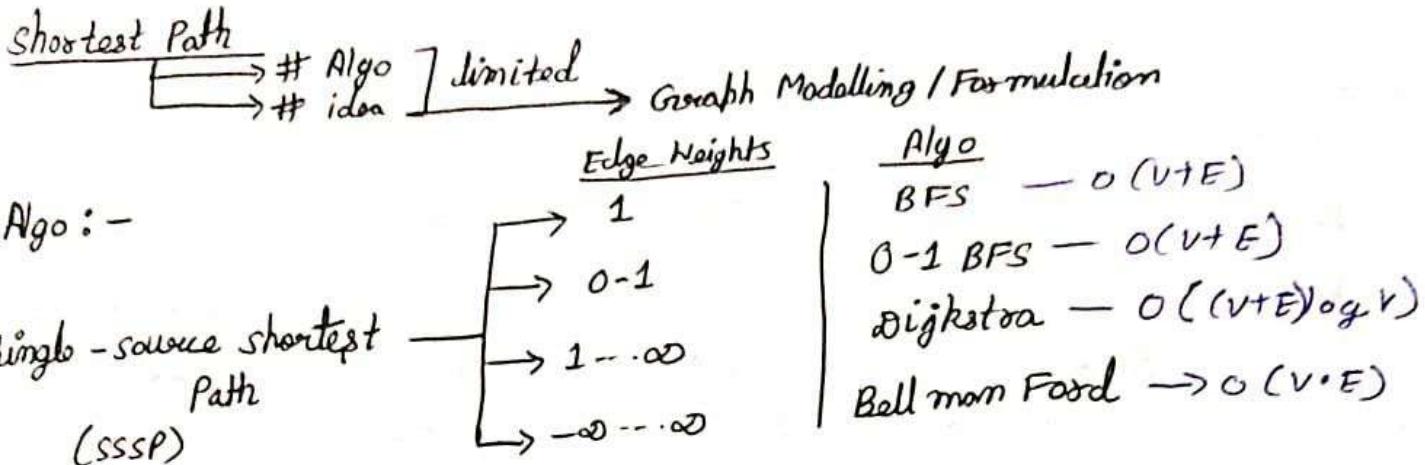


Graph shortest Path Algorithm



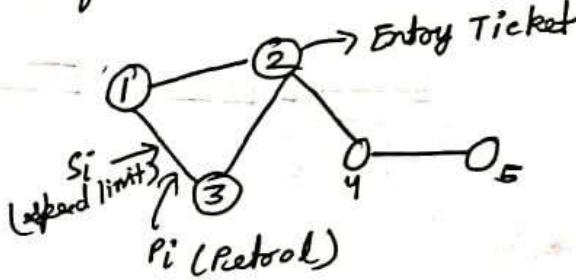
All Pair shortest Path (APSP) \rightarrow $-\infty$: Flyod Marshall Algo
 $\hookrightarrow O(v^3)$

Storing Graph :-

Suppose you are given a graph of cities. In these there are city 1, city 2, ..., city 5. You are given edges. Each city has Entry Ticket which you have to pay when you visit the city. The edges roads (edges) have speed limit (s_i) and petrol (P_i). How will you store this information in your code? (some roads are one-directional).

1. For every i , we need to store some extra information,

$$\text{Entry []} = \{ \begin{smallmatrix} 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 \end{smallmatrix} \}$$



- 2o Previously in edge information, we are storing its neighbour as vector <vector<int>>g

Now, with the neighbors will be store edge info,

1: [(3) 3, 5]
Q Edge Info.

\Rightarrow `vector<vector<pair<int, int>>>` `pair <int, int>>>`

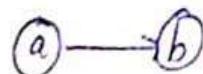
\uparrow \uparrow
 neighbour city
 si
 \uparrow \uparrow
 pi
Edge info.

3. For bidirectional graph,



$g[a].push_back(b);$
 $g[b].push_back(a);$

For undirectional graph,

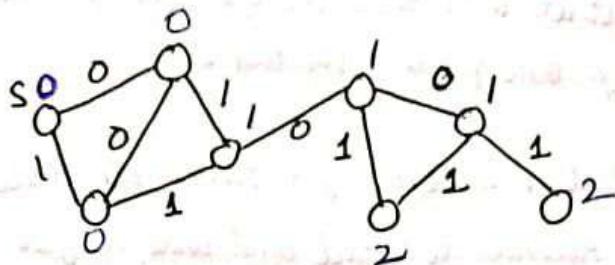


$g[a].push_back(b);$

* So there are three things to store information in graph.

