

ALGORITHMS - DYNAMIC PROGRAMMING

This paradigm is bit different but very similar to Divide and Conquer approach.

In “Dynamic Programming” the word “Programming” doesn’t mean we are going to use some programming language, instead, it means that we are going to use some tables and “dynamic” means we dynamically decide whether to use a function or to call a program (use table).

Whenever our main problem requires solving sub-problems again and again, instead to solving the sub-problem every time we can maintain the solutions in a table and we could use them again.

Let’s demonstrate with an example.

Fibonacci Sum:

Calculate the sum of the Fibonacci series up to ‘n’ terms.

Method: This will be calculated as below

$$f(n) = f(n-1) + f(n-2), n > 1$$

$$f(n) = 1, n = 1$$

$$f(n) = 0, n = 0$$

The program for it will look like below along with the recursion tree for this recursive program.

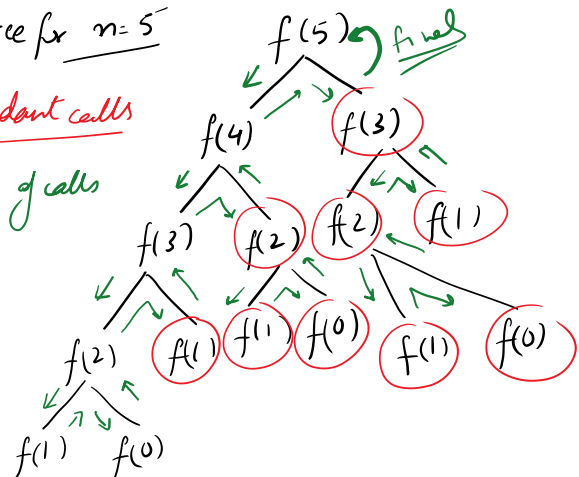
```
F(n)
{
    If(n==0)
        return 0;
    if(n==1)
        return 1;

    return( f(n-1) + f(n-2))
}
```

recursion tree for $n=5$

○ = redundant calls

→ direction of calls



Such kind of computation may reach to a time complexity of $O(2^n)$ which is exponential. We can see that more than half of the calls to the function $F(n)$ are redundant from the tree. What we can do instead is that *we can save the result of the calls for which we know will be used later in a table in order to reduce the time complexity. The saved result can be checked on every function call for reducing redundancy of calls.*

The modified program can look like.

```

F(n)
{
    If(n==0)
        return 0;
    if(n==1)
        return 1;

    Check the table for T(n)

    If(T(n) is empty)
        return( f(n-1) + f(n-2))
}

```

Using this approach, the number of calls will be $\sim n$

Time complexity = $O(n)$ (from $O(2^n)$)

★ *Note: So, if a problem is having some recursive equation and if the recursive equation shows that the problem is having overlapping sub problems, we can apply dynamic programming to optimize the time complexity of the problem.* ★ ★ ✓

★ ☺ ✓ *It will be used when the sub-problems are overlapping. Therefore, there will be recursive problems where DP can't be used like Merge Sort, etc.*

Let's take the problems which apply dynamic programming.

MATRIX CHAIN MULTIPLICATION (Dynamic Programming)

Matrix 1 = 3×2 , Matrix 2 = 2×3 , resulting dimensions = 3×3

So, how many multiplications (scalar) are required to multiply these two matrices?

Result contains: 3 rows, 3 columns and for each element of this matrix the number of scalar multiplications required = 2.

Therefore, $3 \times 3 \times 2$ scalar multiplications are required to multiply these two matrices.

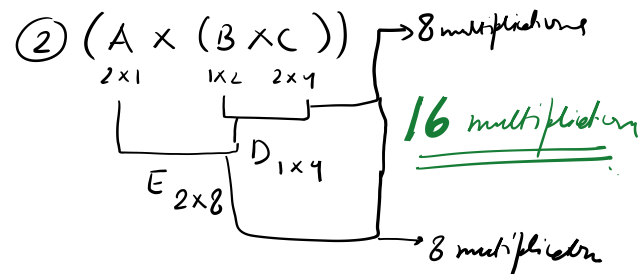
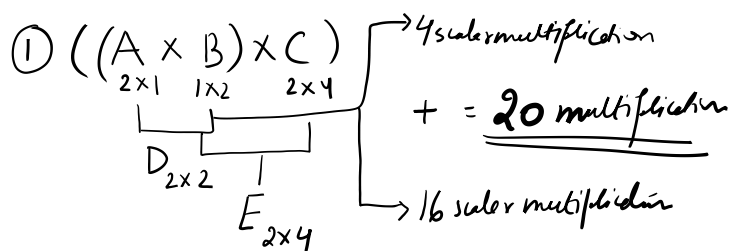
From this result:

1. Matrix A dimensions = $P \times Q$
2. Matrix B dimensions = $Q \times R$
3. Number of scalar multiplications required to multiply these two matrices = $(P \times R) \times Q$

Let's extend it to three matrices.

1. Matrix multiplication is associative in nature this means $((A \times B) \times C) = (A \times (B \times C))$

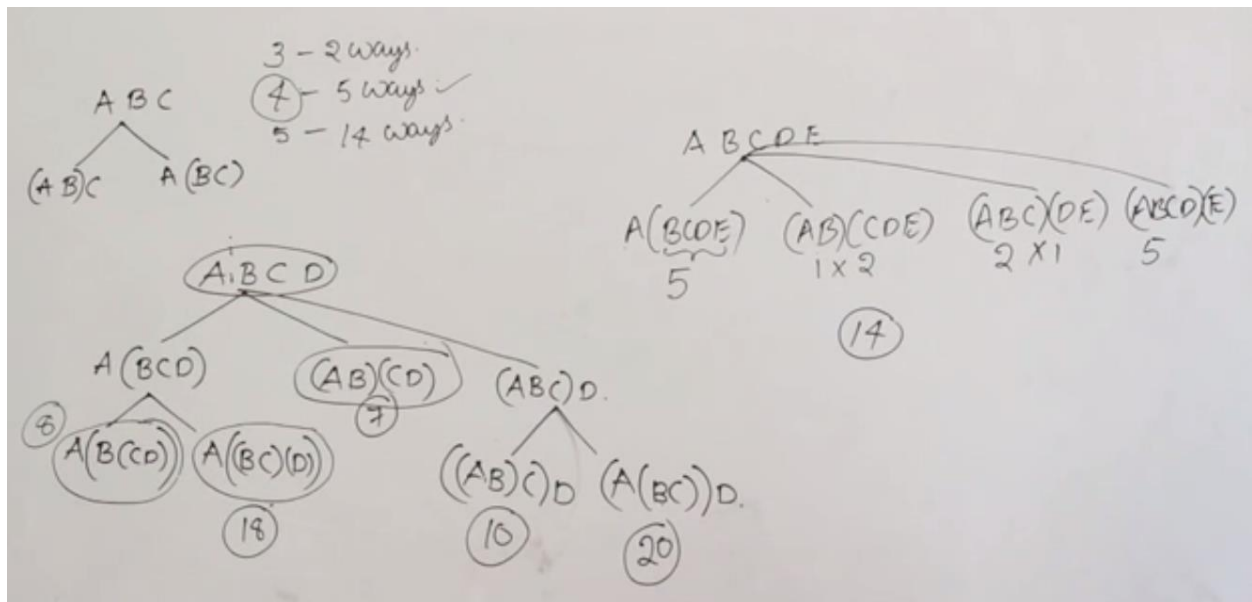
Let's take two examples with different *parenthesized* as associativity can allow and calculate the number of scalar multiplications required to multiply the three matrices.



So, by changing the way we select the multiplication order we can bring down the number of scalar multiplications required to multiply matrices.

If the dimensions of the matrices were significantly large we would have seen very significant effect of the optimal selection of the parenthesized structure which is going to give me least number of scalar multiplications.

For three matrices there are two ways to parenthesize. Let's see for 'n' matrices how many ways there will be to parenthesize.



There is also a formula for calculating the number of ways a matrix chain multiplication can be parenthesized. It is given as:

$$\frac{(2n)!}{(n+1)!n!} \text{ where } n = (\text{number of matrices in the chain} - 1)$$

For 6 matrices, number of ways it can be parenthesized = $\frac{10!}{6!5!} = \frac{10 \times 9 \times 8 \times 7}{5 \times 4 \times 3 \times 2 \times 1} = 42 \text{ ways}$

To solve this problem using dynamic programming, I need to have

1. **Optimal Substructure:** Given the big problem, I should be able to write it in terms of small problems in such a way that the solutions to the small problems put together should make the solution to the big problem.
2. **Overlapping Problems:** When I am solving some problem there should be a smaller problem that is a part of the other small problem as well. Thus the smaller problems should be overlapping

OPTIMAL SUBSTRUCTURE and RECURSIVE SOLUTION for MATRIX CHAIN MULTIPLICATION

Optimal Substructure

To solve the problem using dynamic programming, first we divide the problem into smaller ones and then after that we can find a recursive solution for it.

