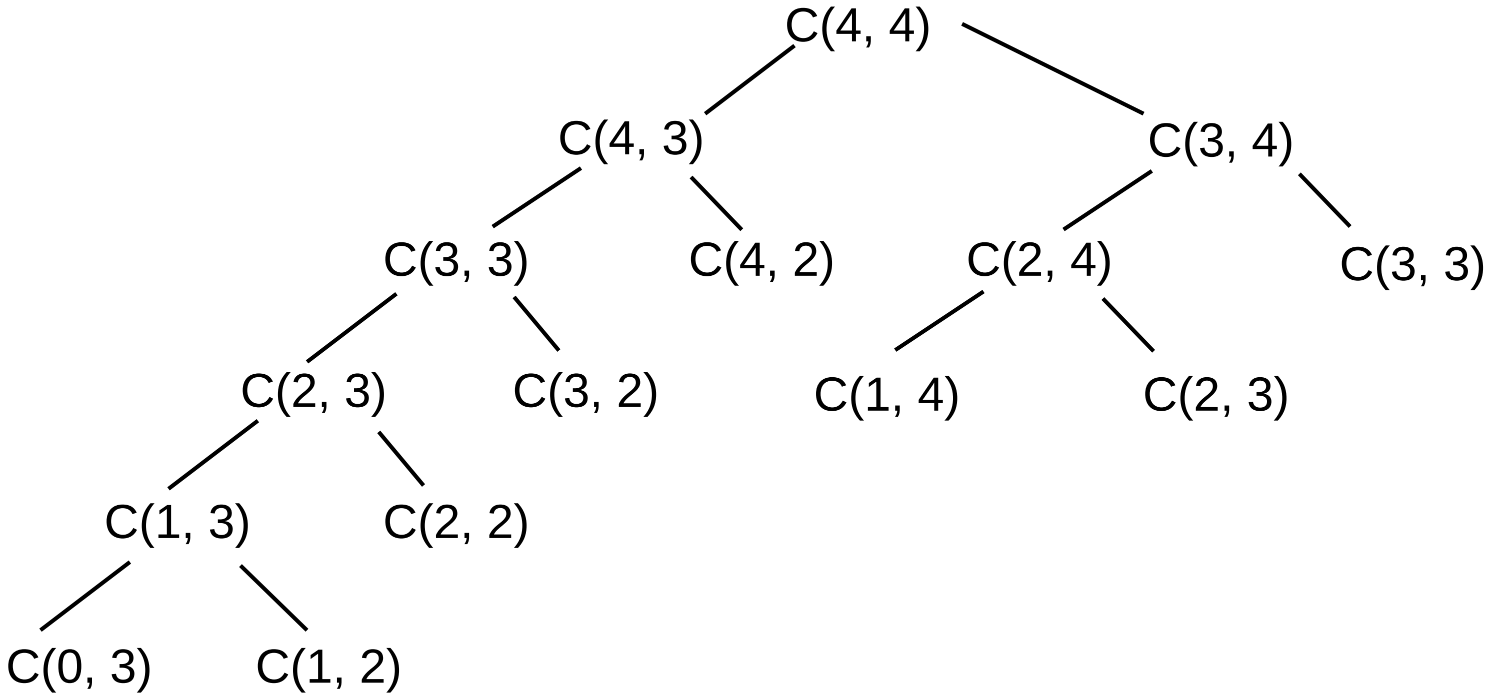
$$C(i, j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1+C(i-1, j-1) & \text{if } i>0, j>0, \text{ and } Xi=Yj \\ \max[C(i, j-1), C(i-1, j)] & \text{if } i>0, j>0, \text{ and } Xi!=Yj \end{cases}$$

Longest Common Subsequence (LCS)

Suppose we have two string *X= {A, A, A, A}* and *Y= {B, B, B,B}.* Then following tree can be constructed for the following recursive formula.

$$C(i, j) = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1+C(i-1, j-1) & \text{if } i>0, j>0, \text{ and } Xi=Yj \\ \max[C(i, j-1), C(i-1, j)] & \text{if } i>0, j>0, \text{ and } Xi!=Yj \end{cases}$$





Longest Common Subsequence (LCS)

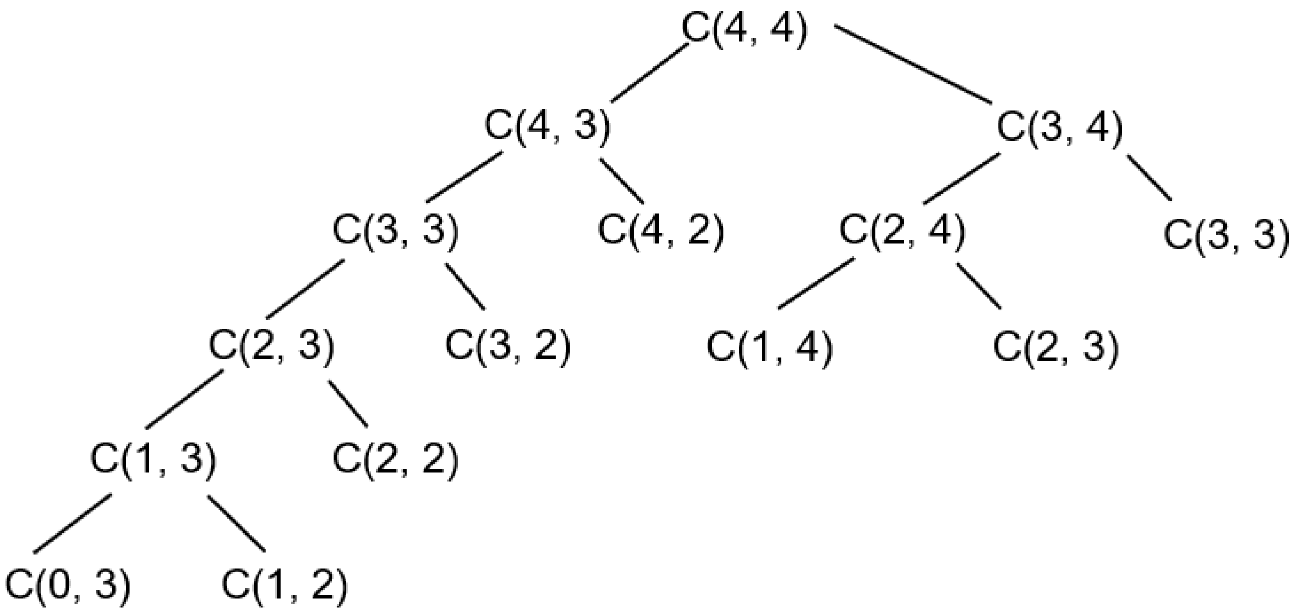
The maximum depth of the tree could be ***m+n***, since ***longest path*** may include the worst case, that decrease n by 1 in one step and m by 1 in another step as given in the following figure.

C(n, m)
C(n-1, m)
C(n-2, m-1)
C(n-2, m-2)
.
.
.
C(0, --) or C(--, 0)

X= {A, A, A, A}

Y= {B, B, B,B}

No match



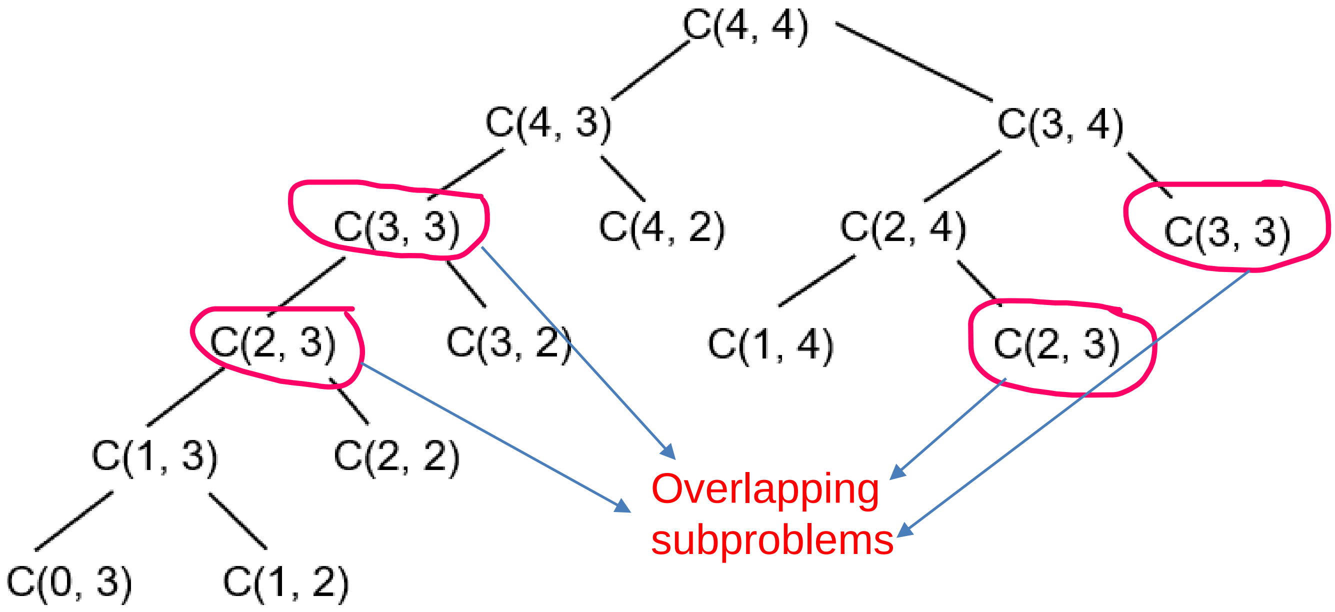
Since, the depth of the root is ***m+n*** so, total number of function will be ***$O(2^{m+n}/2)$,*** if tree will be half filled

Longest Common Subsequence (LCS)

Since, the depth of the root is ***m+n*** so, total number of function will be ***O(2^{m+n}/2),*** if tree will be half filled. So, can we reduce time complexity using DP? DP will only applicable, if we can find optimal substructure and overlapping subproblems.

