→ IA
→ Autotag

→ PSE
→ EO
→ WX-PSE
→ WX-PRE

Shortest path

Bellman ford's Algo

```
vector < int > v [MAX];  ← All the edges.
int dis [N+5];

                                Negative cycle

for ( i=0 ; i<n+2 ; i++)
{
    v[i]. clear ();
    dis [i] = INF ;
}.

for ( i=0 ; i<m ; i++)
{   cin >> & from >> next >> weight )

    v[i] . pb ( from);
    v[i] . pb ( next);
    v[i]. bb (weight);
}

dis [0] = 0;
for ( int i=0 ; i<n-1 ; i++){
    j=0 ;
    while ( v[j]. size() != 0) {
     if dis [v[j][0]] + v[j][1] < dis [v[j][1]]
        dis[ v[j][1] ] = dis [v[j][0]] +
                                v[j][1] ;
}   } j++ ;
```

(BF) Bellman ford: $O(V \cdot E)$ if $E = V^2$ $O(E^3)$

Dijkstra's Algo

① dis = [INF] except source = 0.

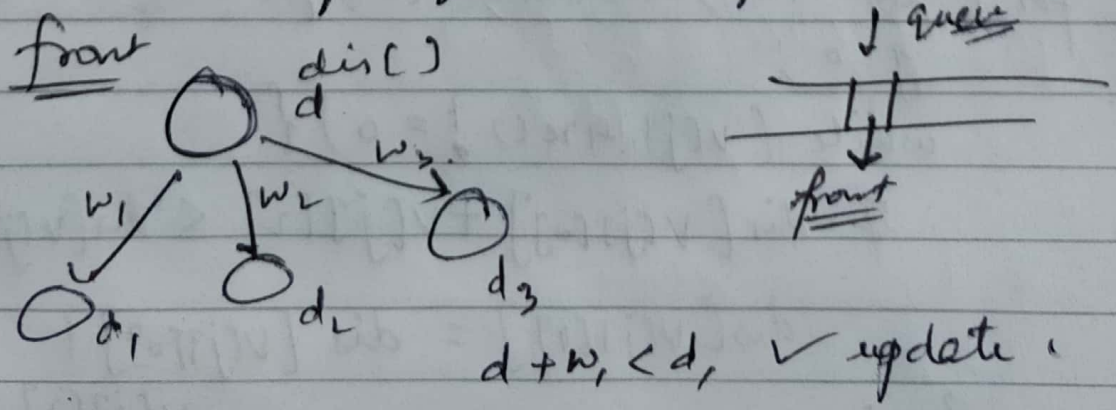
② push source vertex in min-priority queue.
(distance, vertex)
↳ for min-priority queue.

③ pop the vertex with min distance
(V) → child

④ update the distances of child if
current + edge weight < next vert dis.
then push this vertex in to priority queue.

⑤ If popped vertex is visited before,
don't use it.

⑥ Until pri queue is empty.



multiset

```
#define SIZE 10000+1
vector < pair < int, int > > v[SZ];   v - u weight.
int dis[SZ];
bool vis[SZ];
void dijkstra(){
    vis[SZ] → set all false;
    dis[source] = 0;
multiset < pair < int, int > > s;
    s.insert ({0,1});

    while ( !s.empty() ){
pair < int, int > p = *(s.begin())   → first element
    s.erase ( s.begin() );   (pop vertex)
int v = p.second; int wei = p.first;
 if (vis[v]) continue;
  vis[v] = true;
for ( int i=0; i < v[v].size(); i++ ){
  int e = v[v][i].ff; int w = v[x][i].ss;
    if ( (dis[x]+w) < dis[e] ){
        dis[e] = dis[x]+w;
        s.insert ( { dis[e], e } );
}  }
}
```

$O(V^2)$ →

$O(V+E\log V)$ → min priority queue

Floyd-Warshall Algorithm

$O(V^3)$

shortest path b/w All pair of vertices.

N vertices dis[N][N] {INF} each.

→ Initialize → Path dis to ∞.

→ find all pair shortest path that use 0 intermediate vertices. then using 1 inter. 2 etc.