Some conventions:

Chain Multiplication: $A_1 \ A_2 \ A_3 \ \dots \ A_n$
Dimensions: $p_0 \times p_1 \ p_1 \times p_2 \ p_2 \times p_3 \ \dots \ p_{n-1} \times p_n$

Generally, if I have the Matrix chain multiplication as, I can parenthesize as:

$$A_i A_{i+1} A_{i+2} \dots A_j = A_i A_{i+1} \dots A_k | A_{k+1} \dots A_j$$

$p_{i-1} \times p_k \quad p_k \times p_j$

Dimensions:

Total number of scalar multiplications:

$$\text{Num}_{\text{scalar mul}}(A_i A_{i+1} \dots A_k) + \text{Num}_{\text{scalar mul}}(A_{k+1} \dots A_j) + p_{i-1} \times p_k \times p_j$$

★
 I can split the sequence after $i, i+1, \dots$ anywhere so taking 'k' to represent splitting somewhere

Recursive Solution/Equation

Let's introduce another term $m[i, j]$

$m[i, j]$ = minimum number of scalar multiplications required in order to parenthesize $A[i]$ to $A[j]$

So the base condition for recursive equation will be: whenever I have only one matrix to multiply, the minimum number of scalar multiplications = 0

When $i \neq j$, then we have to parenthesize and calculate the minimum number of scalar multiplications. The recursive equation can look like:

$$m[i, j] = \begin{cases} 0, & i = j \\ \min\{m[i, k] + m[k+1, j] + p_{i-1} \cdot p_k \cdot p_j\}, & i \leq k \leq j \end{cases}$$

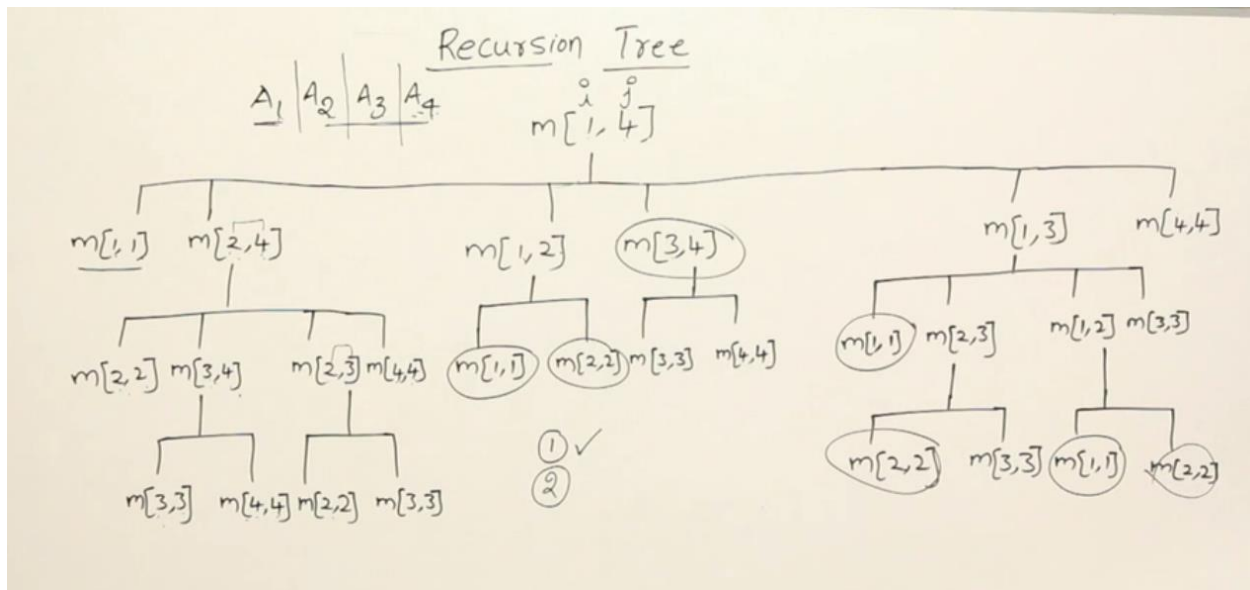
Now we know how to split, but we don't know where to split, so we have to consider all the possibilities. The number of possibilities we get:

K can be $i, i+1, i+2, \dots, j-1$ therefore, $(j-i)$ possibilities

When we are going to split there will be $(j-i)$ ways.

The next part you should be looking at is whether there are overlapping sub-problems or not.

Overlapping sub-problems



Above is the recursion tree for the ways in which 4 matrices can be multiplied. The notation $m[i, j]$ denotes the scalar multiplications required to multiply $m[i, j]$ chain of matrices.

We see that a lot of calls to the functions are repeated, like $m[1, 1]$, $m[2, 2]$, $m[3, 4]$ etc.

Therefore, *there are overlapping sub-problems*.

So, we can reuse the calculations for such repeated calls by storing the values in a table.

This storage of values in a table can be done in two ways:

1. **Bottom-up approach:** go to the bottom from top, calculate the bottom level's values then move one level up until you reach root
2. **Top-down approach:** follow the tree the way recursion happens and as you call the function, calculate the value and store it in the table.

So, how are we going to store them in the table?

DYNAMIC PROGRAMMING BOTTOM UP INITIATION FOR MATRIX CHAIN MULTIPLICATION

For $A(1, 4)$

From the recursion graph above, the unique function calls are made as follows:

(11)	(22)	(33)	(44)	Size of problem = 1	No. of sub-problems of size 1 = 4
(12)	(23)	(34)		Size of problem = 2	No. of sub-problems of size 2 = 3
(13)	(24)			Size of problem = 3	No. of sub-problems of size 3 = 2
(14)				Size of problem = 4	No. of sub-problems of size 4 = 1

Total number of sub-problems = $4+3+2+1 = 10$

For $A(1,n)$

Total number of sub-problems = $n + (n-1) + (n-2) + \dots + 1 = \frac{n(n+1)}{2}$

Therefore, in order to solve a matrix chain multiplication of $A(1\dots n)$ the number of sub-problems you may have to solve is $\frac{n(n+1)}{2}$.

Let's take an example:

Example: Find the least number of scalar multiplications required in multiplying the following 4 matrices with dimensions given.

A_1	A_2	A_3	A_4
1×2	2×1	1×4	4×1
$p_0 \ p_1$	$p_1 \ p_2$	$p_2 \ p_3$	$p_3 \ p_4$

The unique sub-problems are as follows. Let's find the total scalar multiplications required for each sub-problem.

(11) 0 (22) 0 (33) 0 (44) 0

(12) 2 (23) 8 (34) 4

(13) 6 (24) 6

(14) 7

1. (11) (22) (33) (44)

a. The number of scalar multiplications here will be 0.

2. $(1,2) = 1 \times 2 \times 1 = 2$

3. $(2,3) = 2 \times 1 \times 4 = 8$

4. $(3,4) = 1 \times 4 \times 1 = 4$

5. $(1,3) = \min \begin{cases} ((1,1) + (2,3)) + p_0 p_1 p_3 = 0 + 8 + 8 = 16 \\ ((1,2) + (3,3)) + p_0 p_2 p_3 = 2 + 0 + 4 = 6 \end{cases} = 6$

6. $(2,4) = \min \begin{cases} ((2,2) + (3,4)) + p_1 p_2 p_4 = 0 + 4 + 2 = 6 \\ ((2,3) + (4,4)) + p_1 p_3 p_4 = 8 + 0 + 8 = 16 \end{cases} = 6$

7. $(1,4) = \min \begin{cases} ((1,1) + (2,4)) + p_0 p_1 p_4 = 0 + 6 + 2 = 8 \\ ((1,2) + (3,4)) + p_0 p_2 p_4 = 2 + 4 + 1 = 7 \\ ((1,3) + (4,4)) + p_0 p_3 p_4 = 6 + 0 + 4 = 10 \end{cases} = 7$

Now, the time taken to find out the minimum number of scalar multiplications will depend on number of splits and total number of sub-problems.

1. For $A(1,4)$ number I had to look at $4-1 = 3$ splits and each split takes $O(1)$ time so for $A(1,n)$ I will have to look at $(n-1)$ splits for each split $O(1)$ time so for looking at the splits I need $O(n)$ time.
2. In order to solve a matrix chain multiplication of $A(1 \dots n)$ the number of sub-problems you may have to solve is $\frac{n(n+1)}{2}$

Total time complexity for matrix chain multiplication with dynamic programming bottom-up method = $O(\frac{n(n+1)}{2} \cdot n) = O(n^3)$ (WORST CASE)

Space Complexity = $O(n^2)$ (to store the $\frac{n(n+1)}{2}$ sub problems)

MATRIX CHAIN MULTIPLICATION USING DYNAMIC PROGRAMMING TOP DOWN APPROACH (MEMOIZATION)

Using the top down approach, also called memoization, has the same time complexity as that of the bottom-up approach. Since this algorithm is a recursive one the space complexity is a little more than that of the bottom-up approach as the stack for recursive calls is also maintained (stack size = the height of the recursion tree). But, the worst case time and space complexity is same.

Time complexity = $O(n^3)$

Space complexity = $O(n^2) + O(\text{height of recursion tree}) = O(n^2)$ (worst case)

Example: Gate 2011

split

$$\boxed{\boxed{(1)(2 \ 3)}(4)}$$

Find the least number of scalar multiplications required for multiplying the matrices.

