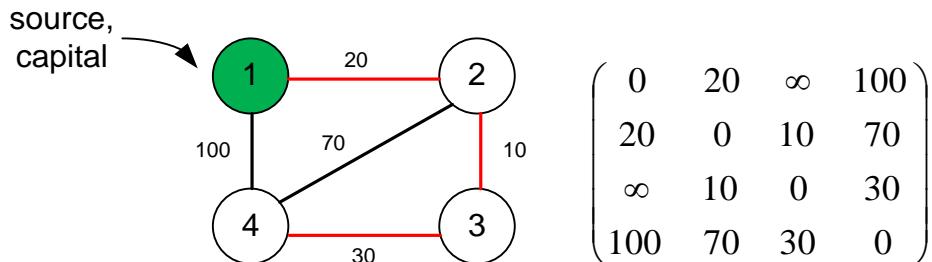


Dijkstra's algorithm and its implementation

Problem. Assume the country G, where there is a lot of cities (let's denote this set as V), and a lot of roads connecting pairs of cities (let's denote them as E). Not the fact that each pair of cities is connected by a road. Sometimes, to get from one city to another, you should visit some other cities. The roads have length. There is a capital s in the country G. You must find the shortest path from the capital to other cities.



Look at the graph. All its edges are weighted – they contains some number. For example it can be the distance between the cities. The corresponding weighted matrix is given on the picture at the right. Consider some values of the matrix:

- $g[1][4] = 100$ means that distance from city 1 to city 4 is 100.
- $g[1][4] = g[4][1]$ means that there is a two-way road between cities 1 and 4.
- $g[i][i] = 0$ for any vertex i means that distance from city i to city i is 0.
- $g[1][3] = \infty$ means that there is no direct way from 1 to 3.

For example, the shortest path from 1 to 4 equals to $20 + 10 + 30 = 60$, which is less than the length of the direct road.

Sometimes we can set $g[i][j] = -1$ (instead of ∞) if there is no edge between i and j .

The mathematical formulation of this problem:

Let $G = (V, E)$ be a directed graph, each its edge is marked with non-negative number (weight of the edge). Let's denote some vertex s as a **source**. You must find the shortest path from the source s to all other vertices of G .

This problem has name “***Find the shortest paths from a single source***”.

If we declare $\text{dist}[v]$ to be the length of the shortest path from **source** to v , then

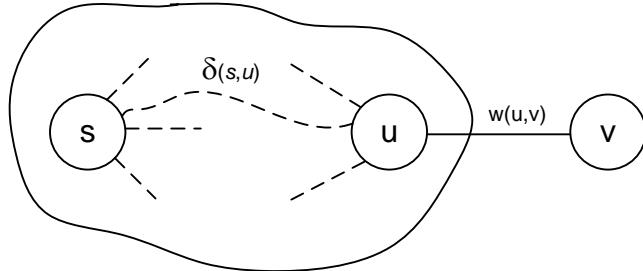
$$\text{dist}[1] = 0, \text{dist}[2] = 20, \text{dist}[3] = 30, \text{dist}[4] = 60$$

Any part of the shortest path is itself a shortest path. This allows you to solve this problem with the implementation of ***dynamic programming***.

Lemma. Let $G = (V, E)$ be a weighted directed graph. If $p = (v_1, v_2, \dots, v_k)$ is the shortest path from v_1 to v_k and $1 \leq i \leq j \leq k$, then $p_{ij} = (v_i, v_{i+1}, \dots, v_j)$ is the shortest path from v_i to v_j .

Let $\delta(s, v)$ be the length of the shortest path between vertices s and v . The weight of an edge between vertices u and v will be denoted by $w(u, v)$. Then if $u \rightarrow v$ is the last edge of the shortest path from s to v , then

$$\delta(s, v) = \delta(s, u) + w(u, v)$$



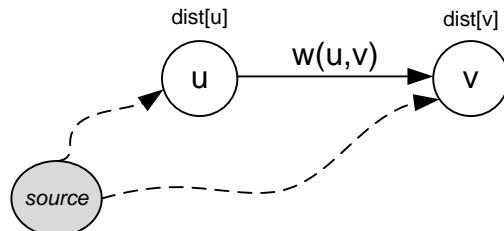
Dijkstra algorithm solves the problem of the shortest path from one source to others. It is greedy algorithm.

Dijkstra algorithm uses next arrays:

- `int g[101][101]` – the weighted matrix;
- `int used[101], used[v] = 1` if the shortest distance from *source* to v is already found;
- `int dist[101], dist[v]` contains the shortest distance from *source* to v ;

Edge relaxation

Let $u \rightarrow v$ be an edge of weight $w(u, v)$. Let $\text{dist}[u]$ and $\text{dist}[v]$ are current shortest distances from *source* to the vertices u and v correspondingly.

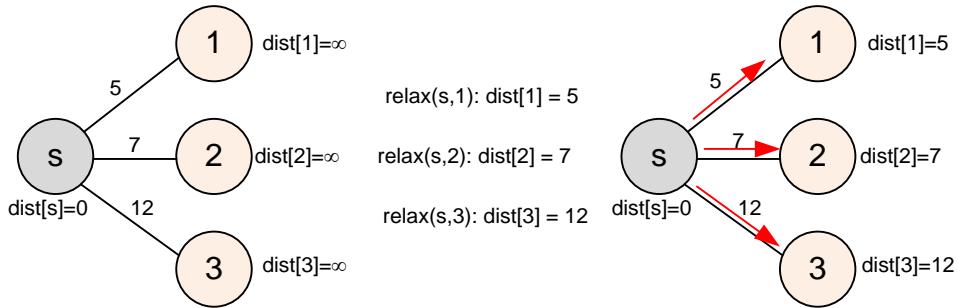


Current shortest distance from *source* to v is $\text{dist}[v]$. But what is we shall go to v through vertex u and along the edge $u \rightarrow v$? The cost of this path is $\text{dist}[u] + w(u, v)$. If this value is less then the value $\text{dist}[v]$ (we are looking for the shortest path), we must update $\text{dist}[v]$:

```
if (dist[u] + g[u][v] < dist[v]) dist[v] = dist[u] + g[u][v];
```

If above condition takes place, we say that edge $u \rightarrow v$ relaxes.

