BELLMAN FORD CODE:

```cpp
class Graph
{
    map<int, map<int,int> > strg ;
    int V ;
    vector<vector<int> > edgeList ;

    public :

    Graph(int V)
    {
        this->V = V ;
    }

    void addEdge(int u, int v, int cost)
    {
        strg[u][v] = cost ;
        edgeList.push_back({u,v,cost}) ;
    }

    void display()
    {
        for(int i = 0 ; i < V ; i++)
        {
            cout << i << "\t" ;

            map<int, int>::iterator itr ;

            for(itr = strg[i].begin() ; itr != strg[i].end() ; itr++)
                cout << itr->first << "@" << itr->second << ", " ;
            cout << endl ;
        }
    }

    void bellmanFord(int src)
    {
        int cost[V] ;
        fill(cost,cost+V,100000) ;
        cost[src] = 0 ;

        // V-1 times, relax every edge
        for(int i = 1 ; i <= V ; i++)
        {
            for(auto edge : edgeList)
            {
                int u = edge[0] ;
                int v = edge[1] ;
                int c = edge[2] ;

                // cost  Relax
                int oc = cost[v] ;
                int nc = cost[u] + c ;
                if(nc < oc)
                {
                    if(i <= V-1)
                        cost[v] = nc ;
                    else
                    {
                        cout << "-ve wt cycle present" ;
                        return ;
                    }
                }
            }
        }

        for(int i =  0 ; i < V ; i++)
            cout << i << " -> " << cost[i] << endl ;
    }
} ;
```

[[0,1,10] [0,2,20] [ ] : [ ]]

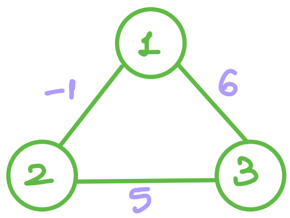
fill → O(V)

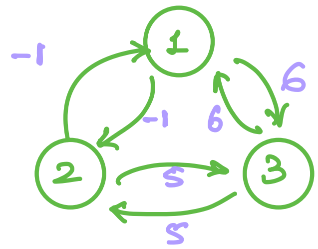
for i loop — O(V)

for edge loop [0,1,10] — O(E)

loop body — O(1)

V+VE = O(EV)





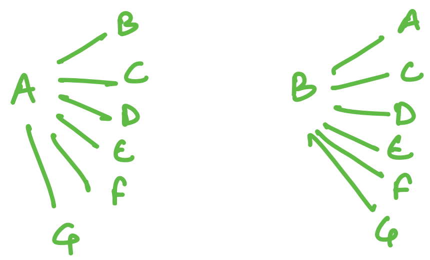
Undirected graph with -ve edge ⇒ Directed graph with -ve wt cycle

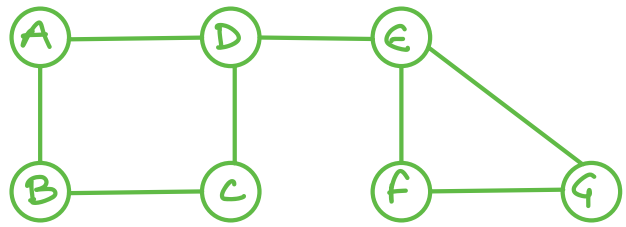
no SSSP algo that works with -ve wt cycle.