(11)	(22)	(33)	(44)	$11 = 0 \quad 22 = 0 \quad 33 = 0 \quad 44 = 0$
(12)	(23)	(34)		$(12) = 10 \times 100 \times 20 = 20000$
(13)	(24)			$(23) = 100 \times 20 \times 5 = 10000$
(14)				$(34) = 20 \times 5 \times 80 = 8000$
				$(13) = \min \left\{ \begin{aligned} &(1,1) + (2,3) + p_1 p_1 p_3 = 0 + 10000 + 5000 = 15000 \\ &(1,2) + (3,3) + p_1 p_2 p_3 = 20000 + 10000 + 0 = 30000 \end{aligned} \right.$
				$(24) = \min \left\{ \begin{aligned} &(2,2) + (3,4) + p_1 p_2 p_4 = 0 + 9000 + 160000 = 169000 \\ &(2,3) + (4,4) + p_1 p_3 p_4 = 10000 + 0 + 40000 = 50000 \end{aligned} \right.$
				$(14) = \min \left\{ \begin{aligned} &(1,1) + (2,4) + p_1 p_1 p_4 = 0 + 50000 + 80000 = 130000 \\ &(1,2) + (3,4) + p_1 p_2 p_4 = 20000 + 8000 + 160000 = 188000 \\ &(1,3) + (4,4) + p_1 p_3 p_4 = 15000 + 0 + 40000 = 55000 \end{aligned} \right.$

LONGEST COMMON SUBSEQUENCE (Dynamic Programming)

Difference between substring and subsequence

Substring has to be made up from consecutive letters of a string, subsequence does not have to be.

To find out the longest common subsequences between two strings A and B where A and B have lengths m and n respectively:

1. Find all the subsequences of the string A (time = $O(2^m)$)
2. Find for each subsequence whether it is a subsequence of B (time = $O(n2^m)$)
3. Find the longest common subsequence (time = $O(2^m)$)

Dynamic programming can be applied to this problem. Let's see how it can be done.

To apply dynamic programming, we need three things:

1. **Optimal substructure:** the problem should be divisible into smaller problems such that the solutions to the smaller problem is a part of the solution to the main problem.
 - a. The optimal substructure for longest common subsequence between two strings will look like below:

If I have two strings $x_1x_2 \dots x_n$ and $y_1y_2 \dots y_m$:

I am going to check if x_n and y_m are equal

If they are equal they will be part of the longest common subsequence and I will

Continue the search for longest subsequence in $x_1x_2 \dots x_{n-1}$ and $y_1y_2 \dots y_{m-1}$

If they are not equal then I can continue the search for it in

1. $x_1x_2 \dots x_n$ and $y_1y_2 \dots y_{m-1}$ or $x_1x_2 \dots x_{n-1}$ and $y_1y_2 \dots y_m$

2. **Recursive Equations:**

The recursive equation or recurrence relation for this problem looks like:

(Convention: $c[i,j]$ = length of longest common subsequence between strings x and y where x is running from 1..i ($x_1x_2 \dots x_i$) and y is running from 1..j ($y_1y_2 \dots y_j$))

$$c[i,j] = \begin{cases} 0; & x = 0, y = 0 \\ 1 + c[i-1, j-1]; & i, j > 0, x_i = y_j \\ \max(c[i-1, j], c[i, j-1]); & i, j > 0, x_i \neq y_j \end{cases}$$

3. **Overlapping Sub problems**

The smaller problems to which the bigger problem has been divided into should be redundant so that their redundant calculations can be ignored.

Let's start understanding:

Best case scenario:

For the best case scenario all the elements in the two strings will have equal length and all elements will match. We will have $O(n)$ time and space complexity.

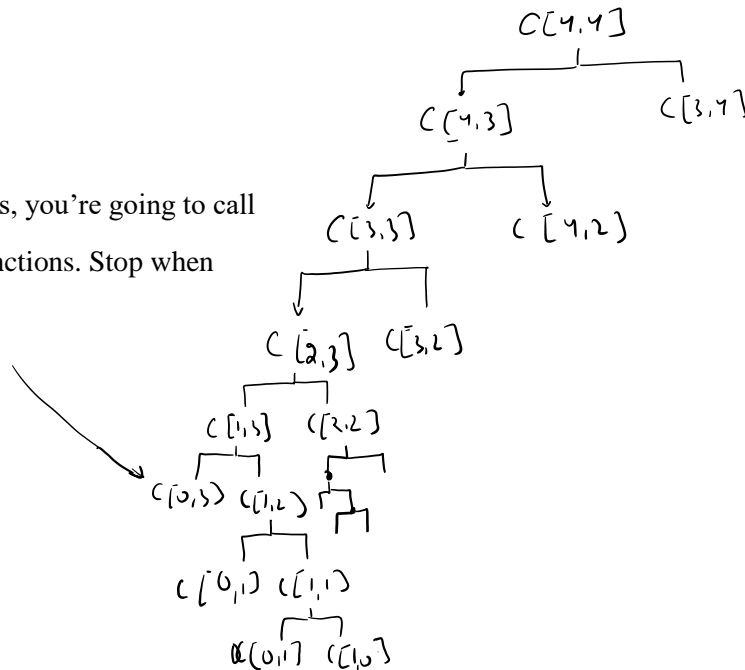
Worst case scenario:

$X = \{A, A, A, A\}$

$Y = \{B, B, B, B\}$

Working of the program in worst case scenario

For such small strings, you're going to call
A lot of recursive functions. Stop when
Something reaches 0

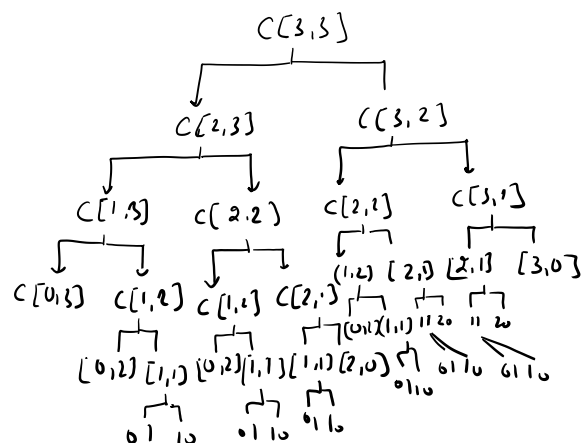


Now what will be the depth of the tree?

Because every time we are going to decrement 'i' first and then 'j' so we can estimate that the height of the tree can go to $O(n+m)$ to reach 0. **Therefore, in worst case the height of the tree is $O(n+m)$. The number of nodes in the tree can go to $O(2^{n+m})$ even if it is half full.**

Inside a function time taken is constant $O(1)$ because we are just making one comparison. But there are 2^{n+m} such function calls, therefore, the time taken for this entire recursive function = $O(2^{n+m})$ which is not that good of a solution.

Dynamic programming approach: Here we can see in the recursion tree that a lot of overlapping sub-problems are there. It turns out that the number of unique function calls will be **$m \times n$** . Let's take an example where the lengths of both the strings are 3.



Unique function calls:

33

23 13 03 02 01 10 11 12

22 21 32 31 30 20 00

Unique sub-problems

(33) (32) (31)

(23) (22) (21)

(13) (12) (11)

Now that we have 16 unique sub-problems, we can create a table and store their calculated values and look them up when next redundant calculation is to be done.

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

Now the question is in which way we are going to evaluate this table. You can do it in bottom-up or top-down approach. Since each element calculation depends on the elements which are on the top, left or diagonal to it. We can start from the first row and then go either row-wise or column wise calculations. Let's see an example of this calculation and algorithm.

Moreover: **Time complexity: $O(mn)$ Space complexity: $O(mn)$** (bottom-up approach)

Time complexity: $O(mn)$ Space complexity: $O(mn + (m+n)) = O(mn)$ (memoization)

Example 1:

$X = (AAB)$ $Y = (ACA)$. Find the longest common subsequence between X and Y.

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

0	0	0	0
0	1	1	1
0	1	1	2
0	1	1	2

→ longest common subsequence
AA

for following:
 $1 + C[i-1, j-1]$
 $\max(C[i, j-1], C[i-1, j])$

Example 2:

$X = \{A, B, C, B, D, A, B\}$, $Y = \{B, D, C, A, B, A\}$. Find the longest common subsequence between these strings.

	0	1	2	3	4	5	6
	Y	B	D	C	A	B	A
0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1
2	0	1	1	1	1	2	2
3	0	1	1	2	2	2	2
4	0	1	1	2	2	3	3
5	0	1	2	2	3	3	3
6	0	1	2	2	3	4	4
7	0	1	2	2	3	4	4

BCBA → longest common subsequence
 BDAB, BCAB

LONGEST COMMON SUBSEQUENCE ALGORITHM

```

LCS(X, Y)
{
    m = X.length
    n = Y.length
    let c[1...m, 1...n] be a new table
    for i = 1 to m
        c[i, 0] = 0
    for j = 0 to n
        c[0, j] = 0
    for i = 1 to m
        for j = 1 to n
            if x[i] == y[j]
                c[i, j] = c[i-1, j-1] + 1
            else
                c[i, j] = max(c[i-1, j], c[i, j-1])
    return c
}

```

GATE 2014

Question: Given A = qpqrr, B = pqpqrqp, calculate $x + 10y$ where x is the length of longest common subsequence and y is the number of such subsequences.