When **Dijkstra** algorithm starts, all values $\text{dist}[v]$ are set to ∞ (only $\text{dist}[\text{source}] = 0$). $\text{dist}[v] = \infty$ means that current shortest distance from *source* to v is **infinity**. Consider the next sample – relaxation of the edges outgoing from the *source*:



Consider an edge $s \rightarrow 1$: $\text{dist}[s] = 0$, $\text{dist}[1] = \infty$. We have the relation:

$$\text{dist}[s] + w(s, 1) < \text{dist}[1], \\ 0 + 5 < \infty,$$

so edge $s \rightarrow 1$ relaxes, and $\text{dist}[1] = \text{dist}[s] + w(s, 1) = 0 + 5 = 5$.

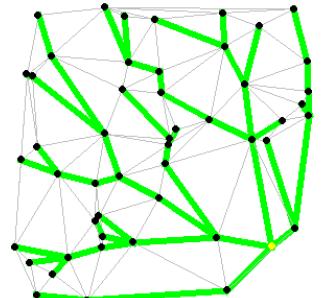
The same way the edges $s \rightarrow 2$ and $s \rightarrow 3$ also relax and we get $\text{dist}[2] = 7$, $\text{dist}[3] = 12$.

Dijkstra algorithm constructs a set of vertices S for which the shortest path from the *source* is known. Initially $S = \{\}$. If vertex $v \in S$, we set $\text{used}[v] = 1$.

Initially $S = \{\}$, so $\text{used}[i] = 0$ for all i ($1 \leq i \leq n$).

If $\text{used}[v] = 1$ for some vertex v , it means that value $\text{dist}[v]$ is already optimal and can't be decreased (improved).

At each step we add to the set S such vertex v for which the distance from the *source* is no more than the distance from the *source* to other vertices from V / S . This is done by finding the minimum among the values of $\text{dist}[v]$ for all $v \in V / S$ (values v which are not in S). This addition of v is just characterizes the principle of a greedy choice. After the addition of v to S the shortest distance from the *source* to v will never be improved, set $\text{used}[v] = 1$.



Since the weights of the edges are non-negative, then the shortest path from the *source* to a particular vertex in S will take place only through the vertices in S . This path we call **special**. At each step of the algorithm there is an array of **dist**, that records the length of the shortest special paths for each vertex. When set S contains all vertices of the graph (for all vertices will be found special way), then array **dist** will contain the length of the shortest paths from the *source* to each vertex.

DIJKSTRA (G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)

$S \leftarrow \emptyset$

$Q \leftarrow V[G]$

while $Q \neq \emptyset$

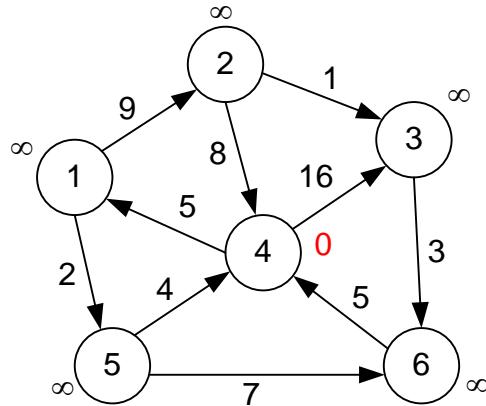
do $u \leftarrow \text{EXTRACT-MIN}(Q)$

$S = S \cup \{u\}$

for each vertex $v \in \text{Adj}[u]$

do $\text{Relax}(u, v, w)$

Consider the graph below. Vertex 4 is the *source*. Initialize $S = \{\}$. For each value of k we set $\text{dist}[k]$ to the maximum positive integer (*infinity* = ∞). Set $\text{dist}[4] = 0$, since the distance from the *source* to itself is 0.

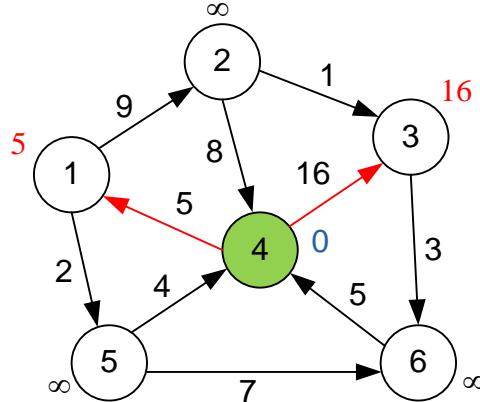


First iteration. We find the smallest $\text{dist}[i]$, where i is the vertex, not included in S .

$$\min \{\text{dist}[1], \text{dist}[2], \text{dist}[3], \text{dist}[5], \text{dist}[6]\} = \text{dist}[4] = 0$$

The first vertex to be included in the set S will be 4: $S = \{4\}$. Relax the edges outgoing from vertex 4:

- $4 \rightarrow 1$: $\text{dist}[1] = \min(\text{dist}[1], \text{dist}[4] + g[4][1]) = \min(\infty, 0 + 5) = 5$;
- $4 \rightarrow 3$: $\text{dist}[3] = \min(\text{dist}[3], \text{dist}[4] + g[4][3]) = \min(\infty, 0 + 16) = 16$;

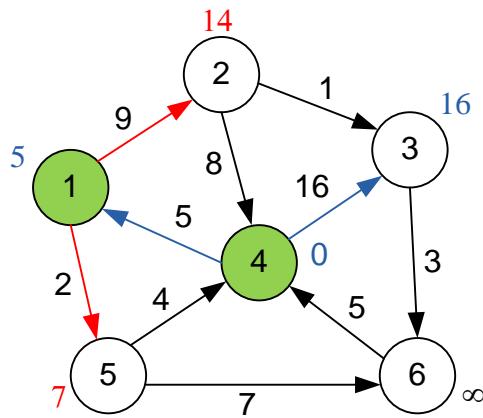


Second iteration. We are looking for the least value among $\text{dist}[i]$, where $i \notin S = \{4\}$:

$$\min \{ \text{dist}[1], \text{dist}[2], \text{dist}[3], \text{dist}[5], \text{dist}[6] \} = \text{dist}[1] = 5$$

