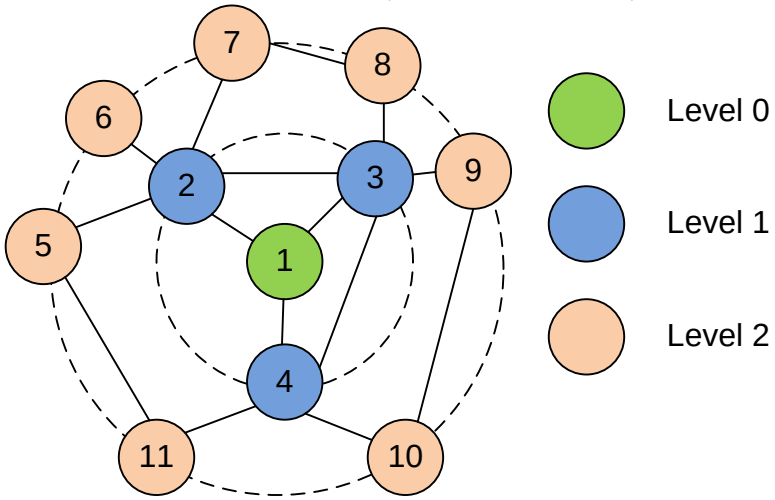
Breadth-First Search

Breadth-first search (bfs) is a traversing algorithm where you starts from a given node (source) and traverse the graph layerwise. We start from the source located on the layer 0. Then we visit all vertices on the layer 1, then on layer 2 and so on.



- We start from the vertex 1 (*source*) that belongs to the level 0.
- Then we visit all vertices on the distance 1 from the *source*: 2, 3, 4. These vertices are located on the level 1.
- Then we visit the other vertices located on the level 2.

The level of the vertex *v* corresponds to the minimum distance from the *source* to the vertex *v*. We shall keep this information in dist[*v*]. So

- dist[1] = 0;
- dist[2] = dist[3] = dist[4] = 1;
- dist[5] = dist[6] = …= dist[11] = 2;

Breadth-first search algorithm finds the ***shortest path*** from one vertex of the unweighted graph to all others. If we start bfs(*source*), then dist[*v*] ($1 \le v \le n$) contains the length of the shortest path from *source* to *v*. Here ***length of the path*** equals to the number of edges in the path.

Complexity of the algorithm O($n + m$), where n is the number of verteces, m is the number of edges.

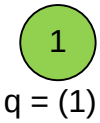
Algorithm can be understood as a process of "***lighting***" the graph: at *zero* step lights only the vertex *source*. At each next step fire spreads from the burning vertex to all its neighbors; that is, in one iteration of the algorithm there is an expansion of "***Ring of Fire***" in breadth per unit (hence the name of the algorithm).

How can we organize a "***Ring of Fire***"? Let's take a queue and push *source* vertex 1 into it:

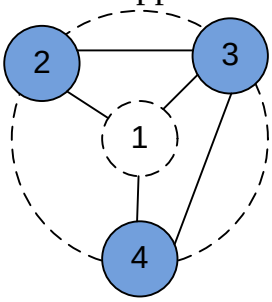
```
queue<int> q;
q.push(1);
```

Now q = (1), queue contains only one vertex. Let's *pop* a vertex (v = 1) and *push* into queue all vertices connected to it. Vertex 1 is connected with 2, 3 and 4. So we'll push these vertices. Now q = (2, 3, 4). What is in the queue? All vertices at the level 1!

Imagine like vertex 1 is exploded and vertices connected to 1 appeared. Vertices of level 0 are exploded and vertices of level 1 are appeared.

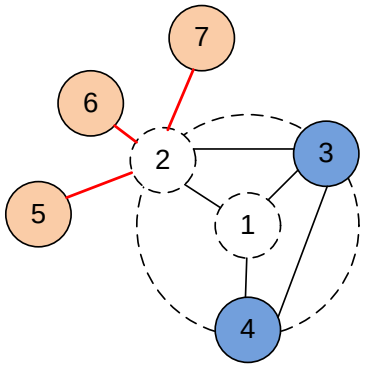


q = (1)



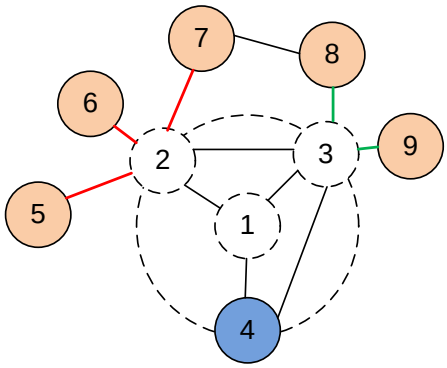
q = (2, 3, 4)

Let's continue the process of *explosion* (popping vertex from the queue and pushing into the queue all the vertices connected to it – which are not visited yet).



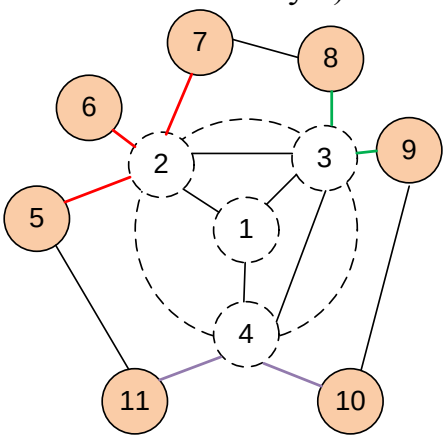
ball 2 exploded,
balls 5, 6, 7 appeared

q = (3, 4, 5, 6, 7)



ball 3 exploded,
balls 8, 9 appeared

q = (4, 5, 6, 7, 8, 9)



ball 4 exploded,
balls 10, 11 appeared

q = (5, 6, 7, 8, 9, 10, 11)

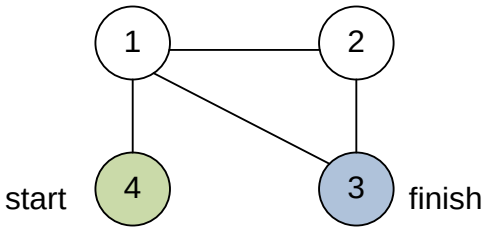
Now vertex 5 will be popped from the queue. No other unvisited vertex is connected to 5. So nothing will be pushed and q = (6, 7, 8, 9, 10, 11). Now the vertices will start to be popped from the queue until queue becomes empty: q = (). When q becomes empty, BFS algorithm terminates.