0-1 BFS Algo :-

Q:- Run SSSP algo from the given node to every other node to find the distance. Each edge is weighted.



$$\text{Distance} = (\Sigma w) \text{ path} \downarrow$$

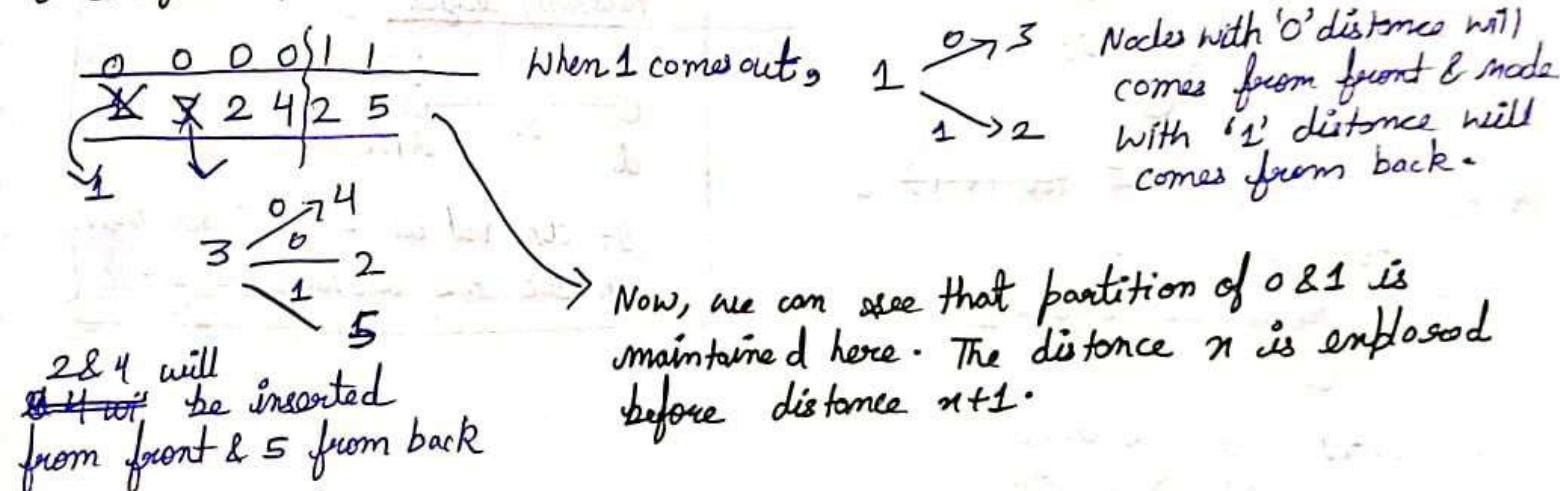
↳ sum of edge weight that are on path

Shortest Path \rightarrow Min distance

① Shortest node not visited comes out

② Process exactly once.

Even if the same things are coming out twice, the shorter one should come first. So instead of a queue we will use a deque and this will solve gives us 0-1 Algorithm.



Main Trick of 0-1 BFS :-

- 1) Change Queue to Deque
- 2) If 0 edge Put it at front and if 1 edge put it at back
- 3) As long as sorted property of distances, it will always work.

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

```
#include<bits/stdc++.h>
using namespace std;
#define F first
#define S second
const int INF = 1e9;

int n,m;
vector<vector<pair<int,int>>> g;

vector<int> dist,vis;
void bfs01(int sc){
    dist.assign(n+1,INF);
    vis.assign(n+1,0);

    // ds
    deque<int> dq;
```

```
// Set up source
dist[sc]=0;
dq.push_back(sc);

// keep exploring until empty
while(!dq.empty()){
    int cur = dq.front(); dq.pop_front();
    if(vis[cur]) continue;

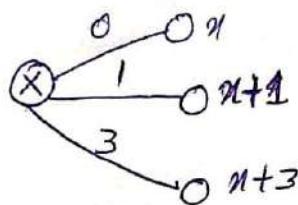
    // explore
    vis[cur] = 1;
    for(auto v:g[cur]){
        if(!vis[v.F] && dist[v.F]>dist[cur]+v.S){
            dist[v.F] = dist[cur]+v.S;
            if(v.S==0){
                dq.push_front(v.F);
            }else{
                dq.push_back(v.F);
            }
        }
    }
}
```

```
void solve(){

    cin>>n>>m;
    g.resize(n+1);
    for(int i=0;i<n;i++){
        int a,b,c;
        cin>>a>>b>>c;
        g[a].push_back({b,c});
        g[b].push_back({a,c});
    }
    bfs01();
    for(int i=1;i<=n;i++){
        cout<<dist[i]<<" ";
    }
}
```

Dijkstra Algorithm :-

If the edges can be anything from $0 \dots \infty$, then we use Dijkstra Algo.



If we release x , then any of these n , $n+1$ and $n+3$ can be possible in a queue, you are putting node with some distance but it will have to remain sorted. We will use priority queue which will keep our queue sorted.