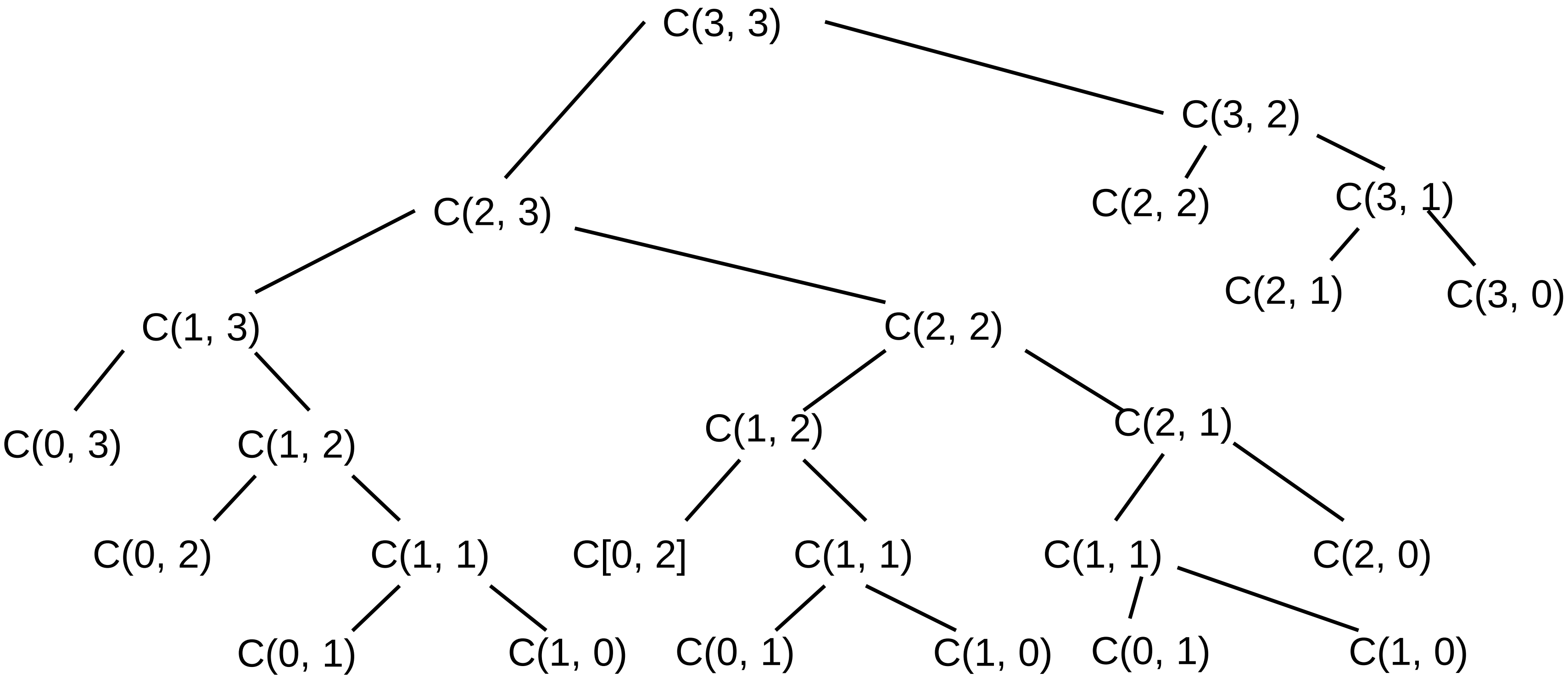
There are many subproblems in the following tree which are overlapping. In DP, only unique subproblems are evaluated. Let us find unique subproblems for the LCS problems.

The unique subproblems could be nearly *O(mxn)*, if string *X* contains *m* characters and string *Y* contains *n* characters.



Longest Common Subsequence (LCS)

There are many subproblems in the following tree which are overlapping. In DP, only unique subproblems are evaluated. Let us find unique subproblems for the LCS problems.



Non-overlapping subproblems are- *C[3,3], C[3,2], C[3,1], C[2,3], C[2,2], C[2,1], C[1,3], C[1,2], C[1,1], C[0,3], C[0,2], C[0,1], [3,0], C[2,0], C[1,0], C[0,0] (9+7=16 sub problems including 7 subproblems with 0 base cases).*

To solve this problem using DP will require to create a table of size ***(m+1xn +1)*** **[1 row and column to store 0 (base case)]**.

Longest Common Subsequence (LCS)

To find the LCS for X= “AAB” and Y =“ACA”, first a table of size (X+1) and (Y+1) is created.

The value at any place will depends on its left diagonal, its previous column, and previous row with same column.

For e.g.
C[1,1] = 1+c[0,0], or max (C[1,0], C[0,1])

So, the value of C[3,3] can only be computed, if we have values of C[2,2] (in case of match) or C[2,1] and C[1,2].

The table should be filled either row wise or column wise to efficiently compute values.

	0	1	2	3
0	**00**	**01**	**02**	**03**
1	**10**	**11**	**12**	**13**
2	**20**	**21**	**22**	**23**
3	**30**	**31**	**32**	**33**

Longest Common Subsequence (LCS)

To find the LCS for X= “AAB” and Y =“ACA”, first a table of size (X+1) and (Y+1) is created.

Step 1: Fill 0 in 0th row and column since, if any of the string will null the LCS will also be NULL.

Step 2: Compute C[1,1] using the formula

$$C[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1+C[i-1, j-1] & \text{if } i>0, j>0, \text{ and } Xi=Yj \\ \max(C[i, j-1], C[i-1, j]) & \text{if } i>0, j>0, \text{ and } Xi!=Yj \end{cases}$$

C[1,1]= 1+C[0,0], since Xi = Yi

= 1+0 =1

C[1,2] = max(C[1,1], C[0,2]), since Xi != Yi

= max(1,0) =1

C[1,3] = 1+C[0, 2], since Xi = Yi

= 1+0 =1

C[2,1] = 1+C[1, 0], since Xi = Yi

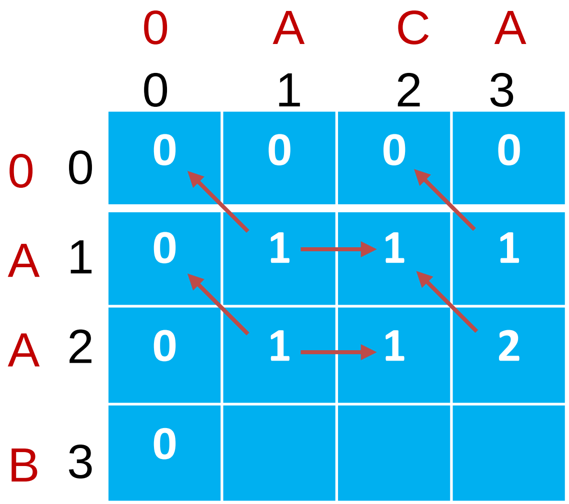
= 1+0 =1

C[2,2] = max(C[2,1], C[1,2]), since Xi != Yi

= max(1,1) =1

C[2,3] = 1+C[1, 2], since Xi = Yi

= 1+1 =2



		0	A	C	A
		0	1	2	3
0	0	0	0	0	0
A	1	0	1	1	1
A	2	0	1	1	2
B	3	0			

Longest Common Subsequence (LCS)

To find the LCS for X= “AAB” and Y =“ACA”, first a table of sizeof (X+1) and (Y+1) are created.

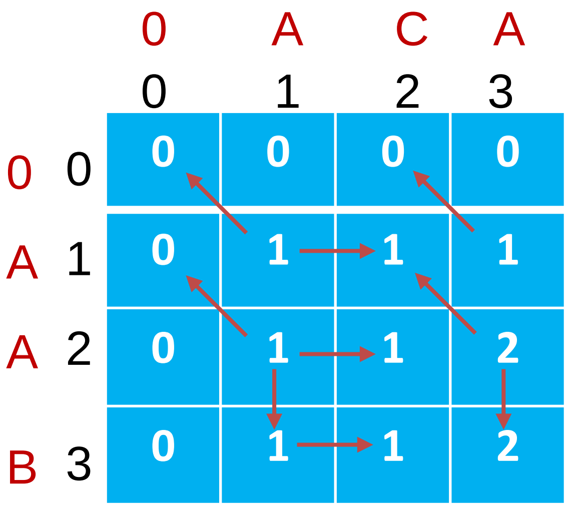
$$C[i, j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ 1+C[i-1, j-1] & \text{if } i>0, j>0, \text{ and } Xi=Yj \\ \max(C[i, j-1], C[i-1, j]) & \text{if } i>0, j>0, \text{ and } Xi!=Yj \end{cases}$$

C[3,1] = max(C[2,1], C[3,0]), since Xi != Yi
= max(1,0) =1

C[3,2] = max(C[2,2], C[3,1]), since Xi != Yi
= max(1,1) =1

C[3,3] = max(C[2,3], C[3,2]), since Xi != Yi
= max(2,1) =2

This is the length of longest common subsequence for the given problem.



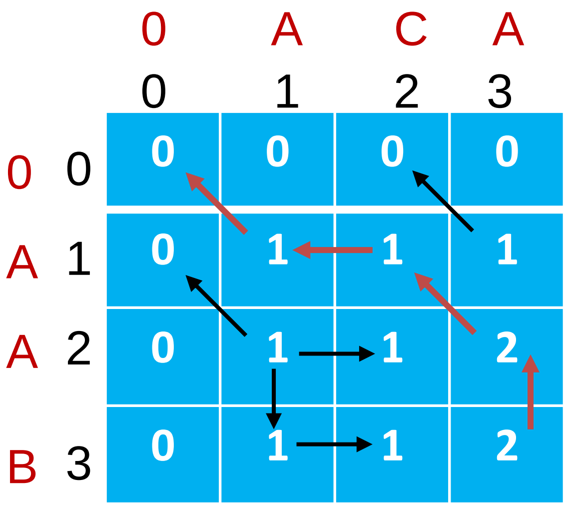
		0	A	C	A
		0	1	2	3
0	0	0	0	0	0
A	1	0	1	1	1
A	2	0	1	1	2
B	3	0	1	1	2

Longest Common Subsequence (LCS)

To find the LCS for X= “AAB” and Y =“ACA”, first a table of sizeof (X+1) and (Y+1) are created.