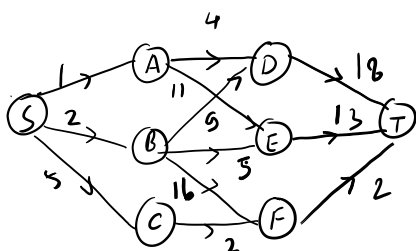
	0	q	p	q	r	r
A	0	0	0	0	0	0
0	0	0	1	1	1	1
1 p	0	1	1	2	2	2
2 q	0	1	2	2	2	2
3 p	0	1	2	2	3	3
4 r	0	1	2	3	3	3
5 q	0	1	2	3	4	4
6 r	0	1	2	3	4	4
7 p	0	1	2	3	4	4

qpqrr qpqrqp ~~qpqrr~~ qprr

$$\Rightarrow x = 4 \text{ \& } y = 3 \Rightarrow x + 10y = 4 + 30 = \underline{\underline{34}}$$

MULTI STAGE GRAPHS (Shortest path using Dynamic Programming)

A graph divided into multiple stages in such a way that there will be no edges within the vertices of a stage but only from one stage to the other stage. This graph is directed weighted graph with stages.



Characteristics of multi-stage graph

1. Within one stage, there will be no edges between the vertices
2. If there is an edge, it will definitely be from stage i to stage $i+1$
3. There is a Source node
4. There is a target node

Problem: We have one source node S and at the end stage we have a Target/Destination node. Find the shortest path between S and T.

I can apply Dijkstra's algorithm for this, but Dijkstra's is designed to find the shortest path from one node to all the other nodes, so before reaching the destination, this algorithm is going to find the shortest path for all the nodes. That is going to take more time. The order of time complexity is $O(E \log V)$.

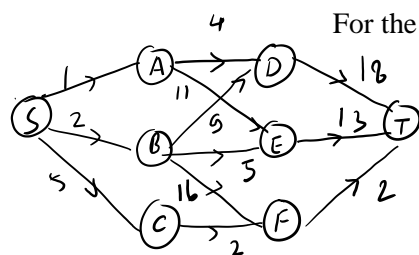
But can I find another way which takes less time? And also the graph is not a normal graph, so I can leverage the properties of such a multi stage graph.

The next thing that we can apply is the greedy method which works like this: Take the Start point and look at the edges going outside of it, connect to the vertex which has the least weight. Repeat this process for the new vertex added and so on.

This method is also not going to work as there are alternate paths which are much smaller. We tried to apply the greedy method because they are going to be faster than the dynamic programming solution. But since we didn't get the right answer, we can fall back to the dynamic programming solution.

Let's see how dynamic programming can be applied to this.

Optimal Sub-structure and recursive equations for finding minimum path in a multi-stage graph



For the optimal sub-structure. If S is the starting point then I can go from $S \rightarrow A$

$S \rightarrow B$ or $S \rightarrow C$ and then find the shortest path from $A \rightarrow T$ or $B \rightarrow T$

Or $C \rightarrow T$ and move on. So this is how I am going to divide the problem

Into smaller ones.

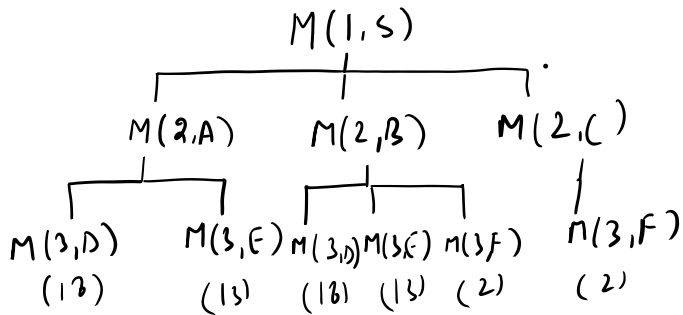
Let us introduce some notations and write the recursive equations.

$M(I, X)$ = minimum cost from stage I from node X . This can be sub-structured as below:

$$M(1, s) = \begin{cases} s \rightarrow A, M(2, A) \\ s \rightarrow B, M(2, B) \\ s \rightarrow C, M(2, C) \end{cases}$$

The recursion tree for the recursive equation is shown on the next page. To apply dynamic programming, I should have overlapping sub-problems in the recursion tree. Let's see if we have them or not.

Recursion tree:

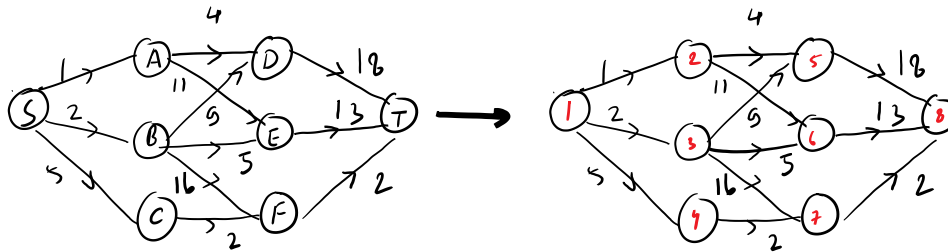


We will stop when we have reached the last
Stage. $M(N-1, i)$

The depth of the tree is number of levels in the graph: If graph has k levels depth = $O(k)$ so space complexity = $O(k)$. The time complexity = $O(2^k)$ so it is an exponential algorithm. Let's see the overlapping sub-problem removal to reduce this time complexity to $O(E)$ where $E = V^2$.

Therefore, the number of unique sub-problems present = number of nodes.

Bottom-up Dynamic Programming for shortest path in multi-stage graph



Let's give a number to the nodes in such a way that the nodes present in the higher stage should have greater number than the nodes present in the lower stage otherwise our algorithm might fail.

This time our dynamic programming DS is an array and for every index we have a vertex.

Let T be the array for storing our result for unique sub-problems:

	22	18	4	18	13	2	0
--	----	----	---	----	----	---	---

index/

1 2 3 4 5 6 7 8

vertex
number

$T[i]$ = Cost of vertex i to T (target vertex). This is calculated as below:

$$T[i] = \min\{Cost(i, j) + T[j]\}$$

$$j = i + 1 \text{ to } n$$

Let's solve the problem:

1. $T[8] = C(8, 8) = 0$
2. $T[7] = C(7, 8) + T[8] = 2 + 0 = 2$

$$\begin{aligned}
3. \quad T[6] &= \min \begin{cases} C(6,7) + T[7] = \infty + 2 \\ C(6,8) + T[8] = 13 \end{cases} = \mathbf{13} \\
4. \quad T[5] &= \min \begin{cases} C(5,6) + T[6] = \infty + 13 \\ C(5,7) + T[7] = \infty + 2 \\ C(5,8) + T[8] = 18 + 0 \end{cases} = \mathbf{18} \\
5. \quad T[4] &= \min \begin{cases} C(4,5) + T[5] = \infty + 18 \\ C(4,6) + T[6] = \infty + 13 \\ C(4,7) + T[7] = 2 + 2 \\ C(4,8) + T[8] = \infty + 0 \end{cases} = \mathbf{4} \\
6. \quad T[3] &= \min \begin{cases} C(3,4) + T[4] = \infty + 4 \\ C(3,5) + T[5] = 9 + 18 \\ C(3,6) + T[6] = 5 + 13 \\ C(3,7) + T[7] = 16 + 2 \\ C(3,8) + T[8] = \infty + 0 \end{cases} = \mathbf{18} \\
7. \quad T[2] &= \min \begin{cases} C(2,3) + T[3] = \infty + 18 \\ C(2,4) + T[4] = \infty + 4 \\ C(2,5) + T[5] = 4 + 18 \\ C(2,6) + T[6] = 11 + 13 \\ C(2,7) + T[7] = \infty + 2 \\ C(2,8) + T[8] = \infty + 0 \end{cases} = \mathbf{22} \\
8. \quad T[1] &= \min \begin{cases} C(1,2) + T[2] = 1 + 22 \\ C(1,3) + T[3] = 2 + 18 \\ C(1,4) + T[4] = 5 + 4 \\ C(1,5) + T[5] = \infty + 18 \\ C(1,6) + T[6] = \infty + 13 \\ C(1,7) + T[7] = \infty + 2 \\ C(1,8) + T[8] = \infty + 0 \end{cases} = \mathbf{9}
\end{aligned}$$

Final Answer: Minimum distance from S \rightarrow T = 9

Bottom up dynamic algorithm is always an ITERATIVE ALGORITHM

Time complexity:

1. How many sub problems need to be evaluated? **n**
2. What is the time taken to evaluate each sub problem?

The time taken to evaluate each subproblem = Calculation of cost for each vertex greater than the current one.

