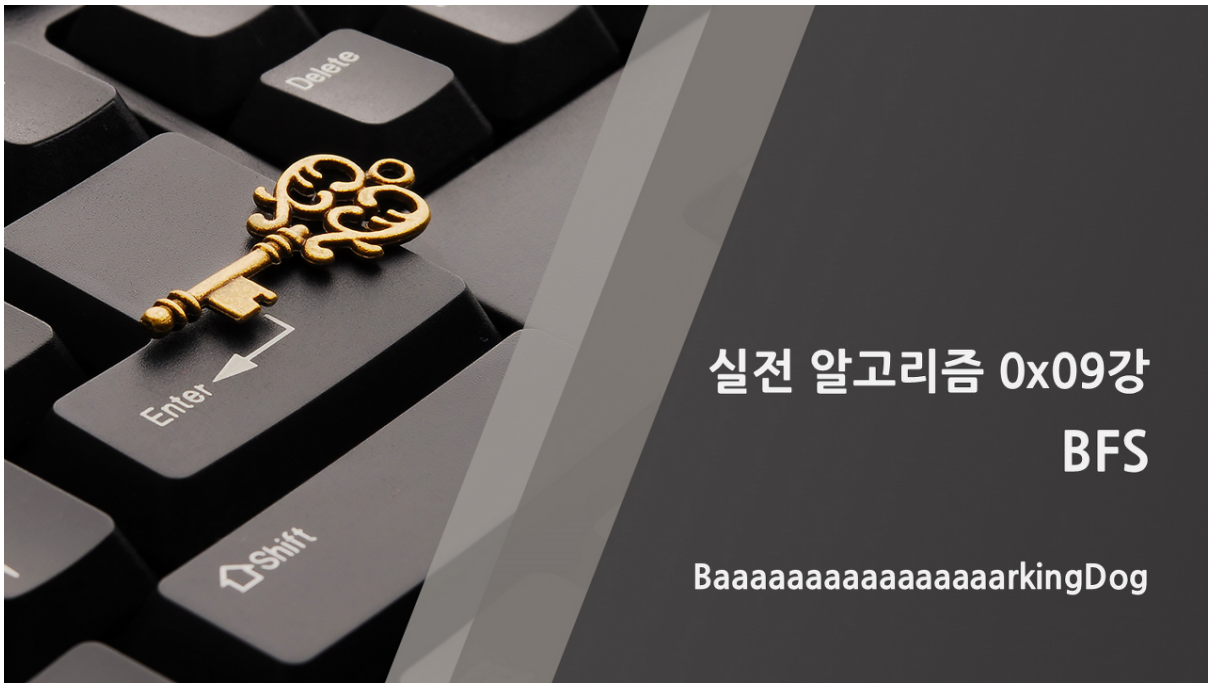
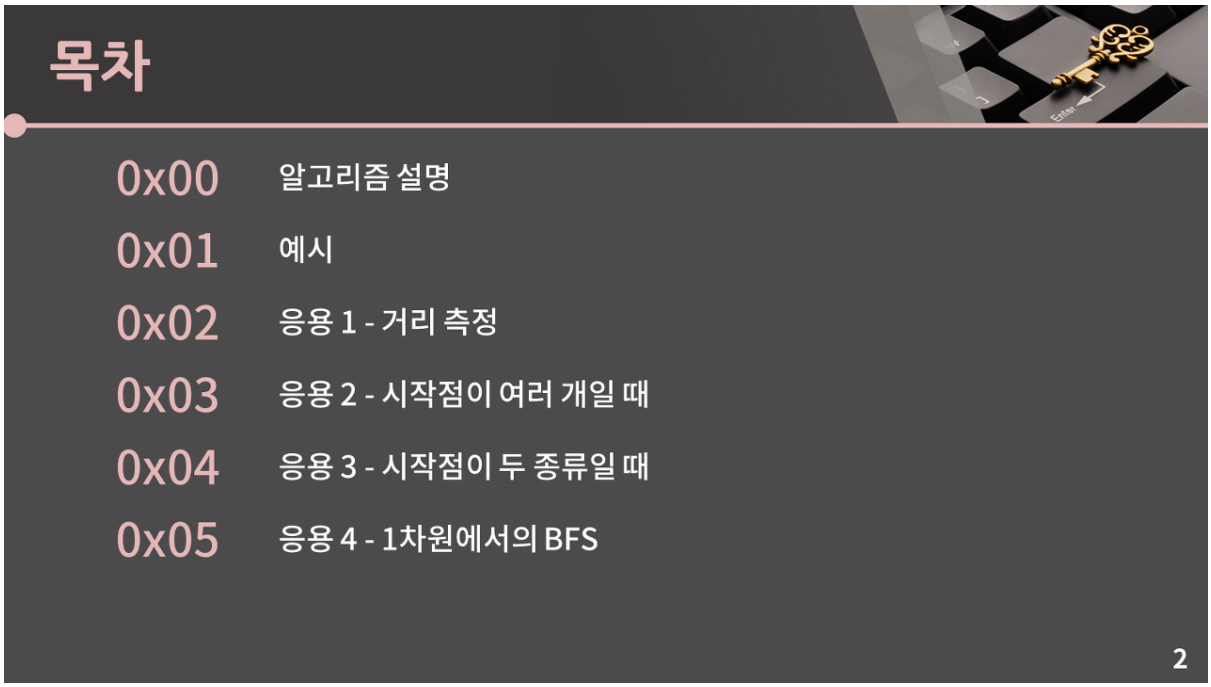
바킹독 0x09 BFS(2) 응용