To find the LCS, check the places where values of the cells an have been updated, which means taken from diagonal and incremented. In this example, values at C[1,1] and C[2,3] have been incremented. The LCS will include characters at X[1] and X[2].

LCS = AA.



		0	A	C	A
		0	1	2	3
0	0	0	0	0	0
A	1	0	1	1	1
A	2	0	1	1	2
B	3	0	1	1	2

Longest Common Subsequence (LCS)

Find the LCS for X= “ABCBDAB” and Y =“BDCABA” in polynomial time.

	0	A	B	C	B	D	A	B
0								
B								
D								
C								
A								
B								
A								

Subset Sum Problem

Given an array of non-negative integers and an integer sum. We have to find whether there exists any subset in an array whose sum is equal to the given integer sum.

Example:

Input: arr [] = {3, 34, 4, 12, 3, 2}, sum = 7

Output: True

Explanation: There is a subset (4, 3) with sum 7.

Subset Sum Problem

Given an array of non-negative integers and an integer sum. We have to tell whether there exists any subset in an array whose sum is equal to the given integer sum.

Example:

Input: arr[] = {3, 34, 4, 12, 3, 2}, sum = 7

Output: True

Explanation: There is a subset (4, 3) with sum 7.

Let us try **brute force approach** to solve the problem

In **brute force** all **possible subsets** are identified. In a **subset**, **element** will **either present** or **not present**. So, there will be **2 possibilities** for **each element**. For *n elements*, the number of *possible subsets* could be **2^n**. So, this approach will be **computationally very expensive**.

Can we reduce time complexity?

Longest Common Subsequence (LCS)

Is there any other approach that find the subsets in efficiently.

DP may be used, however, DP will only applicable if we should-

- Able to find the optimal substructure / Able to write recursive equation
- Able to find overlapping subproblems

So, Let us define a recursive equation to find the optimal substructure and repeating/ overlapping subproblems.

Subset Sum Problem

Let us define recursive formula. Suppose there are *1,2,3, …, i elements* and we have to find whether there is any subset from *i elements* whose sum is *equal to S*. A= { a1, a2, a3, …., ai} Sum =S

This problem is very similar to 0/1 knapsack problem and two conditions are similar to that.

$$SS(i, S) = \begin{cases} SS(i-1, S) & S<ai \\ SS(i-1, S-ai) \text{ OR } SS(i-1, S) & S>=ai \\ True & S=0 \\ False & i=0, S!=0 \end{cases}$$

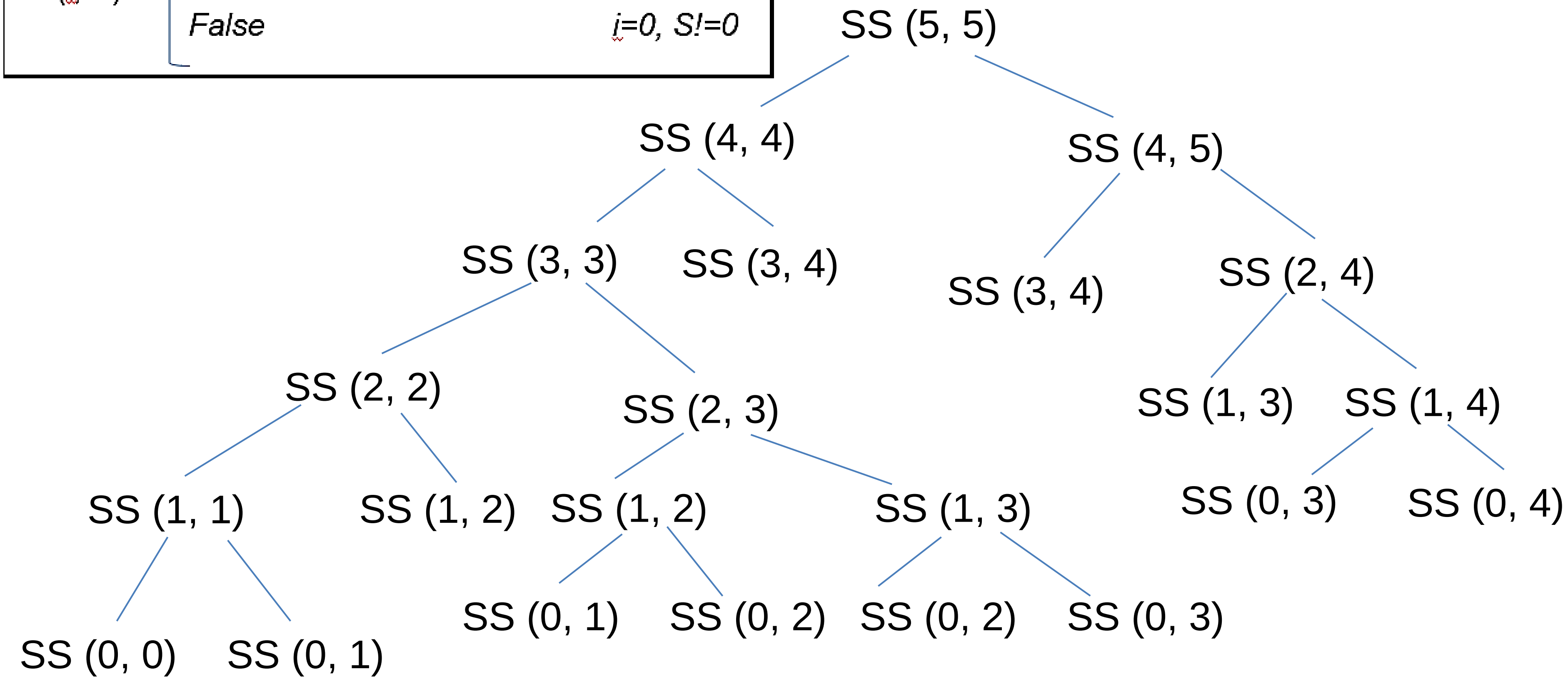
A= { a1, a2, a3, …., ai} Sum =S

Let us draw the recursion tree to identify the overlapping solutions.

Subset Sum Problem

$$SS(i, S) = \begin{cases} SS(i-1, S) & S<ai \\ SS(i-1, S-ai) \text{ OR } SS(i-1, S) & S>=ai \\ True & S=0 \\ False & i=0, S!=0 \end{cases}$$

A= { 1, 1, 1, 1, 1} **Sum =5**



SS (5, 5)

SS (4, 4) SS (4, 5)

SS (3, 3) SS (3, 4) SS (3, 4) SS (2, 4)

SS (2, 2) SS (2, 3) SS (1, 3) SS (1, 4)

SS (1, 1) SS (1, 2) SS (1, 2) SS (1, 3) SS (0, 3) SS (0, 4)

SS (0, 1) SS (0, 2) SS (0, 2) SS (0, 3)

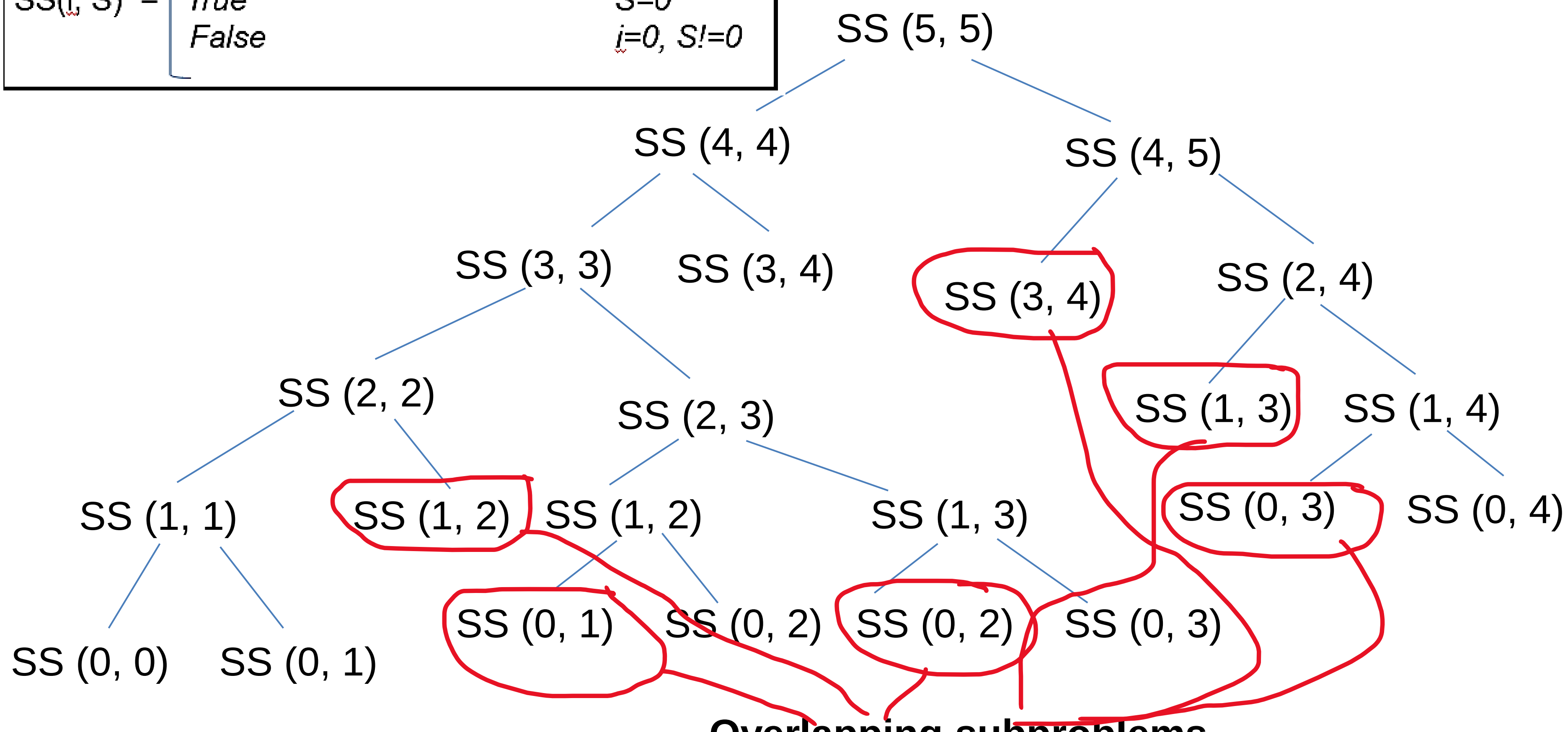
SS (0, 0) SS (0, 1)

In the worst case, the depth of the tree will be **n** and there could be **O(2^n)** number of calls. So, this approach is very expensive. It is also observed from the tree that many subproblems are overlapping. Hence, DP will certainly reduce time complexity.