E-OLYMP **2401. Breadth first search** Undirected graph is given. Find the shortest path from vertex s to vertex f.

Input. First line contains number of vertices n ($n \le 100$) and vertices s and f of a graph. Next n lines describe the adjacency matrix of the graph.

Output. Print the minimum distance from s to f. If path does not exist, print 0.

Sample input	Sample output
4 4 3 0 1 1 1 1 0 1 0 1 1 0 0 1 0 0 0	2



► Let g be the adjacency matrix of the graph (g[*i*][*j*] = 1 if there exists an edge between vertices *i* and *j*, and g[*i*][*j*] = 0 otherwise), *dist* is an array where *dist*[*v*] contains the shortest length from *source* to the vertex *v*. *dist*[*v*] = -1 means that vertex *v* is not used (not visited). The numeration of the vertices in the graph starts from 1 (zero's row and column are not used).

```
#include <cstdio>
#include <vector>
#include <queue>
#include <cstring>
#define MAX 101
using namespace std;

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

// breadth first search dtarts from the vertex `start`
void bfs(int start)
{
  // initialise array dist.
  // dist[i] = -1 means that vertex i is not visited
  memset(dist, -1, sizeof(dist));
  dist[start] = 0;

  // declare and initialize queue
  queue<int> q;
  q.push(start);

  // continue algorithm until queue is not empty
  while (!q.empty())
  {
    // take vertex v from the head of the queue
    // remove vertex v from the queue
    int v = q.front(); q.pop();

    // where can we go from v? Try an edge v -> to
    for (int to = 1; to <= n; to++)
      // if there exists an edge from v to to (g[v][to] == 1)
      // and vertex to is not visited yet (dist[to] = -1)
      if (g[v][to] && dist[to] == -1)
      {
        // push vertex to to queue, calculate dist[to]
        q.push(to);
        dist[to] = dist[v] + 1;
      }
  }
}

int main(void)
```

```
{
  // freopen("bfs.in", "r", stdin);
  // read number of vertices n, starting s and final f vertex
  scanf("%d %d %d", &n, &s, &f);
  // read adjacency matrix
  for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    scanf("%d", &g[i][j]);

  // call bfs from the vertex s
  bfs(s);

  // if dist[f] = -1, path is not found, set dist[f] = 0
  if (dist[f] < 0) dist[f] = 0;

  // print the answer
  printf("%d\n", dist[f]);
  return 0;
}
```

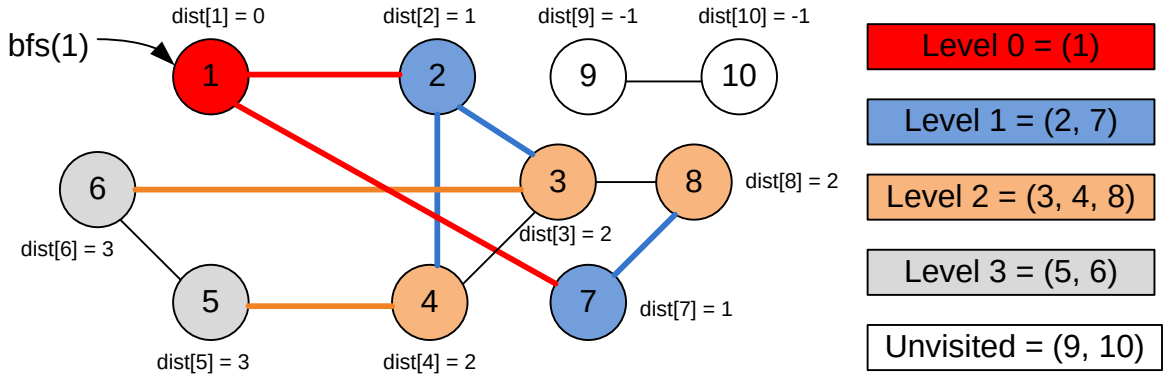
E-OLYMP **5338. Complete Graph - 2** Undirected graph is given with adjacency matrix. Find the shortest path from x to y. If path not found, print -1.

► Use *breadth first search* to find the shortest path.

E-OLYMP **4852. The shortest distance** Directed graph is given. Find the shortest path from the vertex x to other vertices of the graph.

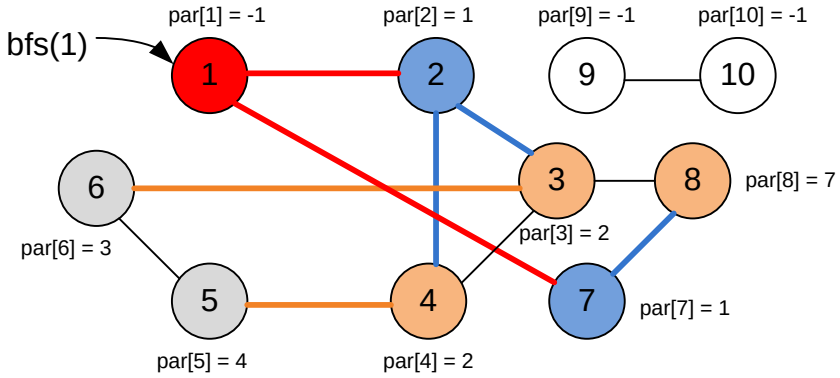
► Run bfs(x). Use *dist* array to store the shortest distances from x to other vertices. If there is no path from x to v, then dist[v] = -1.