In the second step vertex 1 will be included to set S , i.e. $S = \{1, 4\}$. Relax the edges outgoing from vertex 1:

- $1 \rightarrow 2$: $\text{dist}[2] = \min(\text{dist}[2], \text{dist}[1] + g[1][2]) = \min(\infty, 5 + 9) = 14$;
- $1 \rightarrow 5$: $\text{dist}[5] = \min(\text{dist}[5], \text{dist}[1] + g[1][5]) = \min(\infty, 5 + 2) = 7$;



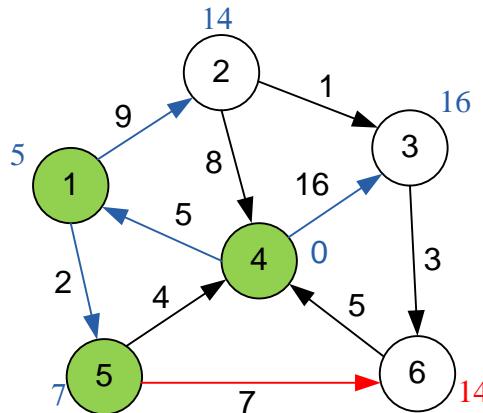
Third iteration. We are looking for the least value among $\text{dist}[i]$, where $i \notin S = \{1, 4\}$:

$$\min\{\text{dist}[2], \text{dist}[3], \text{dist}[5], \text{dist}[6]\} = \text{dist}[5] = 7$$

In the third step vertex 5 will be included to set S , i.e. $S = \{1, 4, 5\}$. Relax the edges outgoing from vertex 5:

- $5 \rightarrow 6$: $\text{dist}[6] = \min(\text{dist}[6], \text{dist}[5] + g[5][6]) = \min(\infty, 7 + 7) = 14$;

We do not consider edge $5 \rightarrow 4$ because vertex 4 is already in S .

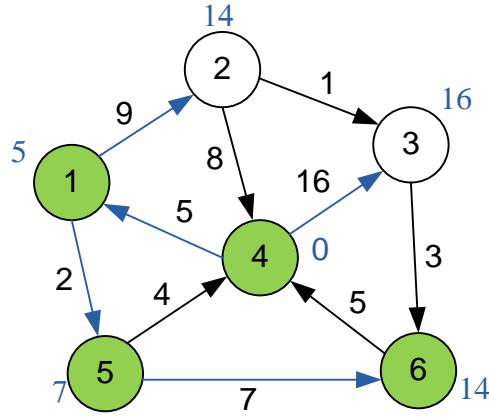


Fourth iteration. We are looking for the least value among $\text{dist}[i]$, where $i \notin S = \{1, 4, 5\}$:

$$\min\{\text{dist}[2], \text{dist}[3], \text{dist}[6]\} = \text{dist}[6] = 14$$

We have two vertices with minimum value of $\text{dist}[i]$: they are 2 and 6 ($\text{dist}[2] = \text{dist}[6] = 14$). We can choose any of two vertices.

In the fourth step vertex 6 will be included to set S , i.e. $S = \{1, 4, 5, 6\}$. Relax the edges outgoing from vertex 6. There is no such edges.



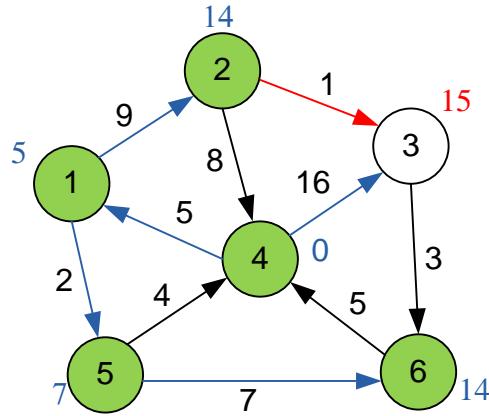
Fifth iteration. We are looking for the least value among $\text{dist}[i]$, where $i \notin S = \{1, 4, 5, 6\}$:

$$\min\{\text{dist}[2], \text{dist}[3]\} = \text{dist}[2] = 14$$

In the fifth step vertex 2 will be included to set S , i.e. $S = \{1, 2, 4, 5, 6\}$. Relax the edges outgoing from vertex 2:

- $2 \rightarrow 3$: $\text{dist}[3] = \min(\text{dist}[3], \text{dist}[2] + g[2][3]) = \min(16, 14 + 1) = 15$;

We do not consider edge $2 \rightarrow 4$ because vertex 4 is already in S .

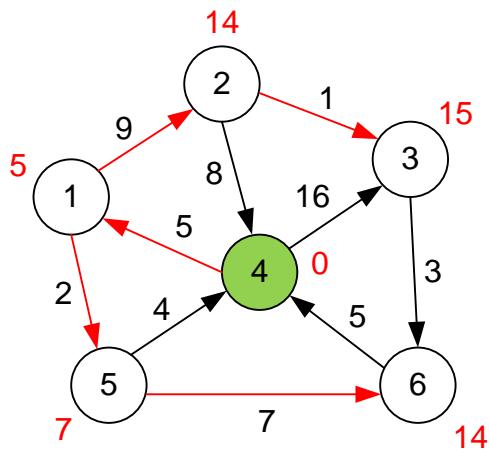


There is no sense to run **sixth iteration**. Vertex 3 will be included to S . But there is no any edge outgoing from 3 that runs into vertex not in S .

The result of all iterations is shown in the table. Vertex v , which is selected and added to the S in each step is highlighted and underlined. The values of $\text{dist}[i]$, for which $i \in S$, are highlighted in *italics*.

Iteration	S	$\text{dist}[1]$	$\text{dist}[2]$	$\text{dist}[3]$	$\text{dist}[4]$	$\text{dist}[5]$	$\text{dist}[6]$
start	{}	∞	∞	∞	<u>0</u>	∞	∞
1	{4}	<u>5</u>	∞	16	0	∞	∞
2	{1, 4}	5	14	16	0	<u>7</u>	∞
3	{1, 4, 5}	5	14	16	0	<u>7</u>	<u>14</u>
4	{1, 4, 5, 6}	5	<u>14</u>	16	0	7	<u>14</u>
5	{1, 2, 4, 5, 6}	5	<u>14</u>	<u>15</u>	0	7	14

The iterative process of Dijkstra's algorithm



E-OLYMP 1365. Dijkstra algorithm You are given a directed weighted graph. Find the shortest distance from vertex s to vertex f .