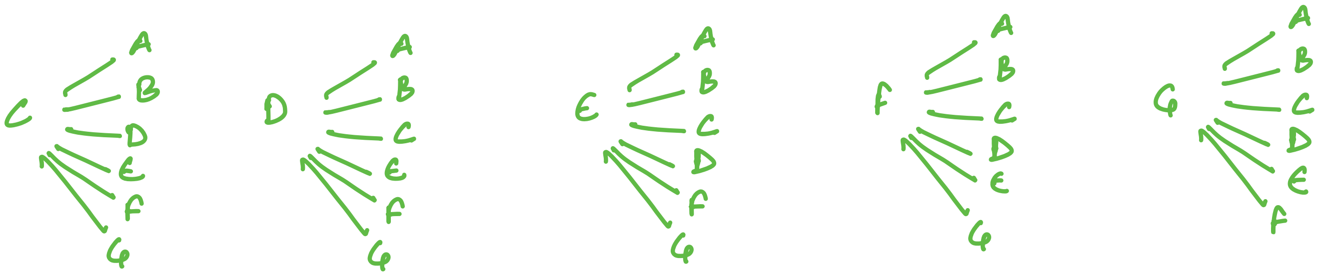
DIJKSTRA	BELLMAN FORD
- undirected graph : works with all +ve wt edges	- undirected graph : works with all +ve wt edges
- directed graph : works with all +ve wt edges	- directed graph : works with +ve wt cycle and -ve wt edge. Doesnot work with -ve wt cycle.
- Time Complexity : $O(E \log V)$	- Time Complexity : $O(EV)$

GRAPH
- MST
  - Prims $O(E \log V)$
  - Kruskal $O(E \log V)$
- SSSP
  - Dijkstra $O(E \log V)$
  - Bellman ford $O(EV)$
- APSP
  - floyd warshall

All Pair Shortest Path







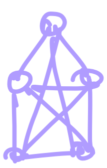
One option:

	Dijkstra	Bellman ford
SSSP:	$E \log V$	EV
APSP	$V \cdot E \log V$	$V \cdot EV = V^2 E$
	$\downarrow$	$\downarrow$
	$V^3 \log V$	V^4

Floyd Warshall V^3

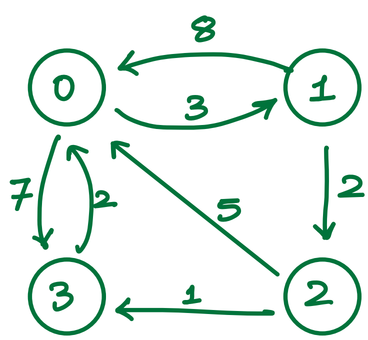
Complete Graph $E = V^2$

$\downarrow$



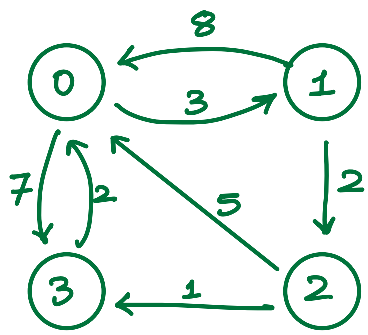
$$ {}^{V}C_2 = \frac{V!}{(V-2)!\,2!} = \frac{(V-2)!(V-1)(V)}{(V-2)!\,2!} = \frac{V(V-1)}{2} = V^2 $$