Let's start breadth first search from the vertex 1. If we go from vertex v to vertex *to*, then dist[*to*] = dist[v] + 1.



If we want to restore the shortest path, for each vertex we need to know from which vertex we arrived there. Let parent[v] contains this information. If we go from vertex v to vertex *to* along the edge $v \rightarrow to$, then parent[*to*] = v. For source s we have parent[s] = -1. If vertex v is not reachable from the source, parent[v] = -1.

v	1	2	3	4	5	6	7	8	9	10
parent[v]	-1	1	2	2	4	3	1	7	-1	-1



How to find the shortest path from *source* to *v*? Let's move backwards from *v* until we reach source:

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

Of course, the path should be printed in the reverse order.

For example, let's find the shortest path from *source* = 1 to *v* = 5:

5, parent[5] = 4, parent[4] = 2, parent[2] = 1

So the shortest path from 1 to 5 is 1, 2, 4, 5

E-OLYMP 4853. The shortest path Undirected graph is given. Find the shortest path from *a* to *b*. Print the length of the shortest path and the path itself.

► Number of vertices is about $5 * 10^4$, use adjacency list to store the graph. Run bfs(*a*). If dist[*b*] ≠ -1, the path is found. Use array *parent* to restore the path.

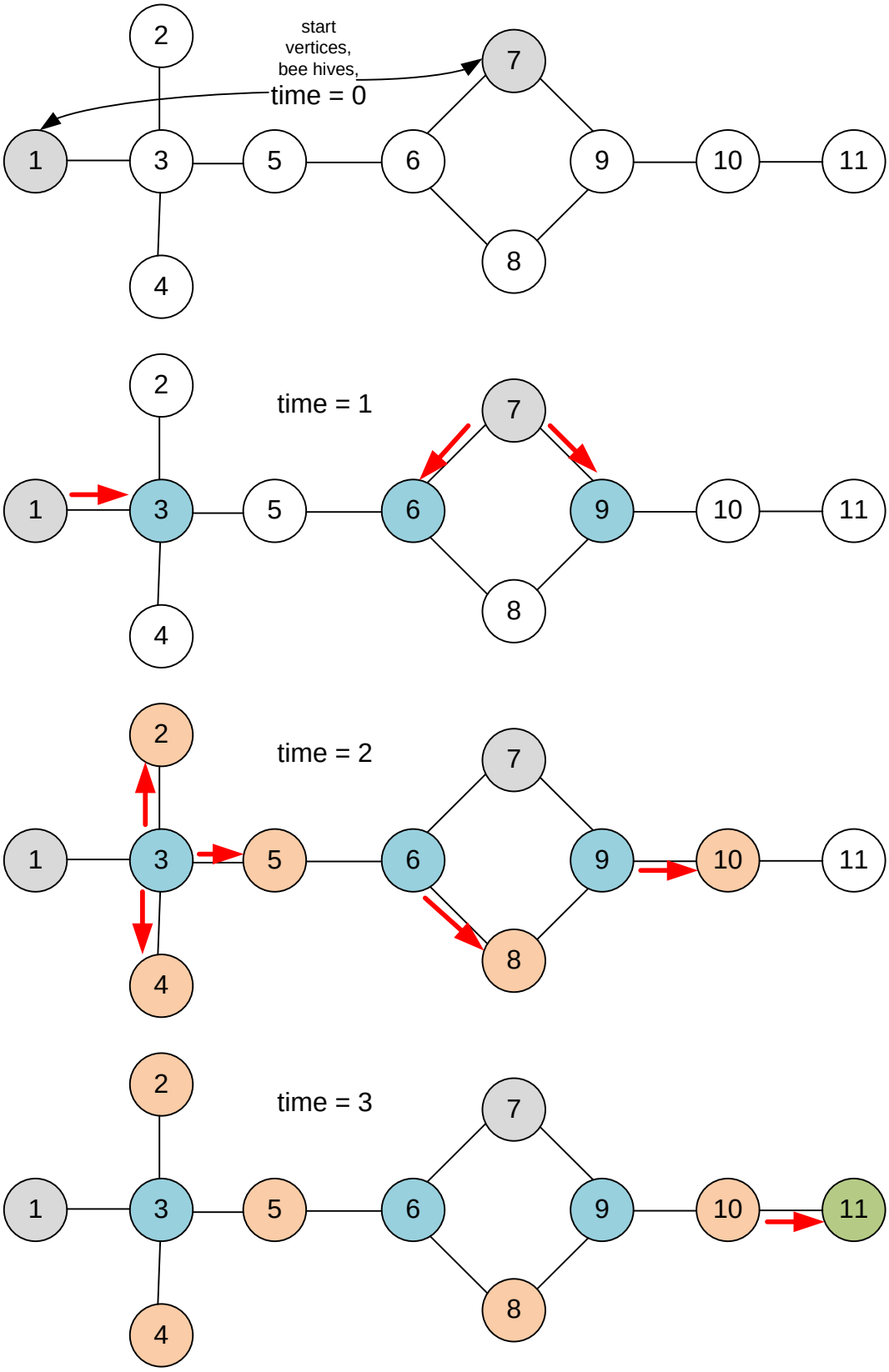
```
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

int i, j, n, m, a, b, u, v;
vector<int> dist, parent;
vector<vector<int> > g;

void bfs(int start)
{
  // declare arrays
  parent = vector<int>(n + 1, -1);
  dist = vector<int>(n + 1, -1);
  dist[start] = 0;

  // initialize a queue
  queue<int> q;
  q.push(start);

  while (!q.empty())
  {
    // remove vertex v from the queue
    int v = q.front(); q.pop();
```

```
      for (int i = 0; i < g[v].size(); i++)
      {
        // there is an edge v -> to
        int to = g[v][i];
        // if vertex v is not visited
        if (dist[to] == -1)
        {
          q.push(to); // push to into the queue
          dist[to] = dist[v] + 1; // recalculate the shortest distance
          parent[to] = v; // if v -> to, parent for to is v
        }
      }
    }
  }

  int main(void)
  {
    scanf("%d %d", &n, &m);
    scanf("%d %d", &a, &b);

    // construct adjacency list
    g.resize(n + 1);
    while (scanf("%d %d", &u, &v) == 2)
    {
      g[u].push_back(v);
      g[v].push_back(u);
    }

    bfs(a); // run bfs from vertex a

    if (parent[b] == -1) // if vertex b is NOT reachable, print -1
      printf("-1\n");
    else
    {
      printf("%d\n", dist[b]); // print shortest distance from a to b
      vector<int> path(1, b); // construct a resulting path

      // b, parent[b], parent[parent[b]], ..., source, -1
      while (parent[b] != -1)
      {
        b = parent[b];
        // insert vertices on the path into vector path
        path.push_back(b);
      }

      // print the shortest path in the order from a to b
      for (i = path.size() - 1; i >= 0; i--)
        printf("%d ", path[i]);
      printf("\n");
    }

    return 0;
  }
```