태그	
공개여부	☑
날짜	
작성일자	





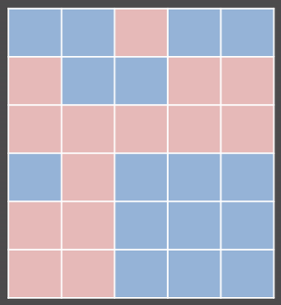

```
//BFS SKELETON
#include <bits/stdc++.h>
using namespace std;
#define X first
```

```
#define Y second // pair 함에서 first, second를 줄여서 쓰기 위해서 사용

int board[502][502] =
{{1,1,1,0,1,0,0,0,0,0},
 {1,0,0,0,1,0,0,0,0,0},
 {1,1,1,0,1,0,0,0,0,0},
 {1,1,0,0,1,0,0,0,0,0},
 {0,1,0,0,0,0,0,0,0,0},
 {0,0,0,0,0,0,0,0,0,0},
 {0,0,0,0,0,0,0,0,0,0} }; // 1이 파란 칸, 0이 빨간 칸에 대응

 bool vis[502][502]; // 해당 칸 방문여부를 저장하는 변
 int n = 7, m = 10; // n = 행의 수, m = 열의 수
 int dx[4] = {1, 0, -1, 0};
 int dy[4] = {0, 1, 0, -1}; // 상하좌우에 있는 칸을 쉽게 접근하기 위해 사용하는 변수

int main(void){

  ios::sync_with_stdio(0);
  cin.tie(0);
  queue<pair<int,int>> Q; // int값 한 쌍(한 지점, 좌표)을 저장하는 큐 선언
  vis[0][0] = 1; // (0, 0)을 방문했다
  Q.push({0,0}) // 큐에 시작점인 (0, 0) 삽입
  while(!Q.empty())
  {
    pair<int,int> cur = Q.front(); Q.pop(); // cur = 현재 위치를 저장하는 변수, 큐에서 좌표값을 가져온 뒤 큐에서 제거해준다.
     for(int dir = 0; dir < 4; dir++) // cur에서 인접한 상하좌우 칸을 살펴볼 것이다.
     {
      int nx = cur.X + dx[dir]
      int ny = cur.Y + dx[dir] // cur에서 인접한 상하좌우칸의 좌표가 nx, ny에 들어간다.

      //예외처리
      if(nx < 0 || nx >= n || ny < 0 || ny >= m) continue; // board 범위 밖일 경우(행은 n, 열은 m줄 존) 넘어감
      if(vis[nx][ny] || board[nx][ny] != 1) continue; // 이미 방문한 칸이거나 파란 칸이 아닐 경우 넘어감

      vis[nx][ny] = 1; // 예외처리에 걸리지 않는 경우! (nx, ny)를 방문했다고 명시해주고
      Q.push({nx,ny}); // 인접했던 그 칸의 좌표값을 큐에 넣어준다.
     }
  }

}



/*
about BFS
1. 시작하는 지점을 큐에 push하고 방문했다는 표시를 남김.
2. 큐에서 행렬(좌표)값이 저장된 원소를 꺼내고, 해당 지점의 상하좌우로 인접한 칸에 대하여 3번 진행
3. 해당 칸을 이전에 방문했다면 아무 동작도 하지 않는다, 만약 방문한 적이 없다면 방문표시를 남기고 해당 칸의 좌표를 큐에 삽입한다.
4. 큐가 빌 때까지 2번을 반복한다.
모든 칸이 큐에 1번씩 들어가므로 시간복잡도는 칸이 N개일 때 O(n)이 된다.
board, visit, n/m, dx/dy, pair queue, pair current, nx/ny
*/
```