Subset Sum Problem

$$SS(i, S) = \begin{cases} SS(i-1, S) & S<ai \\ SS(i-1, S-ai) \ OR \ SS(i-1, S) & S>=ai \\ True & S=0 \\ False & i=0, S!=0 \end{cases}$$

A= { 1, 1, 1, 1, 1} **Sum =5**


SS (5, 5)

SS (4, 4) SS (4, 5)

SS (3, 3) SS (3, 4) SS (3, 4) SS (2, 4)

SS (2, 2) SS (2, 3) SS (1, 3) SS (1, 4)

SS (1, 1) SS (1, 2) SS (1, 2) SS (1, 3) SS (0, 3) SS (0, 4)

SS (0, 0) SS (0, 1) SS (0, 1) SS (0, 2) SS (0, 2) SS (0, 3)

Overlapping subproblems

There will (i+1, S+1) number of unique solution/ calls for the problem having **i** elements with subset sum **S**.

Subset Sum Problem

SS[0,0] = *S=0* is possible, if there is no element,
SS[0,0] = T

For SS[0,1], *S!=0* not possible because there is no element
SS[0,1] = F
Similarly, all other positions of 0th row will be false.

SS[1,0] = T, (If we have one element and we don't choose that then S=0 is possible.) Similarly, all other rows of column 0 will also **T**.

SS[1,1] = SS[0, 1], since s< ai (1<6)
SS[1,1] = F ,

It cannot be included in the subset since, the value of 1st item is 6 which will never be equal to 1. (greater than current sum).
SS[1,2] = F, value of 2nd item will never equal to 2.

All column of this row will be false, since sum =1,2, …,5 will always smaller than item value i.e., 6.

$$SS[i, S] = \begin{cases} SS[i-1, S] & S<ai \\ SS[i-1, S-ai] \text{ or } SS[i-1, S] & S>=ai \\ True & S=0 \\ False & i=0, S!=0 \end{cases}$$

A= { 6, 3, 2, 1} **S =5**

Sum

Items	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T					
3	T					
4	T					

Subset Sum Problem

A= { 6, 3, 2, 1}   S =5

$$SS[i, S] = \begin{cases} SS[i-1, S] & S<ai \\ SS[i-1, S-ai] \text{ or } SS[i-1, S] & S>=ai \\ True & S=0 \\ False & i=0, S!=0 \end{cases}$$

SS[2,1] = SS[1,1] since *1<3*
SS[2,1] = F

S[2,2] = SS[1,2] since *2<3*
SS[2,2] = F

S[2,3] = SS[1,3-3] OR SS[1, 3]
SS[2,3] = SS[1,0] OR SS[1, 3]
SS[2,3] = T
From first two elements, sum 3 is possible. (include 2nd item)

S[2,4] = SS[1,4-3] OR SS[1, 4]
SS[2,4] = SS[1,1] OR SS[1, 4]
SS[2,4] = F
From first two elements, sum 4 is possible.

S[2,5] = SS[1,5-3] OR SS[1, 5]
SS[2,3] = SS[1,2] OR SS[1, 5]
SS[2,3] = F,
From first two elements, sum 5 is not possible.

Sum

Items	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T					
4	T					

Subset Sum Problem

A= { 6, 3, 2, 1}   S =5

$$SS[i, S] = \begin{cases} SS[i-1, S] & S<ai \\ SS[i-1, S-ai] \text{ or } SS[i-1, S] & S>=ai \\ True & S=0 \\ False & i=0,\ S!=0 \end{cases}$$

SS[3,1] = SS[2,1] since *1<2*
SS[3,1] = F

S[3,2] = SS[2,2-2] OR SS[2, 2]
SS[3,2] = SS[2,0] OR SS[2, 2]
SS[3,2] = T
From first three elements, sum 2 is possible. (include 3rd item)

S[3,3] = SS[2,3-2] OR SS[2, 3]
SS[3,3] = SS[2,1] OR SS[2, 3]
SS[3,3] = T
From first three elements, sum 3 is possible. (include 2nd item)

S[3,4] = SS[2,4-2] OR SS[2, 4]
SS[3,4] = SS[2,2] OR SS[2, 4]
SS[3,4] = F
From first three elements, sum 4 is not possible.

Sum

Items

	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T	F	T	T	F	
4	T					

Subset Sum Problem

A= { 6, 3, 2, 1}   S =5

S[3,5] = SS[2,5-2] OR SS[2, 5]
SS[3,5] = SS[2,3] OR SS[2, 5]
SS[3,5] = T

From first three elements, sum 5 is possible (include 2nd and 3rd item.)

S[4,1] = SS[3,1-1] OR SS[3, 1]
SS[4,1] = T

From first four elements, sum 1 is possible (include 4th item.)

S[4,2] = SS[3,2-1] OR SS[3, 2]
SS[4,2] = T

From first four elements, sum 2 is possible. (include 3rd item.)

S[4,3] = SS[3,3-1] OR SS[3, 3]
SS[4,3] = T

From first four elements, sum 3 is possible. (include 2nd item.)

$$SS[i, S] = \begin{cases} SS[i-1, S] & S<ai \\ SS[i-1, S-ai] \text{ or } SS[i-1, S] & S>=ai \\ True & S=0 \\ False & i=0, S!=0 \end{cases}$$

	Sum					
Items	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T	F	T	T	F	T
4	T	T	T	T		

Subset Sum Problem

A= { 6, 3, 2, 1}   S =5

$$SS[i, S] = \begin{cases} SS[i-1, S] & S<ai \\ SS[i-1, S-ai] \text{ or } SS[i-1, S] & S>=ai \\ True & S=0 \\ False & i=0, S!=0 \end{cases}$$

S[4,4] = SS[3,4-1] OR SS[3, 4]
SS[4,4] = T
From first four elements, sum 4 is possible (include 2nd and 4th item.)

S[4,5] = SS[3,5-1] OR SS[3, 5]
SS[4,5] = T
From first four elements, sum 5 is possible (include 2nd and 3rd item.)

Sum

Items

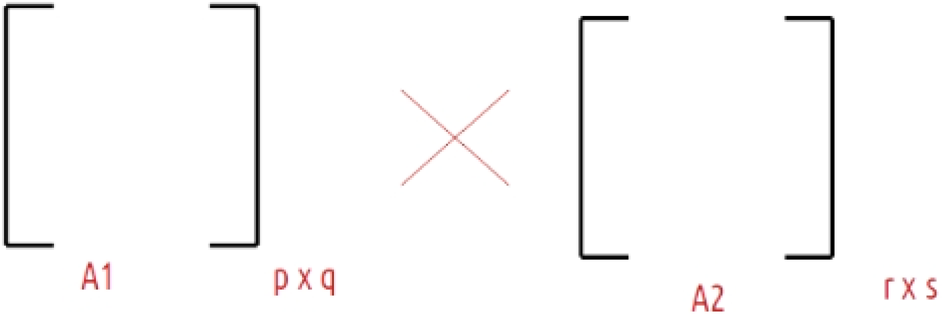
	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	F	F	F	F	F
2	T	F	F	T	F	F
3	T	F	T	T	F	T
4	T	T	T	T	T	T

Time complexity:

O(nxw), if *n* represents the number of items and w is the weight/ sum.
If *w> 2^n*, then brute force approach will the best method to find the subset sum.

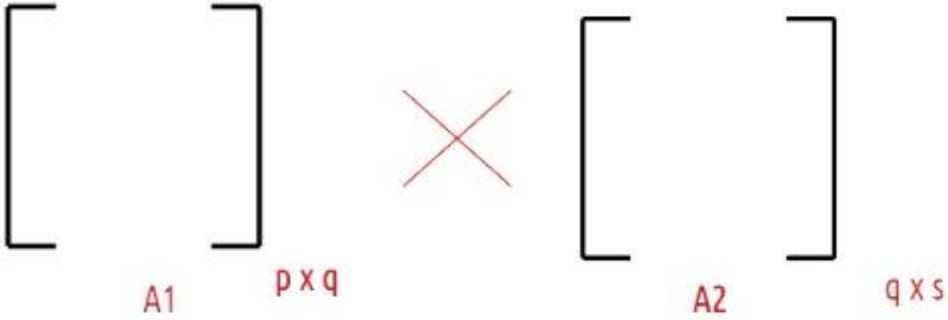
Matrix Chain Multiplication

Two matrices *A1* and *A2* of dimensions *[p x q]* and *[r x s]* can only be multiplied if and only if *q=r.*



A1 and A2 can be multilied only if q == r

The total number of scaler multiplications required to multiply *A1* and *A2* are *p*q*s*



Total number of multilications required to multiply A1 and A2= p x q x s

Matrix Chain Multiplication

Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that finds the most efficient way to multiply a sequence of matrices.

The problem is not actually to perform the multiplications but just to decide the sequence of the matrix multiplications involved.