	0	1	2	3
0	0	3	∞	7
1	8	0	2	∞
2	5	∞	0	1
3	2	∞	∞	0



Vertex 0

	0	1	2	3
0	0	3	∞	7
1	8	0	2	~~∞~~ 15
2	5	~~∞~~ 8	0	1
3	2	~~∞~~ 5	∞	0



$(1,1):0$ $\quad \underbrace{1 \to 0}_{8} \underbrace{\to 1}_{3} = 11$ ×

$(1,2):2$ $\quad \underbrace{1 \to 0}_{8} \underbrace{\to 2}_{\infty}$ ×

$(1,3):\infty$ $\quad \underbrace{1 \to 0}_{8} \underbrace{\to 3}_{7} = 15$ ✓

$(2,1):\infty$ $\quad \underbrace{2 \to 0}_{5} \underbrace{\to 1}_{3} = 8$ ✓

$(2,3):1$ $\quad \underbrace{2 \to 0}_{5} \underbrace{\to 3}_{7} = 12$ ×

$(3,1):\infty$ $\quad \underbrace{3 \to 0}_{2} \underbrace{\to 1}_{3} = 5$ ✓

$(3,2):\infty$ $\quad \underbrace{3 \to 0}_{2} \underbrace{\to 2}_{\infty}$ ×

Vertex 1

	0	1	2	3
0	0	3	~~∞~~ 5	7
1	8	0	2	15
2	5	8	0	1
3	2	5	~~∞~~ 7	0

$(0,2):\infty$ $\quad \underbrace{0 \to 1}_{3} \underbrace{\to 2}_{2} = 5$ ✓

$(0,3):7$ $\quad \underbrace{0 \to 1}_{3} \underbrace{\to 3}_{15} = 18$ ×

$(2,0):5$ $\quad \underbrace{2 \to 1}_{8} \underbrace{\to 0}_{8} = 16$ ×

$(2,3):1$ $\quad \underbrace{2 \to 1}_{8} \underbrace{\to 3}_{15}$ ×

$(3,0):2$ $\quad \underbrace{3 \to 1}_{5} \underbrace{\to 0}_{8}$ ×

$(3,2):\infty$ $\quad \underbrace{3 \to 1}_{5} \underbrace{\to 2}_{2} = 7$ ✓

Vertex 2

	0	1	2	3
0	0	3	5	~~7~~ 6
1	~~8~~ 7	0	2	~~15~~ 3
2	5	8	0	1
3	2	5	7	0

$(0,1):3$ $\quad \underbrace{0\to 2}_{5}\underbrace{\to 1}_{8}$ ✗

$(0,3):7$ $\quad \underbrace{0\to 2}_{5}\underbrace{\to 3}_{1} = 6$ ✓

$(1,0):8$ $\quad \underbrace{1\to 2}_{2}\underbrace{\to 0}_{5} : 7$ ✓

$(1,3):15$ $\quad \underbrace{1\to 2}_{2}\underbrace{\to 3}_{1} = 3$ ✓

$(3,0):2$ $\quad \underbrace{3\to 2}_{7}\underbrace{\to 0}_{5}$ ✗

$(3,1):5$ $\quad \underbrace{3\to 2}_{7}\underbrace{\to 1}_{8}$ ✗

Vertex 3

	0	1	2	3
0	0	3	5	6
1	~~7~~ 5	0	2	3
2	~~5~~ 3	~~8~~ 6	0	1
3	2	5	7	0

$(0,1):3$ $\quad \underbrace{0\to 3}_{6}\underbrace{\to 1}_{5}$ ✗

$(0,2):5$ $\quad \underbrace{0\to 3}_{6}\underbrace{\to 2}_{7}$ ✗

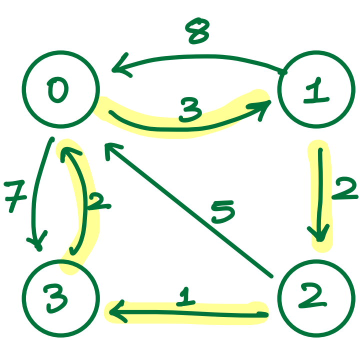
$(1,0):7$ $\quad \underbrace{1\to 3}_{3}\underbrace{\to 0}_{2} = 5$ ✓

$(1,2):2$ $\quad \underbrace{1\to 3}_{3}\underbrace{\to 2}_{7}$ ✗

$(2,0):5$ $\quad \underbrace{2\to 3}_{1}\underbrace{\to 0}_{2} = 3$ ✓

$(2,1):8$ $\quad \underbrace{2\to 3}_{1}\underbrace{\to 1}_{5} = 6$ ✓

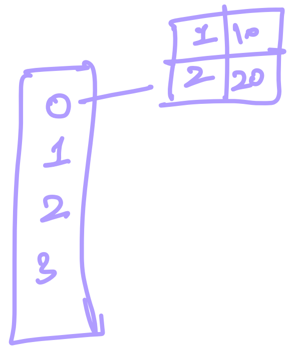
	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0