Input. The first line contains three integers n , s , and f ($1 \leq n \leq 100$, $1 \leq s, f \leq n$), where n is the number of vertices in the graph. Each of the following n lines contains n integers, representing the adjacency matrix of the graph. The integer at the i -th row and j -th column indicates the weight of the edge from vertex i to vertex j . A value of -1 means there is no edge between the vertices, while a non-negative value represents the weight of the edge. The main diagonal of the matrix always contains zeros.

Output. Print the shortest distance from vertex s to vertex f . If there is no path between the two vertices, print -1.

Sample input

```
3 1 2
0 -1 2
3 0 -1
-1 4 0
```

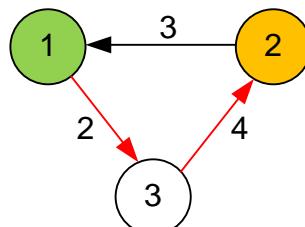
Sample output

```
6
```

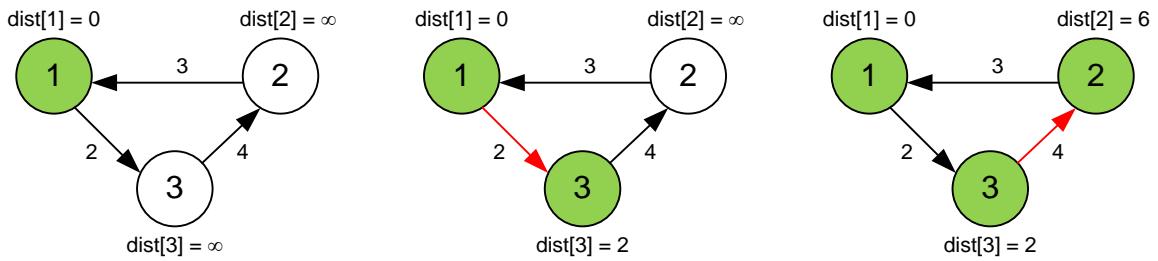
► In this problem, your task is to find the shortest distance between two vertices in a directed weighted graph. To solve it, you need to implement Dijkstra's algorithm.

Example

The graph given in the example looks like this:



The shortest distance from vertex 1 to vertex 2 is $2 + 4 = 6$.



Algorithm realization

Let's declare the constants and arrays we'll use.

```
#define MAX 110
#define INF 0x3F3F3F
int m[MAX][MAX], used[MAX], d[MAX];
```

Read the input data.

```
scanf("%d %d %d", &n, &s, &f);
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
{
    scanf("%d", &m[i][j]);
    if (m[i][j] == -1) m[i][j] = INF;
}
```

Initialize the arrays.

```
memset(used, 0, sizeof(used));
memset(d, 0x3F, sizeof(d));
d[s] = 0;
```

Start the loop for Dijkstra's algorithm. Since the graph contains n vertices, $n - 1$ iterations will be sufficient.

```
for (i = 1; i < n; i++)
{
```

Find the vertex with the smallest value of $d[j]$ among those for which the shortest distance from the source is not calculated (i.e., for which $used[j] = 0$). Let this vertex be w .

```
mind = INF;
for (j = 1; j <= n; j++)
    if (used[j] == 0 && d[j] < mind) { mind = d[j]; w = j; }
```

If it is impossible to find a vertex to include in the set of vertices for which the shortest distance is already calculated, terminate the algorithm.

```
if (mind == INF) break;
```

Relax all the edges outgoing from vertex w .

```
for (j = 1; j <= n; j++)
```

```
if (used[j] == 0) d[j] = min(d[j], d[w] + m[w][j]);
```

Mark that the shortest distance to w is calculated (it is stored in $d[w]$).

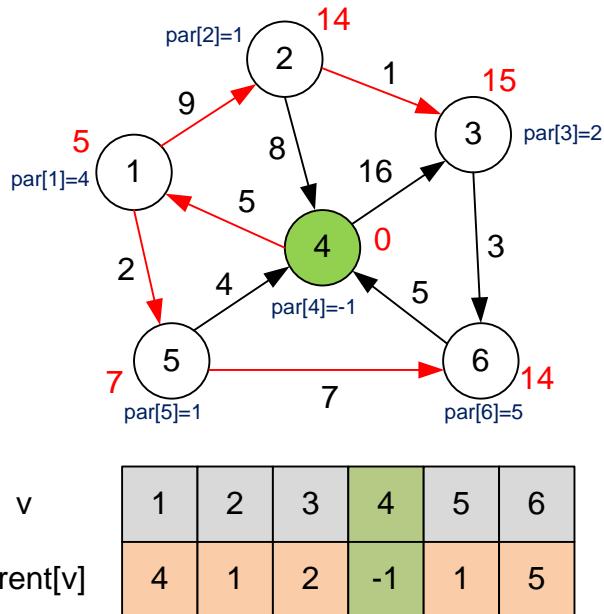
```
used[w] = 1;
}
```

Print the answer – the value of $d[f]$. If it is equal to infinity, then there is no path to vertex f .

```
if (d[f] == INF) d[f] = -1;
printf("%d\n", d[f]);
```

How to restore the shortest path between two vertices? What if we need not only to print the shortest distance between the vertices, but also the path itself? Let's use **parent** array.

Let $\text{parent}[u] = v$ means that after the relaxation of the edge $v \rightarrow u$ the shortest distance $\text{dist}[u]$ becomes optimal.



To find the shortest path from *source* to v , we must move backwards starting from v :

$v, \text{parent}[v], \text{parent}[\text{parent}[v]], \dots, \text{source}, -1$

For example, the shortest path from 4 to 6 can be found next way:

6, parent[6] = 5, parent[5] = 1, parent[1] = 4, parent[4] = -1

And we must print the vertices in the reverse order: 4, 1, 5, 6.