→ Minimize the shortest path b/w any pair of vertices (i, j), minimize dis using the k nodes, so shortest path will be

min (dis[i][k] + dis[k,j], dis[i][j])

```
for (k=1; k<=n; k++){
    for(int i=1; i<=n; i++){
        for(int j=1;j<=n;j++){
            dis[i][j] = min( dis[i][j],
                            dis[i][k] + dis[k][j])
        }
    }
}
```

$O(V^3)$ →

```
int main   nodes, edges.
  init().
  for (i=0; i<edges; i++){
      cin>>x>>y>>weight;
      p[i] = { weight, m_p(x,y)};
  }.
  sort (p, p+edges);

  mincost = kruskal(p);
}.

LL kruskal( pair<LL, pair<int,int>> p[]){
  int x, y;
  LL cost, min = 0;
  for( i=0; i<edges; i++) {
      x = p[i].ss.ff; y = p[i].ss.ss;
      cost = p[i].ff;
// check if cycle.
      if( root(x) != root(y) ) {
          mincost += cost;
          union(x, y);
      }
  }
  return mincost;
}.
```

→ E log V → sorting. (E Log E.)
↳ disjoint set operations.

Prims - Greedy Approach - / Add vertex in each
we add vertex in prims. / step.

Algo $O((V+E)\log V)$

→ Maintain 2 disjoint sets of V. One contain Vs. that are growing spanning tree, & other that are not.

→ select cheapest vertex connected to growing spanning tree & not in growing tree. Add it.
using priority queue. → Insert vertices that are connected growing tree into PQ.

→ check for cycles. Mark the node which have been already selected & insert only and inserted to PQ that are not marked.

```
ll prim ( int x) {        // start vertex
    priority_queue < PII , vector<PII> , greater<PII> > Q;
                     ↓ type      ↓ container type.
    int y; mincost = 0; , PII p;
    Q.push( make_pair(0, x) );  → Insert edge length, vertex.
    while ( !Q.empty() ) { // select edge with min edge.
    p = Q.top(); Q.pop();
      x = p.ss;   if (marked[x]) continue;
    mincost += p.ff;   marked[x] = true;
      for ( int i=0 ; i< adj[x].size(); i++) {
          y = adj[x][i].ss;
            if(marked[x] == false)
                Q.push ( adj[x][i] );
      }
    }
    return mincost;
}
```

Flood-Fill Algorithm

Helps in visiting each & every point in a given area.
Determines the area connected in a multi-dimensional array.

① Bucket Fill

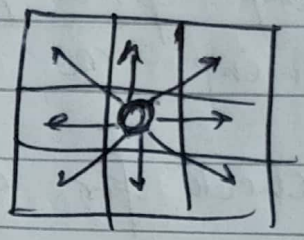
② Solving a Maze. (Matrix) start → Dest^n (with obstacles) helps in finding path.

③ Mine Sweeper. → blank cell discovered, it helps in revealing cells. this done recursively till cells having no. are discovered.

(x,y)

8 cells around it.

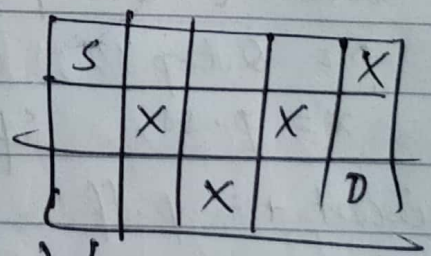
or 4 cells —"—.



Graph Traversal

```
DFS ( x, y, visited, n, m ).
{
    if (x>=n or y>=m) return;
    if (x<0 or y<0)   return;
    if (visited[x][y]) return;
    visited[x][y] = true;
    ↳ call dfs for each connected cell.
    ↳ for ( _____ )
          dfs (x+dx, y+dy, vis, n, m);
}
```

Solving a Maze →




```
DFS (x, y, vis, n, m,
     mat, dest_x, dest_y){
    if (x==dest_x && y==dest_y) return true;
    if (x>=n or y>=m or x<0 or y<0) return false;
    if (visited[x][y]) return false;
    if (mat[x][y]==X) return false;
```

visited = true;

↳ call Dfs for neighbour cells.

if (DFS(x+dx, y+dy, vis . . . .) == true)
return true;

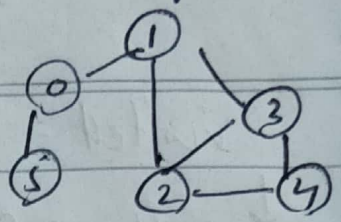
return false; // Not found;

}

→ Articulation Points and Bridges

vertex is AP if removing it and all edges associated with it results in ↑s in no. of connected components.

0 & 1 → are a AP.



Brute-force → remove each and check.

$O(V \cdot (V+E))$.

Back-Edge

↳ All AP in a single DFS traversal.

edge that connects a vertex to a vertex that is discovered before it's parent.

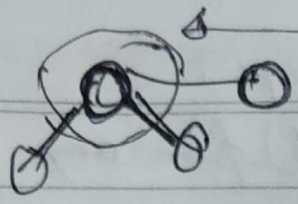
↳ edge connecting to its ancestor.

