Uva 11374

Airport Express

Time: 1 second

Problem Descriptions (1/2)

- In a small city called Iokh, a train service, Airport-Express, takes residents to the airport more quickly than other transports.
- There are two types of trains in Airport-Express, the Economy-Xpress and the Commercial-Xpress.
- They travel at different speeds, take different routes and have different costs. Jason is going to the airport to meet his friend. He wants to take the Commercial-Xpress which is supposed to be faster, but he doesn't have enough money.

Problem Descriptions (2/2)

- Luckily he has a ticket for the Commercial-Xpress which can take him one station forward. If he used the ticket wisely, he might end up saving a lot of time. However, choosing the best time to use the ticket is not easy for him. Jason now seeks your help. The routes of the two types of trains are given. Please write a program to find the best route to the destination. The program should also tell when the ticket should be used.

Input (1/2)

- The input consists of several test cases. Consecutive cases are separated by a blank line.
- The first line of each case contains 3 integers, namely N, S and E ($2 \le N \le 500$, $1 \le S, E \le N$), which represent the number of stations, the starting point and where the airport is located respectively.
- There is an integer M ($1 \le M \le 1000$) representing the number of connections between the stations of the Economy-Xpress. The next M lines give the information of the routes of the Economy-Xpress. Each consists of three integers X, Y and Z ($X, Y \le N$, $1 \le Z \le 100$).

Input (2/2)

- This means X and Y are connected and it takes Z minutes to travel between these two stations.
- The next line is another integer K (1 ≤ K ≤ 1000) representing the number of connections between the stations of the Commercial-Xpress. The next K lines contain the information of the Commercial Xpress in the same format as that of the Economy-Xpress.
- All connections are bi-directional. You may assume that there is exactly one optimal route to the airport. There might be cases where you MUST use your ticket in order to reach the airport.

Output

- For each case, you should first list the number of stations which Jason would visit in order.
- On the next line, output 'Ticket Not Used' if you decided NOT to use the ticket; otherwise, state the station where Jason should get on the train of Commercial-Xpress.
- Finally, print the total time for the journey on the last line.
- Consecutive sets of output must be separated by a blank line.