FLOYD WARSHALL CODE:

```
void floydWarshall()
{
    int cost[V][V] ;

    for(int i= 0 ; i < V ; i++)
    {
        for(int j = 0 ; j < V ; j++)
        {
            if(i == j)
                cost[i][j] = 0 ;
            else
                cost[i][j] = 100000 ;
        }
    }
```

0, ∞




```
    for(int i = 0 ; i < V ; i++)
    {
        map<int, int>::iterator itr ;
        for(itr = strg[i].begin() ; itr != strg[i].end() ; itr++)
            cost[i][itr->first] = itr->second ;
    }
```

edges wt

```
    for(int k = 0 ; k < V ; k++)
```

all vertex

```
    {
        for(int i= 0 ; i < V ; i++)
        {
            for(int j = 0 ; j < V ; j++)
            {
                int oc = cost[i][j] ;
                int nc = cost[i][k] + cost[k][j] ;

                if(nc < oc)
                    cost[i][j] = nc ;
            }
        }

    }
```

(i, j)

i → k → j

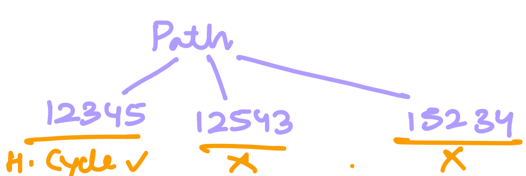
$O(V^3)$

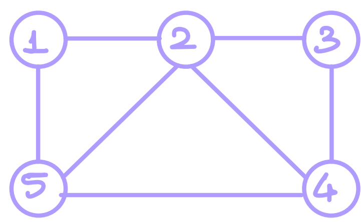
```
    for(int i= 0 ; i < V ; i++)
    {
        for(int j = 0 ; j < V ; j++)
        {
            cout << cost[i][j] << " ";
        }
        cout << endl ;
    }

}
```

TRAVELLING SALESMAN PROBLEM (TSP):

HAMILTONIAN GRAPH:

A Hamiltonian Path in an undirected graph is a path that visits every vertex exactly once.