Presence of backedge means there is alternate path if parent is removed.

```
→ time = 0;
DFS(adj[][], disc[], low[], vis[], par[], AP[], vertex, V)
{   vis[vertex] = true;
    dis[vertex] = low[vertex] = time+1;
    child = 0;
for( i=0 to V )                    // if edge.
    if (adj[vertex][i] == true)
        if (vis[i] == false)
            child = child+1;
            parent[i] = vertex;
        DFS(adj, dis, low, vis, parent, AP, i, n, time+1)
        low[vertex] = min( low[i], low[i] );
    if ( parent[vertex] == nil && child > 1 ) AP[vertex] = 1;
    if ( parent[vertex] != nil && low[i] >= disc[vertex]
            AP[vertex] = 1;
    else parent[vertex] != i
        low[vertex] = min (low[vertex], disc[i])
```

Bridges

Removing edge b/w $u \not\leftrightarrow v$ T^s no. of components



Remove this No of roots, that can be winner. 1

Bi-connected Graphs

Even after removing any vertex, graph remains connected.

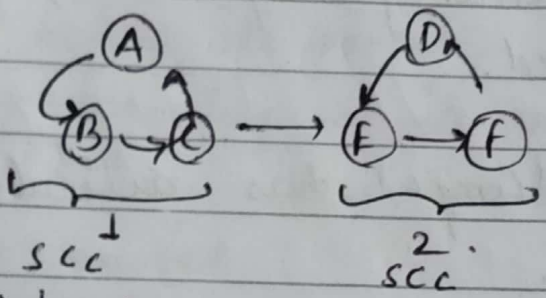
↳ check if it(Graph) has articulation point or not.

Strongly Connected Components (Directed Graph)

Connectivity in undirected graph → means one can go to other, via any path.

→ **Strong Connectivity** (in directed graph)
if there is a path (directed) from any vertex to every other vertex.

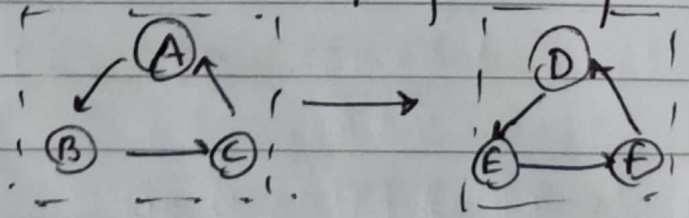
→ Graph → Broken into strongly Connected Components.



Brute force $O(V^3)$

→ Kosaraju's Algo $O(n)$
DFS twice. $O(V+E)$.

→
Condensed Component Graph. $\leq V$ nodes, $\leq E$ edges
in which every node is SCC. & there is an edge from C to C', where C & C' are SCC, if there is an edge from any node of C to any node C'



DAG
Directed Acyclic Graph.
↓
at least one node node with no incoming edges. & at least one node with no outgoing edge.

DFS done from sink,
Only nodes from SCC visited.
↳ to find another sink, remove some edge.

So, if there is an edge from C to C' in condensed component graph, finish time of node C will always be higher than finish time of C'.

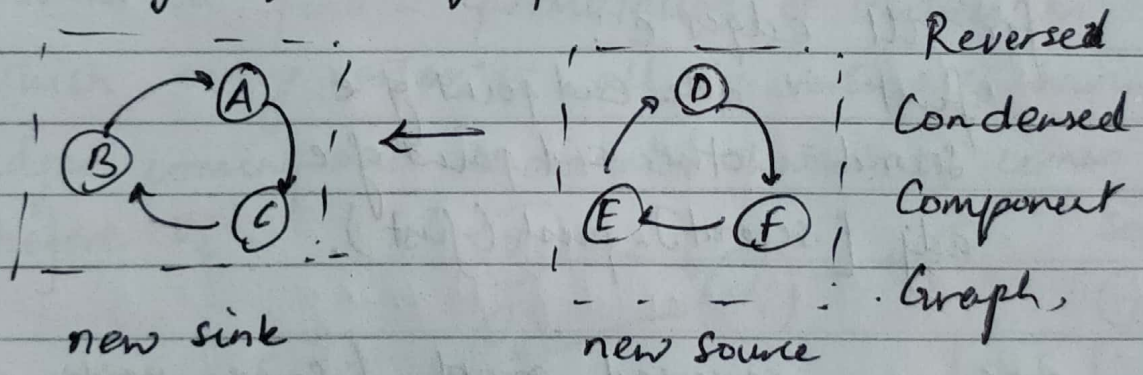
Topological Sorting → (linear arrangements of nodes in which edges go from left to right.) of CCG. can be done, the some node in leftmost SCC will have higher finishing time than all nodes in SCC's to the right in topological sorting.