0x01 예시

BOJ 1926번: 그림

1. 상하좌우로 연결된 그림의 크기를 알아내기
2. 도화지에 있는 모든 그림을 찾아내기



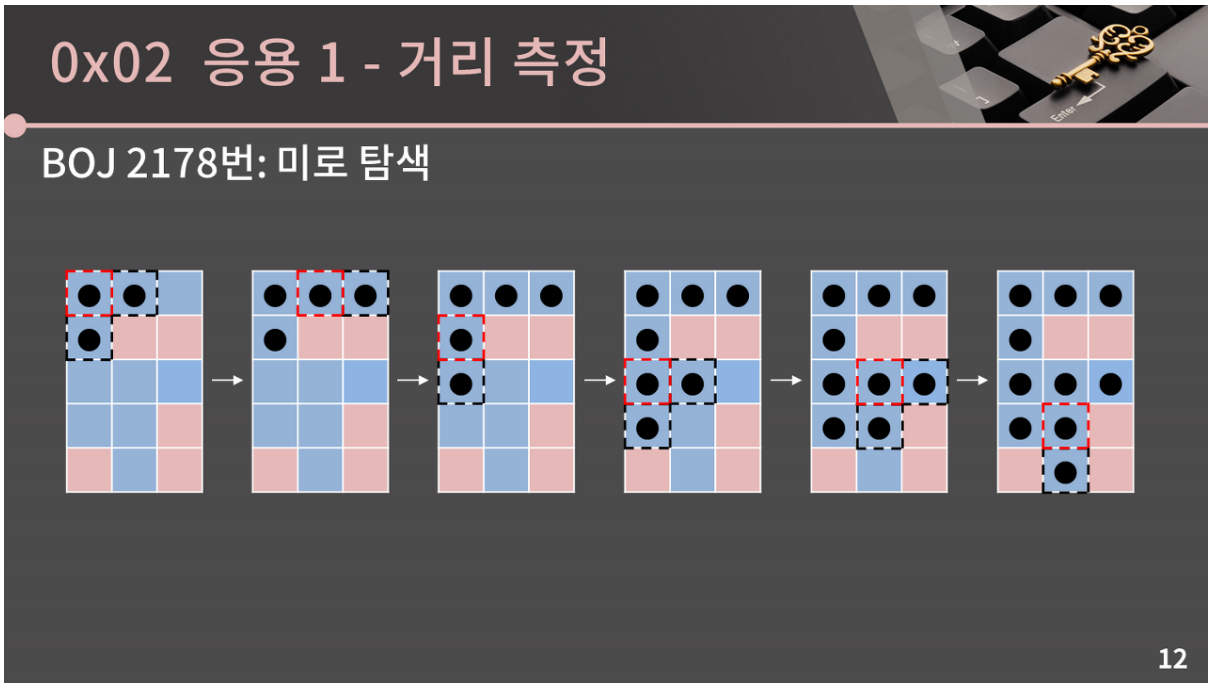
10

1. QUEUE에서 좌표를 POP할 때, SIZE는 1씩 증가한다. 이를 고려하여 코드를 작성하면 됨.
2. bfs의 시작점이 될 수 있는 곳이 여러 곳(4군데), 시작점을 모두 찾아야 함.

   →이중 for문을 이용하여 시작점이 될 수 있는 지점들을 전부 찾아낸다. **파란 칸이고 아직 방문하지 않은 칸을 찾아낸다.**

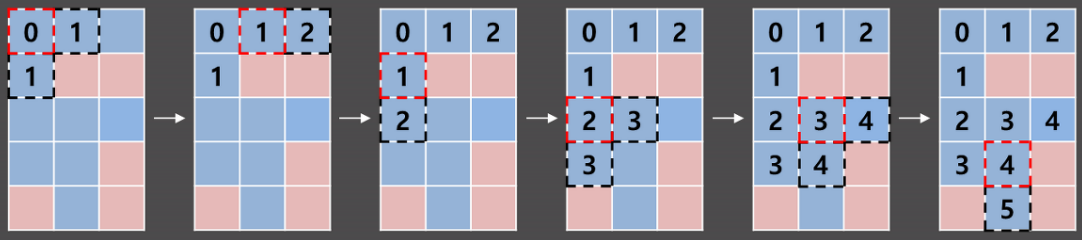
```
#include <bits/stdc++.h>
using namespace std;
#define X first
#define Y second // pair 함에서 first, second를 줄여서 쓰기 위해서 사용
int board[502][502]; // 1이 파란 칸, 0이 빨간 칸에 대응
bool vis[502][502]; // 해당 칸을 방문했는지 여부를 저장
int n,m; // 행렬, i는 행 j는 열
int dx[4] = {1,0,-1,0};
int dy[4] = {0,1,0,-1}; // 상하좌우 네 방향을 의미
int main ()
{
  ios::sync_with_stdio(0);
  cin.tie(0);
  cin >> n >> m;

  int mx = 0; // 그림의 최대 넓이값
  int num = 0; // 그림 수

  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++)
      cin >> board[i][j]; // 보드에 그림그리기

  for(int i = 0; i < n; i++) // 이중 for문으로 시작점 (i,j)찾기, 시작점이 되려면 방문한 적 x, 색칠(1)되어 있어야 함
  {
    for(int j = 0; j < m; j++)
    {
      if( vis[i][j] == 1 || board[i][j] == 0) continue; // continue -> 다음 열로 넘어간다.
      num++; // 그림수 증가

      queue<pair<int,int>> Q; // 좌표를 담을 큐 설정

      vis[i][j] = 1; //(i,j)를 시작점으로 삼고 BFS를 준비한다.
      Q.push({i,j});

      int area = 0; // 그림의 넓이