The matrix multiplication is associative as no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have:

$((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))$

Matrix Chain Multiplication

Let's assume we have three matrices *M1,M2,M3* of dimensions *5×10*, *10×8*, and *8×5* respectively and for some reason we are interested in multiplying all of them. There are two possible orders of multiplication –

1. M1×(M2×M3)

Steps required in *M2×M3* will be *10×8×5 = 400.*

Dimensions of *M23* will be *10×5.*

Steps required in *M1×M23* will be *5×10×5 = 250.*

Total Steps =*400+250 = 650*

2. (M1×M2)×M3

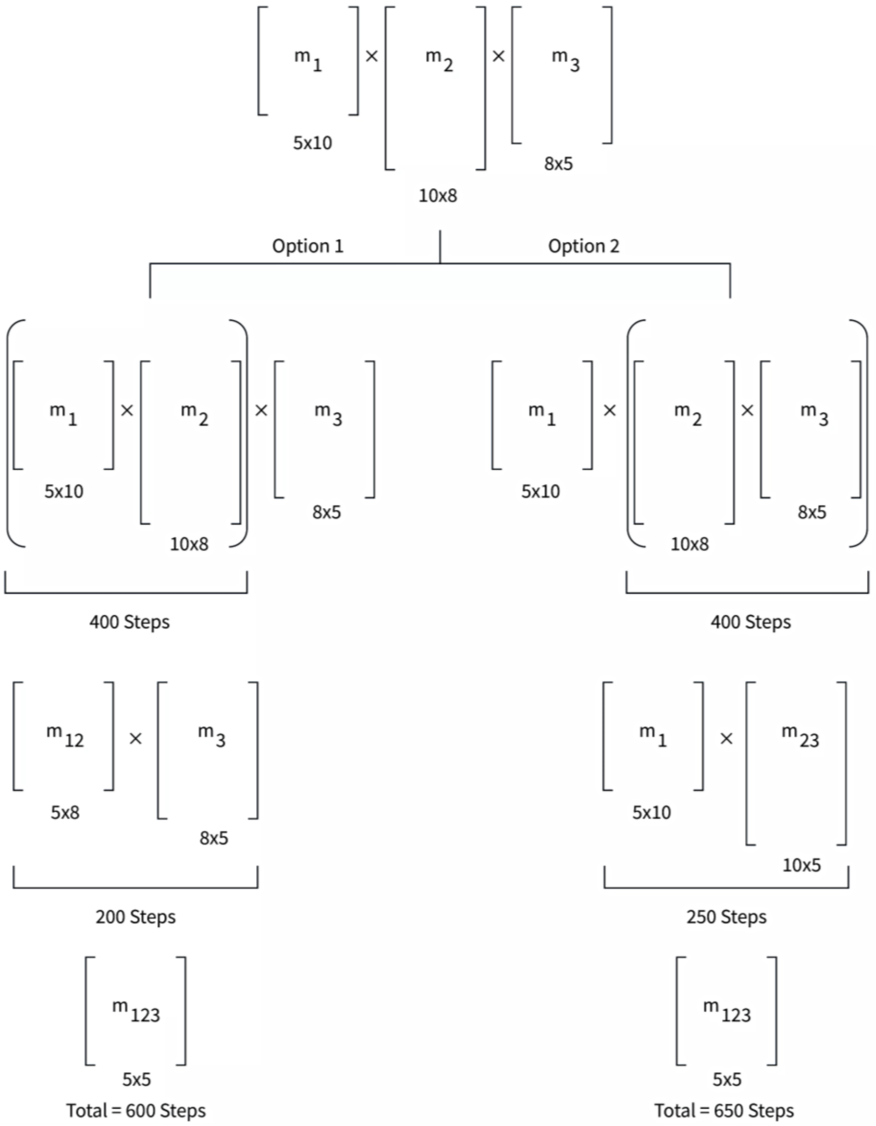
Steps required in *M1×M2* will be *5×10×8 = 400.*

Dimensions of *M12* will be *5×8.*

Steps required in *M12×M3* will be *5×8×5 = 200.*

Total Steps *= 400+200 = 600*

Matrix Chain Multiplication



$m_1 \times m_2 \times m_3$

5x10 10x8 8x5

Option 1

$(m_1 \times m_2) \times m_3$

5x10 10x8 8x5

400 Steps

$m_{12} \times m_3$

5x8 8x5

200 Steps

m_{123}

5x5

Total = 600 Steps

Option 2

$m_1 \times (m_2 \times m_3)$

5x10 10x8 8x5

400 Steps

$m_1 \times m_{23}$

5x10 10x5

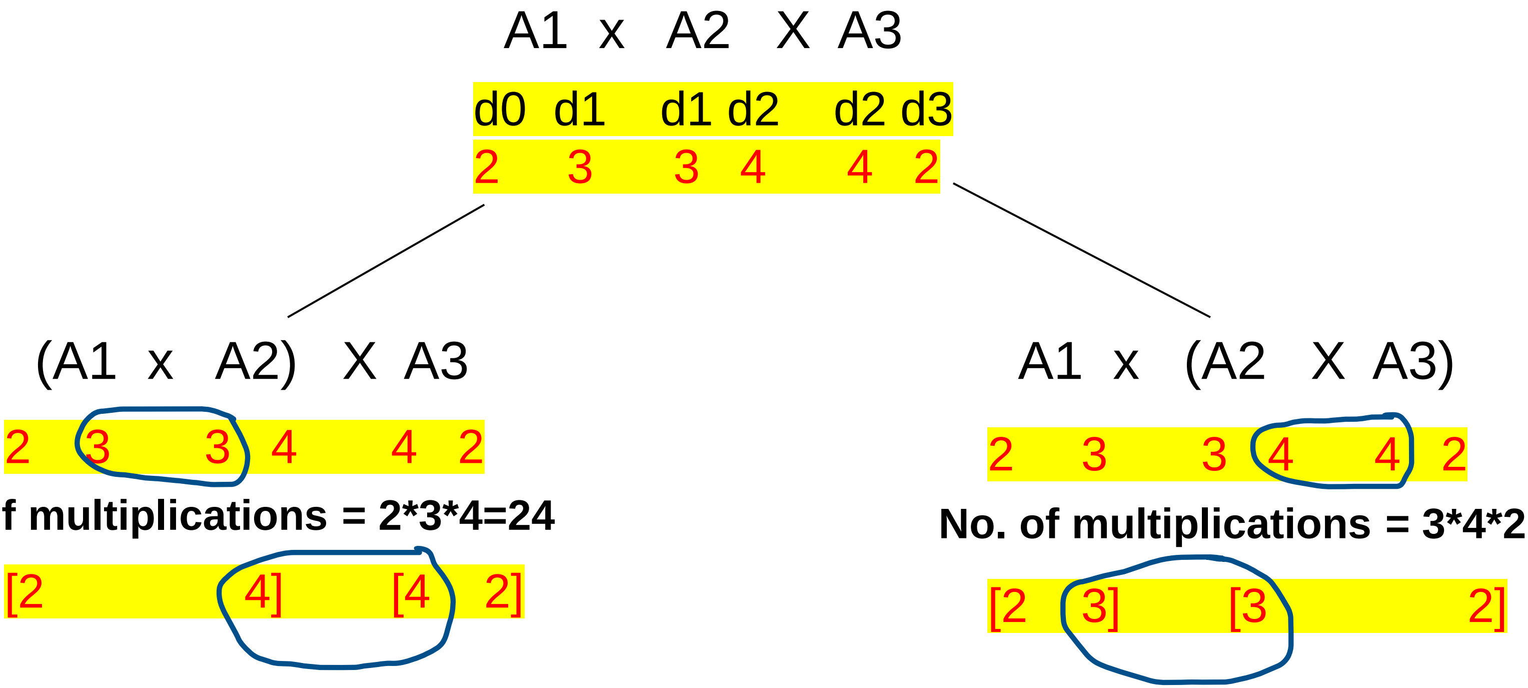
250 Steps

m_{123}

5x5

Total = 650 Steps

Matrix Chain Multiplication



A1 x A2 X A3

d0 d1 d1 d2 d2 d3

2 3 3 4 4 2

(A1 x A2) X A3

2 3 3 4 4 2

No. of multiplications = 2*3*4=24

[2 4] [4 2]

No. of multiplications = 2*4*2=16 +24 =40

A1 x (A2 X A3)

2 3 3 4 4 2

No. of multiplications = 3*4*2=24

[2 3] [3 2]

No. of multiplications = 2*3*2=12+ 24 =36

For this example, 2nd parenthesizing will be better. So, to minimize the number of multiplication, their sequence must be identified in advance. For larger number of matrices, a greater number of possibilities exists. So, trying all will increase time.

Matrix Chain Multiplication

In matrix chain multiplication, the number of ways matrices can be multiplied depends on their size. If there will 3 matrices, the number of way they will be multiplied is 2 while for 4 matrices, they can be multiplied in 5 ways. For 5 matrices, they can be multiplied in

(2n)!/(n+1)! n!, where n= m-1 (number of matrices-1)

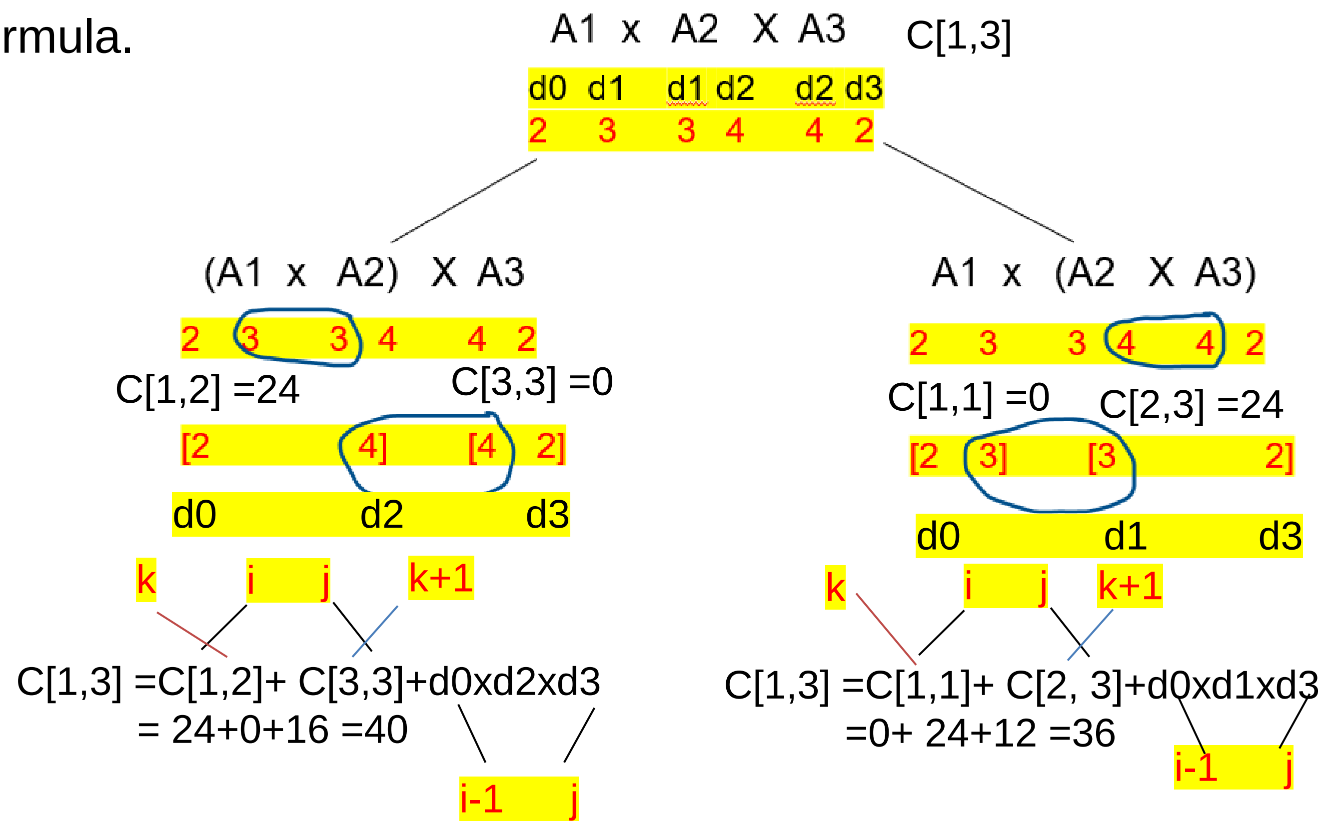
For 5 matrices (2*4)!/ (5)! (4)! = 8!/ 5! 4! = 8*7*6/2*3*4 = 14 ways

