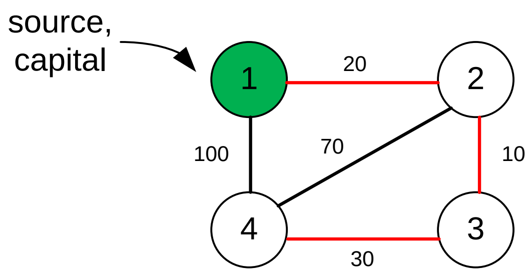
Dijkstra’s algorithm and its implementation

There is decribed an algotirhm of solving problems that require finding of the shortest path from one vertex to the other vertices of the graph named Dijkstra’s Algorithm. We'll concider realization of the algorithm with arrays, STL containers – queues with priorities priority_queue, sets set, and also use of operations over the heap push_heap and pop_heap.

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.



$$\begin{pmatrix} 0 & 20 & \infty & 100 \\ 20 & 0 & 10 & 70 \\ \infty & 10 & 0 & 30 \\ 100 & 70 & 30 & 0 \end{pmatrix}$$

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] = ∞ 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 nust 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 dist[*v*] to be the length of the shortest path from **source** to *v*, then

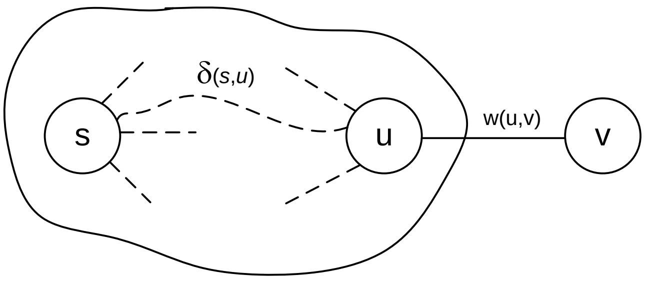
dist[1] = 0, dist[2] = 20, dist[3] = 30, dist[4] = 60

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, …, v_k) is the shortest path from v_1 to v_k and $1 \le i \le j \le k$, then p_{ij} = (v_i, v_{i+1}, …, 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 $u \to v$ is the last edge of the shortest path from *s* to *v*, then

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



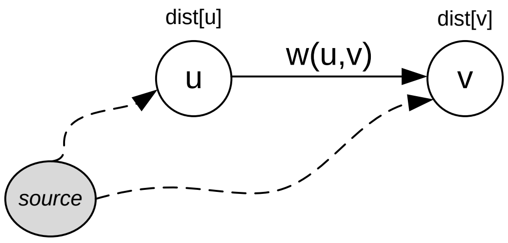
Dijkstra algorithm solves the problem of the shortest path from one source to others. It is greedy algorithm.

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

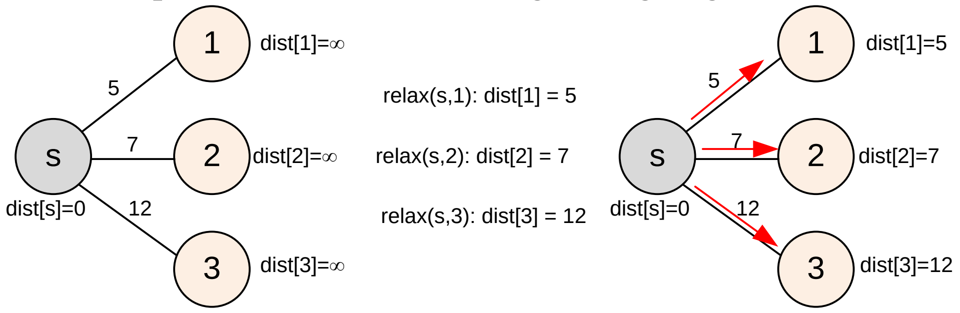


Current shortest distance from *source* to *v* is dist[*v*]. But what is we shall go to *v* through vertex *u* and along the edge $u \to v$? The cost of this path is dist[*u*] + w(*u*, *v*). If this value is less then the value dist[*v*] (we are looking for the shortest path), we must update dist[*v*]:

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

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

When ***Dijkstra*** algorithm starts, all values dist[*v*] are set to ∞ (only dist[*source*] = 0). dist[*v*] = ∞ 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 $s \to 1$: dist[s] = 0, dist[1] = ∞. We have the relation:

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

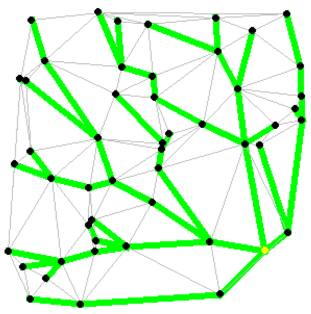
so edge $s \to 1$ relaxes, and dist[1] = dist[*s*] + w(*s*, 1) = 0 + 5 = 5.

