Breadth First Search (BFS)

Though a breadth first search is an advanced technique, it's one that is reasonably intuitive and simply requires the use of a built-in data structure: a queue. More importantly, it's very important in problem solving and competitive programming, so we've decided to include it in this introductory camp.

Queue Data Structure

A queue data structure is essentially one that maintains a line (like one at the grocery store or theme park) in an efficient manner. We have two key operations for a queue:

1. Enqueue – add an item to the back of the queue (push in C++)
2. Dequeue – remove the front item in the queue (pop in C++)

So, during the life of a queue, whenever an item is added, it has to go to the back of the line. Whenever an item is processed from the queue, it has to be the front item (person/item waiting the longest). Both of these operations work in O(1) time. Here is the C++ documentation for using a queue object: https://cplusplus.com/reference/queue/queue/

BFS Overview

A breadth first search explores a network connected by roads (in computer science this is called a graph) in an iterative fashion, starting at some source location and then exploring all neighbors that are one step away, then two steps away, then three steps away, etc. until all reachable locations have been visited. By the end of the algorithm, we can mark each reachable location from the source with the shortest number of steps it takes to get to that location.

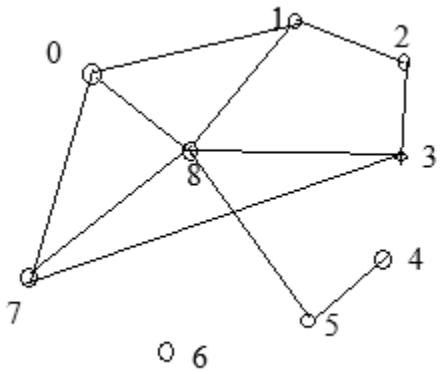
Formally, a graph can be visualized as a set of dots (vertices), of which some distinct pairs are connected by a line (edge). To travel from one dot to the other, we follow the lines. The fewest number of lines we have to move on to get from one dot to the other is the shortest distance between those dots (where we assume that each line/edge has a cost of 1).

The high-level idea behind the algorithm is as follows:

1. Create a distance array, marking the distance from the source vertex to every other vertex. Initialize this to all -1s to indicate that no path has been found. Set distance[source] = 0.
2. Create a queue (of vertices, which in practice is integers) adding the source vertex to the queue.
3. While the queue is not empty, do the following:
    a. Dequeue the front item of the queue, call this current.
    b. Loop through each neighboring item, next of current and do the following:
        i. If the item has not been visited yet (dist[next] == -1)
            1. Set dist[next] = dist[cur] + 1. (Need 1 step from cur…)
            2. Add next to the queue as a new place to explore from.
4. The state of the distance array at this point stores all shortest distances from source. If a stored distance is -1, that location isn't reachable from the source.

BFS Example

Let's take a look at an example run on the traditional graph shown below, starting from vertex 0:



Initially, our distance array looks like this and queue contains 0 only:

Index	0	1	2	3	4	5	6	7	8
Value	0	-1	-1	-1	-1	-1	-1	-1	-1

Once we enter the main BFS loop, we first dequeue 0 and loop through its neighbors, 1, 7 and 8. Since each of these has a current distance of -1, that means each is "new" and we process each vertex in the if statement updating its distance and adding each to the queue. After we do this, our queue looks like:

Queue: 1, 7, 8 (back)

And our distance array now looks like:

Index	0	1	2	3	4	5	6	7	8
Value	0	1	-1	-1	-1	-1	-1	1	1

We go through the BFS loop a second time, this time dequeing vertex 1. Its neighbors are 0, 2 and 8, but of these, only the distance[2] = -1. Thus, this is the only new vertex we process, updating its distance from vertex 0 and adding it to the queue. Our new queue and distance array are:

Queue: 7, 8, 2 (back)

Index	0	1	2	3	4	5	6	7	8
Value	0	1	2	-1	-1	-1	-1	1	1

Next, we dequeue 7 and look at its neighbors 0, 3 and 8. Only distance[3] = -1, so we only process this vertex. After we do so, the queue and distance array look like this:

Queue: 8, 2, 3 (back)

Index	0	1	2	3	4	5	6	7	8
Value	0	1	2	2	-1	-1	-1	1	1

Now, we process vertex 8. Its neighbors are 0, 1, 3, 5 and 7. Of all five of these vertices, only distance[5] = -1. We only process this vertex, updating its distance and adding it to the queue:

Queue: 2, 3, 5 (back)

Index	0	1	2	3	4	5	6	7	8
Value	0	1	2	2	-1	2	-1	1	1

In the next two iterations of the algorithm, 2 and 3 get dequeued, but none of their neighbors has a distance equal to -1, so no processing of vertices happens on those two iterations, leading to the same exact distance array and a queue that only contains 5. When 5 is dequeued, its neighbors are 4 and 8, but of these two neighbors only distance[4] = -1. Processing this vertex leads to the following state of the queue and distance array:

Queue: 4 (back)

Index	0	1	2	3	4	5	6	7	8
Value	0	1	2	2	3	2	-1	1	1

Finally, 4 gets dequeued, but has no unvisited neighbors, at which point the while loop ends since the queue is empty. The distances shown above are indeed the shortest distances from vertex 0 in the example graph. And as is obvious visually, there's no path from vertex 0 to vertex 6.

