Graph

Connected components

→ This is concept when we have disconnected graph so if we have 3 small graphs that ~~be~~ 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

Undirected → DFS
→ BFS
→ DSU (disjoint set union)

Directed → DFS
→ BFS
→ Topological sort (Kahn's Algorithm)

---

cycle Detection

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

True

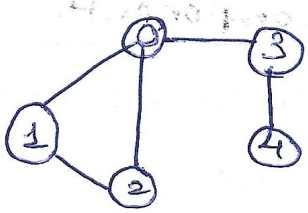
i) Case 1: vis[neigh] ✓ parent ✗

ii) Case 2: vis[neigh] ✓ parent ✓ → continue

iii) vis[neigh] ✗ → normal DFS

Cycle detection (Undirected)

T.C = O[V+E]



Main function

```
Public static boolean detectCycle (ArrayList<Edge> [] graph){
    boolean vis [] = new boolean [graph.length];   ← visited array
    for(int i=0; i<graph.length; i++){
        if (!vis [i]){   ← we are checking this again and again because graph may be different parts
            if (detectCycleUtil (graph, vis, i, -1)){
                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
    if (!vis[e.dest]){   detectCycleUtil(graph, vis, e.dest, curr)){
        return true; }
    }

    //case 1
    else if (vis[e.dest] && e.dest != par){   ← this means there is definately another way to access this node.
        return true;
    }

    //case 2 → do nothing → continue

}
```

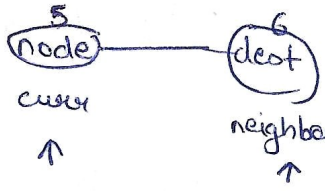
next curr will be destination

curr will be his parent

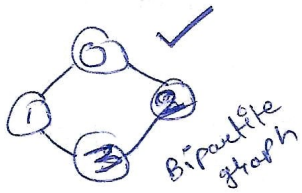
Bipartite → A graph where go through each node and let say the node is curr and its neighbour is destination

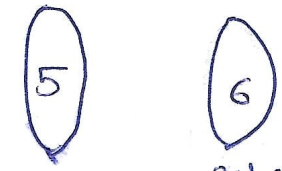
2 parts

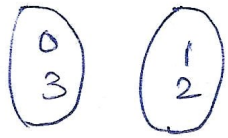
- A bipartite graph is whose vertices can be divided into two independent sets.

→ 

Now both should be in different set

ex- 



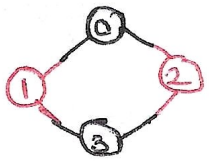


set1(u) set2(v)

using modified BFS T.C = O(V+E) vertices + no. of Edges. no. of

Approach → colouring the graph.

⇒ if a graph is bipartite then each adjacent node will have different colour.



1) So we will make array to store colors of node

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

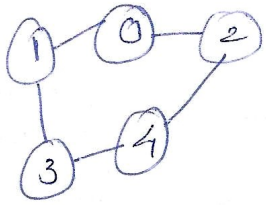
2) So we have 3 case

case 1: neigh → color → same return false

case 2: neigh → color → diff ✗ continue;

case 3; neigh → color ✗ give opp color than self.

Graph in code

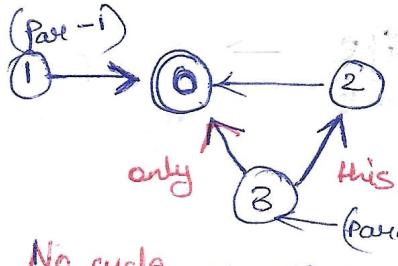


1) Acyclic ✓ Bipartite
2) Even cycle ✓ Bipartite
3) Odd cycle ✗ Bipartite

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

Cycle detection

(Directed graph)



↳ No cycle as we can not go any where from 3

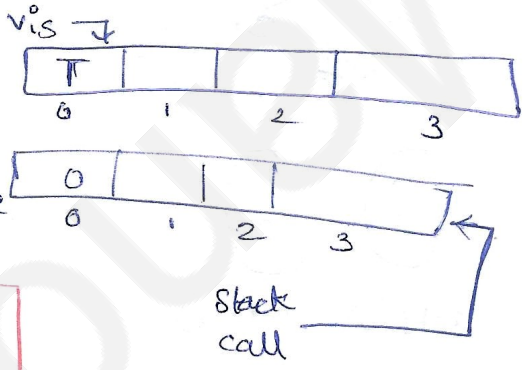
Approach

DFS + stack (boolean vis)

↳ 1 and 3 has no parent

Pseudo code

```
dfs(curr) {
① vis[curr] = true
② for (all neigh)
    if (stack[neigh] = true)
        → cycle exists True
③   if (!vis[neigh])
        isCycle(neigh) → True
    Stack[curr] = false;
}
```



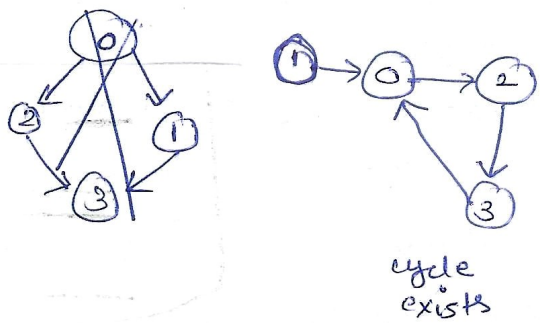
if the stack already contains the neighbour

if not visited visit the node and its neighbour.

while returning remove the elements from stack as there is not same way to visit that node again using that route.