How to start *bfs* from ***multiple vertices*** simultaneously? Imagine we have a graph and some of its vertices are **bee hives**. At the moment *time* = 0 bees start to spread through the graph. Not fly, but spread. It means that at the *time* = 1 bees will be located in the bee hives and in all vertices at distance 1 from bee hives.

Solution is very simple:

push all started vertices (bee hives) into the queue and start bfs



E-OLYMP **4369. Arson** Undirected connected graph is given. Some vetices were fired. Find how many seconds will pass until the last vertex lights up and find this vertex.

► Number of vertices is about 10^5, use adjacency list to store the graph. Insert all vertices that were fired initially into the queue. Run bfs. Find minimum vetex v for which dist[v] is maximum. Print dist[v] and v.

E-OLYMP 10049. Bitmap Rectangular bitmap of size $n * m$ is given. Each pixel of the bitmap is either white or black, but at least one is white. The pixel in i-th line and j-th column is called the pixel (i, j). The distance between two pixels $p_1 = (i_1, j_1)$ and $p_2 = (i_2, j_2)$ is defined as:

$$d(p_1, p_2) = |i_1 - i_2| + |j_1 - j_2|$$

For each pixel find the distance to the nearest white pixel.

► Put the coordinates of all one's in the bitmap into the queue. Start a breadth first search from multiple sources.

Declare the constants.

```
#define INF 0x3F3F3F3F
#define MAX 1002
```

Store the bitmap in an array of strings g. The shortest distance from the point (i, j) to the nearest one (array of shortest distances) is stored in dist[i][j].

```
string g[MAX];
int dist[MAX][MAX];
```

Declare a queue that will contain the coordinates of the points.

```
deque<pair<int, int> > q; // (x, y)
```

Adding point (x, y) to the queue. The shortest distance from it to the nearest point with one is d.

```
void Add(int x, int y, int d)
{
```

If you go beyond the rectangular area, then ignore the point.

```
  if ((x < 1) || (x > n) || (y < 1) || (y > m)) return;
```

If the value dist[x][y] has already been computed, then ignore the point.

```
  if (dist[x][y] != INF) return;
```

Assign the value dist[x][y] = d. Push the point (x, y) into the queue.

```
  dist[x][y] = d;
  q.push_back(make_pair(x, y));
}
```

Function ***bfs*** implements the breadth first search.

```
void bfs(void)
{
```

```
  int x, y;
```

While the queue is not empty, pop the point *temp* and push the coordinates of its four neighbors into the queue.

```
  while (!q.empty())
  {
    pair<int, int> temp = q.front();
    q.pop_front();
    x = temp.first; y = temp.second;
    Add(x + 1, y, dist[x][y] + 1); Add(x - 1, y, dist[x][y] + 1);
    Add(x, y + 1, dist[x][y] + 1); Add(x, y - 1, dist[x][y] + 1);
  }
}
```

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

```
cin >> tests;
while (tests--)
{
  cin >> n >> m;
  for (i = 1; i <= n; i++)
  {
    cin >> g[i];
    g[i] = " " + g[i];
  }
```

Initialize the array of shortest distances with infinity.

```
  memset(dist, 0x3F, sizeof(dist));
```

Push to the queue *q* the coordinates of all points with ones.

```
  for (i = 1; i <= n; i++)
  for (j = 1; j <= m; j++)
    if (g[i][j] == '1')
    {
      q.push_back(make_pair(i, j));
      dist[i][j] = 0;
    }
```

Run the breadth first serch.