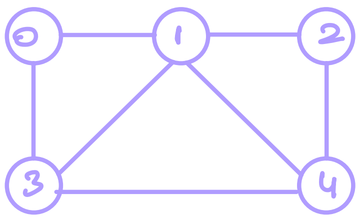


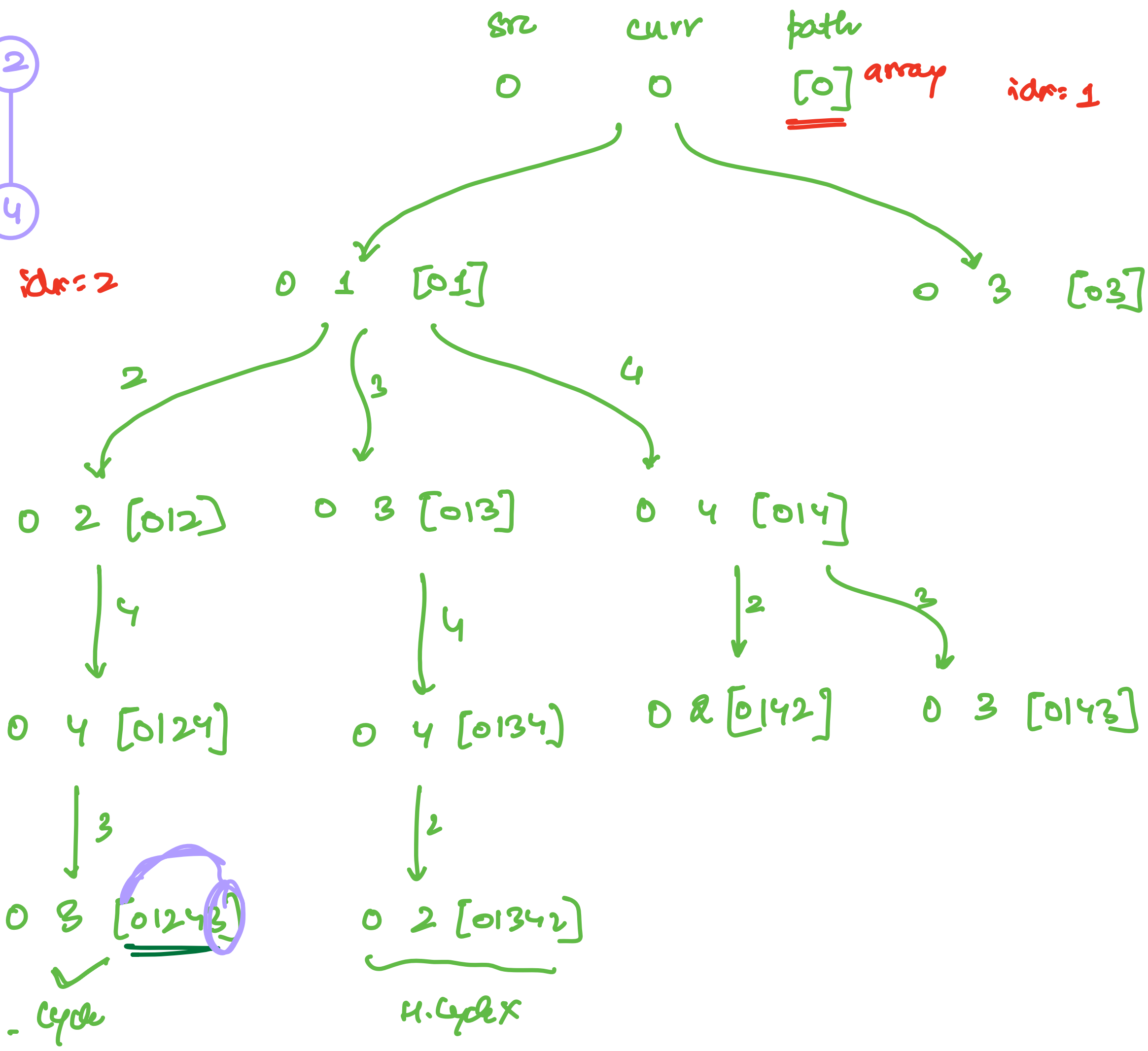
A Hamiltonian Cycle (or Hamiltonian Circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path.

If graph contains a Hamiltonian Cycle, it is called Hamiltonian graph otherwise it is non-Hamiltonian.

Task: Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path.

Print all Hamiltonian Cycle:





PRINT ALL HAMILTONIAN CYCLE CODE

```
#include<iostream>
#include<map>
#include<queue>
#include<vector>

using namespace std ;

class Graph
{
    map<int, map<int,int> > strg ;
    int V ;

    public :

    Graph(int V)
    {
        this->V = V ;
    }

    void addEdge(int u, int v, int cost)
    {
        strg[u][v] = cost ;
        strg[v][u] = cost ;
    }

    void display()
    {
        for(int i = 0 ; i < V ; i++)
        {
            cout << i << "\t" ;

            map<int, int>::iterator itr ;

            for(itr = strg[i].begin() ; itr != strg[i].end() ; itr++)
                cout << itr->first << "@" << itr->second << ", " ;
            cout << endl ;
        }
    }

    bool isItSafe(int *path, int nbr)
    {
        for(int i = 0 ; i < V ; i++)
        {
            if(path[i] == nbr)
                return false ;
        }

        return true ;
    }
    void hamiltonainCycle(int src, int curr, int *path, int idx)
    {
        if(idx == V)
        {
            if(strg[curr].count(src) != 0)
            {
                for(int i = 0 ; i < V ; i++)
                    cout << path[i] << " " ;
                cout << endl ;
            }
            return ;
        }

        map<int, int>::iterator itr ;
        for(itr = strg[curr].begin() ; itr != strg[curr].end() ; itr++)
        {
            int nbr = itr->first ;

            if(isItSafe(path,nbr))
            {
                path[idx] = nbr ;
                hamiltonainCycle(src, nbr, path, idx+1) ;
                path[idx] = -1 ;
            }
        }
    }
} ;
```

