

graph

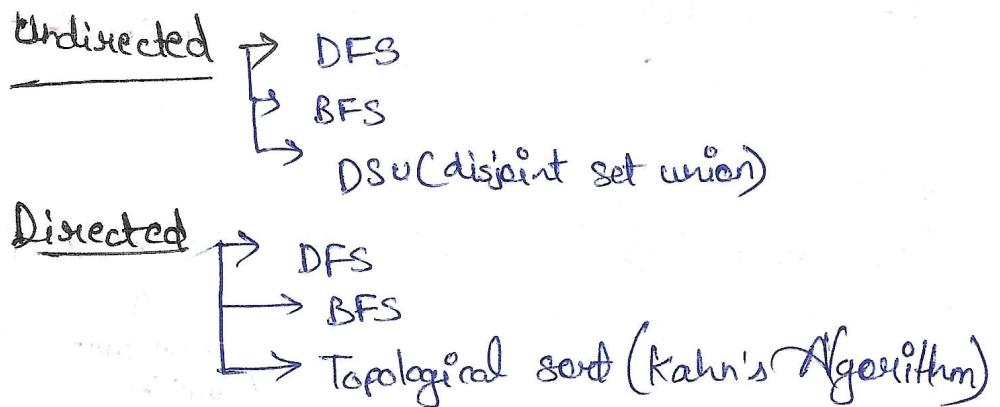
#Connected components

- This is concept when we have disconnected graph so if we have 3 small graphs that make a single graph but they are disconnected.
- So here we at start only we make the boolean array of visited []
- we also make a utility function who checks that all the graphs are travelled or not.

Outer function

```
public static void dfs/bfs (ArrayList<Edge> [] graph) {
    boolean vis [] = new boolean [graph.length];
    for (int i=0; i<graph.length; i++) {
        dfsUtil (graph, i, vis)
```

Cycle in graphs



cycle Detection

Approach → If we find a node that is already visited then it have a cycle in graph.

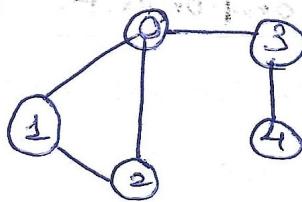
i) Case 1: $\text{vis}[\text{neigh}] \checkmark$
Parent X

ii) $\text{vis}[\text{neigh}] \times$
→ normal DFS

ii) Case 2: $\text{vis}[\text{neigh}] \checkmark$
Parent ✓ → continue

Cycle detection (Undirected.)

$$T.C = O[V+E]$$



Main function

```
public static boolean detectCycle (ArrayList<Edge> graph) {
    boolean vis[] = new boolean [graph.length];
    for (int i=0; i<graph.length; i++) { ← visited array
        if (!vis[i]) { ← we are checking this again and again because
            if (detectCycleUtil (graph, vis, i, -1)) { parts
                return true; //cycle exists in one of the part
            }
        }
    }
}
```

util function.
i is current
and -1 because at
start there is no parent

util function

```
public static boolean detectCycleUtil (ArrayList<Edge> graph[], boolean vis[],
                                      int curr, int par) {
    vis[curr] = true;
```

```
for (int i=0; i<graph[curr].size(); i++) {
```

```
    Edge e = graph[curr].get(i)
```

//case 3

```
if (!vis[e.dest]) { ← (detectCycleUtil (graph, vis, e.dest, curr)) { ↑
    | return true; ↑
    | } next curr will be destination
    } curr will be his parent
```

//case 1

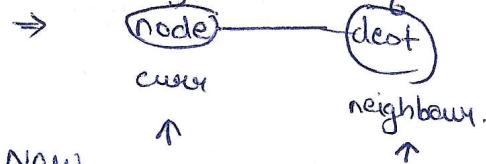
```
else if (vis[e.dest] && e.dest != par) { ← this means
    | return true;
```

there is definitely
another way to access this
node.

//case 2 → do nothing → continue

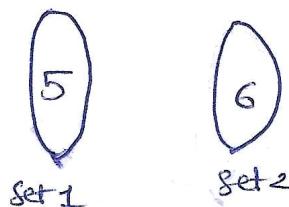
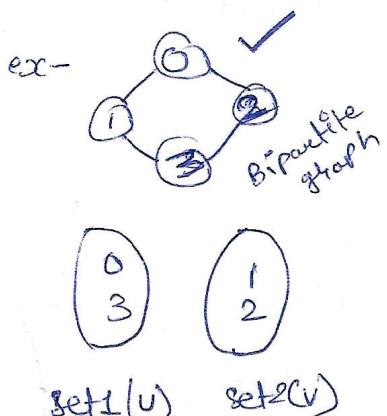
Bipartite → A graph where go through each node and let say the node is curr and its neighbours is destination

- A bipartite graph is whose vertices can be divided into two independent sets.



Now

both should be in different set



using modified BFS

$$T.C = O(V+E)$$

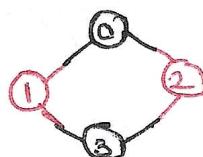
vertices + no. of edges.

- Approach → colouring the graph.

→ if a graph is bipartite then each adjacent node will have different colours.

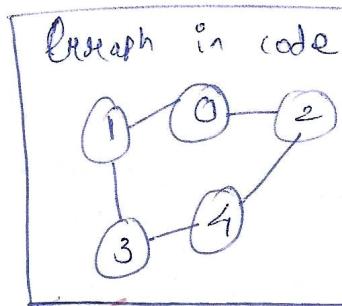
- So we will make array to store colors of node

- 1 → no color
- 0 → yellow/red
- 1 → blue/black



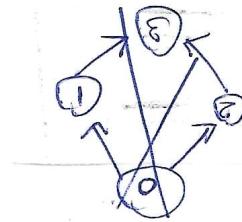
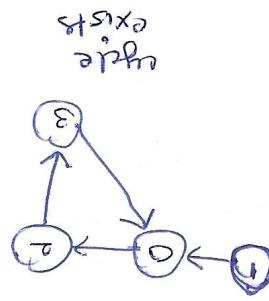
- So we have 3 case

- case 1: $\text{neigh} \rightarrow \text{color} \rightarrow \text{same}$
return false
- case 2: $\text{neigh} \rightarrow \text{color} \rightarrow \text{diff}$
→ continue;
- case 3: $\text{neigh} \rightarrow \text{color} \times$
give opp color than self.