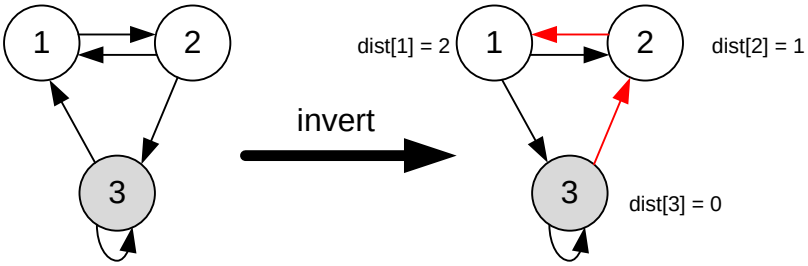
```
  bfs();
```

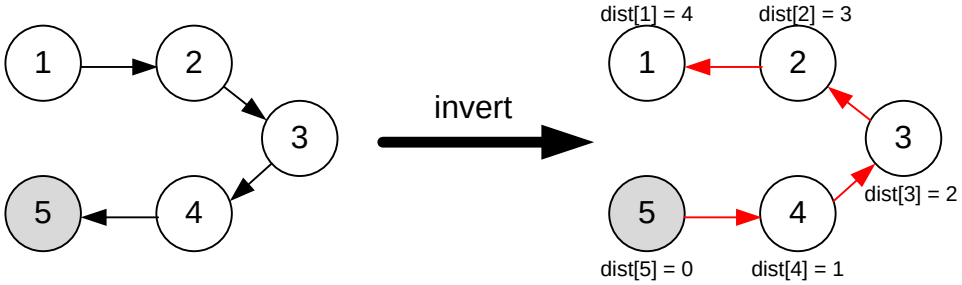
Print the answer – the required distances.

```
  for (i = 1; i <= n; i++)
  {
    for (j = 1; j <= m; j++)
      cout << dist[i][j] << " ";
    cout << endl;
  }
}
```

E-OLYMP 4819. Maximum by minimum Directed graph is given. Find in it a vertex, the shortest distance from which to the given one s is maximum, and print this distance.

► Number of vertices is about 5000, you can use adjacency matrix to store the graph. Reverse all edges. Find the shortest distance from s to all other vertices using bfs. Print the maximum distance.





Let s be a vertex, from which the bfs starts. We denote by $d[u] = \delta(s, u)$ the length of the shortest path from s to u. If the paths from s to u does not exist, then $d[u] = \infty$.

Theorem. Let $s \in V$ – an arbitrary vertex of the graph. Then, for any edge $(u, v) \in E$ the relation $\delta(s, v) \le \delta(s, u) + 1$ takes place.

Theorem. Let during the procedure bfs q contains all vertices $(v_1, v_2, \ldots, v_r)$, where v_1 is the head of the queue and v_r is the tail. Then we have the following relations:

- $d[v_r] \le d[v_1] + 1$
- $d[v_i] \le d[v_{i+1}]$

Corollary. If the vertex v_i is entered into the queue till the vertex v_j, then $d[v_i] \le d[v_j]$.

Theorem. At the end of the procedure bfs for each vertex u, reachable from s, we have the equality $d[u] = \delta(s, u)$. At the same time one of the shortest paths from s to u will be the path from s to parent[u], followed by the edge (parent[u], u).

Classification of edges

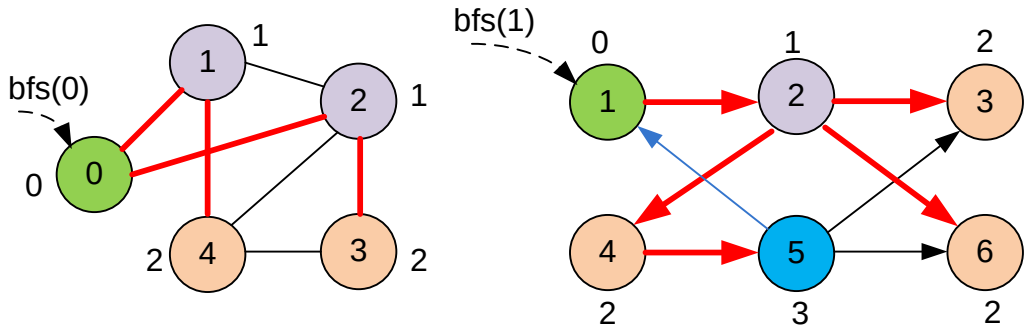
While bfs on an ***undirected graph***, we have the following properties:

• there are no back and no forward edges;

• for each tree edge (u, v) we have $d[v] = d[u] + 1$;

• for each cross edge (u, v) we have $d[v] = d[u]$ or $d[v] = d[u] + 1$;

While bfs on a ***directed graph***, we have the following properties:
• there are no forward edges;
• for each tree edge (u, v) we have $d[v] = d[u] + 1$.
• for each cross-edge (u, v) we have $d[v] \le d[u] + 1$.
• for each back edge (u, v) we have $0 \le d[v] \le d[u]$.



BFS on an undirected graph (left) and on a directed graph (right),
back edges are blue, crossed edges are black

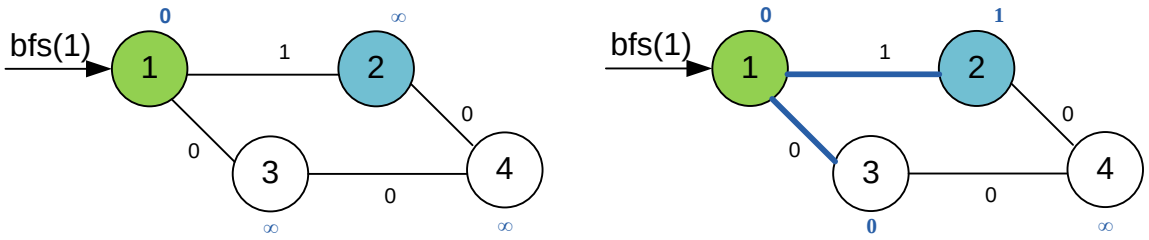
Applications of the algorithm
Search for the connected components in an undirected graph on $O(n + m)$.

