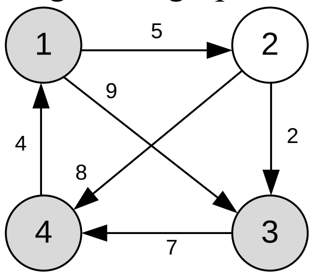
Floyd – Warshall algorithm

The Floyd–Warshall algorithm (also known as Floyd's algorithm) is an algorithm for finding ***shortest paths*** in a ***weighted graph*** with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm.

Below given a graph, its weight matrix and matrix of shortest distances.



$$\begin{pmatrix} 0 & 5 & 9 & \infty \\ \infty & 0 & 2 & 8 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 5 & 7 & 13 \\ 12 & 0 & 2 & 8 \\ 11 & 16 & 0 & 7 \\ 4 & 9 & 11 & 0 \end{pmatrix}$$

weight matrix

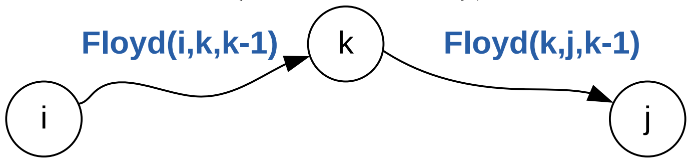
shortest distances

Consider a graph with vertices numbered 1 through *n*. Further consider a function Floyd(*i*, *j*, *k*) that returns the shortest possible path from *i* to *j* using vertices only from the set {1, 2 , …, *k*} as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each *i* to each *j* using any vertex in {1, 2 , …, *n*}.

For each of these pairs of vertices, the Floyd(*i*, *j*, *k*) could be either

- a path that *does not go* through *k* (only uses vertices in the set {1, 2 , …, *k* – 1}) or
- a path that *does go* through *k* (from *i* to *k* and then from *k* to *j* both only using intermediate vertices in {1, 2 , …, *k* – 1})

We know that the best path from *i* to *j* that only uses vertices 1 through *k* – 1 is defined by Floyd(*i*, *j*, *k* – 1), and it is clear that if there was a better path from *i* to *k* to *j*, then the length of this path would be the concatenation of the shortest path from *i* to *k* (only using intermediate vertices in {1, 2 , …, *k* – 1}) and the shortest path from *k* to *j* (only using intermediate vertices in {1, 2 , …, *k* – 1}).



If w(*i*, *j*) is the weight of the edge between vertices *i* and *j*, we can define Floyd(*i*, *j*, *k*) in terms of the following recursive formula:

the base case is:

- Floyd(*i*, *j*, 0) = w(*i*, *j*);

and the recursive case is

- Floyd(*i*, *j*, *k*) = min(Floyd(*i*, *j*, *k* – 1), Floyd(*i*, *k*, *k* – 1) + Floyd(*k*, *j*, *k* – 1))

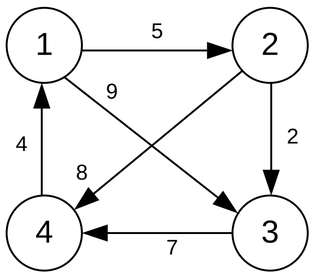
This formula is the heart of the Floyd – Warshall algorithm. The algorithm works by first computing Floyd(i, j, k) for all (i, j) pairs for k = 1, then k = 2, and so on. This process continues until k = n, and we have found the shortest path for all (i, j) pairs using any intermediate vertices.

Time complexity of Floyd – Warshall algorithm is O(n^3).

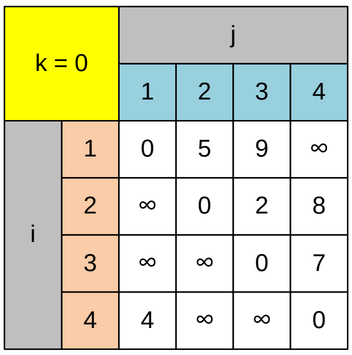
E-OLYMP 974. Floyd - 1 The complete directed weighted graph is given with the adjacency matrix. Construct a matrix of shortest paths between its vertices. It is guaranteed that the graph does not contain cycles of negative weight.

► Use Floyd – Warshall algorithm.