So, there will 14 ways to multiply 5 matrices.

The complexity of identifying the optimal sequence involve computations of all possible sequences. So, the naïve approach would be very expensive. Let us check the time complexity of naïve recursive approach.

Matrix Chain Multiplication

Le us derive recursive formula.



A1 x A2 X A3 C[1,3]

d0 d1 d1 d2 d2 d3

2 3 3 4 4 2

(A1 x A2) X A3

2 3 3 4 4 2

C[1,2] =24

C[3,3] =0

[2 4] [4 2]

d0 d2 d3

k i j k+1

C[1,3] =C[1,2]+ C[3,3]+d0xd2xd3

= 24+0+16 =40

i-1 j

A1 x (A2 X A3)

2 3 3 4 4 2

C[1,1] =0

C[2,3] =24

[2 3] [3 2]

d0 d1 d3

k i j k+1

C[1,3] =C[1,1]+ C[2, 3]+d0xd1xd3

=0+ 24+12 =36

i-1 j

C[i,j] = 0 i=j

min[(C[i,k] +C[k+1,j] +di-1xdkxdj)]

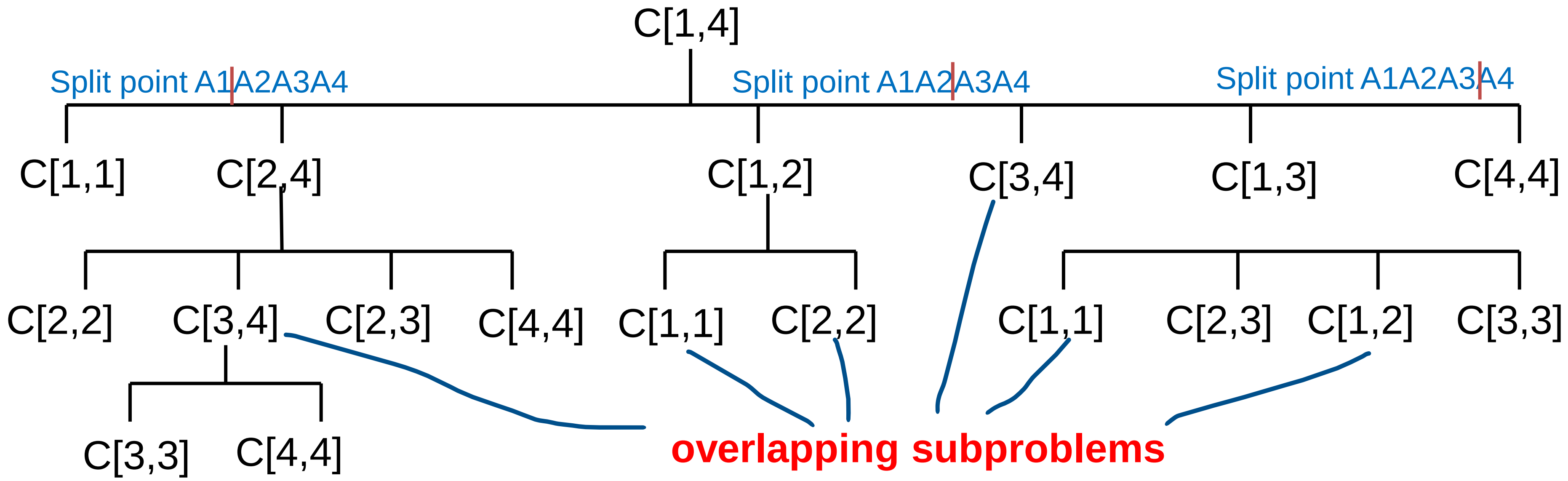
for i<=k<j, since splitting can be anywhere b/w i and j

Matrix Chain Multiplication

Le us draw a recursion tree for four matrices-A1A2A3A4. The cost can be represented by C[1,4].

$$C[i,j] = \begin{cases} 0 & i=j \\ \min[(C[i,k] + C[k+1,j] + di-1xdkxdj)] & \end{cases}$$

for i<=k<j, since splitting can be anywhere b/w i and j



From the recursion tree, the depth in worst case will be **O(n)**. So, if the tree will half fill, then too it will have exponential function call **$O(2^n)$** (as per the property of the complete binary tree and this tree is bigger than a binary tree). As, the recursion tree has many **overlapping subproblems.** The time complexity can be reduced using DP. DP uses bottom-up computation and stores unique results in a table. Let us find the unique results.

Matrix Chain Multiplication

For the previous example, the number of unique solutions are-

Unique function calls	No. of subproblems	Size
C(1, 1), C(2, 2), C(3, 3), C(4,4)	*4*	*1*
C(1, 2), C(2, 3), C(3,4)	*3*	*2*
C(1, 3), C(2,4)	*2*	*3*
C(1,4)	*1*	*4*

Total no, of function calls/ subproblems - 1+2+3+4= 10

Suppose we have *n* matrices *A1 A2 A3, …, An*, then

Subproblems	Size
n	*1*
n-1	*2*
n-2	*3*
.	.
1	*n*

No. of subproblems= 1+2+3+…+n

= n(n+1)/2 = $O(n^2)$

Therefore, for 4 matrices, a table with 4 row and 4 column need to be created.
We create one more table to store the k value for which minimum cost has been found.

Matrix Chain Multiplication

Find the best order to multiply following matrices-

A1	A2	A3	A4
d0x d1	*d1x d2*	*d2x d3*	*d3x d4*
3x2	*2x4*	*4x2*	*2x5*

Step 1: Fill 0 to the matrices of size 1

C[1,1], C[2,2], C[3,3], C[4,4] =0

Step 2: Find the value of matrices of size 2

C[1,2] = *min { C[1,1]+C[2,2]+d0xd1xd2}*
1<=k<2
=min { 0+0+3x2x4} = 24
1<=k<2

We got C[1,2] for k value 1. so, put 1 in k table at (1,2).

C[2,3] = *min { C[2,2]+C[3,3]+d1xd2xd3}*
2<=k<3
=min { 0+0+2x4x2} = 16
2<=k<3

Put 2 in k table at (2,3).

	1	2	3	4
1	**0**	**24**		
2		0	16	
3			0	
4				0

C table

	1	2	3	4
1		**1**		
2			2	
3				
4				

K table

$$C[i,j] = \begin{cases} 0 & i=j \\ \min[(C[i,k] + C[k+1,j] + d_{i-1} \times d_k \times d_j)] & \end{cases}$$

for i<=k<j, since splitting can be anywhere b/w i and j

Matrix Chain Multiplication

Find the best order to multiply following matrices-

A1	A2	A3	A4
d0x d1	*d1x d2*	*d2x d3*	*d3x d4*
3x2	*2x4*	*4x2*	*2x5*

$$C[3,4] = \min_{3<=k<4} \{ C[3,3]+C[4,4]+d2xd3xd4 \}$$

$$= \min_{3<=k<4} \{ 0+0+4x2x5 \} = 40$$

Put 3 in k table at (3,4).

Step 3: Find the value of matrices of size 3

$$C[1,3] = \min_{1<=k<3} \begin{matrix} K=1 \\ K=2 \end{matrix} \left\{ \begin{matrix} C[1,1]+C[2,3]+d0xd1xd3, \\ C[1,2]+C[3,3]+d0xd2xd3 \end{matrix} \right.$$

$$= \min_{1<=k<3} \{ 0+16+3x2x2, 24+0+3x4x2 \}$$

= min {28, 48} = 28 (for k=1)

Put 1 in k table at (1,3).

	1	2	3	4
1	**0**	**24**	**28**	
2		0	16	
3			0	40
4				0

C table

	1	2	3	4
1		**1**	**1**	
2			2	
3				3
4				

K table

$$C[i,j] = \begin{cases} 0 & i=j \\ \min[(C[i,k] + C[k+1,j] + di-1xdkxdj)] & \end{cases}$$

for i<=k<j, since splitting can be anywhere b/w i and j

Matrix Chain Multiplication

Find the best order to multiply following matrices-

A1	A2	A3	A4
d0x d1	*d1x d2*	*d2x d3*	*d3x d4*
3x2	*2x4*	*4x2*	*2x5*

$$C[2,4] = \underset{2<=k<4}{min} \quad \begin{array}{l} K=2 \\ K=3 \end{array} \begin{cases} C[2,2]+C[3,4]+d1xd2xd4, \\ C[2,3]+C[4,4]+d1xd3xd4 \end{cases}$$

$$= \underset{2<=k<4}{min} \{ 0+40+2x2x5,\ 16+0+2x2x5 \}$$

= min {60, 36} = 36 (for k=3)

Put 3 in k table at (2,4).