E-OLYMP 4856. The shortest path The undirected weighted graph is given. Find the shortest path between two given vertices.

Input. The first line contains two positive integers n and m ($n \leq 2000$, $m \leq 50000$) – the number of vertices and edges in a graph. The second line contains positive integers s and f ($1 \leq s, f \leq n$, $s \neq f$) – the numbers of the vertices, the minimal length between which must be found. Each of the next m lines contains three numbers b_i , e_i and w_i – the numbers of the ends of i -th edge and its weight correspondingly ($1 \leq b_i, e_i \leq n$, $0 \leq w_i \leq 10^5$).

Output. Print in the first line one number – the length of minimal path between the vertices s and f . Print in the second line all the vertices on the shortest path from s to f in the traversal order. If the path from s to f does not exist, print -1.

Sample input

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

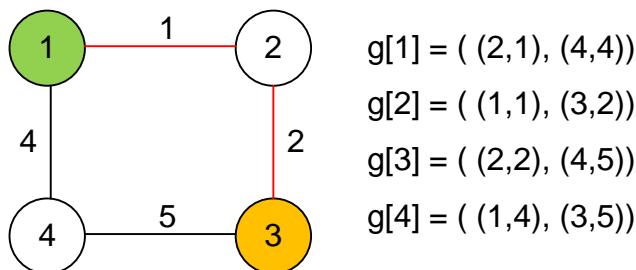
Sample output

```
3
1 2 3
```

► To solve the problem, it is necessary to implement Dijkstra's algorithm. Given the restrictions on the number of vertices and edges, the graph should be represented by an adjacency list.

Example

Sample graph has the form:



```
g[1] = ( (2,1), (4,4))
g[2] = ( (1,1), (3,2))
g[3] = ( (2,2), (4,5))
g[4] = ( (1,4), (3,5))
```

Algorithm realization

The graph is stored in an adjacency list g , where $g[i]$ contains a vector of pairs (vertex j , edge length between i and j). Let's declare additional global arrays for Dijkstra's algorithm:

- $\text{dist}[i]$ stores the shortest distance to the vertex i ;
- $\text{parent}[i]$ stores the vertex number from which we arrived to i moving from the source along the shortest path.

```
#define MAX 5010
#define INF 0x3F3F3F3F
int used[MAX], dist[MAX], parent[MAX];
vector<vector<pair<int, int>> > g;
```

Function **Relax** relaxes the edge (v, to) with weight $cost$.

```

void Relax(int v, int to, int cost)
{
    if (dist[to] > dist[v] + cost)
    {
        dist[to] = dist[v] + cost;
        parent[to] = v;
    }
}

```

Function ***Find_Min*** searches for the vertex with the smallest distance $\text{dist}[i]$ from the source among those to which the shortest distance has not yet been calculated (for which $\text{used}[i] = 0$).

```

int Find_Min(void)
{
    int i, v, min = INF;
    for(i = 1; i <= n; i++)
        if (!used[i] && (dist[i] < min)) min = dist[i], v = i;
    if (min == INF) return -1;
    return v;
}

```

Function ***path*** prints the shortest path from the source to the vertex v . For this, we use *parent* array.

```

void path(int v)
{
    if (v == -1) return;
    path(parent[v]);
    if (parent[v] != -1) printf(" ");
    printf("%d", v);
}

```

The main part of the program. Read the input data. Construct an adjacency list g of the graph.

```

scanf("%d %d", &n, &m);
scanf("%d %d", &s, &f);
g.resize(n+1);
for(i = 0; i < m; i++)
{
    scanf("%d %d %d", &b, &e, &w);
    g[b].push_back(make_pair(e, w));
    g[e].push_back(make_pair(b, w));
}

```

Initialization of global arrays. The distance from source to source ($\text{dist}[s]$) is set to 0.

```

memset(dist, 0x3F, sizeof(dist));
dist[s] = 0;
memset(parent, -1, sizeof(parent));
memset(used, 0, sizeof(used));

```

Start the cycle of Dijkstra's algorithm. Since the graph contains n vertices, it is enough to perform $n - 1$ iterations.

```
for(i = 1; i < n; i++)
{
    v = Find_Min();
```

If no vertex can be included into the set of vertices to which the shortest distance is already calculated, then end the algorithm.

```
if (v == -1) break;
```

Relax all the edges outgoing from the vertex v .

```
for(j = 0; j < g[v].size(); j++)
{
    to = g[v][j].first;
    cost = g[v][j].second;
    if (!used[to]) Relax(v,to,cost);
}
```

The shortest distance to v is already computed (it is in $\text{dist}[v]$).

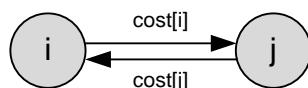
```
used[v] = 1;
}
```

Print the answer depending on the value of $\text{dist}[f]$. If it equals to infinity, then there is no path to the required vertex. Otherwise, print the required shortest distance and shortest path.

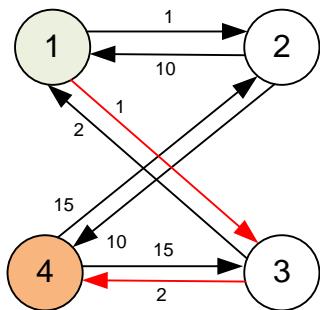
```
if (dist[f] == INF)
    printf("-1\n");
else
{
    printf("%d\n", dist[f]);
    path(f); printf("\n");
}
```

E-OLYMP 1388. Petrol stations There are n cities, some of which are connected by roads. In order to drive along one road, you need one tank of gasoline. In each city the petrol tank has a different cost. You need to get out of the first city and reach the n -th one, spending the minimum possible amount of money.

► Let $\text{cost}[i]$ be the cost of petrol in the city i . For each pair of cities between which there is a road, create two directed edges: $i \rightarrow j$ of weight $\text{cost}[i]$ and $j \rightarrow i$ of weight $\text{cost}[j]$.



We must find the path of minimum cost from 1 to n using Dijkstra algorithm. Graph, given in the first sample input, has the form:



The path of minimum cost is $1 \rightarrow 3 \rightarrow 4$, its cost is $1 + 2 = 3$.