Solving any game with the smallest number of moves, if each state of the system can be represented by a vertex of the graph, and the transitions from one state to another – edges of the graph.

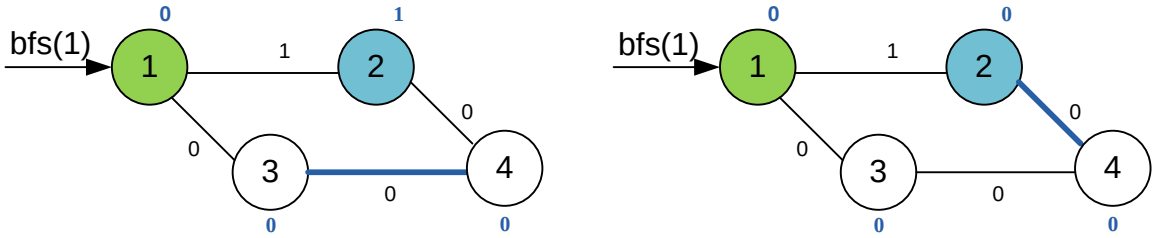
E-OLYMP **10056. Breadth first search 0 - 1** Undirected graph with edges of weight 0 and 1 is given. Find the shortest distance between s and d.

► 0 – 1 graph is given. It is sufficient to slightly modify the breadth-first search. If the distance to vertex is shorter than current found distance, then if the current edge is of zero weight, we add it to the front of the queue, otherwise we add it to the back of the queue.

Initialization: $dist[i] = \infty$ $(2 \le i \le 4)$, $dist[1] = 0$, queue = (1).
Consider the edges outgoing from vertex 1: 1 – 2 and 1 – 3. Set $dist[2] = 1$, $dist[3] = 0$, queue = (3, 2) because vertex 3 will be added to the start of the queue, and vertex 2 will be added to the end of the queue.



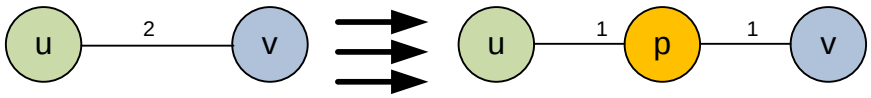
But the value dist[2] = 1 is not final. Pop the vertex 3 from the queue and relazate the edge 3 – 4, we get dist[4] = 0, queue = (4, 2) because vertex 4 will be added to the start of the queue. Now we ralaxate the edges outgoing from vertex 4 and after considering the edge 4 – 2 the value dist[2] becomes equal to 0.



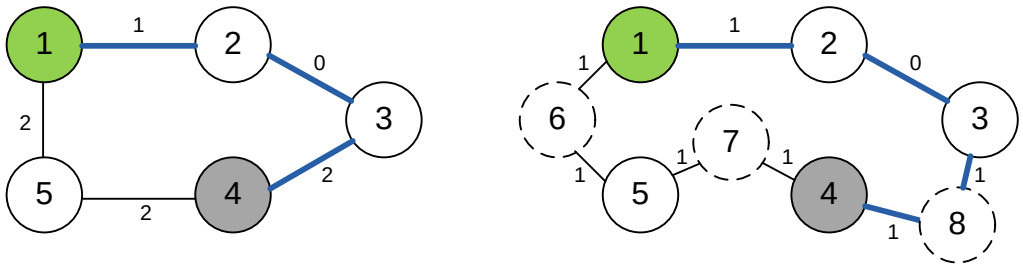
So the value dist[2] was assigned two different values: 1 and 0.

E-OLYMP 10058. Breadth first search 0 – 1 - 2 Undirected graph with edges of weight 0, 1 and 2 is given. Find the shortest distance between *s* and *d*.

► For each edge (*u*, *v*) of weight 2, we introduce a fictitious vertex *p* and replace it with two edges of weight 1: (*u*, *p*) and (*p*, *v*). Initially, the vertices of the graph are numbered 1, 2,…., *n*. Vertices *n* + 1, *n* + 2,… will be declared fictitious. Since there are at most *m* edges in the graph, there will be at most *m* fictitious vertices.



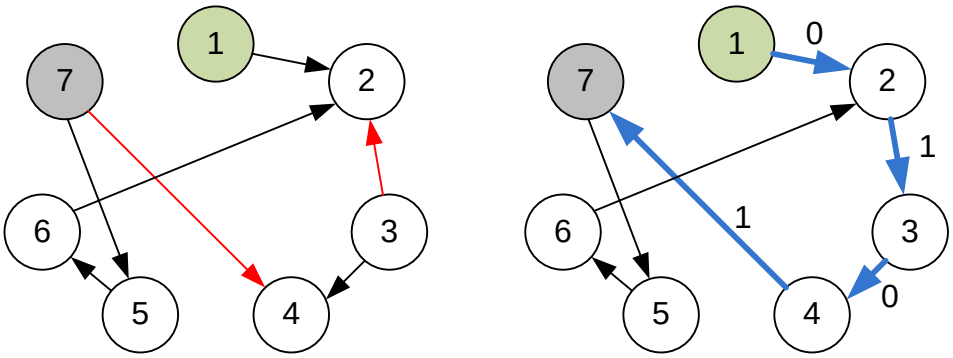
Problem is reduced to breadth-first search on 0 - 1 graph.



E-OLYMP 10048. Reverse the graph You are given a directed graph with *n* vertices and *m* edges. The vertices in the graph are numbered from 1 to *n*. What is the minimum number of edges you need to reverse in order to have at least one path from vertex 1 to vertex *n*.