The same way the edges $s \to 2$ and $s \to 3$ also relax and we get dist[2] = 7, 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 used[*v*] = 1.

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

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



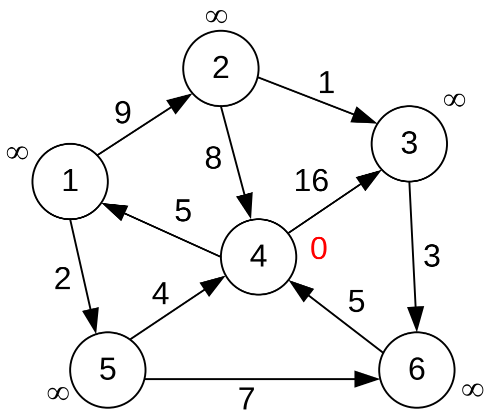
At each step we add to the set S such vertex *v* for which the distance from the *source* is no more than the distance from the *source* to other vertices from V / S. This is done by finding the minimum among the values of dist[*v*] for all $v \in$ V / S (values *v* which are not in S). This addition 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 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 ← ∅
   Q ← V[G]
   while Q ≠ ∅
```

```
do u ← EXTRACT-MIN(Q)
   S = S ∪ {u}
   for each vertex v ∈ Adj[u]
      do Relax(u, v, w)
```

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

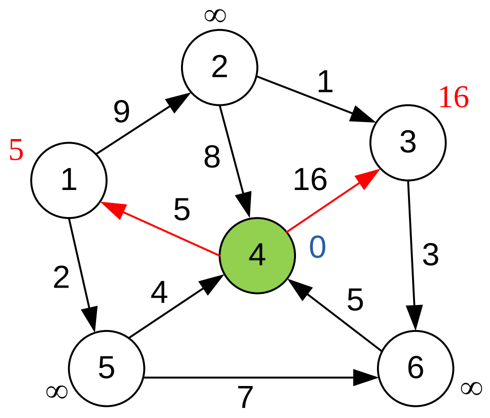


First iteration. We find the smallest dist[*i*], where *i* is the vertex, not included in S.

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

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

- 4 → 1: dist[1] = min(dist[1], dist[4] + g[4][1]) = min(∞, 0 + 5) = 5;
- 4 → 3: dist[3] = min(dist[3], dist[4] + g[4][3]) = min(∞, 0 + 16) = 16;

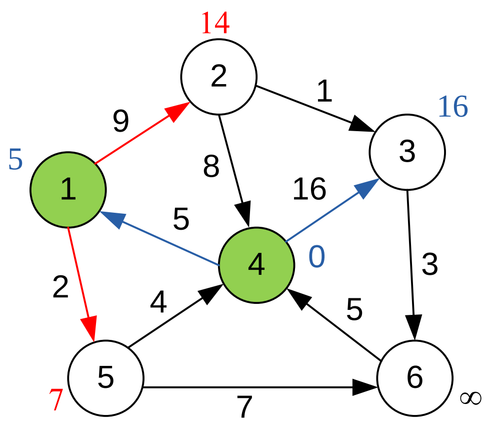