So for vertex 8: 1 unit calculation

For vertex 7: 1 unit calculation

6: 2 units 2: 6 units **Total work = 1 + 2 + 3 + ... + (n-1) = O(n²) = O(V²) = O(E)**

5: 3 units 1: 7 units Therefore, **TIME COMPELXITY = O(E)**

4: 4 units

3: 5 units

0-1 KNAPSACK PROBLEM

The difference between fractional knapsack and 0-1 knapsack problem is that in 0-1 knapsack you can't include the fraction of the objects.

Greedy method can't be applied in 0-1 knapsack problem because if the problem was like the following:

I have two objects with weight 1 and w with their associated profits 2 and w respectively, and, the capacity of the knapsack is w , then the profit/weight ratio would be 2 and 1 respectively. The greedy algorithm will always give priority to the object with profit 2 when clearly object with profit w will produce the maximum profit.

So we need to see if there is a dynamic programming way for this problem. We know that to apply dynamic programming, we need three things:

1. Optimal sub-structure for 0-1 knapsack problem

The idea is that either I am going to include the object or not include the object. So if there are $1, 2, 3, \dots, n$ objects, for each object I have 2 possibilities of inclusion or rejection. So the total possibilities is of $O(2^n)$.

If there are 'i' objects I can include each of them one by one reducing the number of objects to include with each iteration. This can be the optimal sub-structure. The order of complexity = $O(2^n)$ right now. Let's see the recursive equations.

2. Recursive equations for 0-1 knapsack problem

Let,

KS(i, W) = "profit associated with i elements when W is the remaining knapsack capacity".

p_i = profit of i^{th} element

w_i = weight of i^{th} element

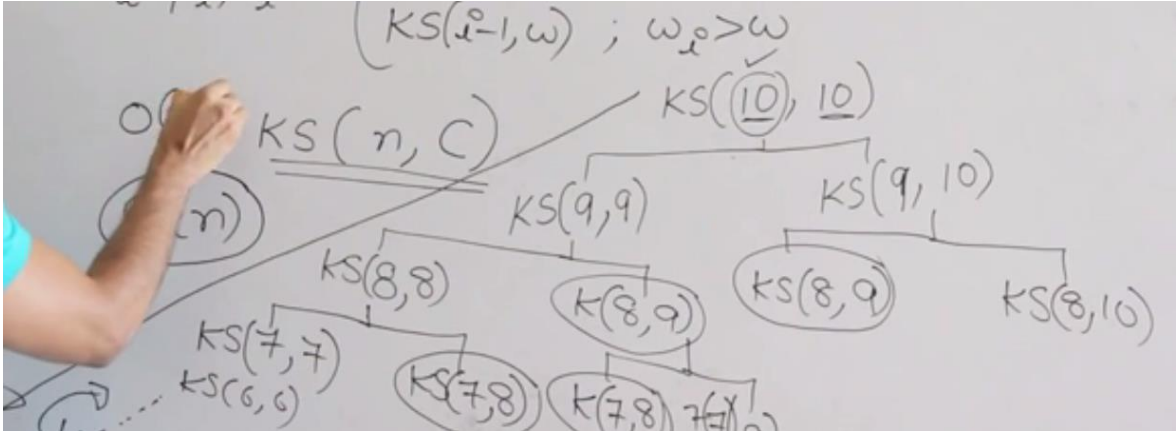
The equations look like the following:

$$ks(i, w) = \begin{cases} \max(p_i + KS(i - 1, W - w_i), KS(i - 1, w)) & \\ 0 ; i = 0 \text{ or } w = 0 & \\ KS(i - 1, W); w_i > W & \end{cases}$$

We are going to call this equation as follows:

K(n, C) = Given n elements and a knapsack of capacity C , the recursion tree will look like below:

Assumption: All objects have a weight of 1. This is the worst case scenario calculation



Depth of recursion tree: $O(n)$

Number of nodes: $O(2^n)$

Space complexity: $O(n)$

Now we know that for dynamic programming to work, we need repeating sub-problems which we are lucky with in this problem. The number of calculations will depend on how many non-repeating sub-problems we have.

In this problem the number of non-repeating sub-problems for $KS(n, W)$ will be $n \times W$. That means if I have **n objects** and the capacity of the knapsack is **W** , then the number of unique sub-problems will be **$n \times W$** .

Time complexity: $O(n \times W)$

Space complexity: $O(n \times W)$

The table size: $(n+1) \times (W+1)$ to accommodate ZEROS!!

BOTTOM-UP DYNAMIC PROGRAMMING FOR 0-1 KNAPSACK PROBLEM

Let's understand with the help of an example:

Question: For the given objects find the maximum profit one can get when the capacity of the knapsack is 6

Objects	1	2	3
Weight	1	2	4
Profit	20	12	28

The recursive equations are as follows:

$$KS(i, w) = \begin{cases} \max(p_i + KS(i-1, W - w_i), KS(i-1, w)) \\ 0 ; i = 0 \text{ or } w = 0 \\ KS(i-1, W); w_i > W \end{cases}$$

DP table: $(n+1) \times (W+1) = (4 \times 7)$

$W = 6$

weight
0 1 2 3 4 5 6

object

0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	12	22	22	22	22
3	0	10	12	22	28	38	40

KS(1, 1) = if the capacity of the knapsack is 1 and I have only first object to include, can I include it or not?

If I include it or not then: $\max(p_i + KS(i-1, W - w_i), KS(i-1, w)) = \max(10 + KS(0, 0), KS(0, 1)) = \max(10, 0) = 10$

KS(1,2) = if the capacity of the knapsack is 2 and I have only first object to include, can I include it or not?

If I include it or not then: $\max(p_i + KS(i-1, W - w_i), KS(i-1, w)) = \max(10 + KS(0, 1), KS(0, 2)) = \max(10, 0) = 10$

KS(1,3) = KS(1,4) = KS(1,5) = KS(1, 6) = 10

KS(2,1) = if the capacity of the knapsack is 1 and I have two objects to include, can I include it or not?

But $w_i > W$, so $KS(i-1, W) = KS(1,1) = 10$ will be the answer

KS(2,2) = if the capacity of the knapsack is 2 and I have two objects to include, can I include it or not?

If I include it or not then: $\max(p_i + KS(i-1, W - w_i), KS(i-1, w)) = \max(12 + KS(1, 0), KS(1, 2)) = \max(12, 10) = 12$

KS(2,3) = if the capacity of the knapsack is 3 and I have two objects to include, can I include it or not?

If I include it or not then: $\max(p_i + KS(i-1, W - w_i), KS(i-1, w)) = \max(12 + KS(1, 1), KS(1, 3)) = \max(22, 10) = 22$

KS(2, 4) = KS(2, 5) = KS(2, 6) = 22

KS(3, 1) = $KS(i-1, W); w_i > W = 4 > 1 = KS(2, 1) = 10$

KS(3, 2) = $KS(i-1, W); w_i > W = 4 > 2 = KS(2, 2) = 10$

KS(3,3) = $KS(i-1, W)$; $w_i > W = 4 > 3 = KS(2, 3) = 22$

KS(3, 4) = if the capacity of the knapsack is 4 and I have three objects to include, can I include it or not?

If I include it or not then: $\max(p_i + KS(i-1, W - w_i), KS(i-1, w)) = \max(28 + KS(2, 0), KS(2, 4)) = \max(28, 22) = 28$

KS(3, 5) = if the capacity of the knapsack is 5 and I have three objects to include, can I include it or not?

If I include it or not then: $\max(p_i + KS(i-1, W - w_i), KS(i-1, w)) = \max(28 + KS(2, 1), KS(2, 5)) = \max(38, 22) = 38$

KS(3, 6) = if the capacity of the knapsack is 4 and I have three objects to include, can I include it or not?

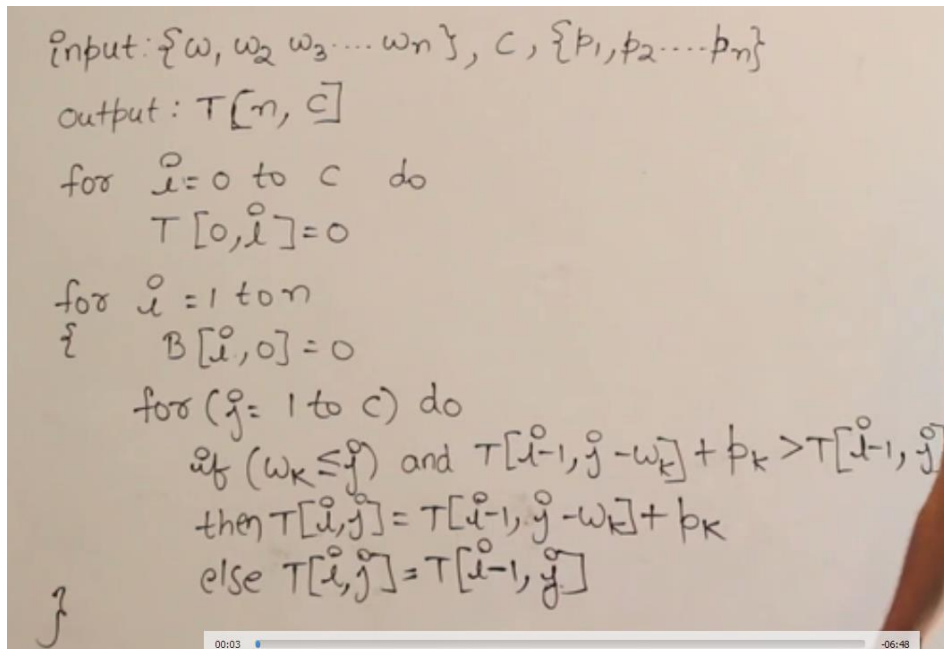
If I include it or not then: $\max(p_i + KS(i-1, W - w_i), KS(i-1, w)) = \max(28 + KS(2, 2), KS(2, 6)) = 40$

TIME COMPLEXITY: $O(n \times W)$ SPACE COMPLEXITY: $O(n \times W)$

If W is $O(2^n)$ then use BRUTE FORCE else use DP.