- 1) Acyclic ✓ Bipartite
- 2) Even cycle ✓ Bipartite
- 3) Odd cycle ✗ Bipartite

* If graph don't have cycle then it is Bipartite



as these are not same
hence it is not same
as elements from stack
while returning remove
array to visit first node
again visiting first node.

Graph used in code // code 1

If not visited visit
the node and its
neighbors.

$\text{Stack}[current] = \text{false}$

$\text{if } (\text{!} \text{vis}[\text{neighbor}]) \leftarrow \text{true}$

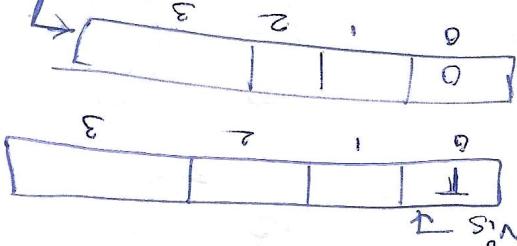
(3) $\text{if } (\text{!} \text{vis}[\text{neighbor}]) \leftarrow \text{true}$

$\text{if } (\text{stack}[\text{neighbor}] = \text{true}) \leftarrow$

(2) for all neighbors

$\text{if } (\text{stack}[\text{neighbor}] = \text{true}) \leftarrow$

call stack



(1) $\text{vis}[current] = \text{true}$

$\text{dfs}(current)$

Final code

① and ③ has no path

where graph is
any

No cycle as we can not go any

(parent - 1)

This direction

Par

(parent - 1)

Directed graph

Cycle detection

vis

(boolean)

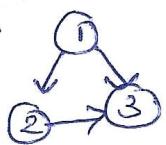
DFS + Stack

APPROACH

Topological Sort → it is linear order of vertices such that every directed edge $u \rightarrow v$ the vertex v comes before v in the order.

DAG → Directed Acyclic Graph

example →



$$\begin{aligned} &\Rightarrow (1 \rightarrow 2) \\ &(2 \rightarrow 3) \\ &(1 \rightarrow 3) \end{aligned}$$

Order
→ 1, 2, 3

* There can be multiple topological order.

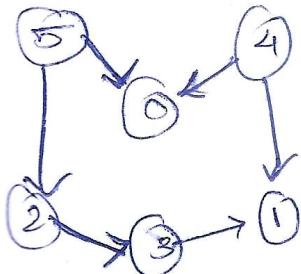
this topological order depend on dependency

① should happen
for ② and ③

ex-
OS
↓
Vs code
↓
Extensions.

Approach

```
→ dfs(graph, curr, vis, stack) {
    vis[curr] = true;
    neigh → call (unvisited)
    stack.add(curr) ] ← just adding the elements
} } as it is called
                                but in recursive manner.
                                so the most dependent node
                                will be at top
                                as it is stack is LIFO do.
```



Output

→ 5 4 2 3 1 0

Util function

```
public topSortUtil (ArrayList<Edge> graph[],  
                    int curr, boolean vis[],  
                    Stack<Integer> s){
```

```
vis[curr] = true;  
for (int i=0; i<graph[curr].size(); i++) {  
    Edge e = graph[curr].get(i);  
    if (!vis[e.dest]) {  
        topSortUtil(graph, e.dest, vis, s);  
        s.push(curr);  
    }  
}
```

Leetcode 101

Q. check the tree is symmetric or not?

mirrored

\Rightarrow main (TreeNode root) {
if ($root == null$) { \leftarrow tree is empty
return true;
}

return check (root.left, root.right);
}

boolean
public \wedge check (Node1, TreeNode Node2) {

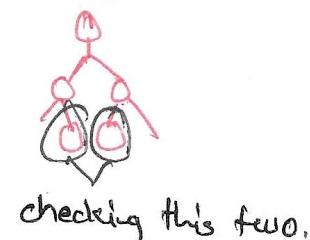
if ($Node1 == Node2 == null$) return true; \leftarrow if both node are null
tree ends evenly

if ($Node1 == null \text{ || } Node2 == null$) return false; \leftarrow tree ends un-evenly

return $Node1.val == Node2.val \wedge (\underline{\text{check} (Node1.left, Node2.right)} \wedge \underline{\text{check} (Node1.right, Node2.left)})$



check (Node1.right, Node2.left);

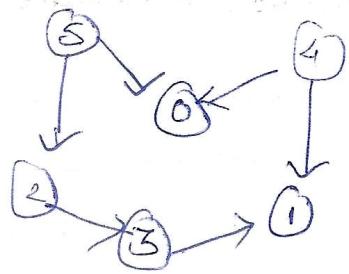


This page is out of topic

Topological sort using (BFS)

• Kahn's Algorithm

	0	1	2	3	4	5
in deg (incoming edges)	2	2	1	1	0	0
out deg (outgoing edge)	0	0	1	1	2	2



Fact: A DAG has at least one vertex with in-degree 0 and one vertex without out-degree 0.



Approach:

→ So, this algorithm is based on dependency.

→ we will print nodes with indegree=0 to outdegree=0.

- ① we will make a array of indegree

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

→ this is done by travelling whole graph and if we find dest of node increase it's in-degree.

- ② Make a queue

- ③ first add the elements in queue who's in-degree = 0

4	1	5	1	
---	---	---	---	--

- ④ Then take the first element from queue

a) reduce it's all neighbour's in-degree by 1

b) add the neighbours to queue who's in-degree has become zero

c) remove the print first element of queue.

- ⑤ repeat step ④ until queue is empty.

Output →

→ 4 5 0 2 3 1

printAllpath(graph, e.dest, dest, Path + src);

}

}

greedy algo.