• Graph used in code // code 1

→



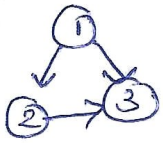
cycle exists

→ only for DAGs

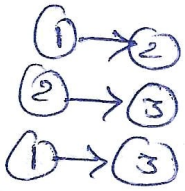
Topological Sort → It is linear order of vertices such that every directed edge u → v the vertex u comes before v in the order.

DAG → Directed Acyclic Graph

example →



⇒



Order → 1, 2, 3

⁂ There can be multiple topological order.

this topological order depend on dependancy ① should happen for ② and ③

ex- OS → Vs code → extensions.

Approach

→ dfs(graph, util, vis, stack) {

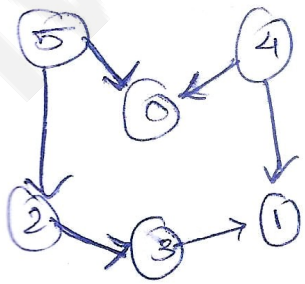
vis[curr] = true;

neigh → call (unvisited) → normal bfs

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 ~~this~~ stack is Lifo so.



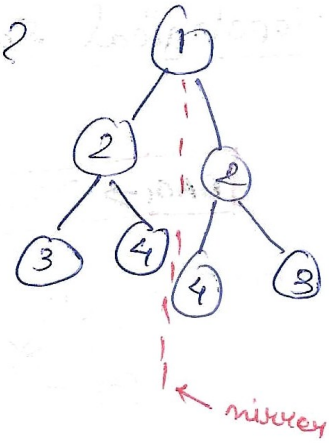
Output → 5 4 2 3 1 0

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. chelek the tree is symmetric (mirror) is not?



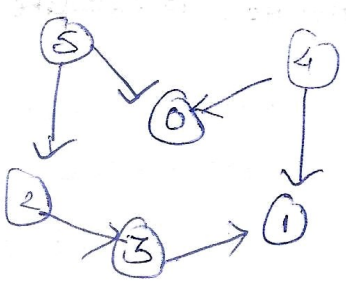

```
=> main (TreeNode root) {
    if (root = null){          ← tree is empty
        return true;
    }
    return check(root.left, root.right);
}

boolean
Public ^ check (Node 1, Tree Node Node 2){
    if (Node1 == Node 2 == null) return true;   ← if both node are null tree ends evenly
    if (Node1 == null || Node 2 = null) return false;   ← tree ends un evenly
Return Node1.val == Node 2. val && (check (Node 1. left, Node 2. right) && check (Node 1. right, Node 2. left);
```


checking this two


checking this two.

This page is out of topic

Topological sort using (BFS)



- Kahn's Algorithm

	0	1	2	3	4	5
in deg (in coming edges)	2	2	1	1	0	0
out deg (out going edge)	0	0	1	1	2	2

Fact: A DAG has at least one vertex with in-degree 0 and one vertex with 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.

→ 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	5		

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

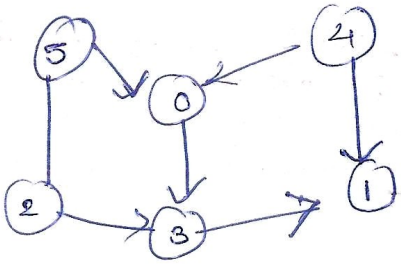
function to calculate indeg

```
Public static void calcIndeg(ArrayList<Edge> graph[], int indeg[]){
    for(int i=0; i<graph.length; i++){
        int v = i;
        for(int j=0; j<graph[v].size(); j++){
            Edge e = graph[v].get(j);
            indeg[e.dest]++;
        }
    }
}
```

→ two for loops and when we find the dest → Increase its indeg

v → vertex

#11 paths from Source to Target | T.C = O(V^V) worst case



s = 5, d = 1

Answer
1) 5-2-3-1
2) 5-0-3-1

→ here we use recursion to travel from one node ~~the~~ ~~through~~ to other through all paths.