Second iteration. We are looking for the least value among dist[*i*], where *i* ∉ S = {4}:

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

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 → 2: dist[2] = min(dist[2], dist[1] + g[1][2]) = min(∞, 5 + 9) = 14;
- 1 → 5: dist[5] = min(dist[5], dist[1] + g[1][5]) = min(∞, 5 + 2) = 7;



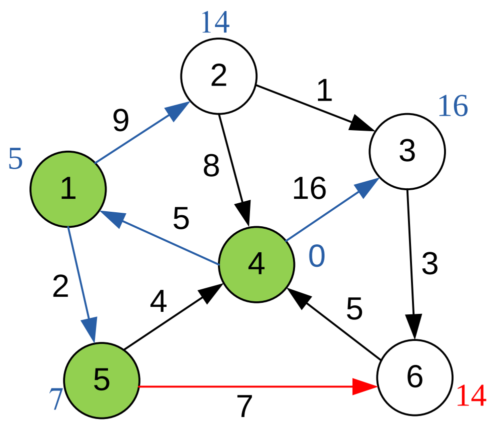
Third iteration. We are looking for the least value among dist[*i*], where $i \notin S = \{1, 4\}$:

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

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 → 6: dist[6] = min(dist[6], dist[5] + g[5][6]) = min(∞, 7 + 7) = 14;

We do not consider edge 5 → 4 because vertex 4 is already in S.

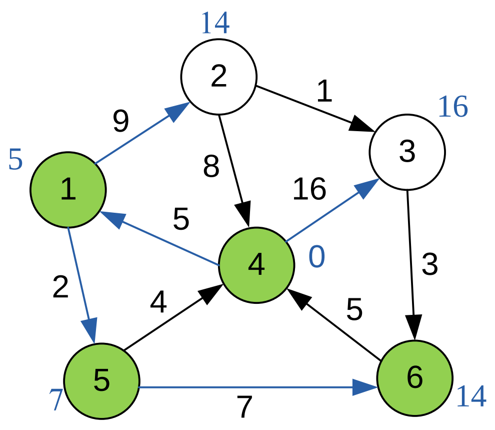


Fourth iteration. We are looking for the least value among dist[*i*], where $i \notin S = \{1, 4, 5\}$:

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

We have two vertices with minimum value of dist[*i*]: they are 2 and 6 (dist[2] = 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.



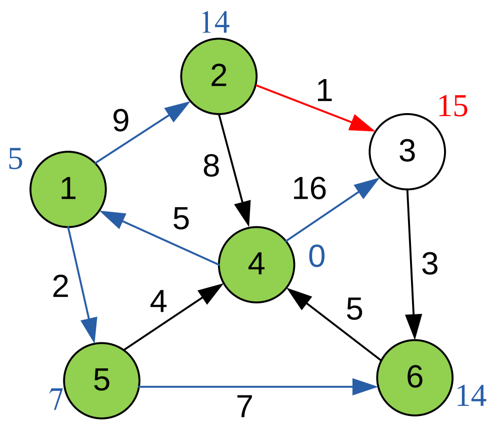
Fifth iteration. We are looking for the least value among dist[*i*], where *i* ∉ S = {1, 4, 5, 6}:

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

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 → 3: dist[3] = min(dist[3], dist[2] + g[2][3]) = min(16, 14 + 1) = 15;