      while(!Q.empty()) // BFS start
      {
        area++; // queue에 있는 원소 하나를 불러올 때마다 area는(그림의 크기) 1씩 커진다.
        pair<int,int> cur = Q.front(); Q.pop();
        for(int dir = 0; dir < 4; dir++)
        {
          int nx = cur.X + dx[dir];
          int ny = cur.Y + dy[dir];

          if(nx >= n || ny >= m || nx < 0 || ny < 0) continue; // 범위를 벗어난 경우 -> 넘어가고 다른 인접 칸 체크
```

```
          if(vis[nx][ny] == 1 || board[nx][ny] == 0) continue; // 방문 한 적 있거나 그림칸이 아닌(0) 경우 -> 넘어가고 다른 인접 칸 체크

          vis[nx][ny] = 1; // 인접 칸에 방문표시
          Q.push({nx,ny}); // 큐에 삽입
        }
      } // bfs end, while 문에서 빠져나온 후 이중 for 문으로 돌아가 새로운 bfs 시작지점을 찾는다.

      mx = max(mx, area); // area가 mx보다 큰 경우 mx에 area 대입. 그림의 크기가 가장 큰 경우를 구하는 함수
    }

  }
   cout << num << '\n' << mx;


}


/*
about BFS
1. 시작하는 지점을 큐에 push하고 방문했다는 표시를 남김.
2. 큐에서 행렬(좌표)값이 저장된 원소를 꺼내고, 해당 지점의 상하좌우로 인접한 칸에 대하여 3번 진행
3. 해당 칸을 이전에 방문했다면 아무 동작도 하지 않는다, 만약 방문한 적이 없다면 방문표시를 남기고 해당 칸의 좌표를 큐에 삽입한다.
4. 큐가 빌 때까지 2번을 반복한다.
모든 칸이 큐에 1번씩 들어가므로 시간복잡도는 칸이 N개일 때 O(n)이 된다.

board, visit, n/m, dx/dy, pair queue, pair current, nx/ny


1926
상하좌우로 연결된 그림의 크기를 알아내라. -> queue에 있는 원소 하나를 불러올 때마다 area는(그림의 크기) 1씩 커진다.
도화지에 있는 모든 그림을 찾아내라. -> 이중 for문을 사용하여 적절한 시작지점을 골라(그림의 일부면서, 아직 방문한 적이 없는 부분) bfs를 시작한다.
*/
```