Sample I/O

N S E

Number of stations

```
4 1 4
4
1 2 2
1 3 3
2 4 4
3 4 5
1
2 4 3
```

No. of airport

No. of starting point

M: number of E-train

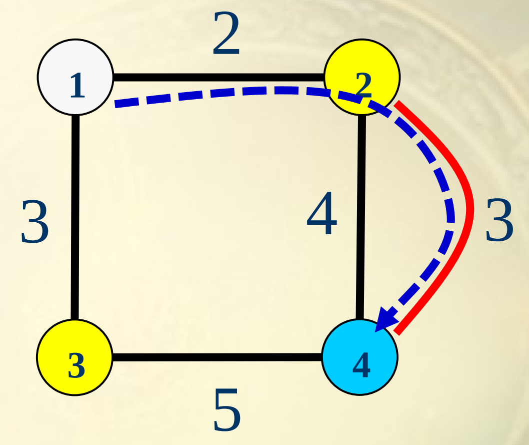
K: number of C-train

```
1 2 4
2
5
```

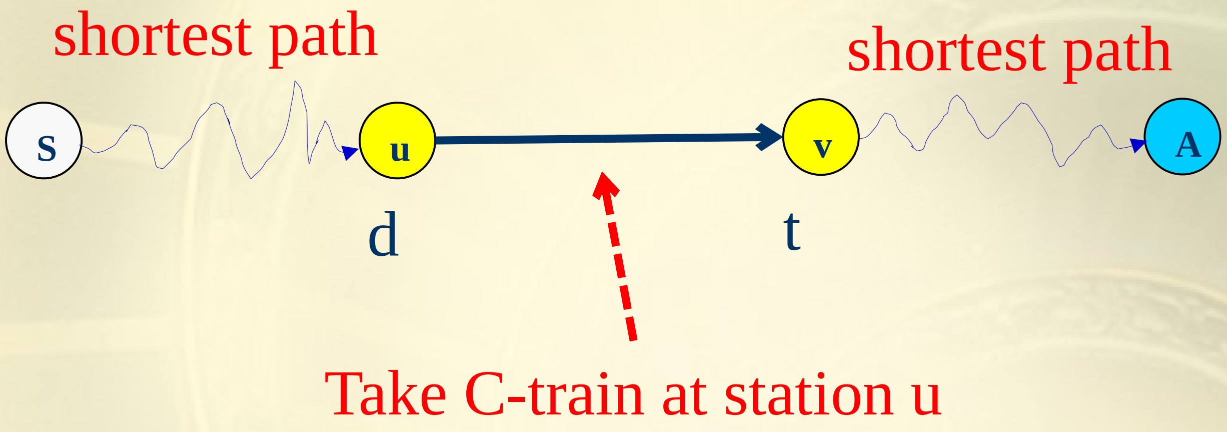
Route

Take C-Train

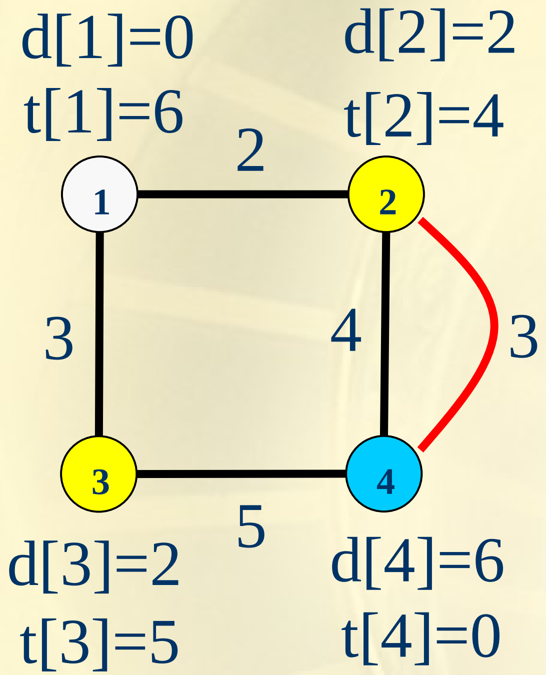
Total cost

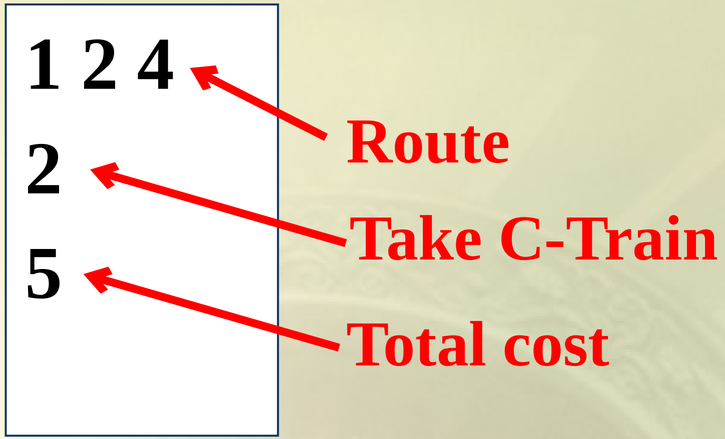


Solution


shortest path
shortest path
s
u
v
A
d
t
Take C-train at station u

Sample I/O





1 2 4
2
5

d[4]=6

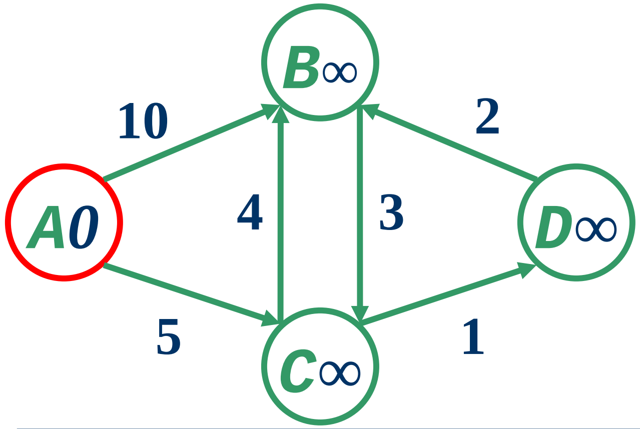
(2 to 4): 3

d[2]+(2 to 4)+t[4]=5<d[4]

So 5

Dijkstra’s Algorithm

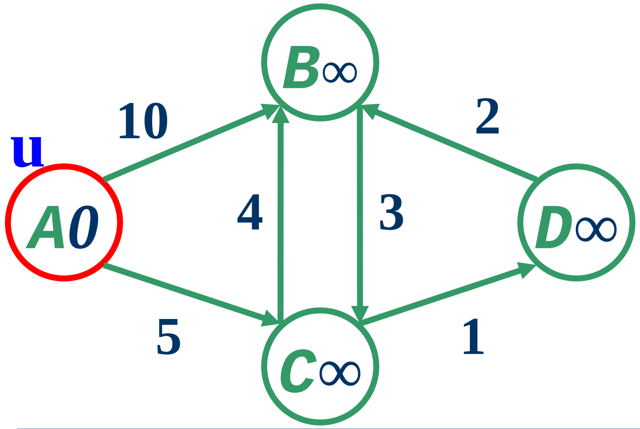
```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q    )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```



```
S =
Q = A, B, C, D
```

Dijkstra’s Algorithm

```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q   )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

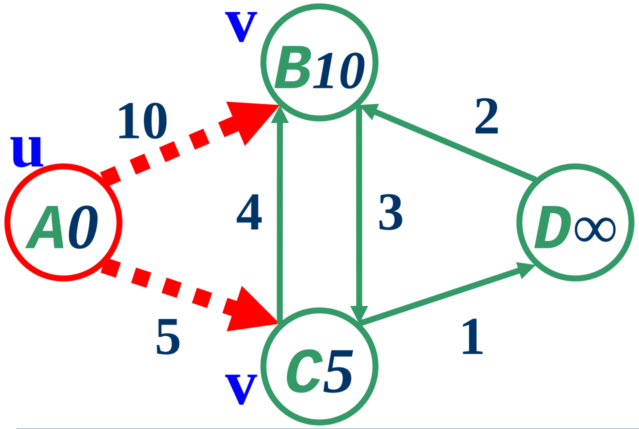


S = A

Q = B, C, D

Dijkstra’s Algorithm

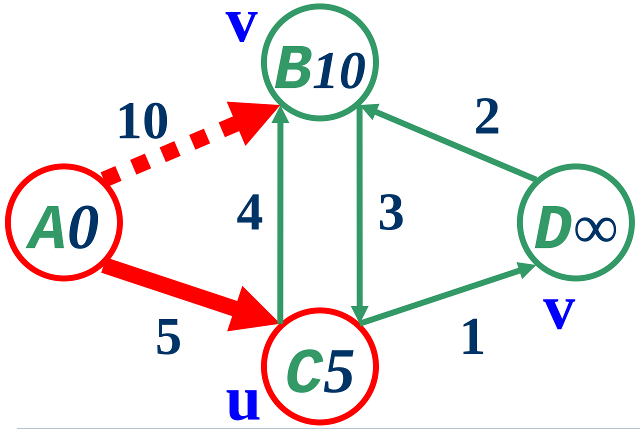
```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q    )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
               d[v]=d[u]+w(u,v);
```