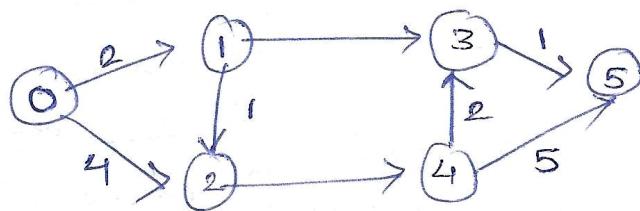
Dijkstra's Algorithm →

shortest path from source to all vertices (weighted graph)

src = 0

$$TC \Rightarrow V + E \log V$$

with PQ without PQ $\rightarrow [V^2]$

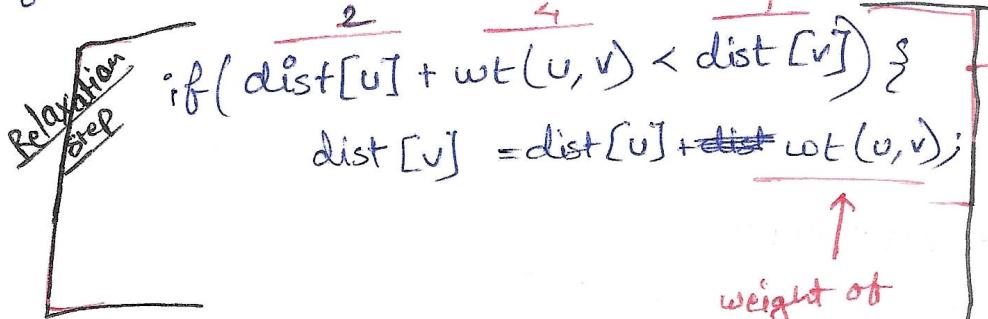
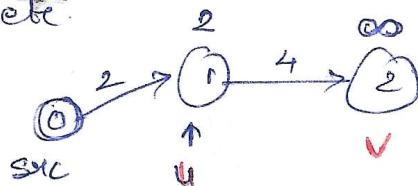


Now for (0) to (2) shortest path is $0 \rightarrow 1 \rightarrow 2$
because sum of weight of vertices in between are $\rightarrow 3$

→ There are many ways to write this algo $dist[u] = \frac{\text{distance from src to } u}{\text{src to } u}$
ex - PQ, hash map, etc.

we will use

P.Q \rightarrow (min heap)
default



Pair (node, dist)

Pseudo code \rightarrow shortest dist node

① Initialize all dist

② Create P.Q \leftarrow using this our heap will be all time sorted on the basis of dist.
 Pair (n, dist)

while (P.Q is not empty) {

curr \rightarrow visit

if ($via[curr] = \text{false}$) {

neighbors
 $\rightarrow dist[u] + wt(u, v) < dist[v]$
 update distance[v].

$$dis[V] = dis[U] + wt(U, V)$$

$f(dis[U]) = \text{Integer MaxValue}$ iff $dis[U] + wt < dis[V]$

to negative.

affter to the no. gets

$$+ \infty + 1 = -1$$

this because in coding

$$wt = e.wt,$$

$$V = e.dest,$$

$$\text{but } U = e.succ;$$

// Relaxation

$$\text{Edge } e = \text{graph}[j].get(k),$$

$\text{for } (i \neq k = 0, k < \text{graph}[j].size(); k++)$

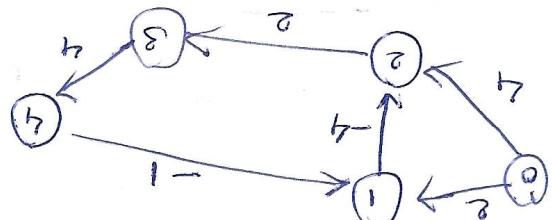
$\text{for } (i \neq j = 0, j < \text{graph.length}; j++)$

$\text{for } (i \neq j = 0; i > v - 1; i++)$

$\text{if } V = \text{graph.length}$

// Edge

work in negative weighted cycle
← This algorithm does not



out nodes at max we can get $V-1$ times because $V-1$ Holes

shortest distance we found out shows all distances.

	0	1	2	3	4	5
0	0	∞	∞	∞	∞	4
1	0	2	4	0	4	-2
2	0	2	-2	0	4	0
3	0	2	-2	0	4	0
4	0	2	-2	0	4	0

$$\text{succ} = 0$$

$$dis[V] = dis[U] + wt(U, V)$$

if $dis[U] + wt(U, V) < dis[V]$

for all edges (U, V)

Applicable.

(Negative Edges).

shortest path from the source to all vertices.

Bellman FordAlg

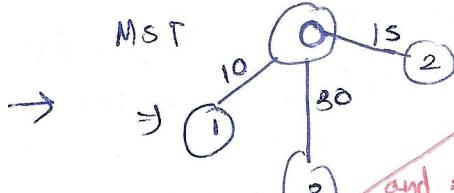
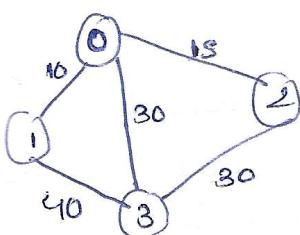
Minimum Spanning Tree (MST)

→ A minimum spanning tree (MST) or minimum weight spanning tree is a subset of all edges of connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

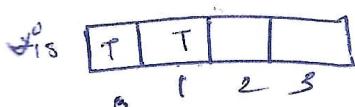
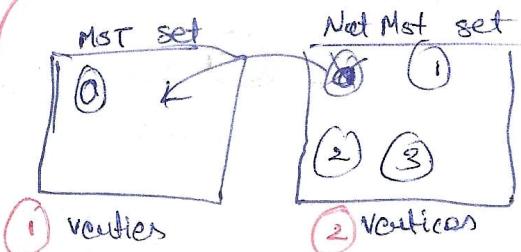
→ A subset graph → (1) In which all the vertices are present
(2) No cycle
(3) least total weight
 (4) If could be that betⁿ two vertices we have not taken the minimum but over all weight is minimum.

Prim's Algorithm

to get MST using MST set
① put the value from set② to set ①
② check every time the shortest dist from all vertices in set① to vertices in set②



③ take shortest dist/wt and move that vertices from set② to set①.



→ vis []
PQ <vertex, cost> → min pair (cost)

while (PQ, isNotEmpty){

curr → (v, cost)

if (not vis[v]) {

visit v
edge | cost ✓
MST(ans ✓) ~~edge~~ neighbors → add PQ ↗ c: size
c: size ↗

~~Q~~ Cheapest flight within K stops

~~tmp~~

→ we have given a array of flights → from, to, price

find the cheapest way to go from src to dest

but only in K stops ← src and dest is not added here.

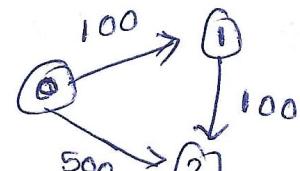
→ if possible return price else -1

→ every values given are positive

we created a graph

flights $[E_0, 1, 100], [1, 2, 100], [0, 2, 500]$

src = 0, dest = 2, K = 1



Approach using

→ Dijkstra's algo with some modification

→ Now we will make PQ < Inf

① vertex, ② path ③ stops
(cost)

We found that

sorting will be done on the basis of this so the options with big stops should not be considered.