We do not consider edge 2 → 4 because vertex 4 is already in S.

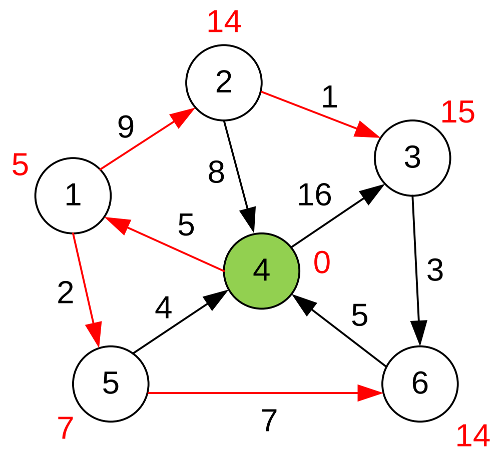


There is no sence to run ***sixth iteration***. Vertex 3 will be included to S. But there is no any edge outgoing from 3 that runs into vertex not in S.

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 dist[*i*], for which *i* ∈ S, are highlighted in *italics*.

Iteration	S	dist[1]	dist[2]	dist[3]	dist[4]	dist[5]	dist[6]
start	{}	∞	∞	∞	**0**	∞	∞
1	{4}	**5**	∞	16	*0*	∞	∞
2	{1, 4}	*5*	14	16	*0*	**7**	∞
3	{1, 4, 5}	*5*	14	16	*0*	*7*	**14**
4	{1, 4, 5, 6}	*5*	**14**	16	*0*	*7*	*14*
5	{1, 2, 4, 5, 6}	*5*	*14*	**15**	*0*	*7*	*14*

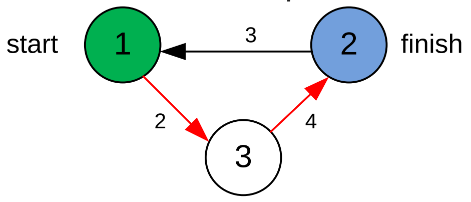
The iterative process of Dijkstra's algorithm



E-OLYMP 2351. Dijkstra The *directed weighted* graph is given. Find the shortest distance from one vertex to another.

Input. First line contains number of vertices *n*, starting *s* and final *f* vertices. Next *n* lines describe weighted matrix.

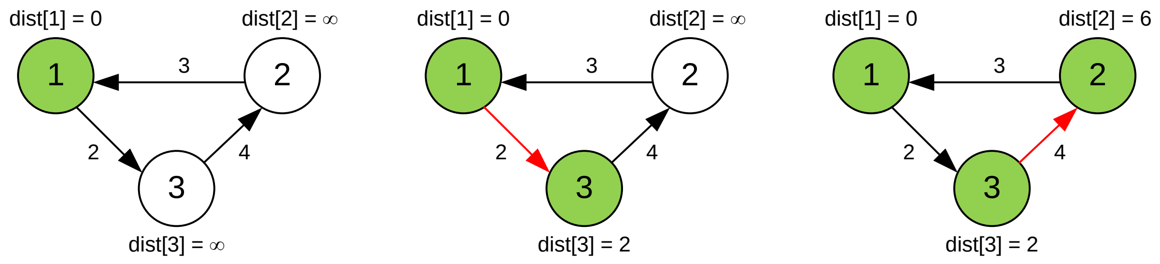
Output. Print the shortest distance from *s* to *f* or -1 if the path does not exist.



Sample input
```
3 1 2
0 -1 2
3 0 -1
-1 4 0
```

Sample output
```
6
```

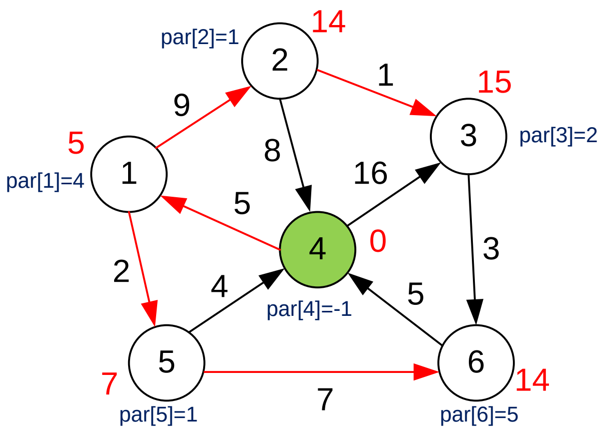
► Read weighted matrix g. Run Dijkstra algorithm.