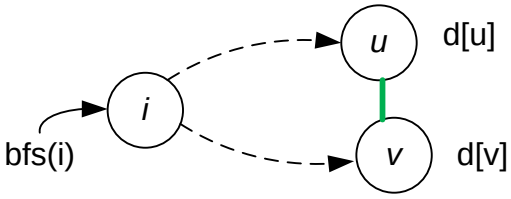
► Construct the 0-1 graph. Assign weight 0 to existing edges, and weight 1 to reversed edges. Run the breadth first search. The value of the shortest path from vertex 1 to vertex *n* equals to the least number of edges to reverse.

Graph given in a sample, has the form:



E-OLYMP [6427. Beehives] Find the shortest cycle in an undirected unweighted graph.

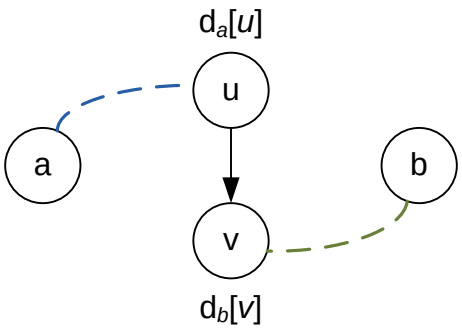
► Start bfs from each vertex. Once in the process we are trying to bypass the current vertex on some edge from an already visited vertex, we find the shortest cycle. Stop the bfs. Among all those found cycles (one from each run bypass) choose the shortest one.



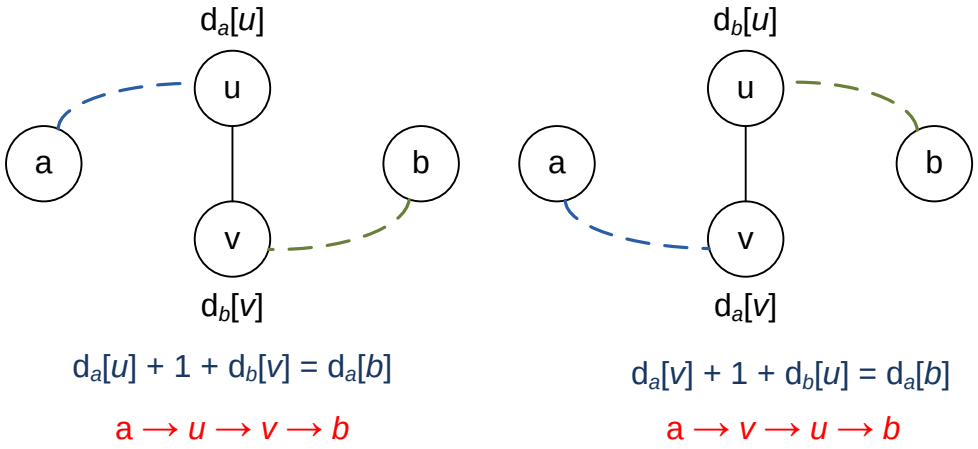
Let we run bfs(i). Let the shortest distance to u is d[u], the shortest distance to v is d[v]. During bfs we try to go from u to v and see that v is already visited. It means that we found a cycle of length d[u] + d[v] + 1.

Consider the **directed graph**. Find all the edges that lie on any shortest path between a given pair of vertices (a, b). To do this, run two breadth first searches: one from a (along the graph edges) and one from b (along the reversed graph edges). Let d_a[] be the array containing shortest distances obtained from the first BFS (from a) and d_b[] be the array containing shortest distances obtained from the second BFS (from b along the reversed edges). Now for every directed edge (u, v) it is easy to check whether that edge lies on any shortest path between a and b: the criterion is the condition

$$d_a[u] + 1 + d_b[v] = d_a[b]$$



In the case of **undirected graph** the path can be either $a \to u \to v \to b$ or $a \to v \to u \to b$.



$d_a[u] + 1 + d_b[v] = d_a[b]$

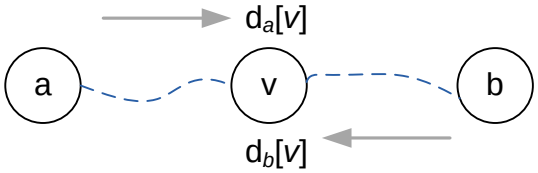
$a \to u \to v \to b$

$d_a[v] + 1 + d_b[u] = d_a[b]$

$a \to v \to u \to b$

Find all the vertices on any shortest path between a given pair of vertices (a, b). To do this, run two breadth first searches: one from a and one from b. Let $d_a[]$ be the array containing shortest distances obtained from the first BFS (from a) and $d_b[]$ be the array containing shortest distances obtained from the second BFS (from b). Now for each vertex v it is easy to check whether it lies on any shortest path between a and b:

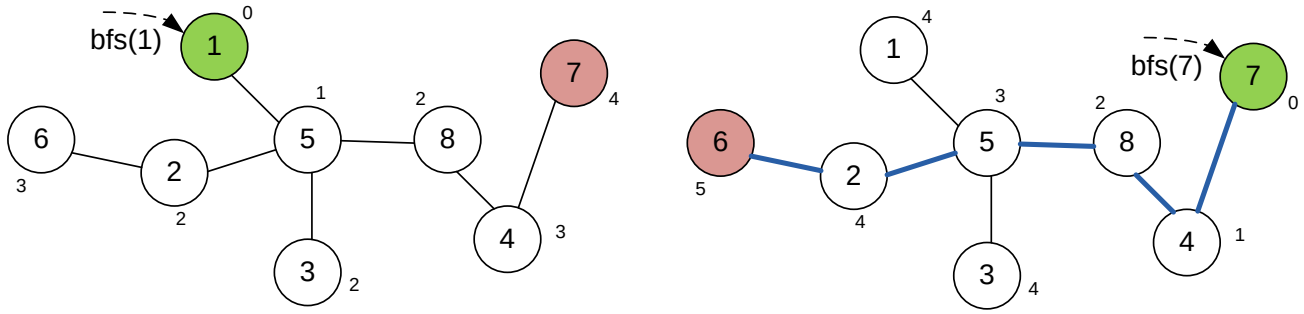
$$d_a[v] + d_b[v] = d_a[b]$$



E-OLYMP **10050. Longest path in a tree** Undirected weighted tree is given. Find the length of the longest path. Find two vertices the distance between which is maximum.