① No need of using a PQ as we need our Q sorted on the basis of stops and as we from src to dest we are already sorted on the basis of PQ so use normal Queue.

② don't stop until all path till dest with stops $\leq K$ are processed.

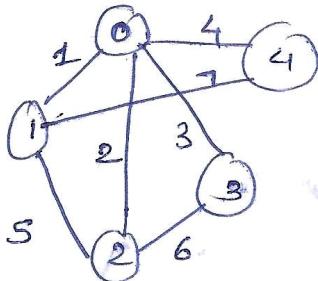
because we can find a more cheaper path than previous.

#Q. Connecting cities with minimum cost

$\Rightarrow \text{cities}[\text{J}][\text{i}] = \begin{cases} \text{if zero no route possible} \\ \text{node} \rightarrow \{0, 1, 2, 3, 4\}, \text{ if not zero route possible} \\ 1 \rightarrow \{1, 0, 5, 0, 7\} \\ 2 \rightarrow \{2, 5, 0, 6, 0\} \\ 3 \rightarrow \{3, 0, 6, 1, 0\} \\ 4 \rightarrow \{4, 7, 0, 0, 0\} \end{cases}$

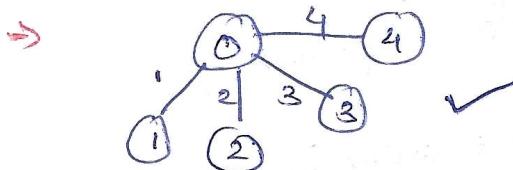
No. of nodes = cities.length

\Rightarrow Map we got.



Now we want routes with min-weight and all node connected.

\rightarrow Do we want MST (minimum Spanning tree)



\therefore We will use (Prim's Algorithm)

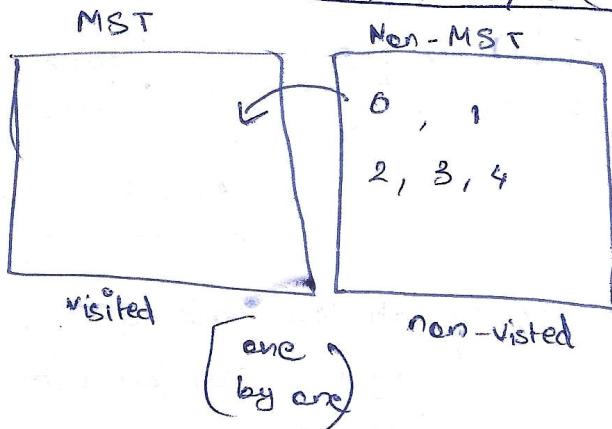
0	1	2	3	4
F	F	F	F	F

visited array

MST

edge	cost
1,0	1
2,0	2
3,0	3
4,0	4

fQ (edges)



- convert the data in graph
- Apply prim's algo.

~~#~~ Disjoint Set DS / Union find DS Constant Time

↳ used to store and track Non-overlapping set.

Two Operations

$$T.C = O(1)$$

- ① Find → To find in which set the element is.
- ② Union → Taking the union of set.

Uses

- 1) Cycle detection (undirected graph)
- 2) Kruskal's Algo (MST)

Implementation (Optimized)

Parent + union by rank

0	1	2	3	4	5	6	7	Parent (leader)
---	---	---	---	---	---	---	---	-----------------

0	1	2	3	4	5	6	7	Rank (height)
0	1	2	3	4	5	6	7	

• Leader joins with leader

Example

1) union(1,3) → Now as both have rank 0

2) find 3 → any one can join with any one →

rank 1 = 1



When we use find then the elements leader is between ∴ leader of 3 is 1

3) union (3,6)

↳ Now the leader of 3 is 1

∴ 6 will directly join with 1



a) Union(1,4)

\hookrightarrow Node 1 is leader

but 4 has leader 2

Node ① is supreme Leader

Pseudo code

find(int n, Par[n], rank[n])

↓
Par[x] = ?

①

find(x)

if (x == par[x]) {

return x; }

return find(par[x])

② union(a,b)

① take out par[A] and
par[B]

② if(rank[A] == rank[B]) {

par[par[A]] = par[B]

rank[par[B]]++;

b) else if (rank A < rank B)

{ par[A] = B }

c) else

{ par[B] = A }

for loop
and union has O(1))

T.C = $O(V + E \log E)$

Kruskal's Algorithm

↳ MST (Greedy algo)

Approach

① Make a ArrayList of <Edge>

② for (i=0 to i=V-1)

③ Now we don't want that node should form cycle
∴ We will use DS.

because we know
in MST there are
(Vertices - 1) edges.

Edge e → a (src)
b (dst)

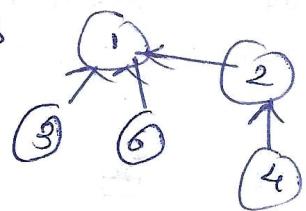
Include ans
Take out
ParA, ParB

if (ParA == ParB)

{ cycle form? }

else → union(ParA, ParB)

As both have same rank
any can join with any one
∴ 2 will join 1



Flood Fill algorithm

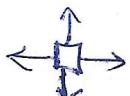
LeetCode 733(Easy)

- ↳ You are given a grid image $[i][j]$ and this (algo) is used to fill the colony.

Ex-  bucket in paint.

$$T.C = O(m \times n)$$

- ① if the cell is of different colour then change the colour.
- ② Move Left, right, up, down
- ③ If got any cell adjacent to this 4 direction ~~change their~~ with different colour change their color.
- ④ If found 'no colour' in that cell then stop.
- ⑤ Repeat step ① and ②



Do this from a $\text{src} \leftarrow$ will be given cell.

example \Rightarrow

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

startRow (sr) = 1

startCol (sc) = 1

$sc = 1 \leftarrow$ previous color

Color = 2.

left ($sr, sc-1$)

right ($sr, sc+1$)

up ($sr-1, sc$)

down ($sr+1, sc$)

Ans \Rightarrow

$$\begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix}$$

 we cannot reach here from any node.