→ dFS

Approach:

```
dfs(graph, src, dest, path){
    if(src == dest)
        print(path);
    for(neigh)
        dfs(graph, e.dest, dest, src)
}
```

this is our new source that is our neighbour

Code:

```
void printPath(ArrayList<Edge> graph[], int src, int dest, string Path){
    if(src == dest){
        syso(Path + dest);
        return
    }
    for(int i=0; i<graph[src].size(); i++){
        Edge e = graph[src].get(i);
```

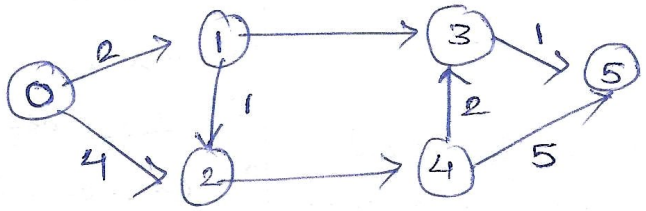
```
        printAllpath(graph, e.dest, dest, path + src);
    }
}
```

Dijkstra's Algorithm → shortest path from source to all vertices (weighted graph)

greedy algo.

src = 0

TC ⇒ V + E log V (with PQ) | without PQ → [V²]



Now for 0 to 2 shortest path is 0 → 1 → 2 because sum of weight of vertices in between are → 3

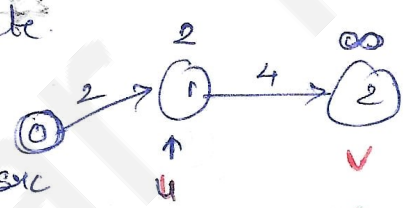
→ There are many ways to write this algo

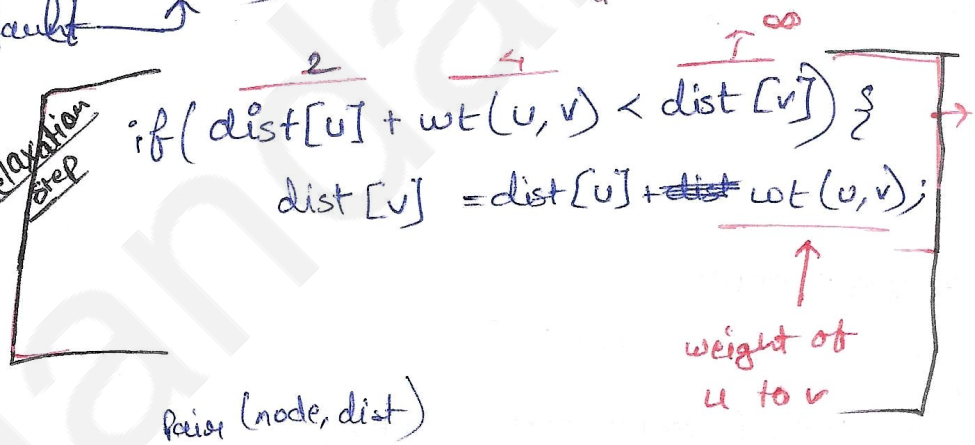
ex- PQ, hash map, etc

we will use

→ P.Q → (min heap) default ↑

dist[u] = distance from src to u





Pair (node, dist)
↳ shortest dist node

Pseudo code

1. initialize all dist
2. create P.Q ← using this our heap will be all time sorted on the basis of dist.
   <pair> (n, dist)

```
while (P.Q is not empty){
    curr → visit
    if (vis[curr] = false){
        neighbours
            ↳ dist[u] + wt(u,v) < dist[v]
              update distance[v].
    }
}
```

edge, dest

Manan Tuc

Bellman Ford Algo

→ shortest path from the source to all vertices. (Negative Edges)

→ Approach.

```
for all edges (u, v)
    if dist[u] + wt(u,v) < dist[v] {
        dist[v] = dist[u] + wt(u,v)
```

→ as there are (-ve) edges we need to perform this step V-1 times

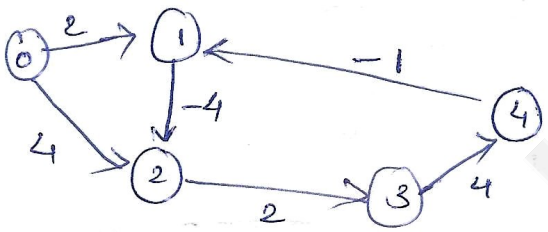
↓

Src = 0

V-1 times because at max we can get V-1 nodes at our longest path.

	0	1	2	3	4
initial	0	∞	∞	∞	∞
1	0	2	4	0	4
2	0	2	-2	0	4 ✓
3	0	2	-2	0	4
4	0	2	-2	0	4

we found out shortest distance for all vertices.



→ This algorith does not work in negative weighted cycle.

→

```
//algo
    int V = graph.length
for (int i=0 ; i<V-1 ; i++) {
    for (int j=0; j< graph.length; j++) {
        for( int k=0; k< graph[j].size(); k++){
            Edge e = graph[j].get(k);
            int u = e.src;
                v = e.dest;
                wt = e.wt;
//relaxation.
if (dist[u] != IntegerMaxValue && dist[u] + wt < dist[v]) {
    dist[v] = dist[u] + wt;
}
}
}
```

this because in coding

+∞ + 1 = -1

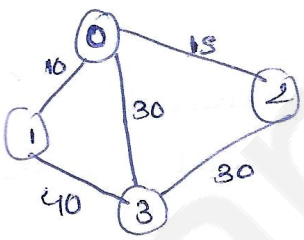
after +∞ the no. gets to negative.