Therefore Time complexity for 0-1 knapsack problem = $\min(O(2^n), O(n \times W))$

ALGORITHM:



```
input:  $\{w_1, w_2, w_3, \dots, w_n\}, C, \{p_1, p_2, \dots, p_n\}$ 
output:  $T[n, C]$ 
for  $i = 0$  to  $C$  do
     $T[0, i] = 0$ 
for  $i = 1$  to  $n$ 
     $B[i, 0] = 0$ 
    for  $(j = 1$  to  $C)$  do
        if  $(w_k \leq j)$  and  $T[i-1, j-w_k] + p_k > T[i-1, j]$ 
        then  $T[i, j] = T[i-1, j-w_k] + p_k$ 
        else  $T[i, j] = T[i-1, j]$ 
    }
```

Let's move on to the next problem that uses DP.

SUBSET SUM (Dynamic programming)

Given a set $\{a_1, a_2, a_3, \dots, a_n\}$ find a subset of this set such that the sum of the elements of this subset is 'W' (given).

Is the greedy method going to work on it? No!

Then, we have the brute force method which is going to take a time of $O(2^n)$ because the power set of this set is going to have the correct solution if it exists.

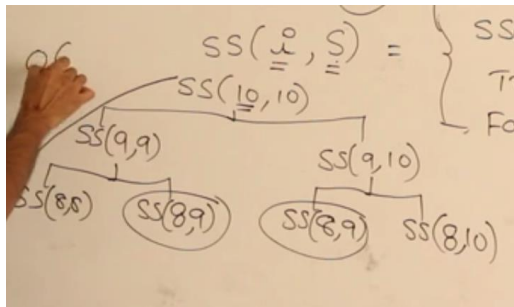
Moreover, this problem is like the 0-1 knapsack because each of the elements have a choice to be in the subset or not to be in the subset.

So, let's apply the dynamic programming.

Recursive equation for subset sum

$$SS(i, S) = \begin{cases} SS(i-1, S); S < a_i \\ SS(i-1, S - a_i) \text{ OR } SS(i-1, S); S > a_i \\ True; S = 0 \\ False; i = 0, S \neq 0 \end{cases}$$

So, if I create the recursion tree (assuming $S = 10$ and $I = 10$) then, in the worst case scenario



The depth of the tree = $O(n)$ and number of nodes = $O(2^n)$. But the problems are repeating.

The unique sub-problems the recursion tree $SS(n, W)$ where n = number of elements and W is the sum of those elements then the number of unique sub-problems = $O(nW)$

Bottom-up dynamic programming for Subset Sum problem

$S = \{6, 3, 2, 1\}$ and $W = 5$

Here I am not worried about the solution of the problem (which subset). Here, I need to know if there is a subset in the given set that can give a sum of 5.

\Rightarrow This is a decision problem

Now $|S| = 4$ and $W = 5$, therefore the table will be 4×5 array

	Sum	0	1	2	3	4	5
Set	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

Sum required = 5

$S = \{6, 3, 2, 1\}$

$$SS(i, S) = \begin{cases} SS(i-1, S); S < a_i \\ SS(i-1, S - a_i) \text{ OR } SS(i-1, S); S \geq a_i \\ True; S = 0 \\ False; i = 0, S \neq 0 \end{cases}$$

$SS(1,1)$ = Taking only first element is the sum of 1 possible? No! because $1 < 5$. Therefore

$$SS(i-1, S); S < a_i: SS(0, 1) = F$$

$SS(1,2) = SS(1,3) = SS(1,4) = SS(1, 5) = F$

$SS(2,1)$ = Taken two elements is the sum of 1 possible? No! because $2 < 5$. Therefore

$$SS(i-1, S); S < a_i: SS(1,1) = F$$

$S(2,2) = F$

$S(2, 3) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(1, 0) \text{ OR } SS(1,3) = T \text{ or } F = T$

$S(2,4) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(1, 1) \text{ OR } SS(1,4) = F \text{ or } F = F$

$S(2,5) = F$

$S(3, 1) = F,$

$S(3, 2) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(2, 0) \text{ OR } SS(2,2) = T \text{ or } F = T$

$S(3,3) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(2, 1) \text{ OR } SS(2,3) = F \text{ or } T = T$

$S(3, 4) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(2, 2) \text{ OR } SS(2,4) = F \text{ or } F = F$

$S(3, 5) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(2, 3) \text{ OR } SS(2,5) = T \text{ or } F = T$

$S(4,1) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(3, 0) \text{ OR } SS(3,1) = T \text{ or } F = T$

$S(4,2) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(3, 1) \text{ OR } SS(3,2) = F \text{ or } T = T$

$S(4,3) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(3, 2) \text{ OR } SS(3,3) = T \text{ or } T = T$

$S(4,4) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(3, 3) \text{ OR } SS(3,4) = T \text{ or } F = T$

$S(4,5) = SS(i-1, S - a_i) \text{ OR } SS(i-1, S) = SS(3, 4) \text{ OR } SS(3,5) = F \text{ or } T = T$

Therefore, the sum of 5 is possible using a subset of the given set. Let's see the time complexity of the algorithm.

Time complexity: n cells and sum of W . For each cell calculation there is a constant time taken. So

Complexity: $O(nW) \times O(1) = O(nW)$

Space complexity: $O(nW)$

IF W is VERY LARGE THEN THE BEST ALGORITHM WILL HAVE TIME COMPLEXITY $O(2^n)$

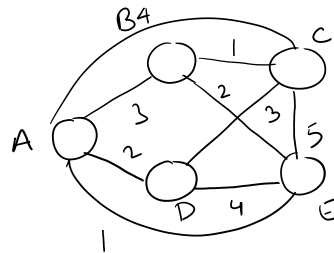
Let us move to the next problem.

TRAVELLING SALESMAN PROBLEM / Min Cost Hamiltonian Cycle(Dynamic Programming)

Let us understand the problem first.

Problem: Given a graph (V, E) where starting and terminal points are same, find out the shortest path from the starting point to itself going through the minimum path obtained by covering all of the vertices at most once.

A sample graph is given below:

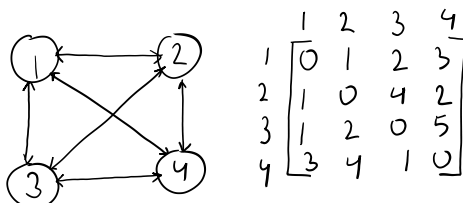


Ways to solve this problem:

1. **Brute Force:** Find out all possible paths, count the weight of each path and find the minimum. The time complexity will be $O((n-1)!)$ (ABCEDA) = $O(n!) > O(2^n)$
2. **Dynamic Programming:** This problem has the highest brute force complexity than any other. Let's see what is the optimal sub-structure, recursive equations and overlapping sub-problems for travelling salesman problem.

Optimal Sub-structure and Recursive Equation for Travelling Salesman problem

Let's understand this with the help of an example:



This graph is a complete graph, having asymmetric distance between the nodes.

Now $T(1, \{2,3,4\})$ = From 1 we want to go to 2, 3, and 4 and return to 1. So I can go from 1 to any of the other and reduce my problem statement to the choices this new vertex will have.

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

Therefore, the main problem has been converted into three sub-problems, hence we have our optimal sub-structures.