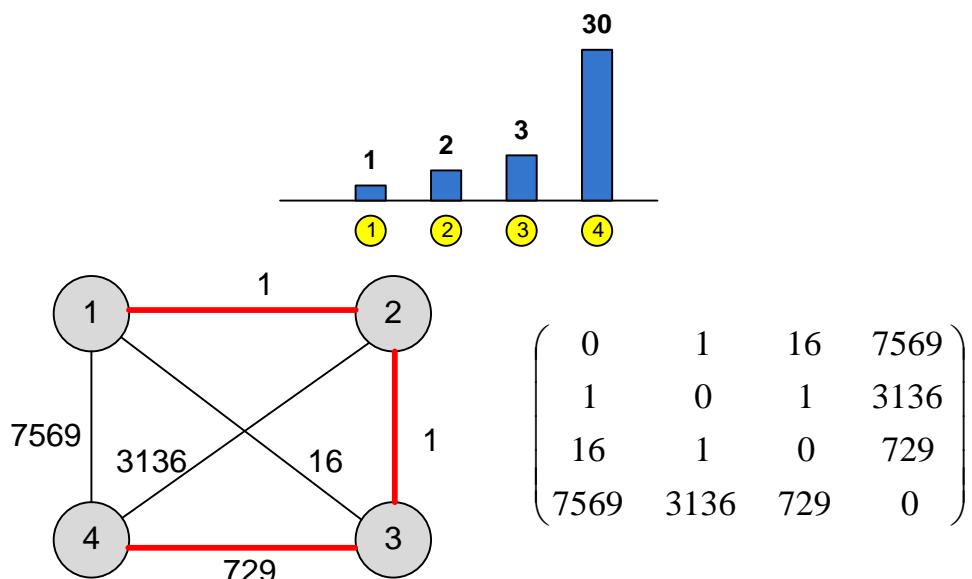
E-OLYMP 5850. Mathematical platforms In older games, which have 2D graphics, one can run into the next situation. The hero jumps along the platforms or islands that hang in the air. He must move himself from one side of the screen to the other. The hero can jump from any platform number i to any platform number k , spending $(i - k)^2 * (y_i - y_k)^2$ energy, where y_i and y_k are the heights where these platforms hang. Obviously, energy should be spent frugally.

You are given the heights of the platforms in order from the left side to the right. Can you find the minimum amount of energy to get from the 1-st (start) platform to the n -th (last)?

► Consider each platform as a vertex of the graph. Between each pair of vertices i and j make an undirected edge $g[i][j]$ with weight $(i - j)^2 * (y_i - y_j)^2$ (the amount of energy required to move between vertices i and j). Now you must find the minimum path between the first and the last vertices, that can be done using Dijkstra's algorithm.

There is no need to keep the graph in memory, since all values of $g[i][j]$ can be calculated using the above formula.

Graph and its weight matrix are given below:



The hero should move along the platforms in the order $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, spending for it $1 + 1 + 729 = 731$ energy units.

Implementation of Dijkstra's algorithm using a priority queue

Implement Dijkstra's algorithm using a priority queue. This data structure is supported by standard template library and is called **priority_queue**. It allows you to store a pair (*key, value*) and to perform two operations:

- insert element with given priority;
- extract the element with the highest priority;

Declare priority queue *pq*, which elements are pairs (*distance, node*), where *distance* is the distance from the source to the *node*. When you insert items, the head of the queue always contains a pair (*distance, node*) with the smallest *distance*. Thus, the vertex to which the distance from the source is minimum, available as *pq.top().second*.

Arbitrary elements cannot be removed from the priority queue (although theoretically heaps support such operation, but in the standard library it is not implemented). Therefore, the relaxation will not remove the old pairs from the queue. As a result, the queue can contain simultaneously several pairs of the same vertices (but with different distances). Among these pairs, we are interested in only one for which the element *pq.top().first* equals to *dist[to]*, all the rest are **fictitious**. Therefore, at the beginning of each iteration, when we take from queue next pair, we will check, if it is fictitious or not (it is enough to compare *pq.top().first* and *dist[to]*). This is an important modification, if it is not done, this will lead to spoilage of the asymptotic behavior to $O(nm)$.

1. Initialize distances of all vertices as infinite.

2. Create an empty priority_queue *pq*. Every item of *pq* is a pair (*distance, vertex*). Distance is used as first item of pair as first item is by default used to compare two pairs

3. Insert source vertex into *pq* and make its distance as 0.

4. While either *pq* doesn't become empty

- a) Extract minimum distance vertex from *pq*. Let the extracted vertex be *u*.
- b) Loop through all adjacent of *u* and do following for every vertex *v*.

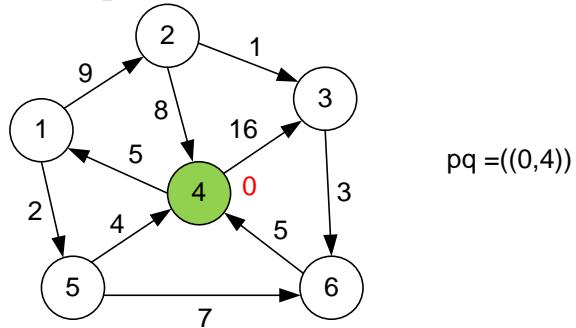
// If there is a shorter path to *v* through *u*.