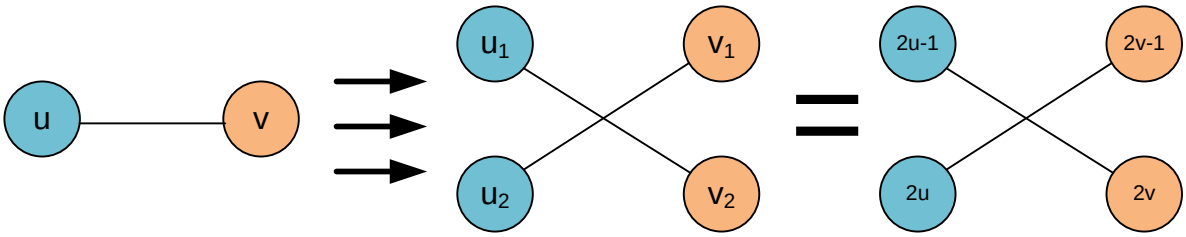
► Select any vertex, for example vertex 1 and run bfs. Find the farthest vertex from vertex 1, let it be v. Run bfs from vertex v and find the farthest vertex from it (let it be u). Path from v to u is the longest.

Example. Run bfs(1), the farthest vertex is 7. Run bfs(7), the farthest vertex is 6. Path from 7 to 6 is the longest.



E-OLYMP **10082. Shortest even path** Undirected unweighted graph is given. Find the shortest path between two vertices of even length.

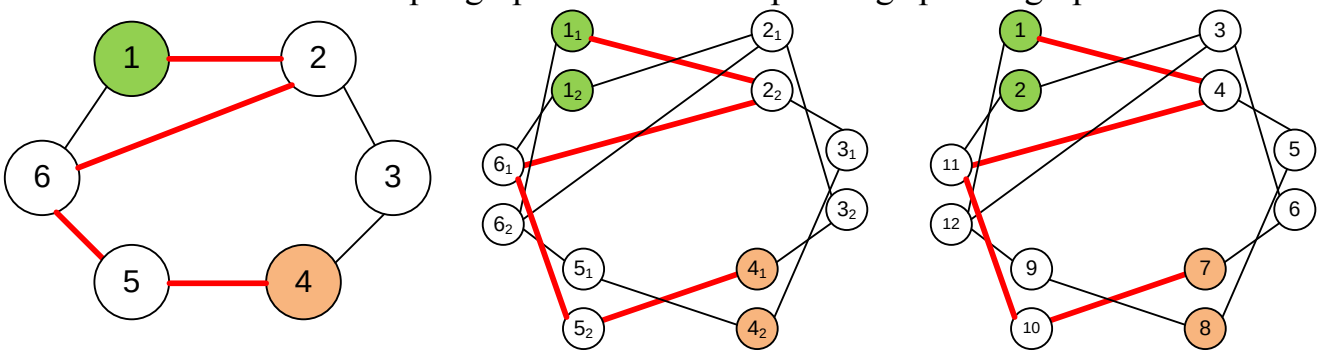
► Split each vertex v of the graph into two: v_1 and v_2. For each undirected edge (u, v) create two edges: (u_1, v_2) and (u_2, v_1).



For example, the vertex v can be associated with vertices $2 * v - 1$ and $2 * v$.

The shortest path between the vertices $2 * s - 1$ and $2 * d - 1$ will be the desired one and will have an even length.

Consider the next sample graph and the corresponding splitted graph:

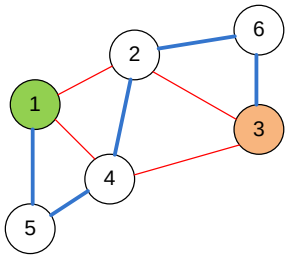


Let we start ***bfs*** from the vertex $v = 1_1$. Then

- if we'll arrive to the vertex x_1, the length of the path will be even;
- if we'll arrive to the vertex x_2, the length of the path will be odd;

Finding the shortest path of *even length* from 1 to 4 in the original graph is equivalent to finding the shortest path from 1_1 to 4_1 (or from 1 to 7) in the splitted graph.

E-OLYMP **10116. Almost shortest path** Undirected unweighted graph is given. Two vertices s and t are given. Let the shortest path from s to t be d. Almost shortest path from s to t is a path of minimum length that does not contain any edge along which a path of length d can pass. Find the length of the almost shortest path or print -1 if such path does not exist.



The shortest path between vertices 1 and 3 equals to 2. The edges that the shortest path can go through are highlighted in red. **Almost shortest path** is the shortest path that does not go along any of the red edges. The almost shortest path is highlighted in blue, its length is 5.

►Run bfs(s), fill shortest distances to distS[] array. Run bfs(t), fill shortest distances to distT[] array. Edge (u, v) is forbidden if and only if

distS[u] + 1 + distT[v] = distS[t] or distS[v] + 1 + distT[u] = distS[t]

Store all forbidden edges to set of edges (set of pairs).

Run bfs(s) again, but movement along the edge (u, v) is allowed if it is not forbidden. Shortest path to t along the not forbidden edges will be the almost shortest path.