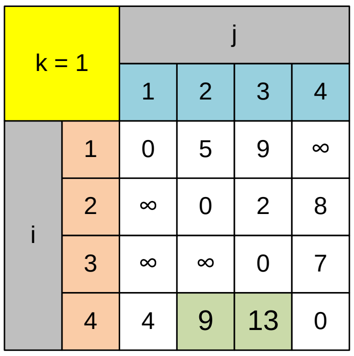
```
#include <stdio.h>
#include <string.h>
#define MAX 101

int n, a, b, i, j, dist;
int g[MAX][MAX];

void floyd(void)
{
  for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++)
    if (g[i][k] + g[k][j] < g[i][j]) g[i][j] = g[i][k] + g[k][j];
}

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

  floyd();

  for (i = 1; i <= n; i++)
  {
    for (j = 1; j <= n; j++)
      printf("%d ", g[i][j]);
    printf("\n");
  }
  return 0;
}
```



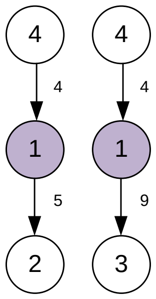
The distance matrix at each iteration of k, with the updated distances, will be:

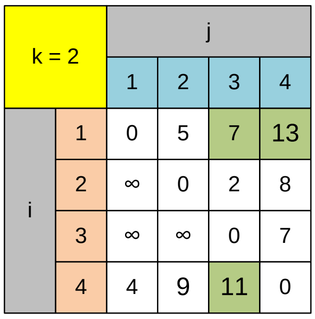


k = 0	j			
i	1	2	3	4
1	0	5	9	∞
2	∞	0	2	8
3	∞	∞	0	7
4	4	∞	∞	0

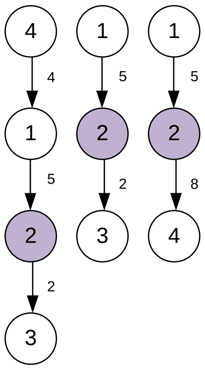


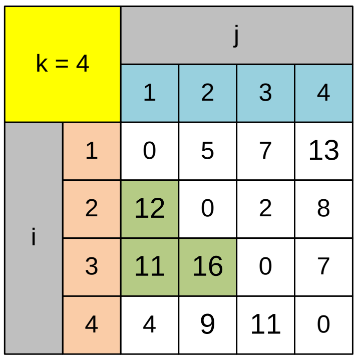
k = 1	j			
i	1	2	3	4
1	0	5	9	∞
2	∞	0	2	8
3	∞	∞	0	7
4	4	9	13	0





k = 2	j			
i	1	2	3	4
1	0	5	7	13
2	∞	0	2	8
3	∞	∞	0	7
4	4	9	11	0





k = 4	j			
i	1	2	3	4
1	0	5	7	13
2	12	0	2	8
3	11	16	0	7
4	4	9	11	0

E-OLYMP 975. Floyd Given a directed weighted graph. Find a pair of vertices, the shortest distance from one of them to another is maximum among all pairs of vertices.

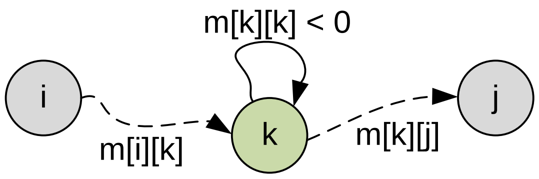
► Use Floyd – Warshall algorithm. Find maximum value in the resulting matrix.

E-OLYMP 976. Floyd – existence The directed weighted graph is given. Using its adjacency matrix, determine for each pair of vertices does there exist the shortest path between them or not.

The shortest path may not exist for two reasons:

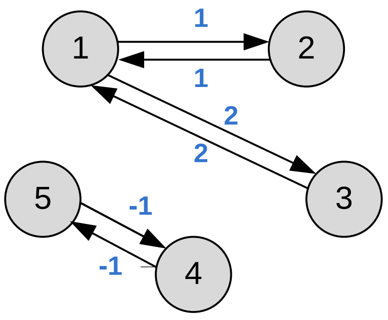
- There is no way.
- There is a way of arbitrarily small weight.

► Run the Floyd-Warshall algorithm on input graph. The path between the vertices *i* and *j* does not exist, if at the completion of algorithm m[*i*][*j*] = +∞. The shortest path between nodes *i* and *j* does not exist, if there is a node *k* such that there exist the paths of any length from *i* to *k* and from *k* to *j*, and between *k* and *k* the cycle of negative length exists (m[*k*][*k*] < 0).



In all other cases the path between *i* and *j* exists.