If *dist[v] > dist[u] + weight(u, v)*

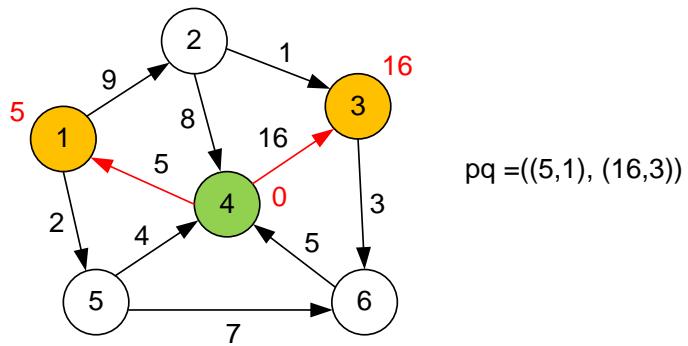
- (i) Update distance of *v*, i.e., do *dist[v] = dist[u] + weight(u, v)*
- (ii) Insert *v* into the *pq* (even if *v* is already there)

5. Print distance array *dist[]* to print all shortest paths.

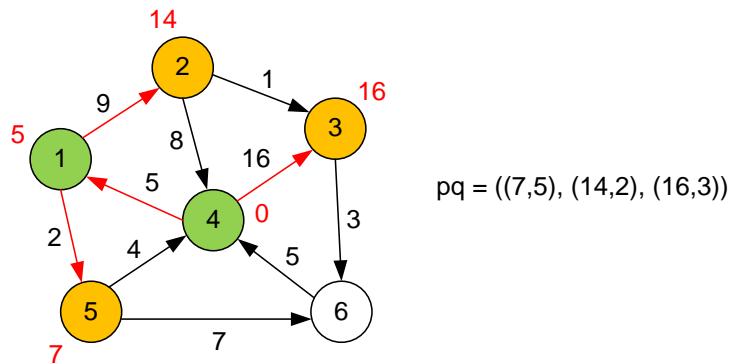
Example. Let's simulate Dijkstra's algorithm using priority queue. We'll insert to priority queue the pairs (*distance*, *vertex*). We start in vertex 4. Shortest path from 4 to 4 is 0. So insert to the queue the pair (0, 4).



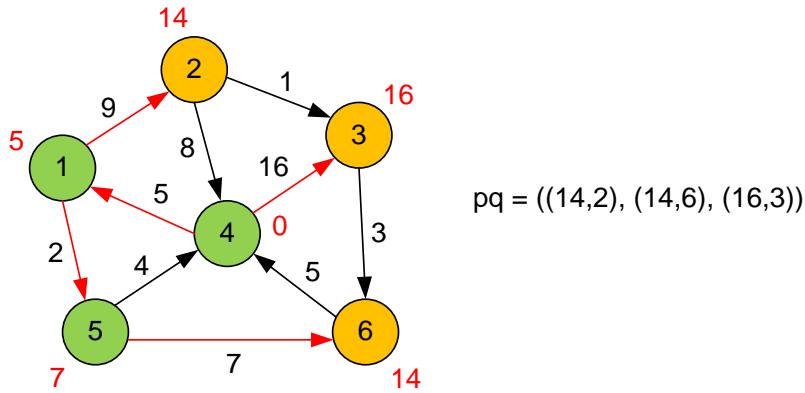
Front of the queue contains vertex 4. Remove it from the queue. Make relaxation of the edges adjacent to the vertex 4.



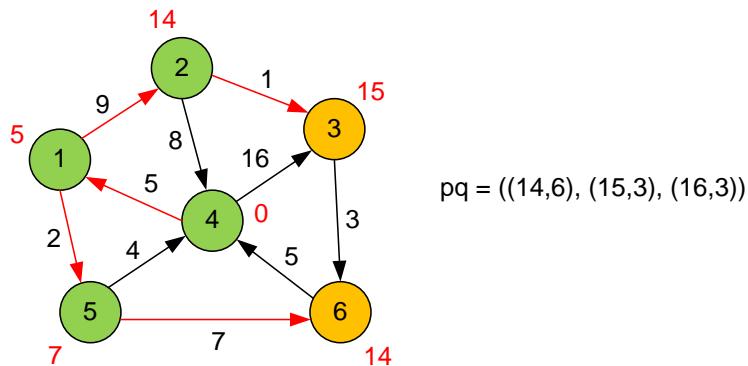
Front of the queue contains vertex 1. Remove it from the queue. Make relaxation of the edges adjacent to the vertex 1.



Front of the queue contains vertex 5. Remove it from the queue. Make relaxation of the edges adjacent to the vertex 5.

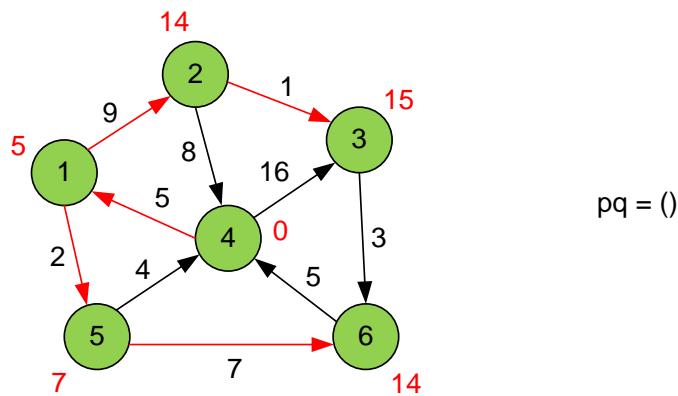


Front of the queue contains vertex 2. Remove it from the queue. Make relaxation of the edges adjacent to the vertex 2.



Make relaxation of edges adjacent to the vertices 6 and 3. None of the edges relax. The next pair (16, 3) is *fictitious*, since $16 > \text{dist}[3] = 15$.

The final graph states are following:



E-OLYMP 2965. Distance between the vertices An undirected weighted graph is given. Find the length of the shortest path between two specified vertices.

Input. The first line contains two positive integers n and m ($1 \leq n \leq 10^5$, $1 \leq m \leq 2 * 10^5$) – the number of vertices and edges in the graph. The second line contains two positive integers s and t ($1 \leq s, t \leq n$, $s \neq t$) – the numbers of the vertices between which you need to find the shortest path.

The following m lines contain the descriptions of the edges. Each edge i is described by three integers b_i , e_i , and w_i ($1 \leq b_i, e_i \leq n$, $0 \leq w_i \leq 100$) – the numbers of the vertices it connects and its weight.