S = A
Q = B, C, D

Dijkstra's Algorithm

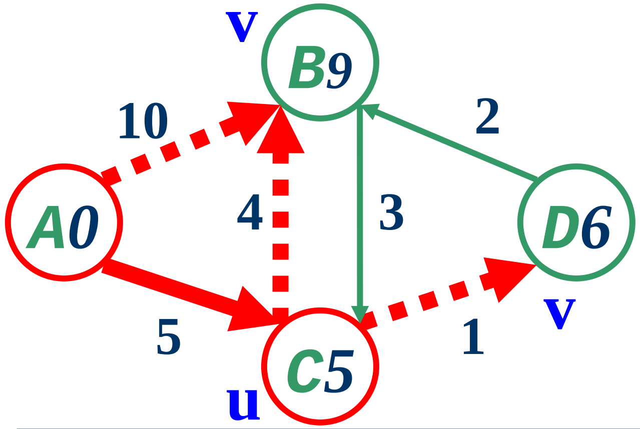
```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q   )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
               d[v]=d[u]+w(u,v);
```



S = A,C
Q = B,D

Dijkstra's Algorithm

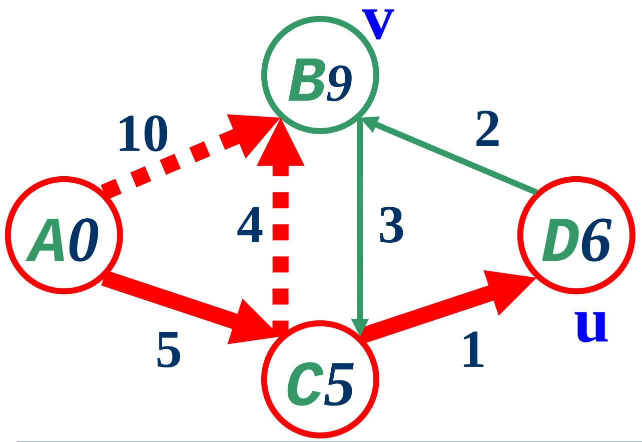
```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q    )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
               d[v]=d[u]+w(u,v);
```



S = A,C
Q = B,D

Dijkstra's Algorithm

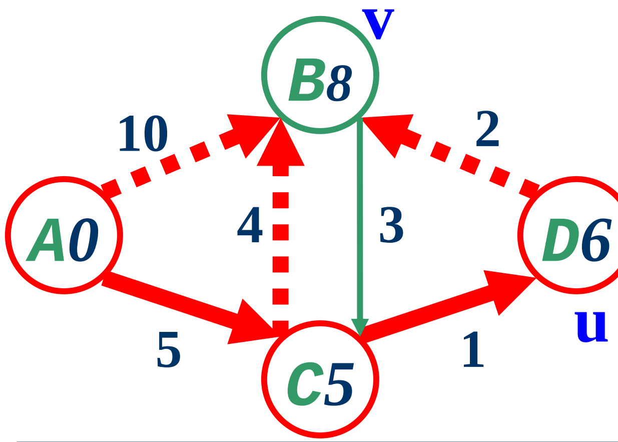
```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q   )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
               d[v]=d[u]+w(u,v);
```



S = A,C,D
Q = B

Dijkstra's Algorithm

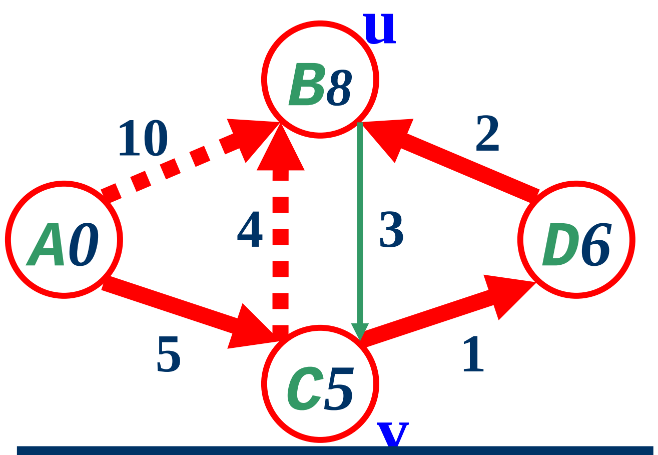
```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q   )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
               d[v]=d[u]+w(u,v);
```



S = A,C,D
Q = B

Dijkstra’s Algorithm

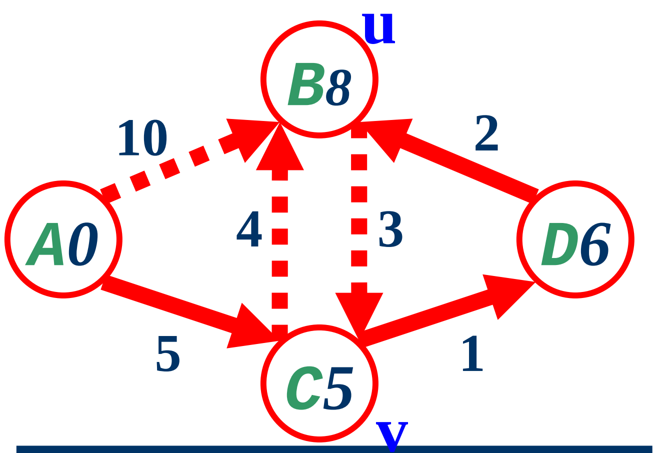
```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q    )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
               d[v]=d[u]+w(u,v);
```