Minimum Spanning Tree (MST)

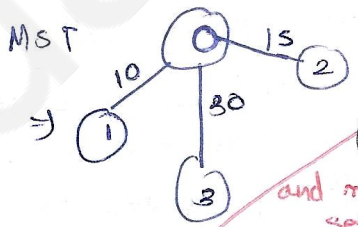
↳ A minimum spanning tree (MST) or minimum weight spanning tree is a subset of all edges of connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

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

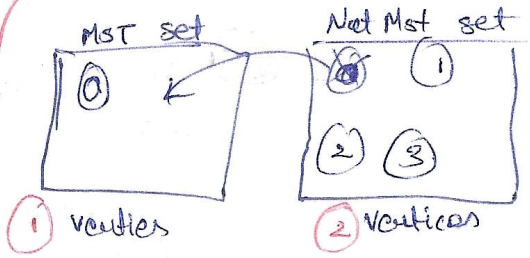
Prim's Algorithm → to get MST using MST set



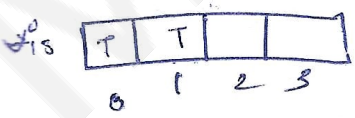
→



(1) Put the value from set (2) to set (1)
(2) Check every time the shortest dist from all vertices in set (1) to vertices in set (2)
(3) take shortest dist/wt and move that vertices from set (2) to set (1).



Approach



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

```
while (PQ.isNotempty) {
    curr → (v, cost)
    if (not vis[]) {
        visit ✓
        MST (ans ✓) edge | cost ✓
        neighbours → add PQ ↳ edge | cost
                                c.dest
                                c.src
    }
}
```

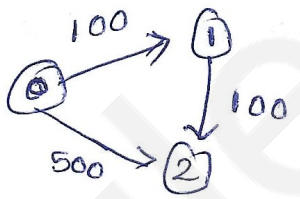
Cheapest flight within K stops

Imp

→ 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 [[0,1,100], [1,2,100], [0,2,500]]

src = 0 , dst = 2 , k = 1

Approach using

→ Dijkstra's algo with some modification

→ Now we will make PQ < Info >

① vertex, ② Path (cost) ③ stops ← sorting will be done on the basis of this so the options with big stops should not be considered.

We found that

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 <= k are processed.
because we can find a more cheaper path than previous.

#Q. Connecting cities with minimum cost

⇒ cities [][] = {

if zero no route possible

node 0 → {0, 1, 2, 3, 4}, ← if not zero route possible with given weight.

1 → {1, 0, 5, 0, 7}

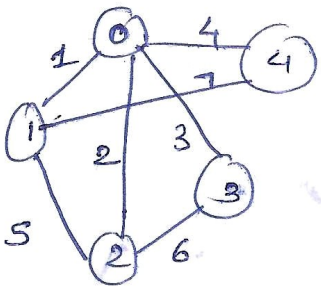
2 → {2, 5, 0, 6, 0}

3 → {3, 0, 6, 0, 0}

4 → {4, 7, 0, 0, 0}}

No. of nodes = cities.length

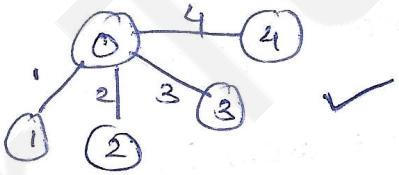
⇒ Map we got.



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

→ So we want MST (minimum Spanning tree)

→



∴ We will use (Prim's Algorithm)

0	1	2	3	4
F	F	F	F	F

visited array

PQ (edges)

edge cost

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

MST

Non-MST

0, 1

2, 3, 4

visited

non-visited

(one by one)

① convert the data in graph

② Apply prims 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) Kruskals Algo (MST)

- Implementation (Optimized)

Parent + union by rank

0	1	2	3	4	5	6	7

Par (leader)

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

Rank (height)

- Leader joins with leader

Example

1) union(1,3) → Now as both have rank 0
   ∴ any one can join with any one →
   rank 1 = 1

2) find 3 → When we use find then the elements leader is returned ∴ leader of 3 is 1

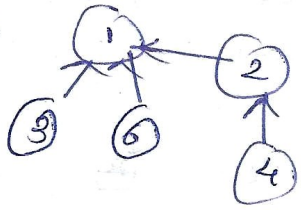
3) union (3,6)
   ↳ Now the leader of 3 is 1 ∴ 6 will directly join with 1 →

4) Union (1, 4)

↳ Now 1 is leader but 4 has leader 2

Now (1) is supreme leader

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



Pseudo code

```
find( int n, Par[n], rank[n]
        ↓
    Par[i] = i
```

(1)

```
find(x)
    if (x == par[x]) {
        return x; }
return find[par[x]]
```

← recursive call for supreme parent.

(2) union (a, b)

① take out par[a] and par[b]

② if (rank are ==) {

a) Par[ParA] = Par[B]

Rank[parB]++;

b) else if (rank A < rank B)

{ Par[A] = B }

c) else

{ Par[B] = A }

for loop and union has O(1) → sorting

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

Kruskal's Algorithm

↳ MST (greedy algo)

① sort edges

② take min cost edge

↳ X form cycle

Include ans

Approach

① Make a ArrayList of <Edge>

② for (i=0 to i = V-1) → because we know in MST there are (Vertices - 1) edges.