→ how to find some node in sink SCC.

the condensed comp Graph can be reversed the all source → sinks.

sinks → sources.

So, SCC of reversed graph will be same as SCC of original graph.



→ DFS on new sinks.

Order in which DFS on new sinks needs to be done is known.

Algo 2 DFS

① DFS on original graph, keep track of finish time using stack. DFS finish → put it to stack. top will be highest finishing time.

```
stack STACK.
void DFS ( int source) {
    visited [s] = true;
    for all child that are not visited:
        DFS (X)

    STACK.push( source)
}
```

② Reverse graph.

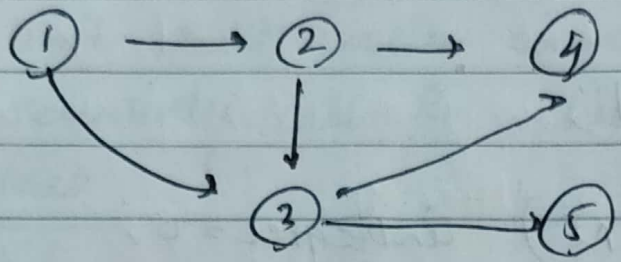
```
clear adj_list;
for all edges e:
    first = one end point of e
    second = other end point of e.
    adj [second]. push (first).
```

③ DFS on reversed graph. (Source vertex as top of the stack.

```
while (!STACK.empty())
    source = STACK.top()
    if source is visited: continue;
    else
        DFS (source)
```

Topological Sort (Only for DAG)

DAG is an ordering of vertices $v_1, v_2 \ldots v_n$ in such a way, that there is an edge directed towards v_j from v_i, then v_i comes before v_j.

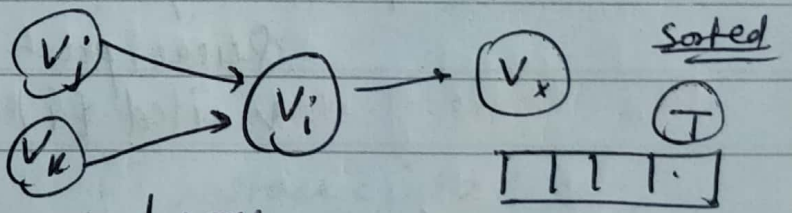


1 2 3 4 5.

there are multiple topological sorting possible for a graph.

→ In order to do topological sorting the graph must not contain any cycles.

→ want to find a permutation of vertices in which every vertex v_i, all the vertices v_j having edges coming out & directed towards v_i comes before v_i.



these sud come earlier then rest.

array indegree[N];

↓

[i] → means no of vertices which are not already inserted in T and there is an edge from them incident on vertex i.

```
topological Sort ( N, adj[N][N] )
{
    T = [];
    visited = [];
    inDegree = [];

    for( i=0; i<N )  inDegree = 0;
                     visited = 0;

Fill indegree;
    for ( i=0 to N)
        for ( j=0 to N)
            if (adj[i][j])
                indegree[j]++;

    for ( i=0 to N)
        if (in_degree==0)
            Queue.push (i);
            visited[i] = true;

    while (!Q.empty())
        vertex = Q.front(); Q.pop();
        T.append (vertex);
        for ( j=0 ; to N)
            if adj[vertex][j] = true && visited[j] = false
                indegree[j] --;
                if (indegree[j]==0)
                    Q.push (j);
                    visited[j] = true;
return T[];
```

Topological Sort (modifying DFS)

start from vertex.
print it, then call for child. } DFS
- use temp stack.

→ Use STACK

Don't print vertex immediatly, recursively call for children. then push to stack.

```
funct ( )
    stack<int> st;
    vis[] = 0;

    for (int i=0; i<V; i++)
        if (visited[i])
            topo (i, vis, stack);

    while ( stack.empty() == false)
    { → stack.top(), st.pop(); }

topo ( int v, vis[], stack<int> & st )
{
    vis[v] = true;

    for (auto i = adj[v].begin to end() )
        if (! vis[i])
            topo (*i, vis, st);

    st.push(v);
}
```

Dynamic Programming

- Break problem into subproblem
- Recursively define value of Sol by expressing in terms of smaller-problems.
- Compute in bottom-up.

→ can be Table-filling

1. Optimization problems → feasible sol, minize maxiz
2. Combinatrial problems → Count No of ways
   Prob. of event

→ Q N → No of diff ways to write it as sum of 1, 3, 4.

$1x + 3y + 4z = N$.