Graph given in a sample, has a form:



Store the adjacency matrix in array m. We shall build the resulting matrix in array res. Let the value “plus infinity” equals to INF.

```
#define INF 0x3F3F3F3F
#define MAX 101
int m[MAX][MAX], res[MAX][MAX];
```

The Floyd-Warshall algorithm. As the graph vertices have negative weights, process them carefully: if at some stage of processing m[*i*][*j*] becomes less than -INF, set m[*i*][*j*] = -INF.

```
void floyd(void)
{
  for(int k = 0; k < n; k++)
  for(int i = 0; i < n; i++)
  for(int j = 0; j < n; j++)
    if ((m[i][k] < INF) && (m[k][j] < INF))
    {
      if (m[i][k] + m[k][j] < m[i][j]) m[i][j] = m[i][k] + m[k][j];
      if (m[i][j] < -INF) m[i][j] = -INF;
    }
}
```

The main part of the program. Read the input adjacency matrix. Set zeroes on its diagonal. If there is no edge between vertices *i* and *j*, set m[*i*][*j*] = INF.

```
scanf("%d",&n);
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
{
  scanf("%d",&m[i][j]);
  if ((m[i][j] == 0) && (i != j)) m[i][j] = INF;
}
```

Run the Floyd-Warshall algorithm.

```
floyd();
```

Build the resulting matrix res. If there is no path between vertices *i* and *j*, set res[*i*][*j*] = 0. Otherwise set res[*i*][*j*] = 1, after which look for the cycle paths.

```
for(i = 0; i < n; i++)
for(j = 0; j < n; j++)
{
  if (m[i][j] == INF) res[i][j] = 0; else
  {
    res[i][j] = 1;
```

The shortest path does not exist between the vertices *i* and *j* if for some vertex *k* there exist a path from *i* to *k* and from *k* to *j*, and also exists a cycle of negative length that starts and finishes at vertex *k* (then m[*k*][*k*] < 0).

```
    for(k = 0; k < n; k++)
      if ((m[k][k] < 0) && (m[i][k] < INF) && (m[k][j] < INF))
        res[i][k] = res[i][j] = res[k][j] = 2;
  }
}
```

Print the resulting matrix res.

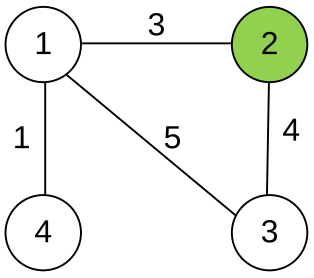
```
for(i = 0; i < n; i++)
{
  for(j = 0; j < n; j++)
    printf("%d ",res[i][j]);
  printf("\n");
}
```

E-OLYMP **2635. Detonating cord** Given an undirected weighted graph. Find the minimum time for which all ropes (edges) will be burned.

► Let g be the distance matrix of a given graph. The weight of the graph edges equals to the burning time of the ropes. The entire cord can burn out, so graph is connected. Find the shortest distances between all pairs of vertices using the Floyd-Warshall algorithm.

If we set fire to a knot (vertex of the graph) *v*, then the entire structure of the ropes will burn out after a time equal to the distance from vertex *v* to the farthest vertex. It remains to find the vertex, the distance from which to the farthest is minimal.

Below given a graph from a sample and a matrix of shortest distances.



	1	2	3	4	max
1	0	3	5	1	5
2	3	0	4	4	4
3	5	4	0	6	6
4	1	4	6	0	6
					4 min

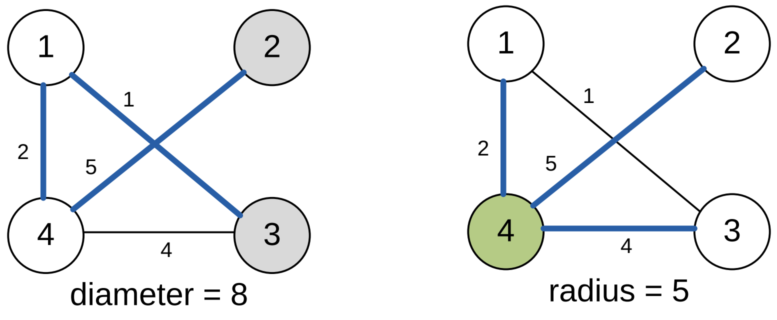
When vertex 2 is set on fire, all other vertices will burn out the fastest, in 4 seconds.

E-OLYMP **1796. Diameter of the graph** Find the ***diameter*** and the ***radius*** of the graph.

► Let $v_1, v_2, \ldots, v_n$ be the graph vertices, $d(v_i, v_j)$ be the shortest distance between v_i and v_j. Let

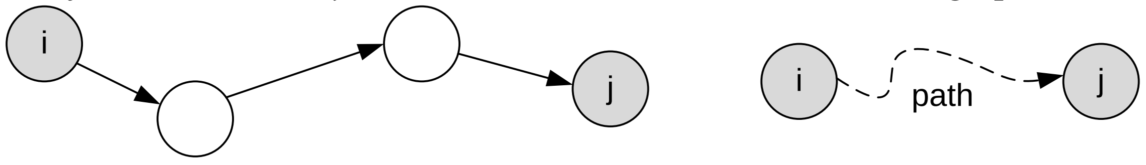
$$r(v_i) = \max_j d(v_i, v_j)$$

Then the minimum value of $r(v_i)$ is the ***radius*** of the graph, and the maximum value of $r(v_i)$ is the ***diameter***.