0x02 응용 1 - 거리 측정

BOJ 2178번: 미로 탐색



13

> 다차원 배열에서의 거리 측정

> 미로의 좌측 상단으로부터 우측 하단으로 가는 최단 경로의 길이를 찾는 문제. BFS를 이용해 시작점에서 연결된 다른 모든 점으로의 최단 경로를 찾을 수 있다. **(0, 0)에서 사방으로 퍼져나가는 것과 같은 느낌**

→ 방문 여부와 거리를 저장하는 2차원 배열을 선언(-1로 초기화)

```
#include <bits/stdc++.h>
using namespace std;
#define X first
#define Y second

int dx[4] ={1, 0, -1, 0};
int dy[4] ={0, 1, 0, -1};

int main()
{
 int n, m; // 행과 열의 수
 cin >> n >> m;
 int dist[n][m]; // 방문여부와 동시에 시작점으로부터의 거리를 저장하는 2차원 배열

 string board[n]; // 입력데이터가 공백으로 구분되어있지 않아서 스트링사용 board[n][m] -> n행에있는 원소에서 m번째 글자를 가져옴

  for(int i = 0; i < n; i++) // n줄 작성
    cin >> board[i];

  for(int i = 0 ; i < n; i++) // dist 전부 -1로 초기화하면 방문여부와 시작점으로 부터의 거리 모두 기록할 수 있다. -1 == 아직 방문 x
    fill(dist[i], dist[i]+m, -1); //dist[0] == dist[0][0], dist[1] == dist[1][0], dist[0]+1 == dist[0][1]
    //dist[i] 부터 dist[i]+m 직전까지(dist[i]+m은 포함하지 않음) -1로 초기화

  queue<pair<int,int>> Q;

  Q.push({0, 0});
  dist[0][0] = 0; // bfs 시작을 위한 준비단계

  while(!Q.empty()) // bfs start
  {
    auto cur = Q.front(); Q.pop(); // auto cur == pair<int, int> cur
    for(int dir = 0; dir < 4; dir++) // 인접한 칸에 대하여 방문
    {
      int nx = cur.X + dx[dir];
      int ny = cur.Y + dy[dir];

      if(nx < 0 || ny < 0 || nx >= n || ny >= m) // 범위 체크
      if(board[nx][ny] != '1' || dist[nx][ny] != -1 ) // 보드에서 1인지, 방문 여부 체크
      dist[nx][ny] = dist[cur.X][cur.Y] + 1; // 방문을 한다는 표시를 함과 동시에 시작점으로 부터의 거리를 삽입
      Q.push({nx, ny});
```

```
    }
  }

  cout << dist[n-1][m-1] + 1;

}

// 거리를 저장할 dist 배열을 두고 상하좌우의 칸을 볼 때 값을 잘 넣어주면 됩니다

/*
about BFS
1. 시작하는 지점을 큐에 push하고 방문했다는 표시를 남김.
2. 큐에서 행렬(좌표)값이 저장된 원소를 꺼내고, 해당 지점의 상하좌우로 인접한 칸에 대하여 3번 진행
3. 해당 칸을 이전에 방문했다면 아무 동작도 하지 않는다, 만약 방문한 적이 없다면 방문표시를 남기고 해당 칸의 좌표를 큐에 삽입한다.
4. 큐가 빌 때까지 2번을 반복한다.
모든 칸이 큐에 1번씩 들어가므로 시간복잡도는 칸이 N개일 때 O(n)이 된다.
board, visit, n/m, dx/dy, pair queue, pair current, nx/ny
*/


/*
16~18 ex1

string 배열

board[0] "101111"
board[1] "101010"
board[2] "101011"
board[3] "111011"

board[0][0] -> 1
board[0][1] -> 0 ...
*/
```

> 토마토가 익어가는 상황 자체가 BFS를 하는 것과 똑같기도 하고, 토마토가 다 익기까지 필요한 최소 일수를 구하려면 **모든 익지 않은 토마토들에 대해 가장 가깝게 위치한 익은 토마토까지의 거리를 구해야한다**는 관점에서 살펴봐도 마찬가지로 BFS를 활용할 수 있겠다는 생각이 들 것입니다.