Traditional BFS Code

In this section we'll look at traditional BFS code on a graph. The input format is as follows:

First line contains two space separated integers, ***n***, the number of vertices in the graph and ***e***, the number of edges in the graph, respectively.

The following ***e*** lines each contain two distinct positive integers, ***u*** and ***v***, in between 1 and ***n***, inclusive, indicating that vertices ***u*** and ***v*** are connected by an edge. Each of these pairs are distinct.

The program will read in the graph and then run a BFS from every vertex. For each BFS, the corresponding distance array will be printed out.

Here is the code:

```
// Arup Guha
// 2/7/2023
// Illustration a BFS in C++
using namespace std;
#include <bits/stdc++.h>

int numV;
vector< vector<int> > graph;
```

```
int* bfs(int v);

int main() {
    int numE, v1, v2;
    cin >> numV >> numE;
    graph.resize(numV);

    // Read in the edges - add both ways.
    for (int i=0; i<numE; i++) {
        cin >> v1 >> v2;
        graph[v1-1].push_back(v2-1);
        graph[v2-1].push_back(v1-1);
    }

    // Run a BFS from each vertex.
    for (int i=0; i<numV; i++) {
        int* dist = bfs(i);

        // Prints out distances from vertex i to all other vertices.
        cout << "Distances from " << i << " to each vertex." << endl;
        for (int j=0; j<numV; j++)
            cout << dist[j] << " ";
        cout << endl;

        delete [] dist;
    }
    return 0;
}

int* bfs(int v) {

    // Set up distances.
    int* dist = new int[numV];
    for (int i=0; i<numV; i++) dist[i] = -1;
    dist[v] = 0;

    queue<int> q;
    q.push(v); // Add source vertex to queue.

    // Run BFS.
    while (q.size() > 0) {
        int cur = q.front(); q.pop(); // Get next.

        // Enqueue all new neighbors.
        for (vector<int>::iterator x = graph[cur].begin(); x<graph[cur].end();
x++) {

            // Been here before
            if (dist[*x] != -1) continue;
            // Add to queue and update distance of next vertex, *x.
            q.push(*x);
            dist[*x]= dist[cur] + 1;
        }
    }
    return dist;
}
```

A few notes about the code:

1. Most competitive programming problems label vertices 1 to n, but in code we label them 0 to n-1, so right when we read in the vertices, we subtract 1.

2. Edges are undirected in most problems, which means that whenever an edge is given between u and v, you have to add both the edge u→v and v→u. This is because v is next to u and u is next to v. One of the most common errors beginning programmers make is to forget adding undirected edges both ways in their graph.

3. If you don't like the iterator syntax of finding all the neighbors of a vertex, you can use a regular for loop with an index.

Kattis Problem: Elevator Trouble
Here is a link to the problem:

https://open.kattis.com/problems/elevatortrouble

Upon reading the problem it's not obvious that a graph is involved in the problem. But, if we imagine that each floor in the building is a vertex and we put **directed** edges from vertex u to vertex v if we can go from floor u to floor v on a single button press, then we can see that even though a graph isn't explicitly given in the input, the structure of the graph is implicitly described in the problem.