Step 4: Find the value of matrices of size 4

	1	2	3	4
1	**0**	**24**	**28**	
2		0	16	36
3			0	40
4				0

C table

	1	2	3	4
1		**1**	**1**	
2			2	3
3				3
4				

K table

$$C[i,j] = \begin{cases} 0 & i=j \\ min[(C[i,k] +C[k+1,j] +di-1xdkxdj)] & \end{cases}$$

for i<=k<j, since splitting can be anywhere b/w i and j

Matrix Chain Multiplication

Find the best order to multiply following matrices-

A1	A2	A3	A4
d0x d1	*d1x d2*	*d2x d3*	*d3x d4*
3x2	*2x4*	*4x2*	*2x5*

Step 4: Find the value of matrices of size 4

$$C[1,4] = \min_{1<=k<4} \begin{cases} K=1 & C[1,1]+C[2,4]+d0xd1xd4, \\ K=2 & C[1,2]+C[3,4]+d0xd2xd4, \\ K=3 & C[1,3]+C[4,4]+d0xd3xd4, \end{cases}$$

C[1,4] = *min {C[1,1]+C[2,4]+d0xd1xd4, C[1,2]+C[3,4]+d0xd2xd4, C[2,3]+C[4,4]+d0xd3xd4}*

= min {0+36+3x2x5, 24+40+3x4x5, 28+0+3x2x5}

= min {66, 124, 58}

= 58 (for k= 3)

Put 3 in k table at (1,4).

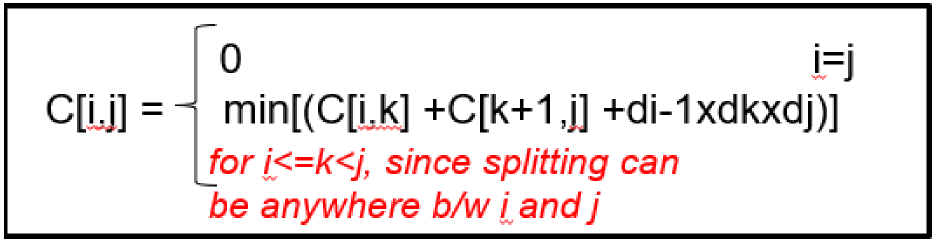
So, to multiply 4 matrices, the minimum Number (optimal) of scaler multiplication will be 58. So, how to parenthesize the matrices to get this cost?

	1	2	3	4
1	**0**	**24**	**28**	**58**
2		0	16	36
3			0	40
4				0

C table

	1	2	3	4
1		**1**	**1**	**3**
2			2	3
3				3
4				

K table



$$C[i,j] = \begin{cases} 0 & i=j \\ \min[(C[i,k] + C[k+1,j] + di\text{-}1xdkxdj)] \end{cases}$$

for i<=k<j, since splitting can be anywhere b/w i and j

Matrix Chain Multiplication

How to parenthesize the matrices to get the cost?

The K table tells how to parenthesize matrices to get the minimum cost.

1. The cost C[1,4] is found for k=3, so, first parenthesis must be place after third matrix.

(A1 A2 A3) (A4)

2. Now, matrices A1A2A3 must be parenthesize in such way so that cost could be minimized. Check the minimum cost found for matrices C[1,3]. It is found for k=1. so, second parenthesis must be place after first matrix.

((A1) (A2 A3)) (A4)

The above parenthesize is the final and it tell that first matrix A2 and A3 must be multiplied, and their resultant matrix is multiplied with A1 and their result will finally be multiplied with A4 to get the final result.

	1	2	3	4
1	**0**	**24**	**28**	**58**
2		0	16	36
3			0	40
4				0

C table

	1	2	3	4
1		**1**	**1**	**3**
2			2	3
3				3
4				

K table

Time Complexity:

Time taken to find the parenthesizing will be equal to the number of values to be filled in the C table = **$n(n+1)/2$ =4*5/2=10 = $O(n^2)$,** for **n** matrices. However, to find any value, such as C[1,4], all possible values of k have been tried out and k = i, i+1, … j. where, j is closed to n. So, to compute any value, it will take O(n) time in worst case. So, overall complexity **= $O(n^* n^2) = O(n^3)$.**

Travelling Salesman Problem

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

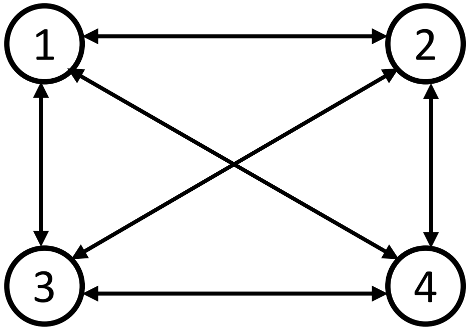
Brute-force Approach: The brute-force approach evaluate every possible path and select the best one. For n number of vertices in a graph, if start and end vertices are fixed, there will be $(n - 1)!$ number of possibilities. This approach will computationally very expensive and require exponential time.

Let us check, can we use DP?

DP will only applicable, if we can find optimal substructure and overlapping subproblems.

Travelling Salesman Problem

Let us take an example to check whether we can find optimal substructure and overlapping subproblems.



	1	2	3	4
1	0	1	2	3
2	1	0	4	2
3	1	2	0	5
4	3	4	1	0

Cost Matrix

If the salesperson starts from *city 1* and visit all other cities, then the problem can be formulated as follows:

$$T(1, \{2, 3, 4\}) = \min \begin{cases} (1, 2) + T(2, \{3, 4\}) & \text{Travel 2 from 1 and continue tour from 2, visit all nodes and get back to 1} \\ (1,3) + T(3, \{2, 4\}) & \text{Travel 3 from 1 and continue tour from 3, visit all nodes and get back to 1} \\ (1,4) + T(4, \{2, 3\}) & \text{Travel 4 from 1 and continue tour from 3, visit all nodes and get back to 1} \end{cases}$$

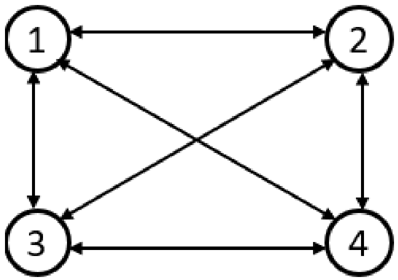
Travel all nodes and get back to 1

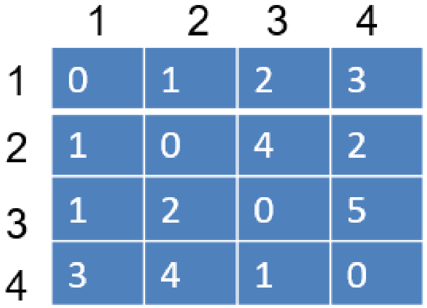
From the above, it has been observed that main problem can be broken into smaller subproblems. So, it has substructure.

Travelling Salesman Problem

$$T(1, \{2, 3, 4\}) = \min \begin{cases} (1, 2) + T(2, \{3, 4\}) \\ (1,3) + T(3, \{2, 4\}) \\ (1,4) + T(4, \{2, 3\}) \end{cases}$$

E(1, 2) = 1, E(1, 3) = 2, E(1, 4) = 3
T(2, {3, 4}), T(3, {2, 4}), T(4, {2, 3}) are unknown.





	1	2	3	4
1	0	1	2	3
2	1	0	4	2
3	1	2	0	5
4	3	4	1	0

Cost Matrix

$$T(2, \{3, 4\}) = \min \begin{cases} (2, 3) + T(3, \{4\}) \\ (2,4) + T(4, \{3\}) \end{cases}$$

E(2, 3) = 4, E(2, 4) = 2, while T(3, {4}), T(4, {3}) are unknown.

$$T(3, \{2, 4\}) = \min \begin{cases} (3, 2) + T(2, \{4\}) \\ (3, 4) + T(4, \{2\}) \end{cases}$$

E(3, 2) = 2, E(3, 4) = 5, while T(2, {4}), T(4, {2}) are unknown.

$$T(4, \{2, 3\}) = \min \begin{cases} (4, 2) + T(2, \{3\}) \\ (4,3) + T(3, \{2\}) \end{cases}$$

E(4, 2) = 4, E(4, 3) = 1, while T(2, {3}), T(3, {2}) are unknown.

$$T(3, \{4\}) = (3,4) + T(4, \{\Phi\})$$
$$T(4, \{3\}) = (4,3) + T(3, \{\Phi\})$$
$$T(2, \{4\}) = (2,4) + T(4, \{\Phi\})$$
$$T(4, \{2\}) = (4,2) + T(2, \{\Phi\})$$
$$T(2, \{3\}) = (2,3) + T(3, \{\Phi\})$$
$$T(3, \{2\}) = (3,2) + T(2, \{\Phi\})$$

E(3, 4) = 5, E(4, 3) = 1, E(2, 4) = 2, E(4, 2) = 4, E(2, 3) = 4, E(3, 2) = 2,
T(4, {Φ}) = E(4,1) = 3, T(3, {Φ}) = E(3, 1) =1, and T(2, {Φ}) = E(2, 1) = 1

Travelling Salesman Problem

T(1, {2, 3, 4}) = min
(1, 2) + T(2, {3, 4}) **=> 1+4 =5**
(1,3) + T(3, {2, 4}) **=> 2+5 =7**
(1,4) + T(4, {2, 3}) **=> 3+4 =7**

E(1, 2) = 1, E(1, 3) = 2, E(1, 4) = 3

T(1, {2, 3, 4}) = min (5, 7, 7)= 5
Min tour cost = 5

T(2, {3, 4}) = min
(2, 3) + T(3, {4}) **=>4+8 = 12**
(2,4) + T(4, {3}) **=> 2+2 = 4**
E(2, 3) = 4, E(2, 4) = 2

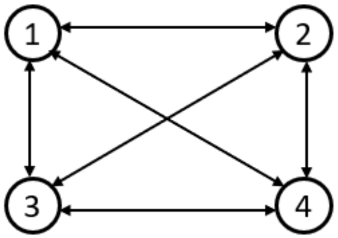
T(2, {3, 4}) = min (12,4)=>4

T(3, {2, 4}) = min
(3, 2) + T(2, {4}) **=> 2+5 = 7**
(3, 4) + T(4, {2}) **=> 5+5 = 10**
E(3, 2) = 2, E(3, 4) = 5