Approach

- ① recursive call is base.

left, right, up, and down

- ② base case

a) stop if sr, sc - invalid (boundary)

b) already visited. $\text{vis}[sr, sc] = \text{true}$

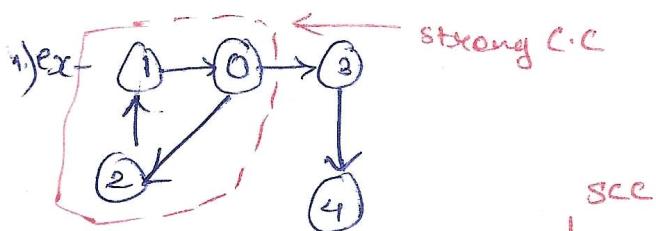
c) $\text{image}[sr, sc] \neq \text{original} \neq 0$ if our cell is 0.

Strongly Connected Component (Directed graph)

↳ It is a component in which we can reach every vertex of the component from every other vertex in that component.

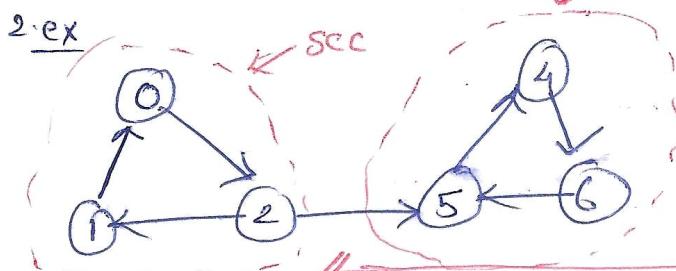
→ It is part of graph which forms cycle and we can go to any vertex from any other vertex.

Find through 'Kosaraju's Algorithm' (DFS) Reverse



Approach

- ① topological sort
- ② get nodes in stack
- ③ transpose the graph
- ④

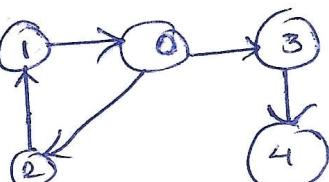


Transpose graph means reverse the direction of each edges.

Pseudo Code

```

1) topSort{
    vis[curr] = T
    for (neigh) {
        if (!vis[neigh]){
            topSort(neigh)
        }
    }
    S.push(curr)
}
  
```



- ② Make DFS on transpose graph according to the stack nodes as sequence.

② Transpose graph.

ArrayList<Edge> transpose[]
(edges will be from dest to src)

Bridge in Graph

↳ bridge is edge in a graph that if removed the graph get divided in two component but if added then 2 components of graph get joined.

Dynamic Programming

→ A technique in Computer Programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.

- ↳ More efficient way of writing code
- ↳ More efficient in time

* Fibonacci

0 1 1 2 3 5 8 13 21 34
→ 0 1 2 3 4 5

$$\rightarrow \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n+2)$$

→ Here normal recursion calls many no. repeated times even if we have the answer of that no.

→ So while writing a DP approach we will make a array that will track and save the answers so that we will not waste time in searching them.

Code fibonacci_DP

```
int fib(int n, int f[5]) {
    if (n == 0 || n == 1) return n;
    if (f[n] != 0) { // means it is calculated
        return f[n];
    }
    f[n] = fib(n-1, f) + fib(n-2, f);
    return f[n];
}

void main (string args[5]) {
    int n=5;
    int f[5] = new int [n+1];
    cout(fib(n, f));
}
```



→ Just making recursion code more time efficient by removing the repeating part.

We used Memoisation above.

• How to identify DP?

- ① when we have asked optimal solution like → least, most, min, max etc.
- ② If there are multiple branches in recursion tree

DP is optimization till $\Theta(n)$.

DP is all about patterns.

Ways of DP

- ① Memoization (Top Down) → using memory to store outputs using stack.
- ② Tabulation (Bottom Up) → using tables to store

Ex- Fibonacci using tabulation

→ we will make a array which will store the values in the index as it's Fibonacci.

Ex-

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

biov

DP approach
using Tabulation

→ $dp[n+1]$

for(int i=2; i<=n; i++) {

$$dp[i] = dp[i-1] + dp[i-2]$$

}

ans = $dp[n]$.

Main concepts

- 1) Fibonacci
- 2) 0-1 knapsack
- 3) Catalan Number
- 4) Unbounded Subsequence (LCS)
- 5) Kadane's
- 6) Catalan
- 7) DP on grid (2D)

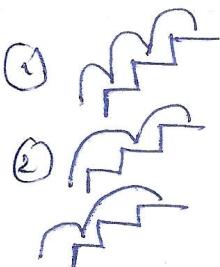
This can help you
to solve 70 DP questions.

Climbing Stairs

$$\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$$

- ↳ person can climb either 1 or 2 stairs at a time
- ↳ Count the no. of ways to reach the top?

→ Ex $n = 3$



ways →

1 → one by one

2 → 2, 1

3 → 2, 1

∴ in $n=3$, there are 3 ways

$n=2$, there are two ways.

$n=1$, there are 1 way

→ Now as in question it is written/given that a person can climb 1 or 2 stairs at a time.

→ So each time when a new stairs is added the ways of climbing stair will be the total ways from last 1 stair plus ways from last two stairs and with recursion we can count that ways so just add.

Pseudocode → ways(n) {

Normal recursion

if($n == 0$) = 1 ← if we are on ground

if($n < 0$) = 0 ← and we want to be on ground then

their is one way

return ways($n-1$) +

ways($n-2$);

if we want to go below then there is no way.

• Using Memoization

→ We just create a array which stores the ~~way~~ answers of previous recursive so we can use them and save time.

Climbing stairs variation

→ You might also be told that can also climb 3 stairs.

→ then the formula will be

$$\begin{aligned} \text{ways}_{(n-1)} + \text{ways}_{(n-2)} + \text{ways}_{(n-3)} &= \text{Total ways} \end{aligned}$$

Climbing stairs (Tabulation DP)

→ 1) Create table.

2) Initialize (base case)

3) Meaning of index

4) Fill (small to large)

Types of Knapsack problems

① Fraction knapsack (greedy)

↳ You are given a sack and some items

- You can fill limited weight items

Fill the sack so that you could get maximum profit.

② 0-1 Knapsack. (DP)

↳ here the items can not be taken in fraction.

③ Unbounded knapsack.

↳ No boundaries for sack unlimited items.

0-1 Knapsack

→ one item can be added one time and we need to fill bag such that we have max profit.