The whiteboard contains the following content:

- Graph:** A complete graph with 4 nodes labeled 1, 2, 3, and 4. Edges connect every pair of nodes.
- Cost Matrix:**

	1	2	3	4
1	0	1	2	3
2	1	0	4	2
3	2	4	0	5
4	3	2	5	0
- Recursive Calculations:**
 - $T(1, \{2,3,4\}) = \min \begin{cases} (1,2) + T(2, \{3,4\}) \\ (1,3) + T(3, \{2,4\}) \\ (1,4) + T(4, \{2,3\}) \end{cases}$
 - $T(2, \{3,4\}) = \min \begin{cases} (2,3) + T(3, \{4\}) \\ (2,4) + T(4, \{3\}) \end{cases}$
 - $T(3, \{2,4\}) = \min \begin{cases} (3,2) + T(2, \{4\}) \\ (3,4) + T(4, \{2\}) \end{cases}$
 - $T(4, \{2,3\}) = \min \begin{cases} (4,2) + T(2, \{3\}) \\ (4,3) + T(3, \{2\}) \end{cases}$
 - $T(3, \{4\}) = (3,4) + T(4, \emptyset) = 8$
 - $T(4, \{3\}) = (4,3) + T(3, \emptyset) = 2$
 - $T(2, \{4\}) = (2,4) + T(4, \emptyset) = 5$
 - $T(4, \{2\}) = (4,2) + T(2, \emptyset) = 5$
 - $T(2, \{3\}) = (2,3) + T(3, \emptyset) = 5$
 - $T(3, \{2\}) = (3,2) + T(2, \emptyset) = 3$
 - $T(2, \emptyset) = (2,1)$
 - $T(3, \emptyset) = (3,1)$
 - $T(4, \emptyset) = (4,1)$

Rebuilding the path

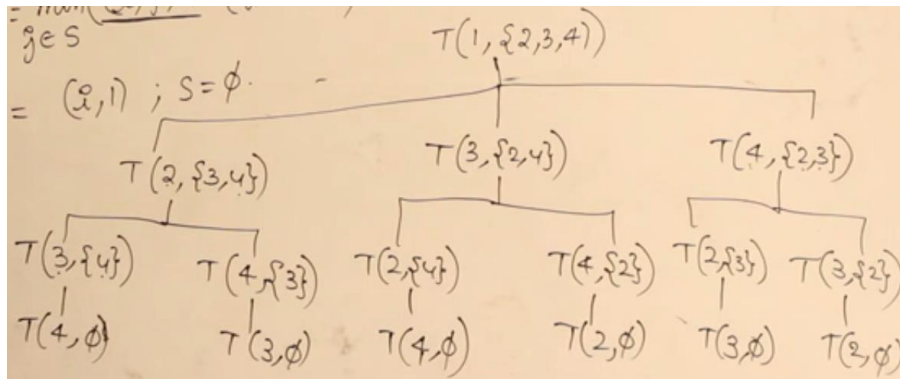
The whiteboard contains the same content as the previous image, but with a red arrow pointing to the graph and the text "Rebuilding the path". A person's hand is visible pointing to the results of the recursive calculations.

Recursive equations for travelling salesman problem

$T(I, S)$ = starting with some node I , cover all the nodes in the set S and return to I covering the least possible weights on the path.

$$T(i, S) = \begin{cases} \min((i, j) + T(j, S - \{j\})); & S \neq \phi \\ (i, 1); & S = \phi \end{cases}$$

Let's see what the recursion tree looks like



The height of the tree = $O(V)$

The number of sub-problems = 15

Number of unique sub-problems = 12

Total number of calculations if dynamic programming is used=

$$(n-1) \sum_{k=0}^{n-2} n - 2C_k = (n-1) 2^{n-2} = O(n2^n) < O(n!)$$

Therefore, even on applying dynamic programming our worst case time complexity is not going to go smaller than $O(n2^n)$

Anyway, let's apply dynamic programming table

Number of distinct sub-problems = $(n-1) 2^{n-2}$

Table size = $(n-1) 2^{n-2}$

For each sub-problem time taken = $O(n)$

Total time taken to solve travelling salesman problem = $O(n) * (n2^n) = O(n^2 2^n)$

Space complexity for travelling salesman problem = $O(n2^n)$

ALL-PAIRS SHORTEST PATH/ FLOYD-WARSHALL ALGORITHM (Dynamic programming)

Here we have to take every node and every other node and find out the shortest path between these two nodes.

Dijkstra's algorithm in all pairs shortest path?

Certainly we can apply this algorithm to calculate shortest path from all vertices to all other vertices. Here's the catch.

1. Time complexity for dijkstra's algorithm : $O(E \log V)$
2. Time complexity for all-pairs shortest path using Dijkstra's: $O(VE \log V)$
3. For dense graph, final time complexity: $O(V \cdot V^2 \log V) = O(V^3 \log V)$

Bellman ford in all pairs shortest path?

Certainly we can apply this algorithm to calculate shortest path from all vertices to all other vertices. Here's the catch.

1. Time complexity for bellman-ford algorithm : $O(EV)$
2. Time complexity for all-pairs shortest path using Bellman Ford: $O(V^2E)$
3. For dense graph, final time complexity: $O(V^2 \cdot V^2) = O(V^4)$

These time complexities are too much. Let's see how dynamic programming can help us reduce the time complexity.

Optimal Sub-structure for all-pairs shortest path algorithm

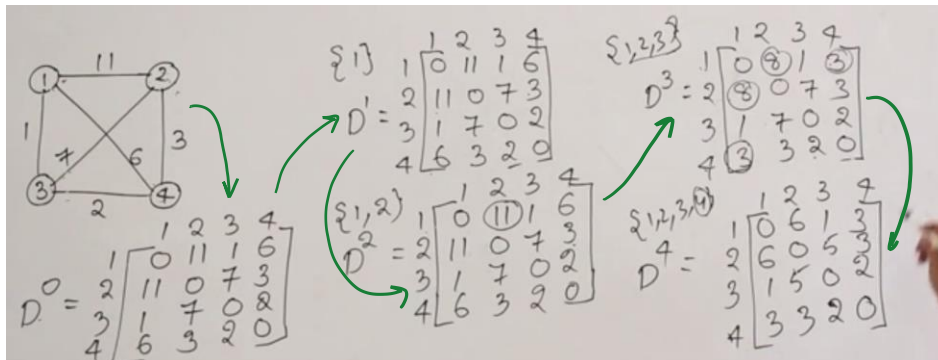
Let $V = \{1, 2, 3, \dots, n\}$ be the vertices of a graph $G(V, E)$ such that $i, j \in V$

Let d_{ij}^k = The shortest path from i to j which will include only the vertices $1 \dots k$

Let's take an example to understand this:

In the exam the graph can be directed and may have negative weight edges but the graph can't have negative weight edge cycles.

Example:



Now let's see how many sub-problems are there.

Sub-problems:

D^0 = given (nothing to solve)

D^1 = n^2 sub-problems

D^2 = n^2 sub-problems

D^3 = n^2 sub-problems

D^4 = n^2 sub-problems

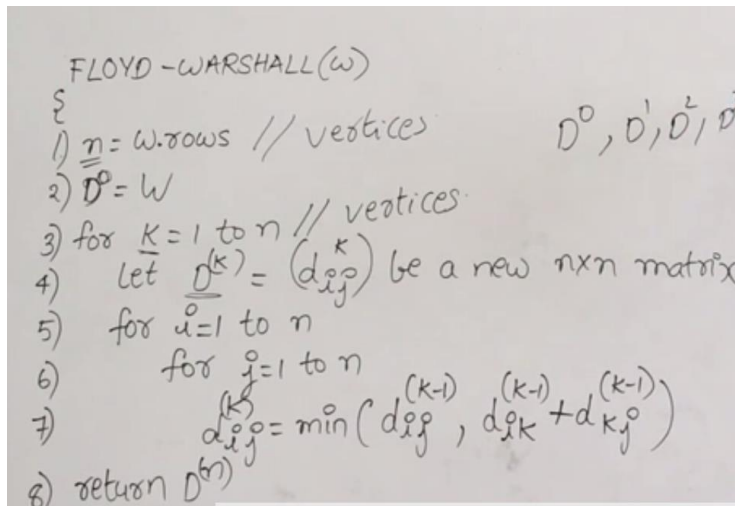
So we have solved n^2 sub-problems n times.

Total number of sub-problems = $nO(n^2) = O(n^3)$

For each sub-problem we are doing constant amount of work, therefore, total time complexity:

$O(n^3) \cdot O(n) = O(n^3)$

Space complexity: At any point in time we are depending on only two matrices. So $O(2n^2) = O(n^2)$

Bottom-up dynamic programming algorithm for all-pairs shortest path algorithm

Let's see the final topic for algorithms:

NP COMPLETENESS

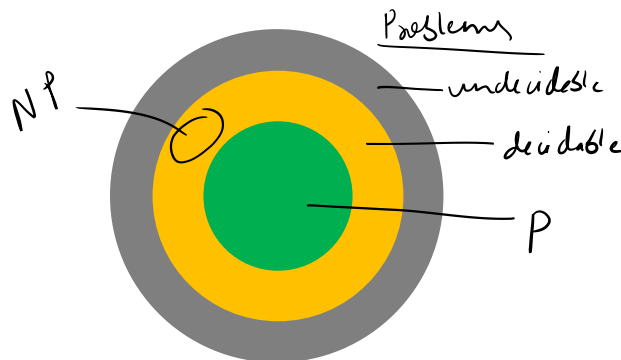
We have seen a lot of algorithms, a question which comes to mind is "Is every problem easy to solve, or some problems are difficult to solve?"

We will have to look at some terminologies for this. Let's start with the types of problems.