> 만약 익은 토마토가 1개였다면 앞의 미로 탐색 문제랑 비슷하게 익은 토마토가 있는 곳을 시작점으로 해서 BFS를 돌려 쉽게 해결이 가능할텐데, **이 문제에서는 익은 토마토의 갯수가 여러 개일 수 있습니다**. 각 익은 토마토들에 대해 해당 위치를 시작점으로 하는 BFS를 한 번씩 다 돌리는 방법을 떠올릴 수 있지만, 그러면 BFS의 시간복잡도가 O(NM)이고 익은 토마토 또한 최대 NM개가 있을 수 있으니 총 O(N^2M^2)이 되어 시간 내로 해결이 안될 것입니다. 과연 어떻게 문제를 해결할 수 있을까요?

시작점이 여러 개인 BFS를 돌 수 있어야합니다. 그리고 그 방법은 그냥 모든 시작점을 큐에 넣고, 앞에서 한 것과 똑같이 BFS를 돌면 끝입니다.

```
#include <bits/stdc++.h>
using namespace std;
#define X first
#define Y second // pair 함에서 first, second를 줄여서 쓰기 위해서 사용

int tomato[1002][1002]; // 맥스값은 1000이지만 넉넉하게 1002로 준다
int n, m; // 행렬
int dist[1002][1002]; // 익는 데 까지 남은 일
int dx[4] = {1, 0, -1, 0};
int dy[4] = {0, 1, 0, -1};


int main(void)
{
  ios::sync_with_stdio(0);
  cin.tie(0);
  queue<pair<int,int>> Q;
  cin >> m >> n;

  for(int i = 0; i < n; i++) // 행
    for(int j = 0; j < m; j++) // 열
    {
    cin >> tomato[i][j];
    if(tomato[i][j] == 1) // 익은 토마토
      Q.push({i,j});
    if(tomato[i][j] == 0) // 익지 않은 토마토
      {
        dist[i][j] = -1; // 안익은 토마토를 익은토마토와 토마토가 없는 상황과 구분하기 위해
      }
    }


    while(!Q.empty()) // bfs start
    {
      auto cur = Q.front(); Q.pop();
      for(int dir = 0; dir < 4; dir++)
      {
        int nx = cur.X + dx[dir];
        int ny = cur.Y + dy[dir];

        if(nx < 0 || ny < 0 || nx >= n || ny >= m ) continue;
        if(dist[nx][ny] != -1) continue; // 아직 방문한 적이 없는 & 안익은 토마토를 방문한다(bfs 한다)

        dist[nx][ny] = dist[cur.X][cur.Y] + 1;
        Q.push({nx,ny});
      }

    }


  int ans = 0;

  for(int i = 0; i < n; i++)
  {
    for(int j = 0; j < m; j++)
    {
      if(dist[i][j] == -1)
        {
          cout << "-1";
          return 0;
        }
      if(dist[i][j] > ans)
      {
        ans = dist[i][j];
      }
    }

  }

  cout << ans;


}


/*bfs의 시작점이 여러개 -> 일단 '모든 시작점을 큐에 넣고' 똑같이 BFS를 돌면 끝 */
/*
안익은 토마토가 bfs가 끝난 이후에도 익지 못한 경우 -> dist가 -1인 경우가 있는지 확인한다
저장될 때부터 모든 토마토가 익어있는 상황-> bfs 생략, line 51로 바로 넘어옴 -> ans는 그대로 0, 0출력
정답은 dist에서 가장 큰 수를 출력하면 된다.
*/
```

```
/*
about BFS
1. 시작하는 지점을 큐에 push하고 방문했다는 표시를 남김.
2. 큐에서 행렬(좌표)값이 저장된 원소를 꺼내고, 해당 지점의 상하좌우로 인접한 칸에 대하여 3번 진행
3. 해당 칸을 이전에 방문했다면 아무 동작도 하지 않는다, 만약 방문한 적이 없다면 방문표시를 남기고 해당 칸의 좌표를 큐에 삽입한다.
4. 큐가 빌 때까지 2번을 반복한다.
모든 칸이 큐에 1번씩 들어가므로 시간복잡도는 칸이 N개일 때 O(n)이 된다.
board, visit, n/m, dx/dy, pair queue, pair current, nx/ny
*/
```

> 불의 전파를 BFS로 처리할 수 있음 → 사방으로 퍼져나간다!