③ Now we don't want that node should form cycle ∴ we will use DS. ④ →

take out ParA, ParB

if (ParA == ParB) { cycle form }

else → union(ParA, ParB)

Edge e → a (src), b (dst)

Flood Fill algorithm

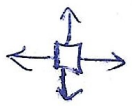
LeetCode 733 (Easy)

T.C = O(m×n)

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

ex-  bucked in paint.

→ 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 src cell. ← will be given

example :- $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$

start row (sr) = 1

~~start = 1~~ start col (sc) = 1

~~sc = 1~~ ← previous color

color = 2.

left (sr, sc-1)

right (sr, sc+1)

up (sr-1, sc)

down (sr+1, sc)

Ans ⇒ $\begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix}$ ← we cannot reach here from any node.

Approach

① recursive call's for.

left, right, up, and down

② Base case

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

2) Already visited. vis[sr, sc] == true

3) image[sr, sc] != original color ~~if 0~~ if our cell is 0.

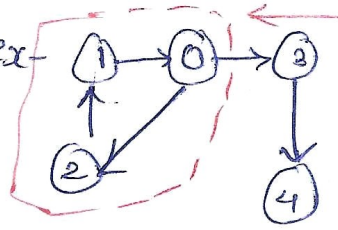
Strongly Connected Component (Directed graph)

→ 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' (Reverse DFS)

1) ex

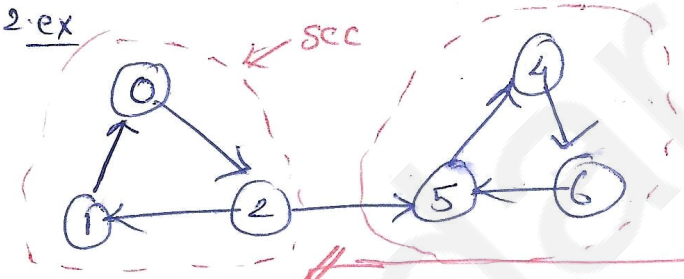


strong C.C

Approach

1. topological sort
2. get nodes in stack
3. Transpose the graph
4.

2. ex

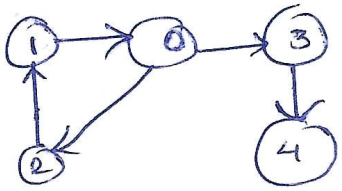


Transpose graph means reverse the direction of each edges.

Pseudo Code

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

3. 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.

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

Dynamic Programming

↳ 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

→ fib(n) = fib(n-1) + 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[]){
    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[]){
    int n=5;
    int f[] = new int[n+1];
    sysO(fib(n, f));
  }
```

DP → 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 O(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

DP approach using Tabulation

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

ex-

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

→ dp[n+1]

```
for(int i=2; i<=n; i++){
    dp[i] = dp[i-1] + dp[i-2]
}
ans = dp[n].
```

Main concepts

1) Fibonnaci
2) 0-1 knapsack
3) Catalan number
4) Unbounded subsequence (LCS)
5) Kadane's
6) Catalane
7) DP on grid (2D)

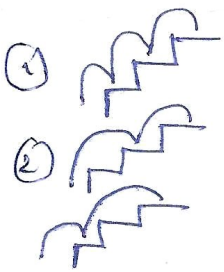
} This can help you to solve 70 DP Questions.

Climbing Stairs

$ways(n) = ways(n-1) + ways(n-2)$

↳ 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 $n=3$, their are 3 way
$n=2$, their are two ways.
$n=1$, their are 1 way

⇒ Now as in question it is written/given that a person can climbs 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 ~~step~~ Stair plus from and last two stairs

and with recursion we can count that ways

So just add

Pseudocode → ways(n) {

Normal recursion

```
if(n==0) = 1
if(n<0) = 0
return ways(n-1) +
       ways(n-2);
}
```

if we are on ground and we want to be on ground then their is one way.

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

- Using Memoization

⇒ we just create a array which stores the ~~way~~ answers of previous recursion 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 $\underbrace{(n-1)}_{\text{ways}} + \underbrace{(n-2)}_{\text{ways}} + \underbrace{(n-3)}_{\text{ways}} = \text{Total ways}$

• Climbing stairs (Tabulation DP)

⇒ 1) create table
+
initialize (base case)

2) Meaning of index

3) fill (small to large)

Types of Knapsack problems

① Fraction knapsack (greedy)
↳ You are given a sack an 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 boundation 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, 3, 4 (weight)

allowed weight = 7

ans = max profit

So with recursion we can do this →

```
// base   knapsack(val[], wt[], W, indx)
```

W → Weight of sack (capacity)

n = total no. of items

```
if (W==0 || i==0) return 0;
```

base case: if sack got full OR items got over

```
if (wt<=W) //valid
```

if wt of item is less than or equal to the capacity emptyness of sack

- include W-wt, i-1
- exclude W, i-1

two choices

```
else // not valid
```

- exclude W, i-1 ← 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] <= 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);
    }
```

T.C = (2^n)