A second observation is that we can run a breadth first search to solve this problem without ever explicitly storing the graph first. Instead, whenever we dequeue an item (which means we have reached a particular floor), instead of looking at a stored graph to figure out which items are adjacent to the current one, we can use the rules in the problem to determine which floor(s) we can reach from the current floor. In this manner, we can figure out the fewest number of button presses to get to our desired floor. Also, since we don't need to know the number of button presses to every floor, we can break out of our breadth first search after we've reached our intended destination.
Here's the code:

```
// Arup Guha
// 12/15/2022
// Solution to 2011 NCPC Problem: Elevator Trouble
// Illustrate BFS for ICPC Lecture

using namespace std;

#include <bits/stdc++.h>

int main() {

    // Read input make floors 0-based.
    int floors, start, end, up, down;
    cin >> floors >> start >> end >> up >> down;
    start--;
    end--;
```

```
    // Store distances here. Use -1 to mean not reached.
    int* dist = new int[floors];
    for (int i=0; i<floors; i++) dist[i] = -1;

    // Set up BFS.
    queue<int> q;
    q.push(start);
    dist[start] = 0;

    // Run bfs.
    while (q.size() > 0) {

        int cur = q.front();
        q.pop();
        if (cur == end) break;

        // Try going up.
        int next = cur + up;
        if (next >= 0 && next < floors && dist[next] == -1) {
            q.push(next);
            dist[next] = dist[cur] + 1;
        }

        // And down.
        next = cur - down;
        if (next >= 0 && next < floors && dist[next] == -1) {
            q.push(next);
            dist[next] = dist[cur] + 1;
        }
    }

    // Ta da!
    if (dist[end] != -1) cout << dist[end] << endl;
    else                       cout << "use the stairs" << endl;

    delete [] dist;

      return 0;
}
```

The key observation here is that when we read in the input, no graph is formed. Instead, we replicate the structure of the breadth first search. When we dequeue an integer in our loop, we use the rules for moving up and down to determine the floors we can reach with a single button press from the current floor we are at (cur in the code).

Use of DX/DY Arrays

Many BFS problems in programming competitions are on two dimensional grids. There's lots of flavor text, but most of these problems have some sort of movement rules, a starting position and some sort of goal position (or positions). A few problems of this nature are an exact BFS without an frills, but most of them require some observations that necessitate minor modifications to a regular BFS. Regardless, almost all of these problems delineate movement rules in a grid. If we are at some location (x, y) in the grid, we can reach some list of positions dictated by directions of movement. Perhaps, we can move up, down, left or right, which means that from (x, y), we can,

in one move, get to (x-1, y), (x, y-1), (x, y+1) and (x+1, y). In other problems, we can move diagonally as well, meaning that we can move to eight possible locations. These are the two most common specifications for movement, but others exist; you just have to read the problem carefully!

No matter, what the specification is, in each case, always create two parallel vectors that store the possible offset of movements in X and Y as follows:

```
const vector<int> DX = {-1,0,0,1};
const vector<int> DY = {0,-1,1,0};
```

In this example, we encode the movement for up, left, right and down respectively, assuming that x is up and down and y is left and right.

If our current position is (curx, cury), here is how we iterate through all possible next locations:

```
for (int i=0; i<DX.size(); i++) {
    int nextx = curx + DX[i];
    int nexty = cury + DY[i];

    // Skip if (nextx, nexty) is out of bounds, previously visited
    // or illegal.

    // If new and valid, add to queue.
}
```

Now we are ready to look at a couple Kattis problems that involve breadth first searches on grids.

Kattis Problem: Escape Wall Maria
Here is a link to the problem:

https://open.kattis.com/problems/escapewallmaria

In this problem, we can move up, down, left and right, usually. The exceptions are fire squares, which we can't move to at all and "special letter squares". For this latter group, we can only move onto these squares from one previous position. For example, to move to a square storing 'U', we must move to it from the square directly above it. (Thus, if a square has an 'U', we only want to enqueue it if we are coming from directly above…)

We can think of this problem similarly to the elevator problem. We'll never form the actual graph. Rather, we'll run the BFS on the grid, and when we are processing a current grid square in our BFS loop, we'll use the rules in the problem to determine which grid squares we can visit next. This will require some additional checks in the for loop structure with the DX/DY vectors specified above.

One other idea will be introduced in this solution: we can store all the items in a 2D vector in an equivalent 1D vector. Thus, the distance array in this problem will be stored as one dimensional while input grid will be two dimensional. This isn't necessary, but just an idea that some

competitive programmers use, so it's illustrated here. If a grid has r rows and c columns, then the item in row x, column y has index cx + y, since each of the previous x rows has exactly c items in it. Similarly, if the index into the distance array is idx, this corresponds to row idx/c and column idx%c. (Integer division reveals the number of full rows we pass, and mod reveals how far along we are in the last row.)