S = A,C,D,B
Q =

Dijkstra's Algorithm

```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q   )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
               d[v]=d[u]+w(u,v);
```



```
S = A,C,D,B
Q =
```

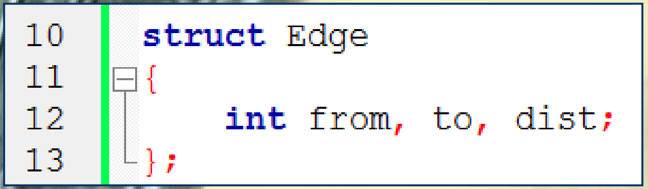
Dijkstra’s Algorithm

```
Dijkstra(G)
   for each v   V
      d[v] =  ;
   d[s] = 0; S =  ; Q = V;
   while (Q   )
      u = ExtractMin(Q);
      S = S U {u};
      for each v   u->Adj[]
         if (d[v] > d[u]+w(u,v))
               d[v]=d[u]+w(u,v);
```

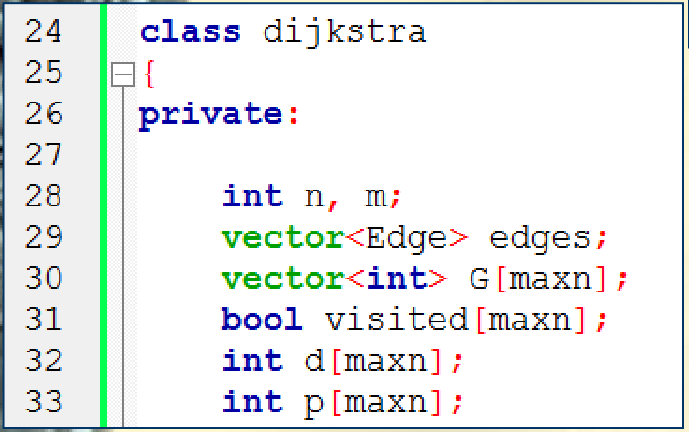
How many times is ExtractMin() called?

How many times is Relax called?

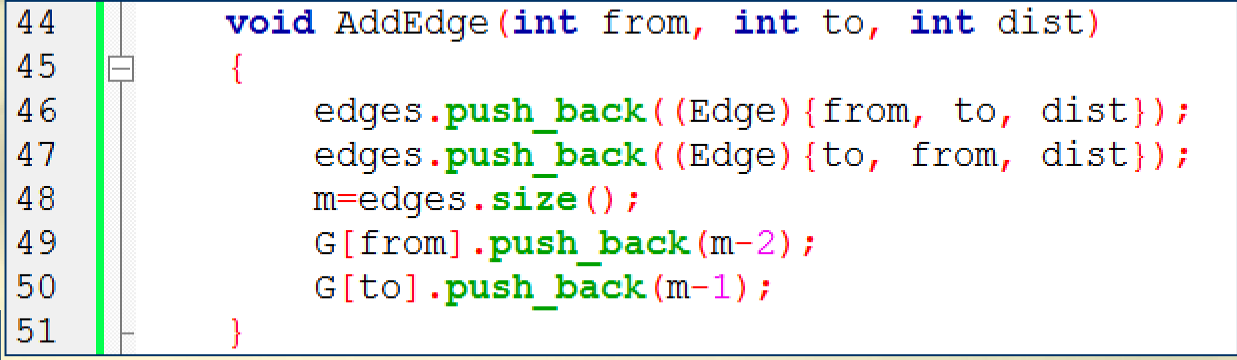
A:O(V lg V + E)



```
struct Edge
{
    int from, to, dist;
};
```

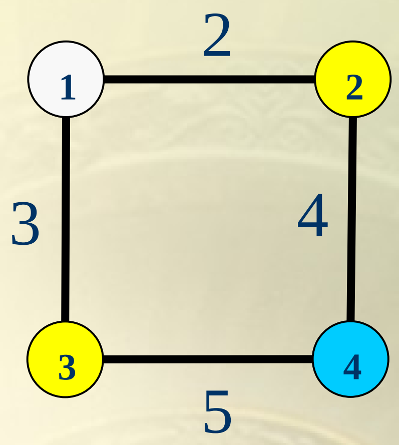


```
class dijkstra
{
private:

    int n, m;
    vector<Edge> edges;
    vector<int> G[maxn];
    bool visited[maxn];
    int d[maxn];
    int p[maxn];
```



```
    void AddEdge(int from, int to, int dist)
    {
        edges.push_back((Edge){from, to, dist});
        edges.push_back((Edge){to, from, dist});
        m=edges.size();
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }
```

```
1 2 2
1 3 3
2 4 4
3 4 5
```