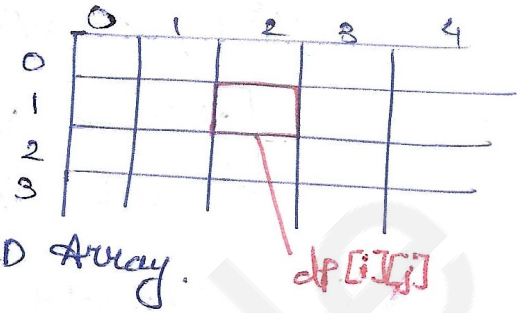
very bad T.C

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

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

So here we need to remember two variables that we need again and changing each time dp[][]

1) Weight (capacity) left W
2) n → index of item

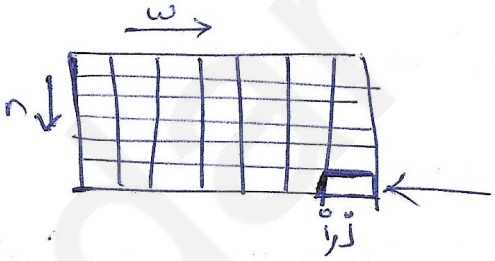


So As two values we will use 2D Array.

→ And initialize the array with -1 then each time we will check weather it is -1 or not and if not then calculate otherwise use the calculate value.

→ Here it is the (profit)

int dp[n+1][W+1]



this cell will contain our final answer.

0-1 Knapsack using tabulisation

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 → (n+1) x (W+1)

A cell will hold the answer for a small problem ex → n = 2, W = 3

like this

So for 3kg sack if we have 2 item (starting) then it hold [this item] and profit will be

Pseudocode

val val[i-1]
wt wt[i-1]

→ for(int i=1 to n+1)
  for(int j=1 to w+1)
    (val, wt)
    if(wt <= j) // valid

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

dp[i][j] = max(val[i-1] + dp[i-1][j-wt], dp[i-1][j])
  } (included, excluded)

else { // invalide
  // 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 their exists a subset whose sum = Target
then return true

→ Here there are 3 such subsets
  ① 7, 3 = 10
  ② 2+7+1 = 10
  ③ 4+2+1+3 = 10

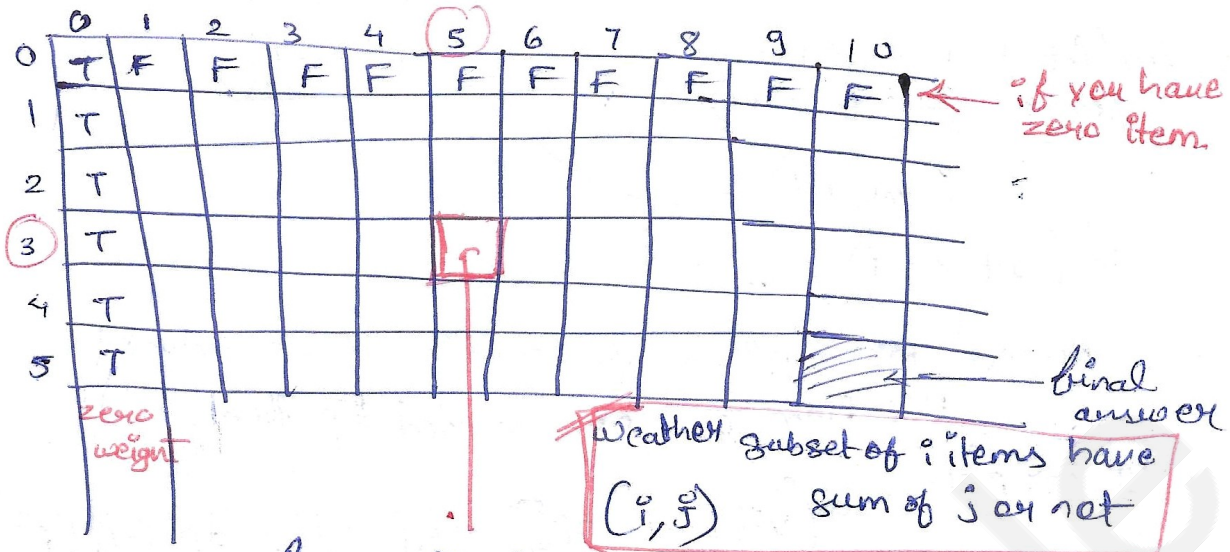
So the ans is true

- **Similarities bet^n knapsack and Target sum**
  1. choice of element ← each element has 2 choice to be in sack or not to be
  2. limit on max allowed capacity ← before putting in sack their 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)



so it will store weather 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

→ for(int i=1 to n+1)
  for(int j=1 to target sum + 1)

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

```
// include (v = arr[i-1])          ← function recursion
if (v <= 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 * W)

→ we can use the same item again

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

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

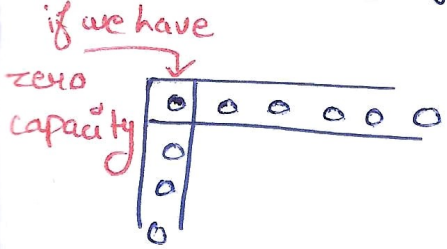
W (total allowed weight) = 7

ans = Max weight.

Approach

① Make a 2D array

② i = item
j = weight



Code