T(3, {2, 4}) = min (7,10)=>7

T(4, {2, 3}) = min
(4, 2) + T(2, {3}) **=> 4+5 =9**
(4,3) + T(3, {2}) **=> 1+3 = 4**
E(4, 2) = 4, E(4, 3) = 1

T(4, {2, 3}) = min (9,4)=>4



	1	2	3	4
1	0	1	2	3
2	1	0	4	2
3	1	2	0	5
4	3	4	1	0

Cost Matrix

T(3, {4}) = (3,4) + T(4, {Φ}) **= 5+3 = 8**
T(4, {3}) = (4,3) + T(3, {Φ}) **= 1+1 = 2**
T(2, {4}) = (2,4) + T(4, {Φ}) **= 2+3 = 5**
T(4, {2}) = (4,2) + T(2, {Φ}) **= 4+1 = 5**
T(2, {3}) = (2,3) + T(3, {Φ}) **= 4+1 = 5**
T(3, {2}) = (3,2) + T(2, {Φ}) **= 2+1 = 3**

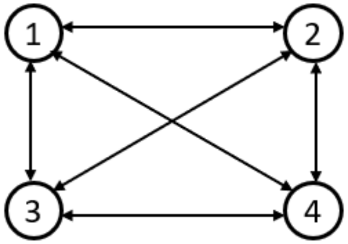
E(3, 4) = 5, E(4, 3) = 1, E(2, 4) = 2, E(4, 2) = 4, E(2, 3) = 4, E(3, 2) = 2,

T(4, {Φ}) = E(4,1) = 3, T(3, {Φ}) = E(3, 1) =1, and T(2, {Φ}) = E(2, 1) = 1

Travelling Salesman Problem

$$T(1, \{2, 3, 4\}) \underset{\text{min}}{=} \begin{cases} (1, 2) + T(2, \{3, 4\}) \Rightarrow 1+4 =5 \\ (1,3) + T(3, \{2, 4\}) \Rightarrow 2+5 =7 \\ (1,4) + T(4, \{2, 3\}) \Rightarrow 3+4 =7 \end{cases}$$

E(1, 2) = 1, E(1, 3) = 2, E(1, 4) = 3



	1	2	3	4
1	0	1	2	3
2	1	0	4	2
3	1	2	0	5
4	3	4	1	0

Cost Matrix

T(1, {2, 3, 4}) = min (5, 7, 7)= 5

Min tour cost = 5

$$T(2, \{3, 4\}) \underset{\text{min}}{=} \begin{cases} (2, 3) + T(3, \{4\}) \Rightarrow 4+8 = 12 \\ (2,4) + T(4, \{3\}) \Rightarrow 2+2 = 4 \end{cases}$$

E(2, 3) = 4, E(2, 4) = 2

T(2, {3, 4}) = min (12,4)=>4

$$T(3, \{2, 4\}) \underset{\text{min}}{=} \begin{cases} (3, 2) + T(2, \{4\}) \Rightarrow 2+5 = 7 \\ (3, 4) + T(4, \{2\}) \Rightarrow 5+5 = 10 \end{cases}$$

E(3, 2) = 2, E(3, 4) = 5

T(3, {2, 4}) = min (7,10)=>7

$$T(4, \{2, 3\}) \underset{\text{min}}{=} \begin{cases} (4, 2) + T(2, \{3\}) \Rightarrow 4+5 =9 \\ (4,3) + T(3, \{2\}) \Rightarrow 1+3 = 4 \end{cases}$$

E(4, 2) = 4, E(4, 3) = 1

T(4, {2, 3}) = min (9,4)=>4

T(3, {4}) = (3,4) + T(4, {Φ}) **= 5+3 = 8**

T(4, {3}) = (4,3) + T(3, {Φ}) **= 1+1 = 2**

T(2, {4}) = (2,4) + T(4, {Φ}) **= 2+3 = 5**

T(4, {2}) = (4,2) + T(2, {Φ}) **= 4+1 = 5**

T(2, {3}) = (2,3) + T(3, {Φ}) **= 4+1 = 5**

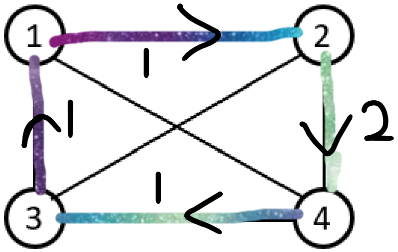
T(3, {2}) = (3,2) + T(2, {Φ}) **= 2+1 = 3**

Find tour

$T(1, \{2, 3, 4\}) = \boxed{(1, 2)} + T(2, \{3, 4\})$

$T(2, \{3, 4\}) = \boxed{(2, 4)} + T(4, \{3\})$

$T(4, \{3\}) = \boxed{(4,3)} + T(3, \{Φ\})$



Travelling Salesman Problem

From the example, it is clear that main problem can be written in terms of small problems and small problem is solved recursively until we reached to the base case. However, the complexity of the recursive approach would very high (exponential). **Can we reduce time complexity?**

Complexity can be reduced using DP. DP will only applicable if we can write recursive equation and find overlapping solutions.

$$T(i, S) = \min_{j \in S} \begin{cases} (i, 1) & S = \Phi \\ ((i, j) + T(j, S-\{j\})) & S \mathrel{!=} \Phi \end{cases}$$

Every vertex is picked from S

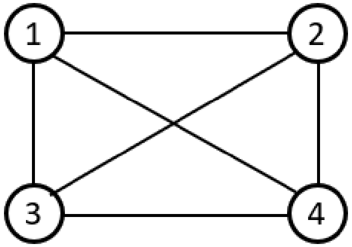
If there will no edge b/w two vertices (vertices are not adjacent) cost will ∞.

Let us construct recursion tree for the previous example.

Travelling Salesman Problem

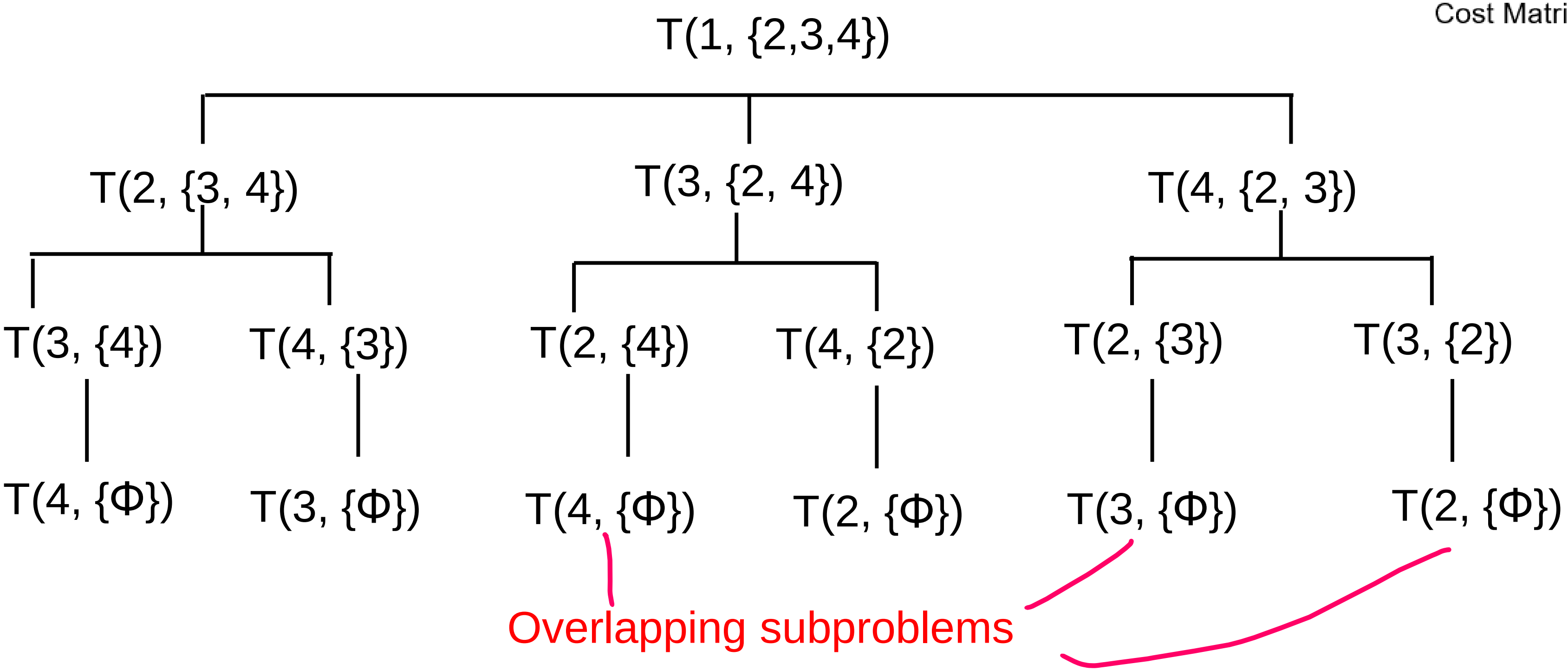
$$T(i, S) = \begin{cases} (i, 1) & S = \Phi \\ \min\limits_{j \in S}\left((i, j) + T(j, S-\{j\}) \right) & S \mathrel{!=} \Phi \end{cases}$$

Every vertex is picked from S



	1	2	3	4
1	0	1	2	3
2	1	0	4	2
3	1	2	0	5
4	3	4	1	0

Cost Matrix



Overlapping subproblems

There are total 15 subproblems and out of which 3 are repeating. So, total unique subproblems are 12, ($(n-2)*2^n$) which is still very large. DP approach can also not be able reduce complexity below exponential. So, it is not recommended to use DP for TSP.

Travelling Salesman Problem

The number of unique subproblems for n vertices will be equal to $(n-2)*2^n$, which is also exponential. The time complexity to find the shortest path = $O(n\ 2^n)$.

The time complexity of the DP approach is lesser than the brute-force approach $O(n!)$. However, it is still very high, so it is normally not recommended to use DP for TSP problem.