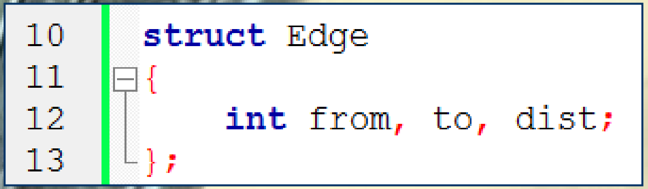


edges

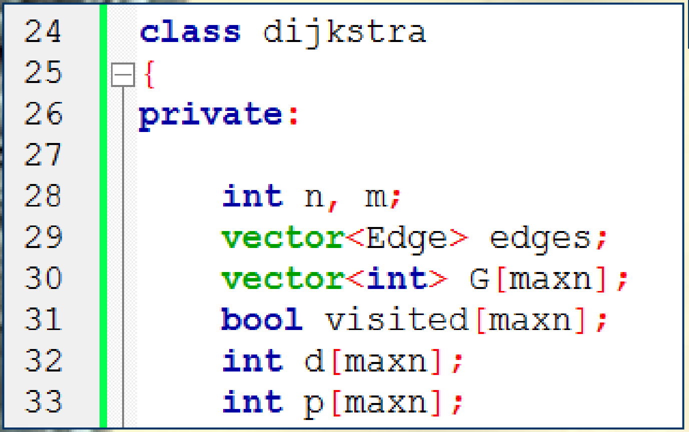
0	1								
(1,2,2)	(2,1,2)								

G

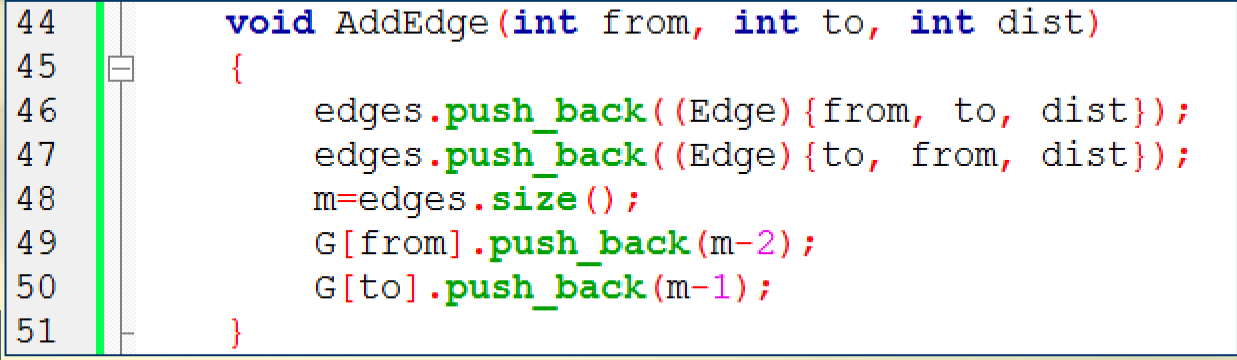
1 →	0		
2 →	1		
3 →			
4 →			



```cpp
struct Edge
{
    int from, to, dist;
};
```

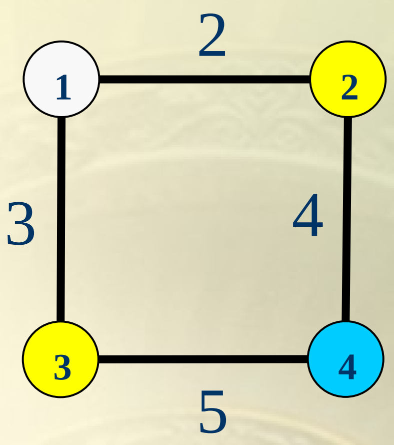


```cpp
class dijkstra
{
private:

    int n, m;
    vector<Edge> edges;
    vector<int> G[maxn];
    bool visited[maxn];
    int d[maxn];
    int p[maxn];
```



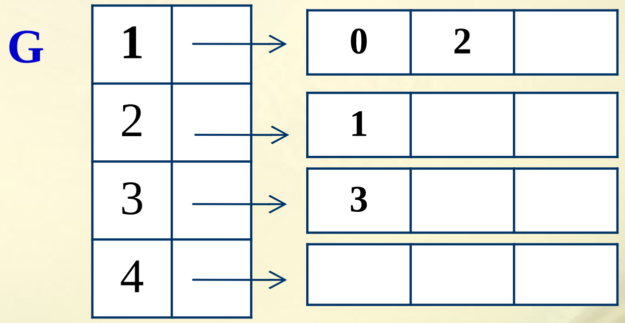
```cpp
    void AddEdge(int from, int to, int dist)
    {
        edges.push_back((Edge){from, to, dist});
        edges.push_back((Edge){to, from, dist});
        m=edges.size();
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }
```

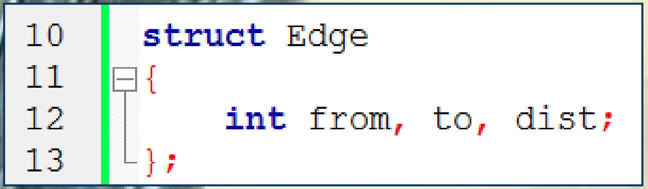
1	2	2
1	3	3
2	4	4
3	4	5



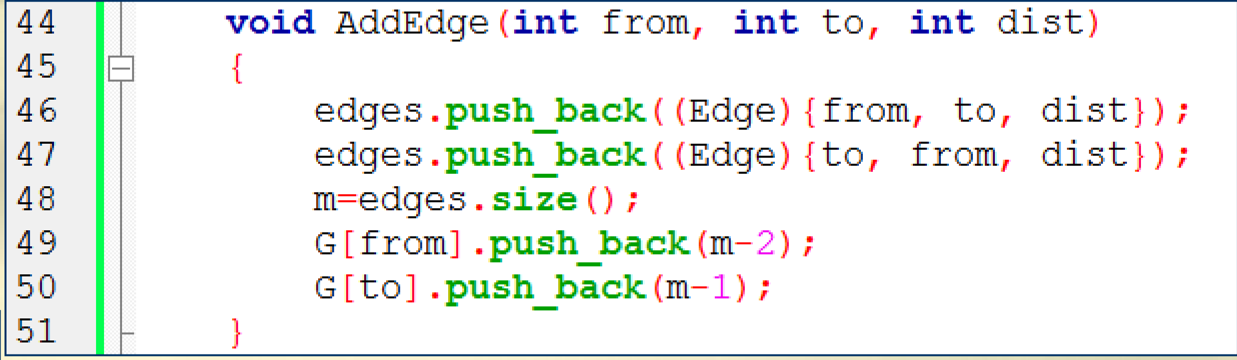
edges

0	1	2	3						
(1,2,2)	(2,1,2)	(1,3,3)	(3,1,3)						