```
for(int i=1; i<n+1; i++){
 for(int j=1; j<W+1; j++){
if(wt[i-1]<=j){ //choice of yes
  dp[i][j] = Math.max(
(val[i-1) + dp[i][j-wt[i+1]], dp[i-1][j]);
}
[i+1][j+1] else {
  dp[i][j] = dp[i-1][j];
}
}
```

we are including same item again

if we have 0 weight capacity item

↑ for invalid condition

Ans → 45 x2 + 10 = 90 + 10 = 100
3Kg + 1Kg

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

length = 1, 2, 3, 4, 5, 6, 7, 8

price = 1, 5, 8, 9, 10, 17, 17, 20

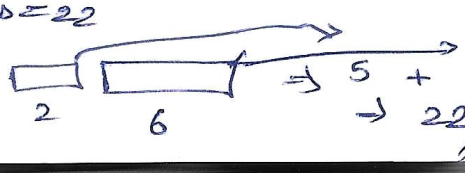
Rod length = 8

Max profit { price = val
length = wt
total Rod = W

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

Same as unbounded knapsack because you can make a piece of same length repeated time.

Ans = 22

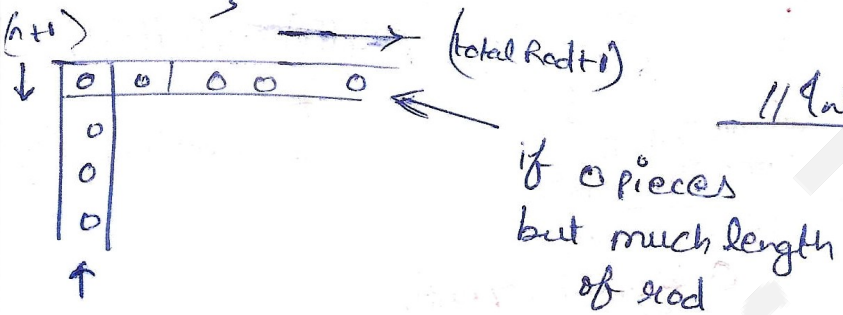


// max profit

Code

wt = length
val = price
W = total Rod

```
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][j-length[i-1]], dp[i-1][j]);
        } else { // invalid
            dp[i][j] = dp[i-1][j];
        }
    }
}
```



// Initialisation

if 0 pieces but much length of rod

if many pices but accepted rod length is 0

Longest Common Subsequence (LCS)

↳ same order And in bet" character can be missing.

str1 = "abcde", str2 = "ace"

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 = "abcdge" str2 = "abedg"

ans = "abdg" = 4.

• Recursion logic → we will check ~~from~~ last characters of string

→ (str1, n) ← last char (str2, m) ← last char.

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

→ (str1, n-1) (str2, m-1)

② case ↳ ans + 1

if n != m then we have two choice

either remove element from str1 or str2

ans1 [(str1, n-1) (str2, m) (str1, n) (str2, m-1)] ans2

return Math.max(ans1, ans2)

• Base case

if (str1 = 0 || str2 = 0)
that is n = 0 that is m = 0

lcs = 0
return 0;

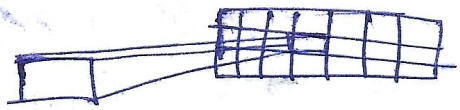
Pseudocode

```
lcs(str1, str2, n, m){
  if(n==0 || m==0){ return 0; }   // base case
  if(str1(n-1) == str2(m-1)){ // same
    return lcs(str1, str2, n-1, m-1) + 1; }   // adding 1 in common string length that is ans.
  else{
    ans1 = lcs(str1, str2, n-1, m);
    ans2 = lcs(str1, str2, n, m-1);
    return max(ans1, ans2);
```

T.C = O(n×m)

LCS (Memonization) → here there are 2 fluctuating integers
length of str1 and length of str2

⇒ As we have some overlapping cases so we are using memonization to reduce T.C → So we will take a 2D Array



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

→ if str1 is of i length and str2 is of j length lcs will be in that cell

Code

→ just check that weather we already have a sol^ for that string lengths.

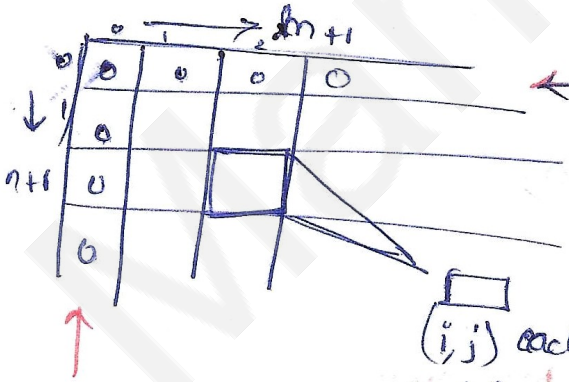
→ so for that initialize the ~~this~~ whole array with -1.

LCS Tabulation

1. table
2. meaning + initialization
3. fill (bottom up)

small problem → large problem

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



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

• Pseudo code