Here is the solution:

```
// Arup Guha
// 6/21/2025
// Solution to Kattis Problem: Escape Wall Maria

using namespace std;
#include <bits/stdc++.h>

const vector<int> DX = {-1,0,0,1};
const vector<int> DY = {0,-1,1,0};

bool inbounds(int x, int y, int r, int c);

int main() {

    int maxT, r, c;
    cin >> maxT >> r >> c;

    // Read the grid.
    int s = -1;
    vector<string> grid(r);
    for (int i=0; i<r; i++) {
        cin >> grid[i];

        // Look for the starting vertex.
        for (int j=0; j<c; j++)
            if (grid[i][j] == 'S')
                s = c*i + j;
    }

    // Initialize distance array.
    vector<int> dist(r*c, -1);
    dist[s] = 0;

    // Set up queue.
    queue<int> q;
    q.push(s);

    int res = -1;

    // I'll run a regular BFS.
    while (q.size() > 0) {

        // Get next location.
        int cur = q.front(); q.pop();

        // If we still haven't got out, it's too late.
        if (dist[cur] > maxT) break;
```

```
        // Where we are.
        int curX = cur/c;
        int curY = cur%c;

        // We got out!
        if (curX == 0 || curX == r-1 || curY == 0 || curY == c-1) {
            res = dist[cur];
            break;
        }

        // Try going all four directions.
        for (int i=0; i<DX.size(); i++) {

            // Next place we can go to.
            int nextX = curX + DX[i];
            int nextY = curY + DY[i];
            int next = c*nextX + nextY;

            // Not inbounds.
            if (!inbounds(nextX, nextY, r, c)) continue;

            // Fire!
            if (grid[nextX][nextY] == '1') continue;

            // Check if we didn't come from the right direction.
            // This is tricky; make sure you have the right value of i!
            if (grid[nextX][nextY] == 'U' && i != 3) continue;
            if (grid[nextX][nextY] == 'D' && i != 0) continue;
            if (grid[nextX][nextY] == 'R' && i != 1) continue;
            if (grid[nextX][nextY] == 'L' && i != 2) continue;

            // Been here before.
            if (dist[next] != -1) continue;

            // Mark distance and add to queue.
            dist[next] = dist[cur] + 1;
            q.push(next);
        }
    } // end BFS.

    // Output accordingly.
    if (res != -1)
        cout << res << endl;
    else
        cout << "NOT POSSIBLE" << endl;

    return 0;
}

bool inbounds(int x, int y, int r, int c) {
    return x>=0 && x<r && y>=0 && y<c;
}
```

Kattis Problem: Grid
Here is a link to the problem:

https://open.kattis.com/problems/grid

In this problem, instead of moving one step in the four cardinal directions, the square which you are at determines how far in each direction you can jump. Thus, instead of adding DX[i] and DY[i], respectively to your current x and y position, you should simply add mult*DX[i] and mult*DY[i], where mult is the specified value at the square you are currently at. Here's the solution to this problem:

```
// Arup Guha
// 6/21/2025
// Solution to Kattis Problem: Grid
// https://open.kattis.com/problems/grid

using namespace std;
#include <bits/stdc++.h>

const vector<int> DX = {-1,0,0,1};
const vector<int> DY = {0,-1,1,0};

bool inbounds(int x, int y, int r, int c);

int main() {

    int r, c;
    cin >> r >> c;

    // Read the grid.
    vector<string> grid(r);
    for (int i=0; i<r; i++)
        cin >> grid[i];

    // Initialize distance array. This time we'll do 2D
    vector<vector<int>> dist(r, vector<int>(c, -1));
    dist[0][0] = 0;

    // Set up queue.
    queue<pair<int,int>> q;
    q.push(pair<int,int>{0,0});

    // I'll run a regular BFS.
    while (q.size() > 0) {

        // Get next location.
        pair<int,int> cur = q.front(); q.pop();

        // We made it!
        if (cur.first == r-1 && cur.second == c-1) break;

        // Try going all four directions.
        for (int i=0; i<DX.size(); i++) {
```

```
            // Next place we can go to.
            int mult = grid[cur.first][cur.second] - '0';
            pair<int,int> next;
            next.first = cur.first + mult*DX[i];
            next.second = cur.second + mult*DY[i];

            // Not inbounds.
            if (!inbounds(next.first, next.second, r, c)) continue;

            // Been here before.
            if (dist[next.first][next.second] != -1) continue;

            // Mark distance and add to queue.
            dist[next.first][next.second] = dist[cur.first][cur.second] + 1;
            q.push(next);
        }
    } // end BFS.

    // Ta da!
    cout << dist[r-1][c-1] << endl;

    return 0;
}

bool inbounds(int x, int y, int r, int c) {
    return x>=0 && x<r && y>=0 && y<c;
}
```

In this implementation, instead of doing a distance array of one dimension, we use a more natural distance array of two dimensions. This transforms the queue to be a queue of pairs, since each location is now stored as an ordered pair. This requires a bit more typing in some places but is perhaps, as mentioned, more natural.

Also notice how similar this solution is to the previous problem. In fact, this file was created as a minor edit of the previous file. (I copied and pasted the old file and mostly deleted things, and changed a couple minor things, such as the multiplier code.)