```
#include <stdio.h>
#include <string.h>
#define MAX 2001
#define INF 0x3F3F3F3F

int i, j, min, n, s, f, v;
int g[MAX][MAX], used[MAX], dist[MAX];

// Relaxation of the edge i -> j
void Relax(int i, int j)
{
    if (dist[i] + g[i][j] < dist[j])
      dist[j] = dist[i] + g[i][j];
}

int main(void)
{
  scanf("%d %d %d", &n, &s, &f);

  memset(g, 0x3F, sizeof(g));
  memset(used, 0, sizeof(used));
  memset(dist, 0x3F, sizeof(dist));
  dist[s] = 0;

  for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    scanf("%d", &g[i][j]);

  for (i = 1; i < n; i++)
  {
    // find vertex v with minimum d[v] among not used vertices
    min = INF; v = -1;
    for (j = 1; j <= n; j++)
       if (used[j] == 0 && dist[j] < min) { min = dist[j]; v = j; }

    // no more vertices to add
    if (v < 0) break;

    // relax all edges outgoing from v
    // process edge v -> j
    for (j = 1; j <= n; j++)
      if (used[j] == 0 && g[v][j] != -1) Relax(v, j);

    // shortest distance to v is found
    used[v] = 1;
  }

  if (dist[f] == INF) printf("-1\n");
  else printf("%d\n", dist[f]);
  return 0;
}
```

How to resore 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 parent[u] = v means that after the relaxation of the edge $v \longrightarrow u$ the shortest distance dist[u] becomes optimal.



v	1	2	3	4	5	6
parent[v]	4	1	2	-1	1	5

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

v, parent[*v*], parent[parent[*v*]], …, *source*, -1

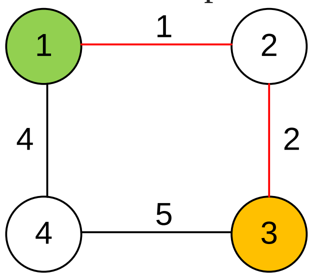
For example, the shortest path from 4 to 6 can be found next way:

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 and print the shortest path between two given vertices.

In sample input we must to find the shortest path from 1 to 3.



► Read list of edges, construct an adjacency matrix g. Run Dijkstra algorithm.

We can print the shortest path from *source* to *v* using function PrintPath(*v*):

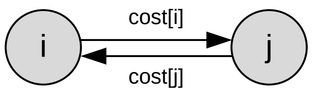
```
void PrintPath(int v)
{
  vector<int> res;
  while (v != -1)
  {
    res.push_back(v);
    v = parent[v];
  }

  for (int i = res.size() - 1; i >= 0; i--)
    printf("%d ", res[i]);
  printf("\n");
}
```

E-OLYMP [8348. Distance between vertices](#) The weighted graph is given. Find the weight of the minimum path between two vertices.

► Read list of edges, construct an adjacency matrix g. Run Dijkstra algorithm.

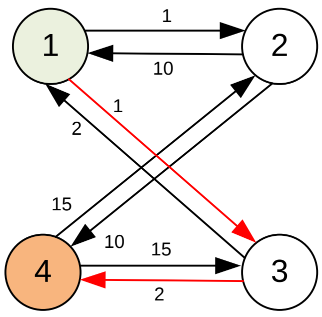
E-OLYMP [1388. Petrol stations](#) There are *n* cities, some of which are connected by roads. In order to drive along one road you need one tank of gasoline. In each city the petrol tank has a different cost. You need to get out of the first city and reach the *n*-th one, spending the minimum possible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

► Let 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* → *j* of weight cost[*i*] and *j* → *i* of weight 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:



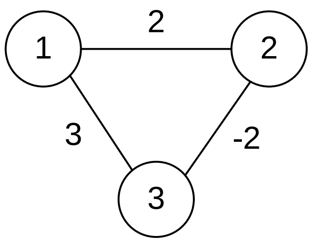
The path of minimum cost is 1 → 3 → 4, its cost is 1 + 2 = 3.

Next, we consider the implementation of Dijkstra's algorithm using different containers and functions described in the template library STL.

STL (Standart Template Library) - a set of template functions and classes in C ++, which includes a variety of data containers (list, queue, set, map, hash table, priority queue) and basic algorithms (sorting, searching). The story of the basic concepts and the template library can be found, for example, in Wikipedia: http://en.wikipedia.org/wiki/Standard_Template_Library. A detailed description of the STL can be found at http://www.sgi.com/tech/stl.

Edges with negative length

Dijkstra's algorithm does not work if there are edges in the graph of negative length. Consider the graph:



Run Dijkstra’s algorithm from the first vertex, initially setting d = (0, ∝, ∝). After the relaxation of edges emanating from the vertex 1, we obtain d [2] = 2 and d [3] = 3. The shortest distance to the top 2 will be considered as calculated, run relaxation edges emanating from the vertex array 2. Finally, the shortest paths will take the form: d = (0, 2, 0). Although the shortest path between vertices 1 and 2 is 3 - 2 = 1 (and it passes the top 3).

Note that adding a large number to the weights of the edges still does not solve the problem.

Implementation of Dijkstra’s algorithm using sets

Simulate priority queue with a set<pair <int, int>>. Its elements are a pair, the second element which contains the number of vertices *i*, and the first - the current optimal distance from the source to the vertex *i* (the value of d[*i*]). The advantage of this data storage is that the elements of the set s always sort of the first component of the pair. That is, a couple with a top added each time during the set S, and the top s is available as * s.begin ().

```
#include <cstdio>
#include <set>
#include <memory>
#define MAX 10000
#define INF 0x3F3F3F3F
using namespace std;

typedef pair<int, int> ii;

int i, v, n, dist, source, b, e;
int m[MAX][MAX], d[MAX];
set<ii> s;

int main(void)
{
  scanf("%d %d", &n, &source);
  memset(m, 63, sizeof(m));
  while(scanf("%d %d %d", &b, &e, &dist) == 3) m[b][e] = dist;
  memset(d, 0x3F, sizeof(d)); d[source] = 0;
  s.insert(ii(0,source));
```

The body of the while loop is executed *n* times (*n* - the number of vertices in the graph). At each iteration, one and only one edge is entered into the set S and removed from further consideration. At the last, *n* - th iteration, no edge relaxes and from the set s is removed information about the latest top.