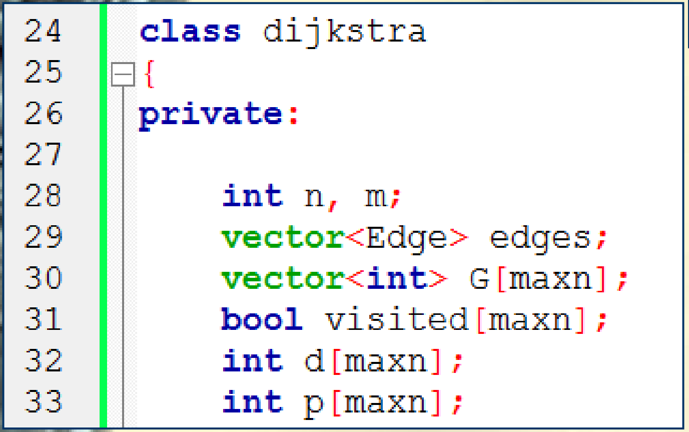




```
struct Edge
{
    int from, to, dist;
};
```

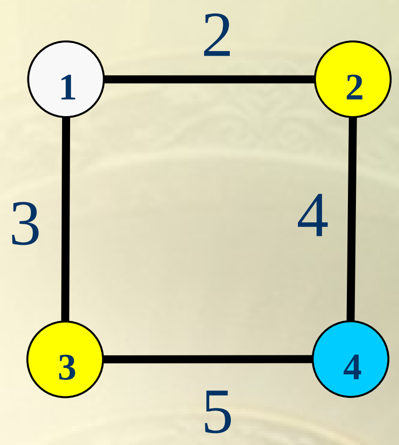


```
    void AddEdge(int from, int to, int dist)
    {
        edges.push_back((Edge){from, to, dist});
        edges.push_back((Edge){to, from, dist});
        m=edges.size();
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }
```



```
class dijkstra
{
private:

    int n, m;
    vector<Edge> edges;
    vector<int> G[maxn];
    bool visited[maxn];
    int d[maxn];
    int p[maxn];
```

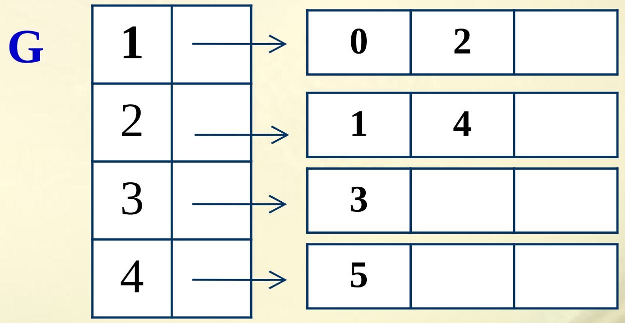
1 2 2
1 3 3
2 4 4
3 4 5



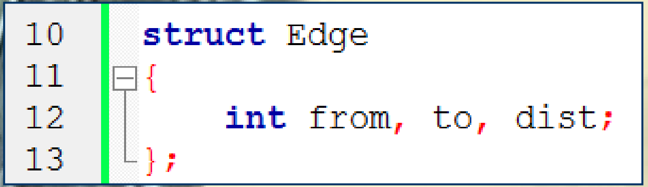
edges

0	1	2	3	4	5				
(1,2,2)	(2,1,2)	(1,3,3)	(3,1,3)	(2,4,4)	(4,2,4)				