Types of Problems

- Problems
 - Undecidable Problems
 - Algorithm doesn't exist
 - Example: Halting problem of Turing machines
 - Decidable Problems
 - Algorithm Exists
 - Types
 - **Tractable Problems:** The easy problems
 - If there exists at least one polynomial bound algorithm i.e., $O(n^k)$
 - It is easy to write them as programs and understand.
 - All the programs that we write are tractable
 - **Intractable Problems:** The hard problems
 - If the problem is not tractable, they are intractable
 - There is no polynomial bound algorithm
 - Example: Traveling salesman problem/0-1 knapsack problem
 - $O(n^{\log n})$ or $O(c^n)$
 - *The polynomial time algorithms do not exist for exact solution but if I don't need exact solution, I can "compromise". If we are not interested in the best answer, then I can modify the algorithm using something called as **heuristics**.*
 - This is quite like approximation of the solution

In order to prove that whether we have to implement the actual algorithm or we have to approximate the solution we need to know if the problem is P(tractable) or NP(intractable).



Let's talk about what are P and NP problems

P and NP Problems

We talk about the decision problems now only.

P = set of all decision problems for which there is an algorithm which solves the problem in polynomial time or $O(n^k)$

NP = the set of all decision problems whose solution can be verified in polynomial time. This means that given the solution, the solution can be verified in polynomial time.

Examples of P and NP

P	NP
Given a graph G, is there an MST whose weight is at most 10?	“
Given a fractional knapsack problem is there any solution where the profit is at least 10?	“
	Travelling Salesman Problem
	0/1 knapsack problem

Now every P class problem has a solution in polynomial time. So, obviously that solution can be verified in the polynomial time. This means for every P-class problem there will always be a verification algorithm that runs in polynomial time.

- ⇒ **Every P class problem falls in the category of NP-class**
- ⇒ **Along with this there will be extra problems in the NP-class which do not belong to P class**

Therefore, P-class is a subset of NP-class and the set (NP-P) is the set of intractable decidable problems.

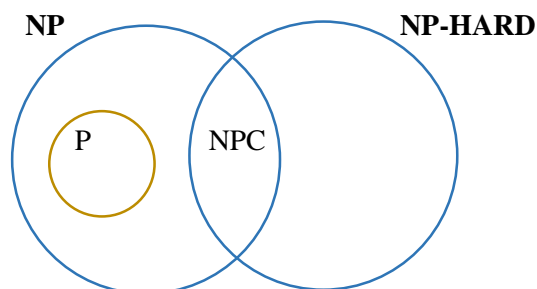
NP hard and NP complete problems

NP-HARD: If every problem in NP can be polynomial time reducible to a problem ‘A’ then A is called NP-hard. This means the problem A is as-hard-as any problem in NP

- ⇒ *If A could be solved in polynomial time, then, every problem in NP is P.*
- ⇒ **$P = NP$**

NP-COMPLETE: A problem is said to be NP-Complete if it is “NP” and NP-HARD

Tentative relationship between P, NP, NP-HARD, NP-COMPLETE (scientific belief)



Notes:

1. IF NP-HARD or NP-COMPLETE is solved in polynomial time then $NP = P$
2. An NP problem or NP-complete problem is proven to be not solvable in polynomial time, then $NP \neq P$
3. Status of NP is unknown
4. If A is an NP-hard problem and in polynomial time it can be converted into a problem B, then B is an NP-HARD problem
5. If A is an NP-hard problem and B is NP problem and if A can be converted to B in polynomial time, then B is NP-COMPLETE

Optimization and decision problems

Most of the problems for which we are interested in proving that they are *hard* or for which there is no polynomial time algorithm is optimization problems.

Every optimization problem can be converted to decision problems. One can work on the decision problems to determine if they are HARD or not thus the optimization problems' hardness can be determined. It is easier to base the theory on decision problems than the optimization problems, that's why we are doing this.

The whiteboard is divided into two columns. The left column contains handwritten notes on optimization problems, and the right column contains handwritten notes on decision problems.

Left Column (Optimization Problems):

- TSP:** A Graph G , shortest path covering all vertices exactly once. A diagram shows a graph with four vertices labeled A, B, C, and D. Edges connect A to B (weight 1), A to C (weight 1), B to D (weight 1), and C to D (weight 2). The path A-B-C-D-A is circled.
- O/KS:** Given c, p, w , find out the maximum profit. Example: $c=10, 10/5, 5/2, 6/6 = \{I_1, I_2\}$.
- LCS:** A, B, find LCS. Example: A = A A B B, B = A B C D. The common subsequence A B is circled.

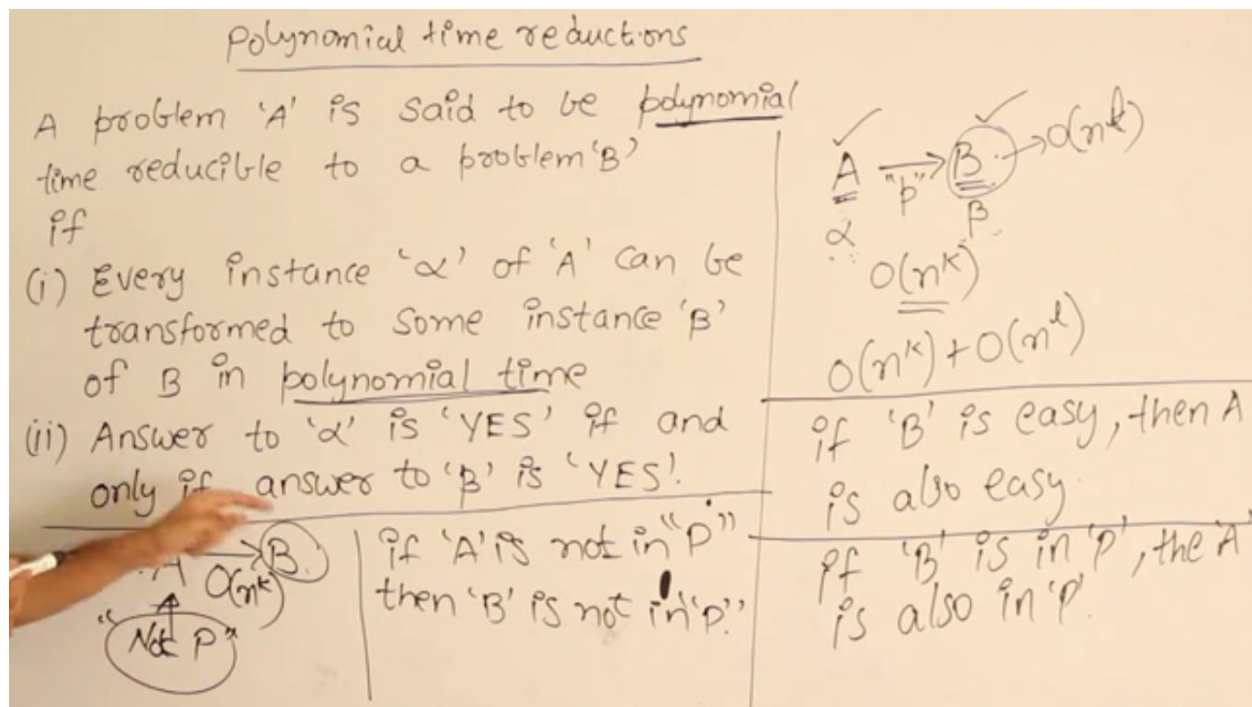
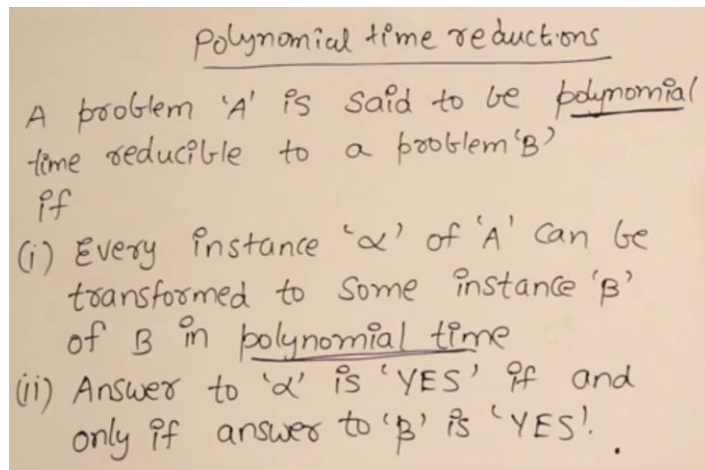
Right Column (Decision Problems):

- decision problem**
- Is there any SP covering all vertices of length at least k ? (Yes/No)
- Is there any solution whose profit is at least k ?
- Is there any SS whose length is at least k ?

We only use decision problems in entire NP completeness theory

1. If the optimization problem is *easy* (polynomial time solution) then decision problem is *easy*
2. If decision problem is *hard* (intractable) then definitely optimization problem is *hard*

Polynomial Time Reduction Algorithms



1. Conversion works for decision problems
2. Conversion should take place in polynomial time
3. If B is easy A is easy
4. If A is hard, B is hard

Verification Algorithms

Any algorithm which verifies whether an answer is right or wrong. The verification in polynomial time is interesting.

NP-Complete Problems

How can we identify the set of NP complete problems?

1. Circuit satisfiability problem
2. Satisfiability problem
3. 3-CNF satisfiability
4. Clique (maximum sub-graph of a graph that is complete)
5. Subset Sum
6. Vertex Cover(how many watchmen should be placed in a corner so that every edge is guarded)
7. Hamiltonian Cycle
8. Traveling Salesman Problem