→ val [] = 15, 14, 10, 45, 30 (^{price}/~~weight~~)

wt [] = 2, 5, 1, 8, 4 (weight)

allowed weight = 7

ans = max profit

So with recursion we can do this → Weight of Sack (capacity)

~~#base~~ knapsack(val[], wt[], W, indexⁱ) n = total no. of items

if ($W=0$ || $i=0$) return 0;] base case
if ($wt[i] > W$) return 0;] if sack got full
else if ($wt[i] \leq W$) / valid [] 04 items got over

[→ include $W - wt[i]$, $i-1$] if tot of item is less
[→ exclude W , $i-1$] than or equal
else // not valid to the capacity
[→ exclude W , $i-1$] emptiness of sack
two choices

if item cannot be fixed.

Code

```
ans_recur(int val[], int wt[], int W, int n) {
```

// base case

if ($W=0$ || $n=0$) return 0;

if ($wt[n-1] \leq W$) {

int ans1 = val[n-1] + ans_recur(val, wt, W-wt[n-1], n-1);

int ans2 = ans_recur(val, wt, W, n-1);

return Math.max(ans1, ans2);

}

else {

return ans_recur(val, wt, W, n-1);

}

→ here during recursion program will go on all outcomes with the help of branches and return the max profit value if find on any branch.

0-1 knapsack using ~~tabulation~~ (memoization) T.C = O(n+w)

So here we need to remember two variables

that we need again and changing each time $dp[i][j]$

- 1) Weight (capacity) left w
- 2) $n \rightarrow$ index of item

So As two values we will use 2D Array.

→ And initialize the array with -1

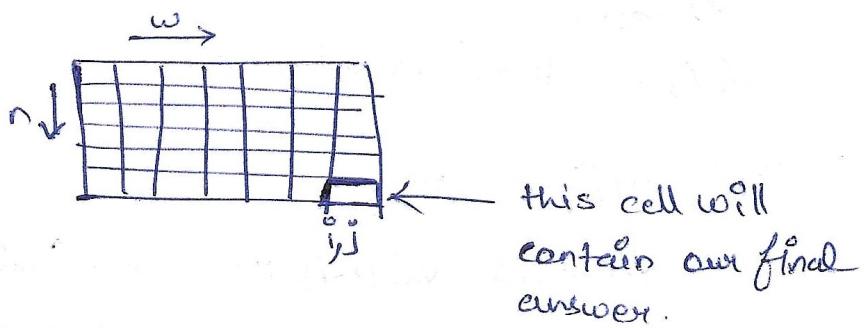
then each time we will check whether it is -1 or not and if not then calculate otherwise use the calculate value.

→ Here it is the (profit)

0	1	2	3	4
0				
1				
2				
3				

$dp[i][j]$

`int dp[n+1][w+1]`



0-1 knapsack using tabulation

- ① Create a table
- ② meaning assign (i, j)
- ③ fill (bottom to up)
small to large.

$DP[i][j]$

i → items + → (Max profit)
j → w (knapsack size)

two fluctuating variable.

w and n

2D array $\rightarrow (n+1) \times (w+1)$

A cell will hold the answer for a small problem ex $\rightarrow n=2, w=3$

like this
so for 3kg sack if (starting) we have 2 item

then it hold [this item] and profit will be

Pseudocode

→ for (int $i=1$ to $n+1$)
 for (int $j=1$ to $w+1$)

(Val, wt) ←

if ($wt \leq j$) // valid

[Now we take maximum of profit if we included
the item or not included the item.]

$$dp[i][j] = \max \left(\begin{array}{l} \text{val}[i-1] + dp[i-1][j-wt], \\ \text{included} \end{array} \right) \quad \left(\begin{array}{l} dp[i-1][j] \\ \text{excluded} \end{array} \right)$$

else { // invalid

// only-excluded

$$dp[i][j] = dp[i-1][j]$$

}

Q. Target sum subset → number [] = 4, 2, 7, 1, 3
TargetSum = 10.

↳ This question is similar to knapsack problem

if there exists a subset whose sum = Target
then return true

→ Here there are 3 such subsets

So the ans is true

$$\textcircled{1} \quad 7, 3 = 10$$

$$\textcircled{2} \quad 2+7, 1 = 10$$

$$\textcircled{3} \quad 4+2+1+3 = 10$$

• Similarities betⁿ knapsack and Target sum

- ① choice of element ← each element has 2 choices to be included or not to be included
- ② limit on max allowed capacity ← before putting in sack there should be enough space.
- ③ imagine val = wt.

Tabulation approach

- ① create table
- ② meaning + initialization
- ③ bottom up manner (small to large)

boolean (T/F)

	0	1	2	3	4	5	6	7	8	9	10
0	T	F	F	F	F	F	F	F	F	F	
1	T										
2	T										
3	T					F					
4	T										
5	T										

zero weight

if you have zero item.

final answer

Weather subset of i items have sum of j or not

So it will store whether are there

3 items whose sum is equal to 5

If yes then True or False.

• Base case / Initialization.

→ if target sum = 0 then it is always true

⇒ loop will go from

→ $\text{for } (\text{int } i=1 \text{ to } n+1)$
 $\text{for } (\text{int } j=1 \text{ to } \text{target}+1)$

→ Code is same as valid / invalid case just like knapsack.

// include ($v = \text{arr}[i-1]$) ↓ function recursion

if ($v \leq j \& (dp[i-1][j-v] = T)$) {
 $dp[i][j] = T$ }

// exclude

if ($dp[i-1][j] = T$) {
 $dp[i][j] = T$ }

Unbounded Knapsack $T.C = O(n \times w)$ → we can use the same item again

$$val[i] = 15, 14, 10, 45, 30$$

$$wt[i] = 2, 5, 1, 3, 4$$

$$w(\text{total allowed weight}) = 7$$

ans = Max weight.

Code

```
for (int i=1; i<n+1; i++) {
    for (int j=1; j<=w+1; j++) {
```

if ($wt[i-1] \leq j$) {choice of yes
 $dp[i][j] = \max(dp[i-1][j],$

$(val[i-1]) + dp[i-1][j-wt[i-1]], dp[i-1][j]);$

$dp[i-1][j]$ else {

$dp[i][j] = dp[i-1][j];$

↑ for invalid condition

① Make a 2D array

② $i = \text{item}$

$j = \text{weight}$

we are including
same item again

if we have

zero capacity

0	0	0	0	0	0
0					
0					
0					

if we have
② Weight capacity
item

Ans =

$$\begin{array}{r} 45 \times 2 \quad 10 \\ 3\text{kg} \quad + \quad 1\text{kg} \end{array} = 90 + 10 = 100$$

Rod cutting → A rod of n inches can be cut and sold piece wise. each piece have a separate price, cut and sell such that you have max profit.