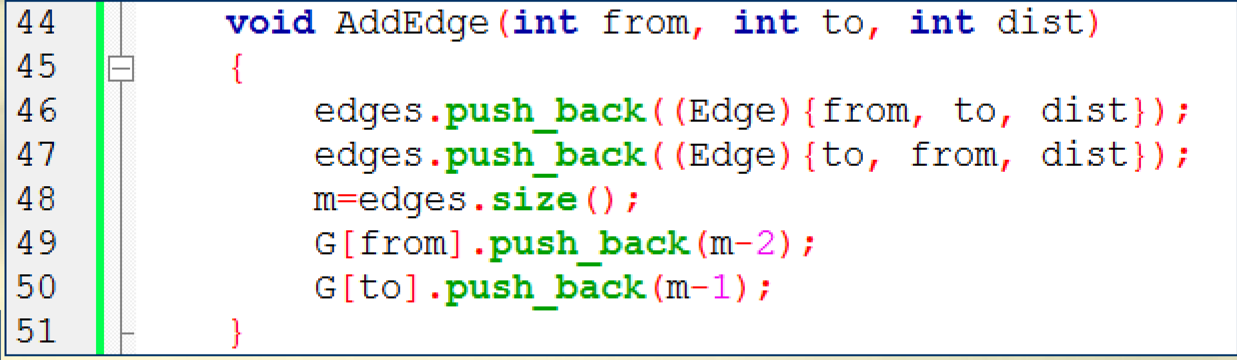
G



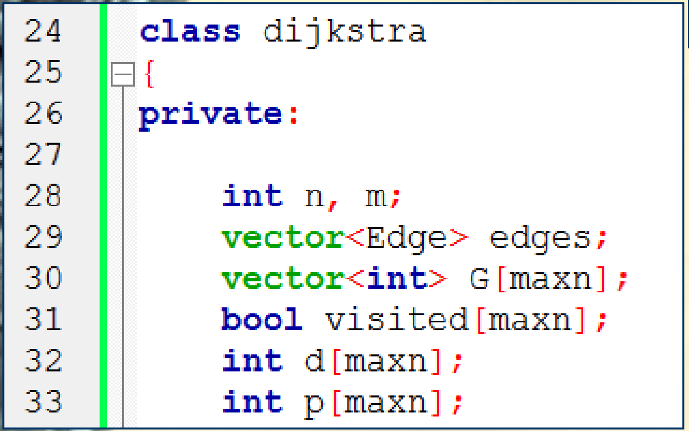
1	0	2	
2	1	4	
3	3		
4	5		



```
struct Edge
{
    int from, to, dist;
};
```

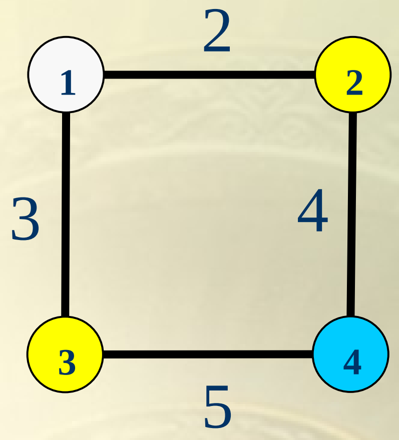


```
    void AddEdge(int from, int to, int dist)
    {
        edges.push_back((Edge){from, to, dist});
        edges.push_back((Edge){to, from, dist});
        m=edges.size();
        G[from].push_back(m-2);
        G[to].push_back(m-1);
    }
```



```
class dijkstra
{
private:

    int n, m;
    vector<Edge> edges;
    vector<int> G[maxn];
    bool visited[maxn];
    int d[maxn];
    int p[maxn];
```

```
1 2 2
1 3 3
2 4 4
3 4 5
```



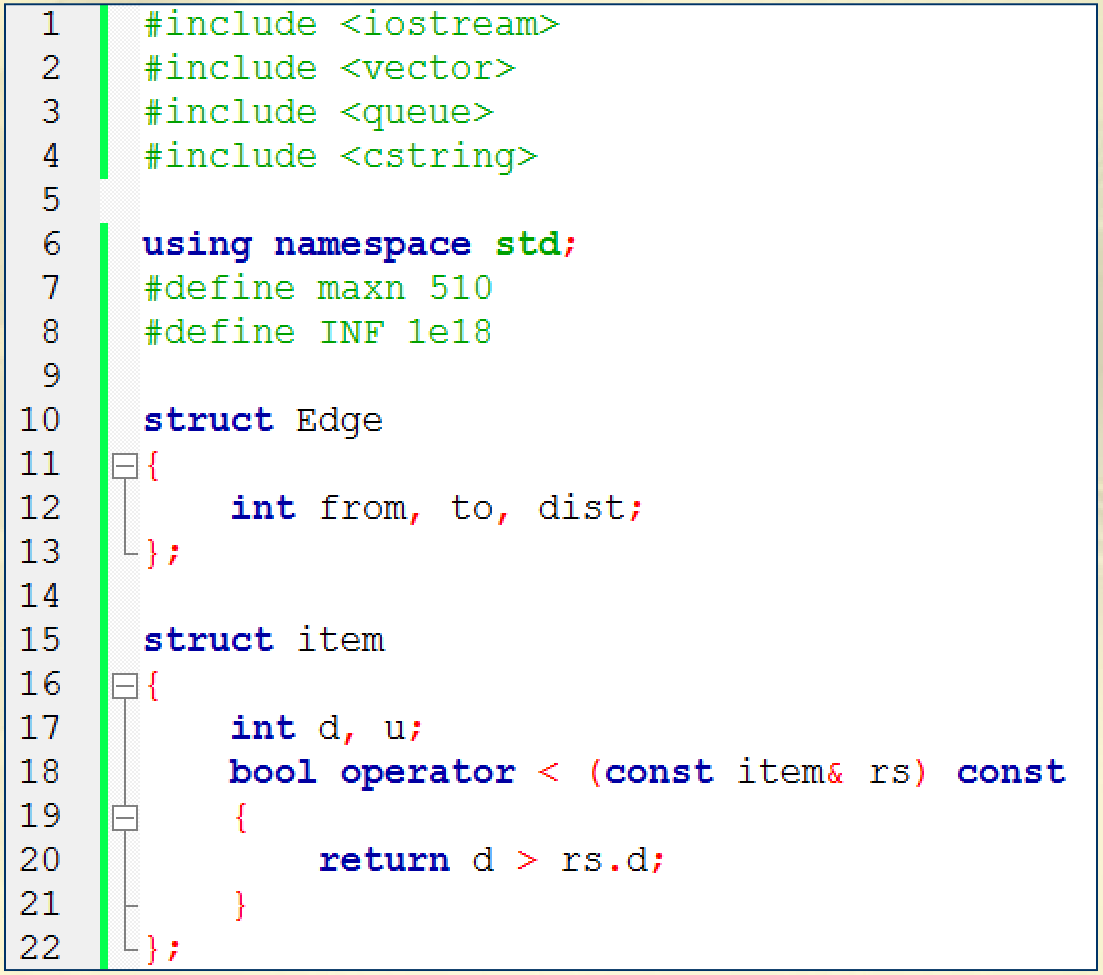
edges

0	1	2	3	4	5	6	7		
(1,2,2)	(2,1,2)	(1,3,3)	(3,1,3)	(2,4,4)	(4,2,4)	(3,4,5)	(4,3,5)		