```
  while(!s.empty())
  {
    ii top = *s.begin(); s.erase(s.begin());
    v = top.second;
```

Vertex *v* is added to the current step in the set S. Produce relaxation of edges leading from *v*. If an edge (*v*, *i*) relaxes, it should be removed from the pair s (d[*i*], *i*)) and added (d[*v*] + m[*v*][*i*], *i*).

```
      for(i = 1; i <= n; i++)
        if (d[i] > d[v] + m[v][i])
        {
          if (d[i] != INF) s.erase(s.find(ii(d[i],i)));
          d[i] = d[v] + m[v][i];
          s.insert(ii(d[i],i));
        }
    }

    for(i = 1; i <= n; i++)
      printf("From source %d to destination %d distance is %d\n",
              source, i, d[i]);
    return 0;
  }
```

Implementation of Dijkstra's algorithm using a priority queue

Implement Dijkstra's algorithm using a priority queue. This data structure is supported by standart 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.

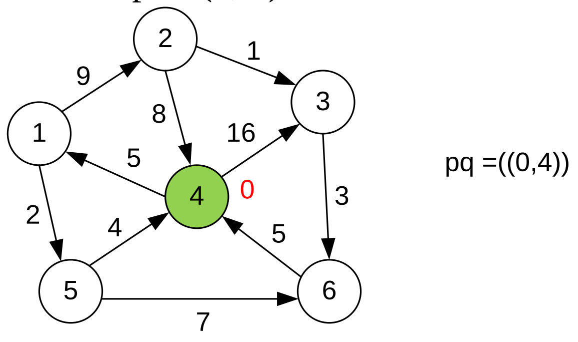
4. While either *pq* doesn't become empty
   a) Extract minimum distance vertex from *pq*. Let the extracted vertex be *u*.

b) Loop through all adjacent of *u* and do following for every vertex *v*.

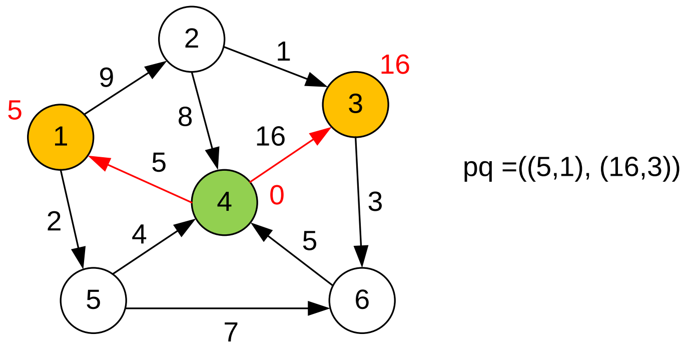
```
    // If there is a shorter path to $v$ through $u$.
    If dist[$v$] > dist[$u$] + weight($u$, $v$)

        (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.

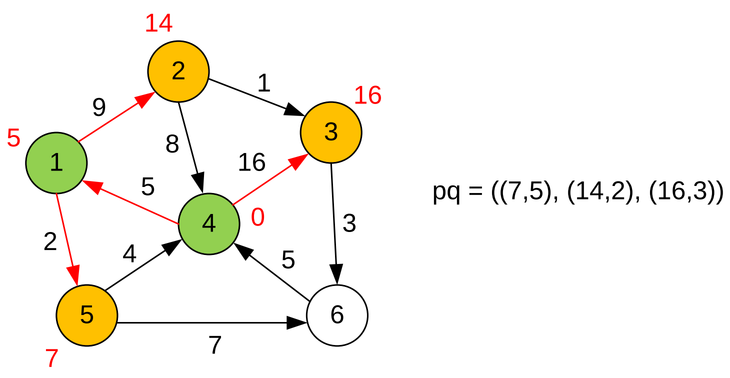
Example. Let's simulate Dijkstra's algorithm using priority queue. We'll insert to priority queue the pairs (*distance*, *vertex*). We start in vertex 4. Shortest path from 4 to 4 is 0. So insert to the queue the pair (0, 4).