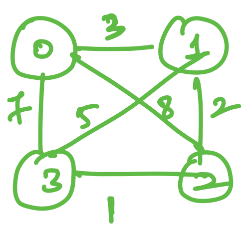
K	V
10	1
20	2

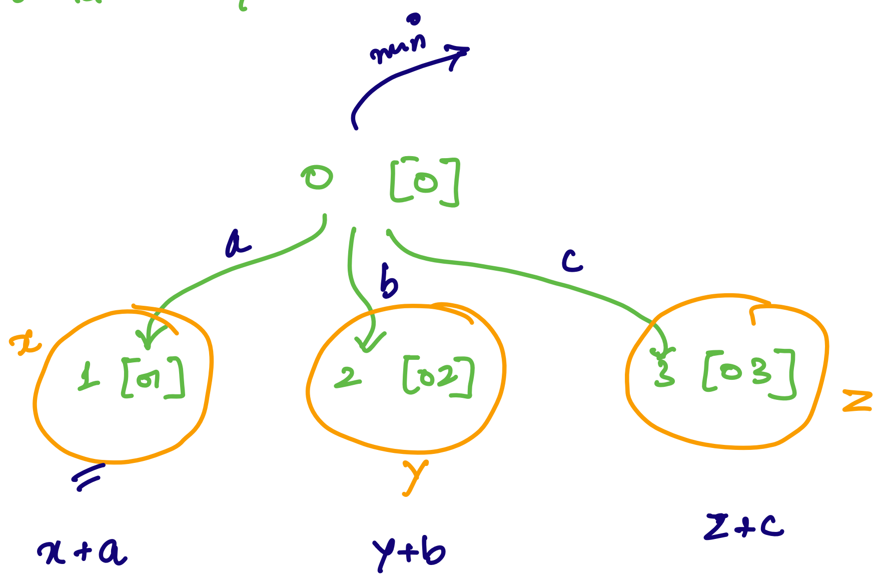
count(10) → 1
count(30) → 0

```cpp
int main()
{
    int n = 5 ;
    Graph g(n) ;

    g.addEdge(0,1,3) ;
    g.addEdge(0,3,7) ;
    g.addEdge(1,2,2) ;
    g.addEdge(1,3,5) ;
    g.addEdge(1,4,1) ;
    g.addEdge(3,4,2) ;
    g.addEdge(2,4,2) ;

    g.display() ;
    int path[n] ;
    for(int i = 0 ; i < n ; i++)
        path[i] = -1 ;
    path[0] = 0 ;
    g.hamiltonainCycle(0,0,path,1) ;

    return 0 ;
}
```

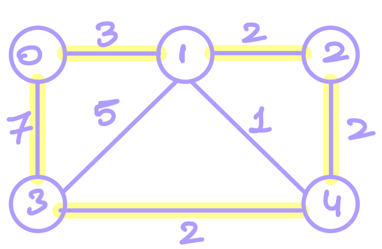
Travelling Salesman Problem (TSP)

- weighted graph
- min cost hamiltonian cycle





min (x+a, y+b, z+c)



TSP Cost : 7+2+2+2+3 = 16

TSP CODE

```
int tsp(int src, int curr, int *path, int idx)
{
    if(idx == V)
    {
        if(strg[curr].count(src) != 0)
            return strg[curr][src] ;
        else
            return 100000;
    }

    int ans = 100000 ;
    map<int, int>::iterator itr ;
    for(itr = strg[curr].begin() ; itr != strg[curr].end() ; itr++)
    {
        int nbr = itr->first ;

        if(isItSafe(path,nbr))
        {
            path[idx] = nbr ;

            int rr = tsp(src, nbr, path, idx+1) ;
            ans = min(ans, rr+strg[curr][nbr]) ;

            path[idx] = -1 ;
        }
    }

    return ans ;
}
```