> 시작 점이 두 종류<불, 지훈>일 때의 BFS

> **이런 문제는 불에 대한 BFS와 지훈이에 대한 BFS를 모두 돌림으로서 해결이 가능합니다.**
> 먼저 지훈이는 신경쓰지 말고 불에 대한 BFS를 돌려서 미리 각 칸에 불이 전파되는 시간을 다 구해둡니다. 두 번째의 맵이 바로 각 칸에 불이 전파시간을 의미합니다.
>
> 그 다음에는 지훈이에 대한 BFS를 돌리며 지훈이를 이동시킵니다. 이 때 **만약 지훈이가 특정 칸을 x시간에 최초로 방문할 수 있는데 그 칸에 x시간이나 그 이전에 불이 붙는다면 그 칸을 못가게 됩니다.**
>
> 예를 들어 **으로 마킹한 칸을 보면 지훈이는 저 칸에 2시간이 될 때 방문하게 됩니다. 그런데 불은 이미 1시간만에 전파되었기 때문에 지훈이는 저 곳을 갈 수 없습니다. *, ***으로 마킹한 칸도 마찬가지 이유로 지훈이가 갈 수 없는 칸입니다.
>
> 원래 BFS의 구현에서는 큐 안에서 (nx, ny)를 살펴볼때 방문했는지 여부를 vis[nx][ny]가 true인지 혹은 dist[nx][ny]가 0 이상인지 확인하고, 이미 방문한 칸이라면 continue를 합니다. **이 문제에서는 추가로 해당 칸에 불이 붙은 시간을 확인해서 필요에 따라 continue를 하면 됩니다. 이렇게 BFS와 지훈이에 대한 BFS를 따로 해서 문제를 그렇게 어렵지는 않게 해결할 수 있습니다.**

```
#include <bits/stdc++.h>
using namespace std;
#define X first
#define Y second
int r, c; // 입력의 첫째 줄에는 공백으로 구분된 두 정수 R과 C가 주어진다. 단, 1
          // ≤ R, C ≤ 1000 이다. R은 미로 행의 개수, C는 열의 개수이다.
string board[1002]; // 행의 max 수만큼 선
int fire[1002][1002];
```

```
int jihoon[1002][1002];
int dx[4] = {0, 1, 0, -1};
int dy[4] = {1, 0, -1, 0};

int main(void) {
  ios::sync_with_stdio(0);
  cin.tie(0);

  cin >> r >> c; // row and column

  queue<pair<int, int>> Q1; // fire bfs queue
  queue<pair<int, int>> Q2; // jihoon bfs queue

  for (int i = 0; i < r; i++) // input data
    cin >> board[i];

  for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
    {
      if (board[i][j] == 'J' || board[i][j] == '.')  // 불과 지훈이가 지나갈 수 있는 공간들을 -1로 초기화 합니다. 지훈이가 있는 공간도 -1로 초기화 하는데
      {
        jihoon[i][j] = -1;
        fire[i][j] = -1;
      }
      if (board[i][j] == 'F')
        Q1.push({i, j}); // 불의 위치 -> Q1
      if (board[i][j] == 'J')
        Q2.push({i, j}); // 지훈의 위치 -> Q2
    }

  while (!Q1.empty()) // FIRE BFS, 불이 지나갈 수 있는 공간에서 퍼지는 데 걸리는 시간들을 계산
  {
    auto cur = Q1.front();
    Q1.pop(); // pair<int,int> cur

    for (int dir = 0; dir < 4; dir++) {
      int nx = cur.X + dx[dir]; // 행
      int ny = cur.Y + dy[dir]; // 열

      if (nx < 0 || ny < 0 || nx >= r || ny >= c)
        continue;
      if (fire[nx][ny] != -1)
        continue;

      fire[nx][ny] = fire[cur.X][cur.Y] + 1;
      Q1.push({nx, ny});
    }

  } // fire bfs end

  auto cur2 = Q2.front();
  jihoon[cur2.X][cur2.Y] = 0; // line 28에서 지훈이의 위치를 0으로 설정합니다 (원상복구)

  while (!Q2.empty()) // jihoon bfs start
  {
    cur2 = Q2.front();
    Q2.pop(); // pair<int,int> cur

    for (int dir = 0; dir < 4; dir++) {
      int nx = cur2.X + dx[dir]; // 행
      int ny = cur2.Y + dy[dir]; // 열

      if ((nx < 0 || ny < 0 || nx >= r || ny >= c))
      {
        cout << jihoon[cur2.X][cur2.Y] + 1;
        return 0;
      } // 행렬범위를 벗어나면 탈출에 성공한 것

      if (jihoon[nx][ny] != -1)
        continue; // 지날 수 있는 공간들만 탐색하고, 전에 방문한 적이 있다면 패스합니다.
      jihoon[nx][ny] = jihoon[cur2.X][cur2.Y] + 1;

      if (jihoon[nx][ny] >= fire[nx][ny] && fire[nx][ny] != -1) // 이미 경로에 불길이 번진 후라면 값을 0으로 만든 후(방문표시) 좌표를 큐에 넣지 않고 패
      {
        jihoon[nx][ny] = 0;
        continue;
      }
      Q2.push({nx, ny});
    }

  }                     // jihoon bfs end
  cout << "IMPOSSIBLE"; // 탈출에 실패했다.

}
```