G

1	→	0	2	
2	→	1	4	
3	→	3	6	
4	→	5	7	

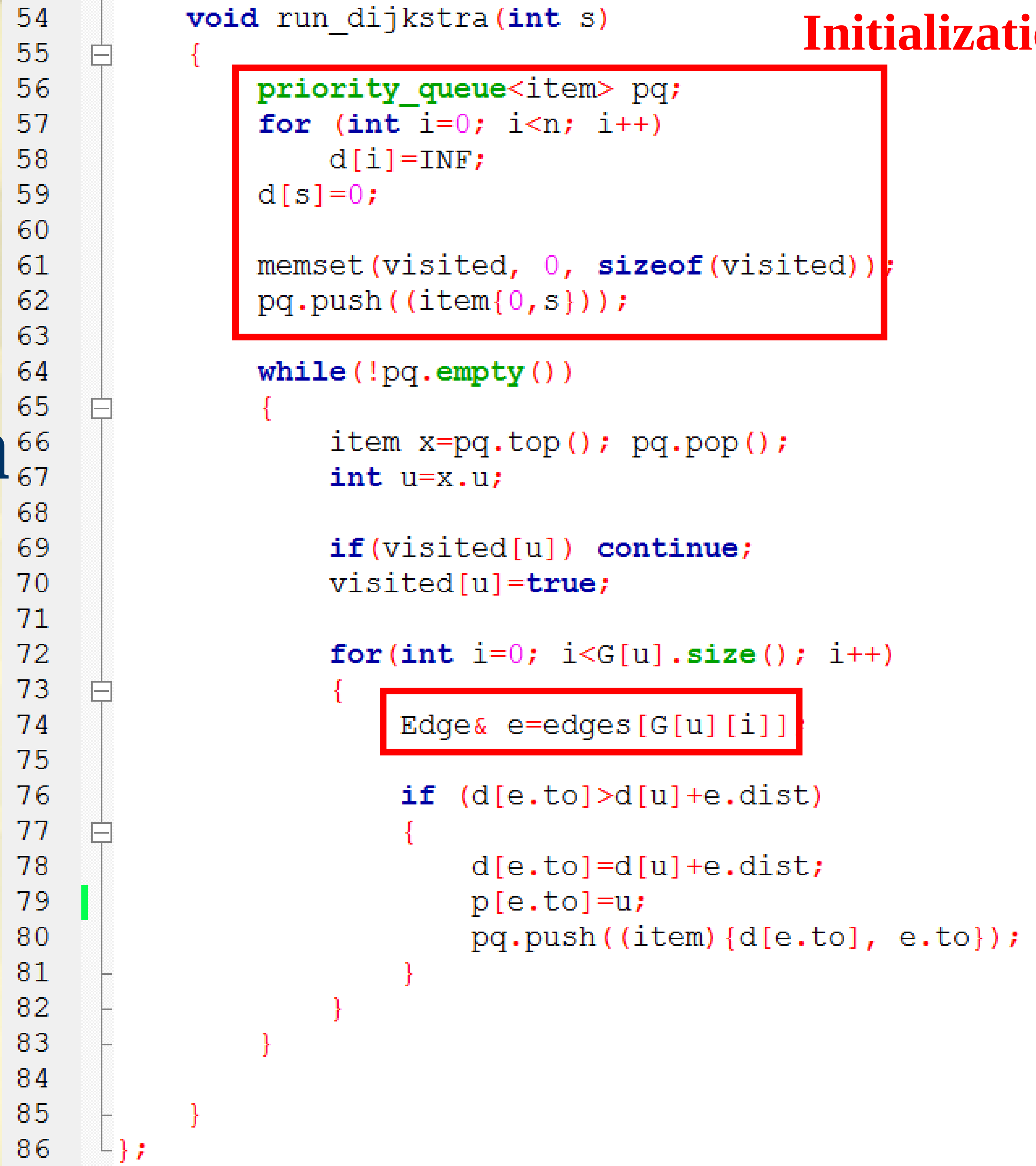
Data Structure



```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>

using namespace std;
#define maxn 510
#define INF 1e18

struct Edge
{
    int from, to, dist;
};

struct item
{
    int d, u;
    bool operator < (const item& rs) const
    {
        return d > rs.d;
    }
};
```

Dijkstra

Initialization



```
    void run_dijkstra(int s)
    {
        priority_queue<item> pq;
        for (int i=0; i<n; i++)
            d[i]=INF;
        d[s]=0;

        memset(visited, 0, sizeof(visited));
        pq.push((item{0,s}));

        while(!pq.empty())
        {
            item x=pq.top(); pq.pop();
            int u=x.u;

            if(visited[u]) continue;
            visited[u]=true;

            for(int i=0; i<G[u].size(); i++)
            {
                Edge& e=edges[G[u][i]];

                if (d[e.to]>d[u]+e.dist)
                {
                    d[e.to]=d[u]+e.dist;
                    p[e.to]=u;
                    pq.push((item){d[e.to], e.to});
                }
            }
        }

    }
};
```