* We require min. distance but PQ output max on pop. Hence we will insert negative distance to make our PQ reverse sorted and will negate it back once it comes out.

```
void dijkstra(int sc){
    dist.assign(n+1, INF);
    vis.assign(n+1, 0);

    // DS
    priority_queue<pair<int, int>> pq;
    pq.push({-0, sc});

    // setup source
    dist[sc]=0;
    pq.push({-0, sc});

    // keep exploring until empty
    while(!dq.empty()){
        auto temp = dq.top(); dq.pop();
        int cur = temp.S;
        if(vis[cur]) continue;

        for(auto &e : adjList[cur]){
            if(dist[e.S] >= dist[cur] + e.W) {
                dist[e.S] = dist[cur] + e.W;
                pq.push({-dist[e.S], e.S});
            }
        }
    }
}
```

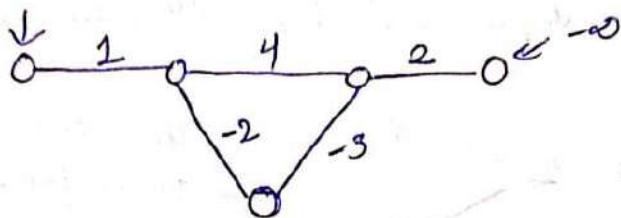
```
// keep exploring until empty
while(!dq.empty()){
    auto temp = dq.top(); dq.pop();
    int cur = temp.S;
    if(vis[cur]) continue;

    // explore
    vis[cur] = 1;
    for(auto v:g[cur]){
        if(!vis[v.F] && dist[v.F]>dist[cur]+v.S){
            dist[v.F] = dist[cur]+v.S;
            dq.push({-dist[v.F],v.F})
        }
    }
}
```

```
void solve(){
    cin>>n>>m;
    g.resize(n+1);
    for(int i=0;i<m;i++){
        int a,b,c;
        cin>>a>>b>>c;
        g[a].push_back({b,c});
        g[b].push_back({a,c});
    }
    dijkstra(1);
    for(int i=1;i<=n;i++){
        cout<<dist[i]<<" ";
    }
}
```

Bellman Ford :-

↳ Used in negative cycles



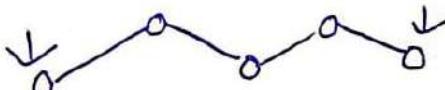
This algorithm says that for $(v-1)$:-

[for $(v-1)$ times)
relax all Edges]

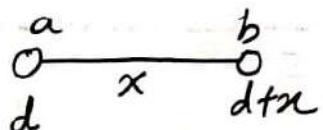
↓
for $(v-1)$ times // O(V)
for $(e \in \text{Edges})$ // O(E)
 relax(e) // O(1)

⇒ So, Total complexity = $O(V \cdot E)$

Why will this always work?



Relaxing edges



If the val at b is bigger than a , set the distance = $d+x$

From one node to any node the longest shortest path will have v nodes because a shortest path cannot repeat a node unless there is a -ve loop which makes the no. of edges in the longest path will be at max $(V-1)$.

```
void bellman(){
    cin>>n>>m;
    // g.resize(n+1);
    vector<vector<int>> edges;
    for(int i=0;i<m;i++){
        int a,b,c;
        cin>>a>>b>>c;
        // g[a].push_back({b,c});
        // g[b].push_back({a,c});
        edges.push_back({a,b,c});
    }
    // dijkstra(1);
    vector<int> dist(n+1,INF);
    dist[1]=0;
    for(int i=0;i<n-1;i++){
        for(auto edge:edges){
            if(dist[edge[1]] > dist[edge[0]]+edge[2]){
                dist[edge[1]] = dist[edge[0]]+edge[2];
            }
        }
    }
}
```

```
r
int changed = 0;
for(auto edge:edges){
    if(dist[edge[1]] > dist[edge[0]]+edge[2]){
        dist[edge[1]] = dist[edge[0]]+edge[2];
        changed = 1;
    }
}

if(changed){
    cout<<"Negative loop reached from 1"<<endl;
}else{
    cout<<dist[n]<<endl;
```

* APSP : Floyd Warshall $O(V^3)$

↳ Used to find shortest path between any two nodes.

```
void floyd_marshall(){
    int n,m;
    int adj[n+1][n+1];
    // adj[i][j] -> cost to go from i to j.
    for( int i=1; i<=n; i++){
        for( int j=1; j<=n; j++){
            if(i==j)adj[i][i]=0;
            else adj[i][j]=INF;
        }
    }
    for( int i=0; i<m; i++){
        int a,b,c;
        cin>>a>>b>>c;
        adj[a][b]=min(adj[a][b],c);
    }
}
```

```
// 4 line magic
for<int k=1;k<=n;k++>{
    for<int i=1;i<=n;i++>{
        for<int j=1;j<=n;j++>{
            adj[i][j] = min(adj[i][j],adj[i][k]+adj[k][j]);
        }
    }
}
// adj[i][j] -> i to j shortest path value.
```