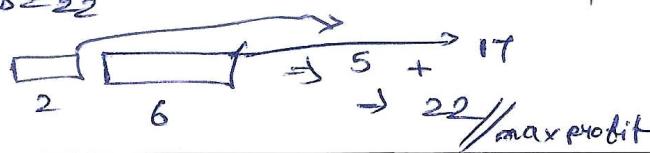
$$\text{length} = 1, 2, 3, 4, 5, 6, 7, 8$$

$$\text{price} = 1, 5, 8, 9, 10, 17, 17, 20$$

$$\text{Total length} = 8$$

Max profit {
 $\text{Price} = val$
 $\text{Length} = wt$
 $\text{Total Rod} = w$

$$\text{Ans} = 22$$



Reimagine this question like knapsack that you have given many small pieces and you need to choose some small pieces which when joined will be equal to the total length, with MAX Profit

Same as unbounded knapsack

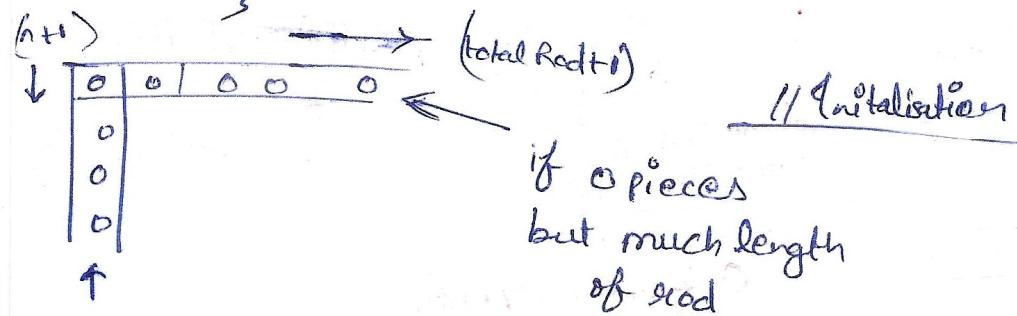
because you can make a piece of same length repeated time.

Code

```

for(int i=1; i<n+1; i++) {
    for(int j=1; j<w+1; j++) {
        if(length[i-1] <= j) { // valid
            dp[i][j] = Math.max(val[i-1] + dp[i-1][j-length[i-1]], dp[i-1][j]);
        } else { // invalid
            dp[i][j] = dp[i-1][j];
        }
    }
}

```



if many pieces
but accepted
Rod length
is 0

Largest Common Subsequence (LCS)

str1 = "abcde", str2 = "ace" ↳ same order And in bet' character can be missing.

subsequence of str1

ans = 3

→ "ace"

Now to write recursion code we should know

- ① base case
- ② big problem & small problem

Ex 2

str1 = "abcedge", str2 = "abedg"

ans = "abdg" = 4.

Recursion logic → we will check ~~from last characters~~ of string
→ (str_1, n) ^{last char} (str_2, m) ^{last char}

① Case

if $n = m$ then call the function
for 1 length less for both.

→

$(\text{str}_1, n-1) (\text{str}_2, m-1)$

② Case

↳ ans + 1

if $n \neq m$ then we have two ~~choice~~

either remove element

from str1

or

str2

ans1 $\left[(\text{str}_1, n-1) \quad (\text{str}_1, n) \right]$ ans2
 $\left[(\text{str}_2, m) \quad (\text{str}_2, m-1) \right]$

return $\text{Math}.\max(\text{ans1}, \text{ans2})$

Base case

if $(\text{str}_1 = 0 \text{ || } \text{str}_2 = 0)$
that is $n = 0 \text{ || } m = 0$,

$\text{lcs} = 0$

return 0;

Pseudocode

$\text{lcs}(\text{str}_1, \text{str}_2, n, m)$ {

~~if~~ $(\text{str}_1 == 0 \text{ || } m == 0)$ { base case
return 0; }

if $(\text{str}_1(n-1) == \text{str}_2(m-1))$ { // same
return $\text{lcs}(\text{str}_1, \text{str}_2, n-1, m-1) + 1$ }

else {

$\text{ans1} = \text{lcs}(\text{str}_1, \text{str}_2, n-1, m);$

$\text{ans2} = \text{lcs}(\text{str}_1, \text{str}_2, n, m-1);$

return $\text{max}(\text{ans1}, \text{ans2});$

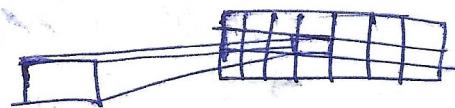
adding 1
common string length
that is ans.

Lcs (Memorization) → here there are fluctuating integers
 ⇒ As we have some overlapping cases so we are using memorization → So we will take a 2D array to reduce T.C

Code

→ just check that whether we already have a solⁿ for that string lengths.

→ so far that initialize the ~~whole~~ whole array with -1.



(i, j) → each cell will have the value of lcs for that length of strings

→ if str1 is of i length & str2 is of j length be in that cell

LCS Tabulation

- ① table
- ② memoizing + initialization
- ③ fill (bottom up)

small problem → large problem

• Tabulation is a safe approach as some times the recursion stack got overflow

		dm+1			
		0	0	0	0
n+1		0	0	0	0
		0	0	0	0
		0	0	0	0

if str1 is empty then lcs = 0

if str2 = 0 then lcs = 0

(i, j) each cell will contain the lcs if str1 is of i length and str2 is of j length.

Pseudo code

```

for(i=1 to n+1)
    for(j=1 to m+1)
        if (str1[i-1] == str2[j-1])
            dp[i][j] = dp[i-1][j-1] + 1
        else
            ans1 = dp[i-1][j];
            ans2 = dp[i][j-1];
            dp[i][j] = Max(ans1, ans2);
    
```

Longest Common Substring T.C = O(mn)

\uparrow contiguous sequence.