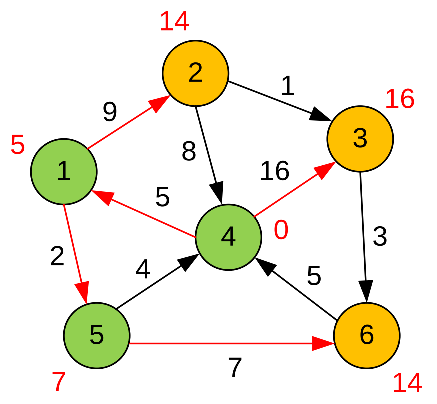
Front of the queue contains vertex 4. Remove it from the queue. Make relaxation of the edges adjacent to the vertex 4.



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

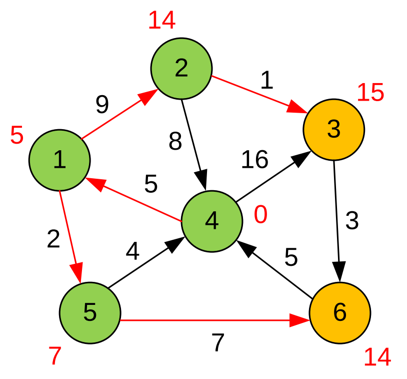


Front of the queue contains vertex 5. Remove it from the queue. Make relaxation of the edges adjacent to the vertex 5.



pq = ((14,2), (14,6), (16,3))

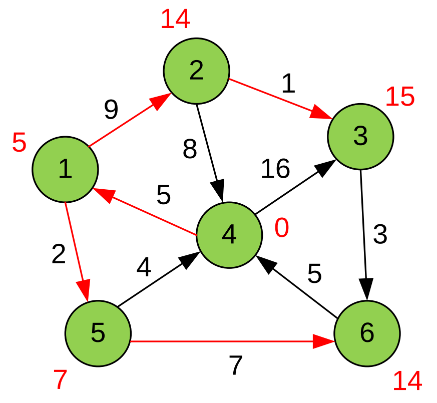
Front of the queue contains vertex 2. Remove it from the queue. Make relaxation of the edges adjacent to the vertex 2.



pq = ((14,6), (15,3), (16,3))

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 > dist[3] = 15.

The final graph states are following:



pq = ()

E-OLYMP **2965. Distance between the vertices** Undirected weighted graph is given. Find the weight of the minimal path between two vertices.

► Number of vertices $n \le 10^5$, let's use priority queue to solve the problem.

```
#include <cstdio>
#include <vector>
#include <queue>
```

```
#define INF 0x3F3F3F3F
using namespace std;

int b, e, w, v, j, i, tests;
int n, m, start, fin;
vector<int> dist;

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

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

vector<vector<edge> > g;

void Dijkstra(vector<vector<edge> > &g, vector<int> &d, int start)
{
  priority_queue<edge> pq;
  pq.push(edge(start, 0));

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

  while (!pq.empty())
  {
    edge e = pq.top(); pq.pop();
    int v = e.node;
    if (e.dist > d[v]) continue;

    for (int j = 0; j < g[v].size(); j++)
    {
      int to = g[v][j].node;
      int cost = g[v][j].dist;
      if (d[v] + cost < d[to])
      {
        d[to] = d[v] + cost;
        pq.push(edge(to, d[to]));
      }
    }
  }
}

int main(void)
{
  scanf("%d %d %d %d", &n, &m, &start, &fin);
  g.resize(n + 1);
  for (i = 0; i < m; i++)
  {
    scanf("%d %d %d", &b, &e, &w);
    g[b].push_back(edge(e, w));
    g[e].push_back(edge(b, w));
  }

  Dijkstra(g, dist, start);

  if (dist[fin] == INF)
```

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

  return 0;
}
```

E-OLYMP 625. Distance between the vertices Undirected weighted graph is given. Find the weight of the minimal path between two vertices.

► Number of vertices $n \leq 5000$, let's use priority queue to solve the problem. In this problem we need not only to find the shortest distance between the vertices, but the shortest path also.

Implementation of Dijkstra's algorithm using the heap

Priority queue, as it is known, can be implemented using the heaps. Instead of direct declaration of priority queue we will use a vector of pairs *pq*, and the operations of insertion / extraction of its elements will be used by the Template Library functions for maintaining the basic properties of the heap pop_heap and push_heap. The main property of the heap is supported on a set of elements from *pq*[0] to *pq*[*len*].