2차원에서의 BFS를 생각하면 지금 이 그림처럼 현재 선택된 칸에서 상하좌우로 뻗어나가는 방식으로 진행이 됐습니다. 왜 상하좌우로 뻗어갔나 생각해보면 그게 불이 됐든 토마토가 됐든 전파가 상하좌우로 이루어졌기 때문입니다.

그럼 약간의 창의성을 발휘해서 아래와 같이 이 문제에서도 수빈이가 X에 있다고 할 때 X-1, X+1, 2X로 이동하는 것을 BFS로 처리할 수 있겠다(dx[3] = {1, -1, 2})는 생각을 해볼 수 있습니다.


```
#include <bits/stdc++.h>
using namespace std;
#define X first
#define Y second

int ary[200002]; // 아무리 많이 벗어나도 음수이거나 20만 이상(100000*2)을 벗어나지는 않음
```

```
int dx[3] = {1, -1, 2}; // 앞, 뒤, 현위치 * 2 순간이동

int main(void){
  ios::sync_with_stdio(0);
  cin.tie(0);

  fill(ary, ary + 200002, -1); // -1로 초기화하여 방문 해야할 곳 & 방문여부 체크
  int n, k;

  cin >> n >> k; // n == 수빈
  ary[n] = 0; // 수빈이가 있는 곳의 값을 0으로 설정한다 (방문했음!)

  queue<int> Q;
  Q.push(n); // 수빈의 위치를 큐에 넣는다 // bfs 초기세팅

  while(!Q.empty()) // bfs start
  {
    int nx;

    auto curX = Q.front(); Q.pop(); // queue에서 좌표 가져오기

    for(int dir = 0; dir < 3; dir++) // 탐색
    {
      if(dir != 2)
        nx = curX + dx[dir]; // 앞으로 갔다 뒤로 갔다 curX*2 순간이동
      else // dir == 2, 2배 순간이동
        nx = curX * dx[dir];

      if(nx < 0 || nx > 200001) continue; // 아무리 많이 벗어나도 음수이거나 20만 이상(100000*2)을 벗어나지는 않음.
                                          //음수번째 칸을 들리는 경우는 무조건 최단거리가 아님.
      if(ary[nx] != -1) continue; // 방문한 적이 있다면 패스

      ary[nx] = ary[curX] + 1;
      Q.push(nx);
    }

  }

cout << ary[k];


}



/*
about BFS
1. 시작하는 지점을 큐에 push하고 방문했다는 표시를 남김.
2. 큐에서 행렬(좌표)값이 저장된 원소를 꺼내고, 해당 지점의 상하좌우로 인접한 칸에 대하여 3번 진행
3. 해당 칸을 이전에 방문했다면 아무 동작도 하지 않는다, 만약 방문한 적이 없다면 방문표시를 남기고 해당 칸의 좌표를 큐에 삽입한다.
4. 큐가 빌 때까지 2번을 반복한다.
모든 칸이 큐에 1번씩 들어가므로 시간복잡도는 칸이 N개일 때 O(n)이 된다.
board, visit, n/m, dx/dy, pair queue, pair current, nx/ny
*/
```