DP_n → No of ways to write N as sum of 1, 3, 4

$$DP_n = DP_{n-1} + DP_{n-3} + DP_{n-4}$$

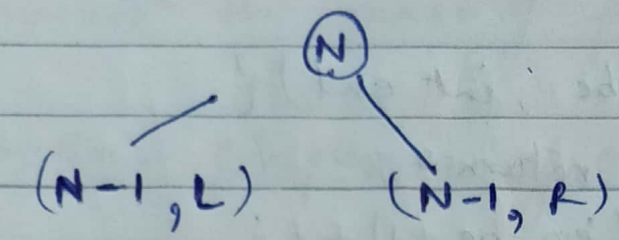
DP_{n-1}: No of ways if last is 1;

Base → $DP_0 = DP_1 = DP_2 = 1$, $DP_3 = 2$, DP

```
for( i=4; i<=N; i++)
    DP[i] = DP[i-1] + DP[i-3] + DP[i-4];
```

Wines 1 2 3 . . . N .

→ a_1 a_2 a_3 . . . a_n ←



DP(N) = max ((N-1, L) , (N-1, R)

return ~~rec~~ max(rec(n-1) , rec(n-1))

rec(arr[], year, n, i, j)

{ if (i > j) return 0;

return (max (rec(y+1, be+1, en) + y * p[be]

rec(y+1, be, en-1) + y * p[en])

} → 2^N → year wise

dp[i]

best profit on year - (y) and (interval) [s, e]

minimize → state space func^n arguments.

year, size [n-year]] (year, idx)

→ look for arguments / if we don't need to calculate

(N) → global total wines

int p[N];

$\rightarrow 2^N$

```
int profit(int be, int end){
    if(be > en) return 0;
    int y = N-(en-be+1)+1;
    return max( profit(be+1, en) + year*p[be]
                profit(be, en-1) + year*p[en]
              )
}
```

→ func' → has (N^2) diff arguments.
Can be called with.

∴ At most (N^2) diff things we need to calculate.

int p[N], Ans[N][N]; → init with [-1];

```
int profit(int be, int en){
    if(be > en) return 0;
    if(ans[be][en] != -1) return ans[be][en];
    int year = N-(en-be+1)+1;
    return ans[be][en] = max(
        profit(be+1, en) + year*p[be];
        profit(be, en-1) + year*p[en]);
}
```

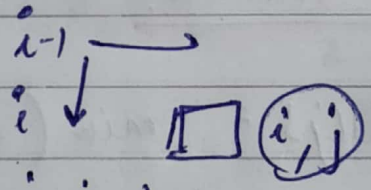
Try to create Backtrack func^n.
Avoid redundant arguments
minimize the range of possible values of func^n Arguments.
- Optimize complexity of one func^n call.

2-D

Finding a min cost path in grid.
finding no of ways to reach a [][]

① Min-Cost path in 2-D.

cost[i][j] →

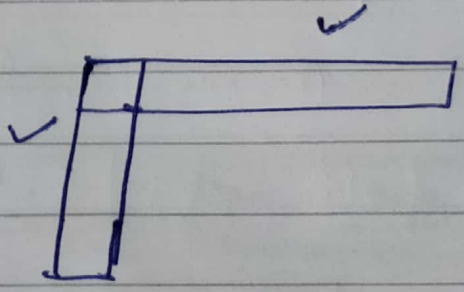


(i-1, j) (i, j-1)

Mincost[i][j] = min(mincost[i][j-1], mincost[i-1][j]) + cost[i][j];

Optimal Sub-structure
~~Opte~~ Overlapping Sub-Problems.

Base Cases →



(2) finding min no of ways to reach from start to end position.

noofways [i][j] = noofways [i-1][j] + noofways [i][j-1] ;

(3) → If some cells are blocked.

(4) Edit Distance →

if not same

dp[i][j] = min (dp[i-1][j], dp[i][j-1]) ↳ add cost of remove insert delete

if same,

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

Greedy Algorithms → (just Techniques)

Always make the smartest choice at the moment.
∴ locally optimal choice.

→ objective func" → to be optimized. (min or max)
it never goes back & reverse the decision.
only one shot.

→ Difficult part is → work harder to understand the correctness issues.

Q → scheduling jobs. Minimize total time for finishing tasks of jobs.

$F = P[1] * C[1] + P[2] * C[2] + \ldots P[N] * C[N]$

same prio → then in sorted order. (same priority)

1
2 ↑

1
1+2 } 4

2
2+1 } 5.

✓ priority, time[]

sort them.

(P[i]/T[i], i)

Som, 20k.

→ Hari Ballon ✓ 20k
→ Ramswamy

Coregaon
Bangalore