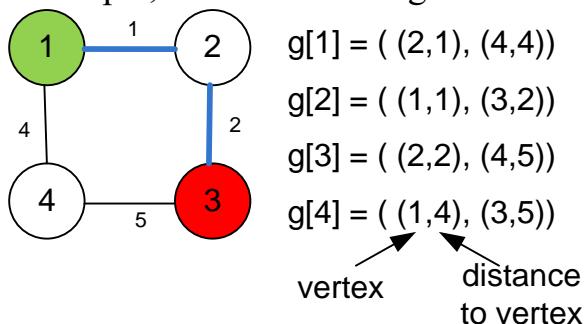
Output. Print the weight of the shortest path between vertices s and t , or -1 if no such path exists.

Sample input	Sample output
4 4	3
1 3	
1 2 1	
2 3 2	
3 4 5	
4 1 4	

- The number of vertices in the graph is large, so we use a priority queue to implement Dijkstra's algorithm.

Example

The graph given in the example, has the following form:



Algorithm realization

Declare a constant for infinity.

```
#define INF 0x3F3F3F3F
```

Declare a structure g for storing the weighted graph. Each element $g[i]$ is a list of pairs (*vertex, distance*).

```
vector<vector<pair<int, int>> q;
```

The function **Dijkstra** implements Dijkstra's algorithm starting from the vertex *start*.

```
void Dijkstra(vector<vector<pair<int, int>>>& g, vector<int>& d,
              int start)
{

```

Declare and initialize the priority queue pq . The elements of this queue will be pairs (*distance from the source to the vertex, vertex*). Thus, the vertex at the top of the queue will always be the one with the smallest distance from the source.

```
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
pq.push({0, start});
```

Initialize the array of shortest distances d . The vertices of the graph are numbered from 1 to n .

```
d = vector<int>(n + 1, INF);
d[start] = 0;
```

Continue the Dijkstra's algorithm until the queue is empty.

```
while (!pq.empty())
{
```

Extract the pair e from the top of the priority queue.

```
pair<int, int> e = pq.top(); pq.pop();
int v = e.second; // node
```

Check if the vertex v is a **dummy** vertex. If the distance to v is greater than the already computed value $d[v]$, ignore this vertex.

```
if (e.first > d[v]) continue; // distance > d[v]
```

Iterate over all vertices to adjacent to v . The distance from v to to is *cost*.

```
for (auto edge: g[v])
{
    int to = edge.first;
    int cost = edge.second;
```

If the edge (v, to) relaxes, update the value of $d[to]$ and insert the pair $(d[to], to)$ into the queue.

```
if (d[v] + cost < d[to])
{
    d[to] = d[v] + cost;
    pq.push({d[to], to});
}
```

The main part of the program. Read the input data.

```
scanf("%d %d %d %d", &n, &m, &start, &fin);
g.resize(n + 1);
for (i = 0; i < m; i++)
{
```

Read the undirected edge (b, e) with weight w and add it to the graph g .

```
scanf("%d %d %d", &b, &e, &w);
g[b].push_back({e, w});
g[e].push_back({b, w});
}
```

Run Dijkstra's algorithm from the starting vertex $start$.

```
Dijkstra(g, dist, start);
```

Print the shortest path distance $dist[fin]$. If $dist[fin]$ is equal to infinity, then the path does not exist.

```
if (dist[fin] == INF)
    printf("-1\n");
else
    printf("%d\n", dist[fin]);
```

Algorithm realization – edge structure

Declare a constant for infinity.

```
#define INF 0x3F3F3F3F
```

Declare an array of shortest distances $dist$.

```
vector<int> dist;
```

The **edge** structure stores information about an edge: the vertex $node$ it leads to and its weight $dist$.

```
struct edge
{
    int node, dist;
    edge(int node, int dist) : node(node), dist(dist) {}
};
```

The **edge** structure will also be used in the priority queue. The queue stores the vertices of the graph, where:

- $node$ is the vertex number;
- $dist$ is the current distance from the start vertex to vertex $node$.

The vertex at the top of the queue will be the one with the smallest $dist$ value.

```
bool operator< (edge a, edge b)
{
    return a.dist > b.dist;
}
```

Declare the adjacency list g of the graph.

```
vector<vector<edge>> g;
```

The function **Dijkstra** implements Dijkstra's algorithm starting from the vertex *start*.

```
void Dijkstra(vector<vector<edge> > &g, vector<int> &d, int start)
```

Declare and initialize the priority queue *pq*. The elements of this queue will be pairs (*distance from the source to the vertex, vertex*). Thus, the vertex at the top of the queue will always be the one with the smallest distance from the source.

```
priority_queue<edge> pq;
pq.push(edge(start, 0));
```

Initialize the array of shortest distances *d*. The vertices of the graph are numbered from 1 to *n*.

```
d = vector<int>(n + 1, INF);
d[start] = 0;
```

Continue the Dijkstra's algorithm until the queue is empty.

```
while (!pq.empty())
{
```

Extract the pair *e* from the top of the priority queue.

```
edge e = pq.top(); pq.pop();
int v = e.node;
```

Check if the vertex *v* is a **dummy** vertex. If the distance to *v* is greater than the already computed value *d[v]*, ignore this vertex.

```
if (e.dist > d[v]) continue;
```

Iterate over all vertices *to* adjacent to *v*. The distance from *v* to *to* is *cost*.

```
for (auto ed : g[v])
{
    int to = ed.node;
    int cost = ed.dist;
```

If the edge (*v, to*) relaxes, update the value of *d[to]* and insert the pair (*d[to], to*) into the queue.

```
if (d[v] + cost < d[to])
{
    d[to] = d[v] + cost;
    pq.push(edge(to, d[to]));
}
```

The main part of the program. Read the input data.

```
scanf("%d %d %d %d", &n, &m, &start, &fin);
g.resize(n + 1);
for (i = 0; i < m; i++)
{
```

Read the undirected edge (b, e) with weight w and add it to the graph g .

```
scanf("%d %d %d", &b, &e, &w);
g[b].push_back(edge(e, w));
g[e].push_back(edge(b, w));
}
```

Run Dijkstra's algorithm from the starting vertex $start$.

```
Dijkstra(g, dist, start);
```

Print the shortest path distance $dist[fin]$. If $dist[fin]$ is equal to infinity, then the path does not exist.

```
if (dist[fin] == INF)
    printf("-1\n");
else
    printf("%d\n", dist[fin]);
```