```
for(int 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[j][j-1];
dp[i][j] = Max(ans1, ans2);
```

Longest Common Substring T.C = O(m×n)

↳ contiguous sequence.

ex → S_1 = "ABCDE",
S_2 = "ABGCE"

Ans = 2 = AB

ex →
S_1 = ABCDGH
S_2 = ACDGHR

ans = 4
⇒ "CDGH"

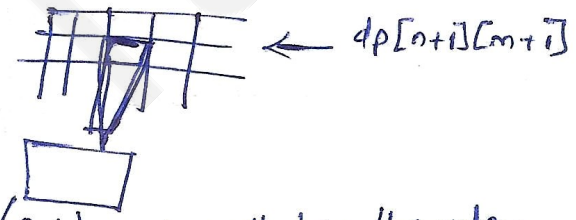
//Logic

→ A B C D E
A B G C E
L.C.Sub → 2 1 0 0 1

Max LCS

if character is diff then the counter reset

two fluctating 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 string is is i and j

#base case

```
if (str1.length() == 0 || str2.length() == 0)
    dcs = 0;

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

    if (str1.charAt(n-i) == str2.charAt(j-i)) {
        dp[i][j] = dp[i-1][j-1] + 1;
    } else
        { dp[i][j] = 0;
        }
  }
}
return ans = Math.max(ans, dp[i][j]);
```

← if same then the length is increasing

← but if not same then the counter get reset

As we want a longest common substring.

Longest Increasing Subsequence (LIS)

T.C = O(n*n)

↳ ordered maintain
↳ Increasing order
↳ sorted in acceding order

Example

arr [] = {50, 3, 10, 7, 40, 80}

LIS = ~~4~~

Ex - 50, 80

3, 10, 40, 80 ✓ } 4 length
3, 7, 40, 80 ✓ }

10, 40, 80

40, 80

Approach to solve

① take a sorted array of unique elements

② then take a longest common sub sequence

To take out this add all the element in Hash set then sort.

Ex → 50, 3, 10, 7, 40, 80

Longest sorted unique → 3, 7, 10, 40, 50, 80

→ 3, 10, 40, 80

longest common subsequence.

~~Edit Distance~~ String Conversion / edit distance

↳ given two strings word1 and word2

Minimum number of operations to convert word 1 to word2

3 operations

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

Ans. ∴ ~~steps~~ 5 steps

ex → word1 → "intention" word2 = "execution"

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

0 Recursion Approach

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

n = str1.length();
m = str2.length();

Case (1) if (str1(n) == str1(m))

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

Case 2 if (str1(n) != str1(m) { ← character are differen

here the (str2.length - 1) because one character satisfied

a) Add str1(n), str2(m+1) + 1 because one operation is performed

b) delete str1(n-1), str2(m) + 1

c) replace str1(n-1) str2(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 str1 = i and str2 = j.

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

→ n+1

0	1	2	3
1			
2			
3			
4			

↓ n+1

and here we need to delete the str1 characters is our str2 is empty

∴ No. of operation = 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;
}
```

Pseudocode

```
if (str1(i-1) == str2(j-1)) {
    dp[i][j] = dp[i-1][j-1]
}
```

if removed char from each string

```
else {
    dp[i][j] = min(dp[i][j-1], dp[i-1][j], dp[i-
}
```

add case

Q String Conversion

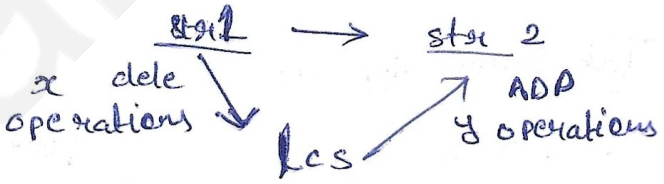
only 2 operation is allowed.

1) insertion
2) deletion

str 1 = "pear"
str 2 = "sea"

Take lcs

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



⇒ x+y = operations required

form

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

#8. Wild card matching.

→ we have given a normal text and a wild card text

→ wild card text contain normal character (alphabets) and special two characters '*' and '?'

We are said to convert str1 to str2 is posible 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 wheater the string/text can be matched or not ?

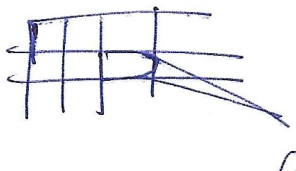
⇒ ex → s = "aa"
p = "*" ← this can be changed to "aa"

output true.

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

→ Tabulation approach
2D array.

1) dp [n+1][m+1]



(i, j) each cell will contain if the string length = i, if pattern length = j, then it is possible T/F to convert or NOT

// ~~base case~~ initialisation

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

if (s="2" p=0 then false

if (s="" p=2 then

empty.

(a) if p="*" then True

(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) }
    }
```

// fill in bottom

s="abc" p="aac"

(1) if last character are equal

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

then we will check for the remaining string.

(2)

if s="abc"

p="ab?"

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

↑ as question mark can be converted we will check for remaining