```
#include <cstdio>
#include <queue>
#include <algorithm>
#define MAX 10000
using namespace std;

vector<pair<int,int> > pq(MAX); //(distance,node)
vector<vector<int> > m(MAX, vector<int> (MAX));
vector<int> dist(MAX,0x3FFFFFFF);
int n, src;

int main(void)
{
  int i, a, b, d, len;
  scanf("%d %d",&n,&src);
  while(scanf("%d %d %d", &a, &b, &d) == 3) m[a][b] = d;

  pq[0] = make_pair(0,src); len = 1; dist[src] = 0;
  while(len)
  {
    pair<int,int> s = pq[0];
    pop_heap(pq.begin(),pq.begin()+len, greater<pair<int,int> >()); len--;

    for(i = 1; i <= n; i++)
      if (m[s.second][i] && (dist[i] > dist[s.second] + m[s.second][i]))
      {
        dist[i] = dist[s.second] + m[s.second][i];
        pq[len] = make_pair(dist[i],i); len++;
        push_heap(pq.begin(),pq.begin()+len,greater<pair<int,int> >());
      }
  }
  for(i = 1; i <= n; i++)
    printf("From source %d to destination %d distance is %d\n",
           src, i, dist[i]);
  return 0;
}
```

The run time of Dijkstra's algorithm

The complexity of Dijkstra's algorithm consists of two basic operations:

• the time spent with the lowest vertex distance d[v];

• time of the relaxation time (time change of the d[to]).

Suppose | V | = N – the number of vertices, and | E | = M – the number of edges. Simple implementation of these operations will require, respectively O(n) and O(1) time. Taking into concideration that the first operation is performed only in O(n) time, and the second in O(m), the asymptotic of the simplest implementation of Dijkstra's algorithm will be O($n^2 + m$).

While using arrays for search the shortest paths algorithm requires $n – 1$ iterations, in each of which the search for v and relaxation of all outgoing edges of it is done during the O(n). Thus, the overall execution time of the algorithm is O(n^2).

The asymptotic behavior of O($n^2 + m$) is optimal for dense graphs, when $m \approx n^2$. The more graph is sparse (ie the less m, compared with the maximum number of edges n^2), the less optimal this estimate becomes because of the first term. Thus, it is necessary to improve the operating times of the first type while not greatly deteriorating the running times of the second type.

For example, the Fibonacci heap allows generate the operation of the first type in O($\log_2 n$), and the second in O(1). Therefore, while using Fibonacci heaps time of Dijkstra's algorithm will be O($\log_2 n+m$), which is nearly the theoretical minimum for the shortest path search algorithm. However, Fibonacci heap is quite difficult to implement, and also have a considerable constant hidden in the asymptotic behavior.

As a compromise, you can use the structure of the data set or design priority_queue, allowing to carry out both types of operations (recovery of minimum and update element) for O($\log_2 n$). Then time of Dijkstra's algorithm will be O($n\log_2 n + m\log_2 n$) = O($m\log_2 n$).

In the Olympiad competitions task of finding ways to fairly widespread. For example, with the following tasks required to use Dijkstra's algorithm:

[Tianshan] http://acm.tju.edu.cn/toj : 2134 (106 miles to Chicago), 2819 (Travel), 2870 (The K-th City).

[Topkoder] www.topcoder.com : SRM 241 (AirTravel).

BIBLIOGRAPHY

1. "The construction and analysis of algorithms" feed T., Charles Leiserson, Rivest, R., K. Stein - Moscow, St. Petersburg, Kiev, 2005 - 1292 p.

2. "Practice and Theory of Programming", Book 2. Vinokurov NA, Vorozhtsov AV - M: Fizmatkniga, 2008 - 288 p.

3. Article "binary pyramid", Journal of potential №7, 2007.

- Runs on a weighted graph;
- Starts with an initial node and a goal node;
- Finds the least cost path to the goal node.

Dijkstra's Algorithm

- Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
- Keep a set of visited nodes. This set starts with just the initial node.
- For the current node consider all of its unvisited neighbours and calculate (distance to the current node) + distance fron the current node to the neighbour). If this is less than their current tentative distance, replace it with this new value.
- When we are done considering all of the neigbours of the current node, mark the current node as visited and remove it from the unvisited set.
- If the destination node has been marked visited, the algorithm has finished.
- Set the unvisited node marked with the smallest tentative distance as the next "current node" and go back to step 3.