Ex → $S_1 = "A B C D E"$,
 $S_2 = "A B C I C E"$
Ans = 2 = AB

Ex → $S_1 = ABCDGH$
 $S_2 = ACDGHIJR$
Ans = 4
 $\Rightarrow "CDGH"$

// Logic

→ A B C D E

A B C I C E

L.C.Sub → 2 1 0 0 1

Max LCS

if character is diff
then the counter reset

base case

if ($str1[i] == 0 \text{ || } str2[j] == 0$)
dcs = 0;

for (int i=1; i<n+1; i++) {

for (int j=1; j<m+1; j++) {

if ($str1.charAt(i-1) == str2.charAt(j-1)$) {

} $dp[i][j] = dp[i-1][j-1] + 1$; \boxed{i} ← if same
else

{ $dp[i][j] = 0$ } ← then the length
is increasing

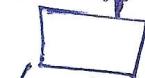
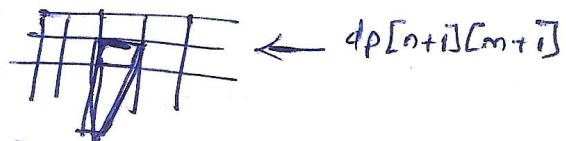
}

As we want a longest common substring,

return ans = Math.max(ans, dp[i][j]);

two fluctuating values

- ① 2D Array
- ② meaning, initialize
- ③ small → large problem problem



(i, j) each cell has the value of LC sub when length of string1 and string2 is i and j .

longest Increasing Subsequence (LIS)

Example

Given $I = \{50, 3, 10, 7, 40, 80\}$

LIS = 3

Ex - 50, 80

3, 50, 40, 80 ✓ } 4 length
3, 7, 40, 80 ✓ }

10, 40, 80

40, 80

To take out
this add all
the element
in Hash set

then sort.

Longest sorted
unique

Ex → 50, 3, 10, 7, 40, 80

3, 7, 10, 40, 50, 80
T T T T T

→ 3, 10, 40, 80

largest
common
subsequence.

T.C = $O(mn)$

↳ ordered maintain

↳ Increasing order

↳ sorted in ascending order

Approach to solve

① Take a sorted array
of unique elements

② Then take a longest
common subsequence

Edit Distance String Conversion / edit distance

Given two strings word1 and word2

Minimum number of operations to convert word1 to word2

3 operations

- 1) Insert a character
- 2) Delete a character
- 3) Replace a character.

Ans.
 \therefore steps

Ex → word1 → "intention" word2 = "execution" 5 steps

- i) intention → intention (remove 't')
- ii) intention → execution (replace 'i' with 'e')
- iii) execution → exention (replace 'n' with 'x')
- iv) exention → exection (replace 'n' with 'c')
- v) exection → exec~~u~~tion (insert 'u')

0 Recursion Approach

→ If we take the both string and check from last character one by one then we have some case

case ① if $(\text{str}(n) == \text{str}(m))$

$$n = \text{str.length}(); \\ m = \text{str}_2.length();$$

then we will check for small problem that is removing 1 character from each string.
and call the function again.

Case 2

if $(\text{str}(n) != \text{str}(m))$ { ← character are different

here the
 $(\text{str}_2.length() - 1)$
because one character satisfied

- a) Add $\text{str}(n), \text{str}_2(m+1) + 1$ because one operation is performed
- b) delete $\text{str}(n-1), \text{str}_2(m) + 1$
- c) replace $\text{str}(n-1), \text{str}_2(m-1) + 1$

↳ As after replacing the character became same and the function will call for smaller strings.

Tabulation Approach

• 2D array

• $[\]$

(i, j) each cell have stored

the value of total operations / or
min operations will be required

if the length of str = i and str2 = j .

if our string is empty then we need to add no. of characters from second string.

$\rightarrow m+1$

0	1	2	3
1			
2			
3			
4			

\downarrow
 $n+1$

and here we need to delete the

str1 characters is our str2 is empty

∴ No. of operations = str1.length.

Initialisation of Pseudocode

```
for (int i=0 to n+1) {
```

```
    for (int j=0 to m+1) {
```

```
        if (i==0) dp[i][j] = j;
```

```
        if (j==0) dp[i][j] = i;
```

}

Pseudo code

```

if (str1(i-1) == str2(j-1)) {
    dp[i][j] = dp[i-1][j-1]
} else {
    dp[i][j] = min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1]) + 1
}

```

removed char from
each string
add
case

Q String Conversion

only 2 operation is allowed.

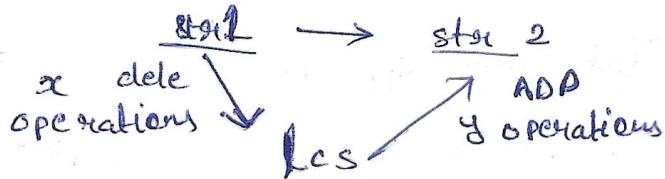
str1 = "pear"

str2 = "sea"

- 1) insertion
- 2) deletion

Take lcs

- steps →
- ① take out lcs of both string
 - ② delete the different characters.
 - ③ Add the remaining characters.



$$\Rightarrow \text{dlt}y = \text{operations required}$$

form

$$dp[i-j] + 1$$

#Q. Wild card matching.

- we have given a normal text and a wild card text
- wild card text contain normal character (alphabet) and special two characters '*' and '?'

We are said to convert str1 to str2 is possible or no.

← this character can be replaced by

- ① any character in string
- ② or multiple characters
- ③ or " " empty space

← this character can be replaced by a single character of string

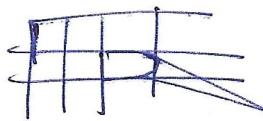
So we need to tell whether the string/text can be matched or not?

⇒ Ex → $s = "aa"$
 $p = "*?"$ this can be changed to "aa"
output true.

Recursion Approach → we are going to solve the smallest problem by decrease the length of string

→ Tabulation approach
2D array.

→ dp [n+1][m+1]



Initialisation

if ($s=0$ and $p=0$) then True

(i, j) each cell will contain

if ($s="2"$ and $p=0$) then false

if the string length = i
if pattern length = j

if ($s="r"$, $p=2$ then

then it is possible T/F

empty. (a) if $p="*"$

to convert or Not

(b) if $p="b"$ False

(c) if $p="?"$ = False

(d) if $p=($ "characters + '*')

then check if after removing

* did our pattern get equals to string?

if yes → True

or

→ False

Pseudo code

```
for( int j= 1 to m ) {  
    if ( p.charAt(j-1) == '*' ) {  
        dp[0,j] = dp[0,j-1] ;  
    }  
}
```

Cell in bottom

"abc" $p="aac"$
① if last character are equal

$dp[i][j] = dp[i-1][j-1]$,

②

if $s="abc"$

then we will check
for the remaining
string.

$p="ab?"$

$dp[i][j] = dp[i-1][j-1]$

As question says
can be converted
we will check for remaining