Transitive closure of a graph

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here *reachable* mean that there is a path from vertex i to j. The *reachability matrix* is called ***transitive closure*** of a graph.



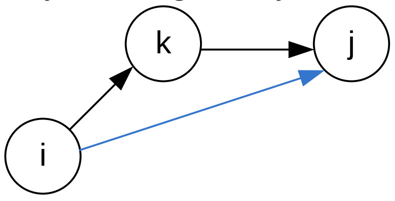
The graph is given in the form of adjacency matrix g[n][n]. Floyd Warshall algorithm can be used to calculate the distance matrix dist[n][n]. If dist[i][j] is *infinite*, then j is not reachable from i, otherwise j is reachable and value of dist[i][j] will be less than n.

Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations:

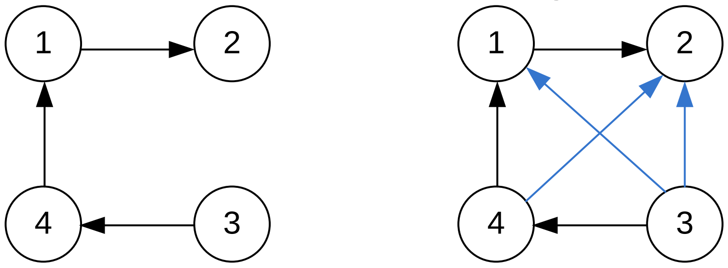
1. Instead of integer resultant matrix dist, we can create a *boolean* reachability matrix dist (we save space). The value dist[i][j] will be ***true*** if j is reachable from i, otherwise ***false***.
2. Instead of using arithmetic operations, we can use logical operations. For arithmetic operation ‘+’, logical **and** (&&) is used (we save time by a constant factor. Time complexity is same though).

E-OLYMP 10157. Transitive closure Find the transitive closure of the directed graph.

► In the problem you must find the transitive closure of the graph. If graph contains the edges $i \to k$ and $k \to j$, the edge $i \to j$ should be added.



Consider the graph at the left. Its transitive closure is given at the right.



In the transitive closure the next edges will be added: 3 → 1 (there is a path 3 → 4 → 1), 3 → 2 (path 3 → 4 → 1 → 2) and 4 → 2 (path 4 → 1 → 2).

Declare adjacency matrix g.

```
#define MAX 101
bool g[MAX][MAX];
```

Read the input data. Create the graph.

```
scanf("%d", &n);
while (scanf("%d %d", &a, &b) == 2)
  g[a][b] = true;
```

Start the transitive closure algorithm. If there are edges *i* → *k* and *k* → *j*, then create an edge *i* → *j*.

```
for (k = 1; k <= n; k++)
for (i = 1; i <= n; i++)
for (j = 1; j <= n; j++)
  if (g[i][k] && g[k][j]) g[i][j] = true;
```

Print the adjacency matrix of transitive closure of the graph.

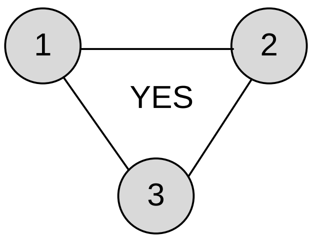
```
for (i = 1; i <= n; i++)
{
  for (j = 1; j <= n; j++)
    printf("%d ", g[i][j]);
  printf("\n");
}
```

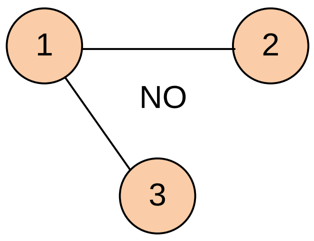
E-OLYMP 3988. Transitivity of a graph Undirected graph without loops and multiple edges is called transitive, if from conditions that vertices *u* and *v* are connected with an edge, vertices *v* and *w* are connected with an edge and all three vertices *u*, *v* and *w* are different, implies that vertices *u* and *w* are connected with an edge.

Verify that a given undirected graph is transitive.

► Run the graph transitive closure algorithm. If graph contains edges *i* → *k* and *k* → *j*, then one should check the existence of an edge *i* → *j*.

Graphs given in samples, have the form:


1
2
YES
3


1
2
NO
3