13주차 결과보고서

전공: 컴퓨터공학과 학년: 4학년 학번: 20191612 이름: 윤기웅

**1. (문제 및 제목은 작성하지 말 것. 답안만 작성하면 됩니다.)**

.................

Readfile : readFile 함수는 .maz 파일을 불러와 미로 데이터를 적절한 자료구조에 저장하는 역할을 합니다.

**작동 상세:**

* **파일 선택:** ofSystemLoadDialog("Select .maz file")을 통해 파일 선택 창을 열어 .maz 파일을 선택합니다.
* **파일 유효성 검사:** filePath.find\_last\_of(".")로 파일 경로에서 마지막 점(.) 위치를 찾고, 확장자가 .maz인지 확인합니다. .maz가 아닐 경우 함수는 false를 반환합니다.
* **파일 존재 확인:** ofFile file(fileName)를 사용해 파일 객체를 생성하고, file.exists()로 파일이 존재하는지 검사합니다. 존재하지 않으면 false를 반환합니다.
* **파일 읽기:** ofBuffer buffer(file)로 파일 내용을 버퍼에 저장합니다. buffer.getLines().begin()과 buffer.getLines().end()을 이용해 파일의 각 줄을 읽고, WIDTH와 HEIGHT를 계산합니다.
* **동적 할당:** 미로 데이터를 저장할 배열들을 동적으로 할당합니다. input, map, visited, dist 배열을 할당하고 초기화합니다.
* **미로 데이터 저장:** 버퍼로부터 읽은 각 줄을 input 배열에 저장하고, 각 문자가 ' ' (빈 공간)일 경우 map 배열에 1을, 그렇지 않으면 0을 저장하여 미로를 구성합니다.

**미로 찾기 역할:** 이 함수는 파일에서 미로 데이터를 읽어 프로그램이 사용할 수 있는 형태로 변환하는 역할을 하며, 미로의 크기를 설정하고 데이터를 적절한 배열에 저장하여 이후 탐색에 필요한 기초 데이터를 제공합니다.

makeNode: makeNode 함수는 주어진 좌표를 가진 노드를 생성합니다.

**작동 상세:**

* **노드 생성:** malloc(sizeof(Node))을 사용해 동적으로 Node 구조체를 할당합니다.
* **좌표 설정:** 생성된 노드의 r와 c 필드를 각각 입력받은 r과 c로 설정합니다.
* **초기화:** 노드의 next 포인터를 NULL로 초기화하여 노드가 아직 다른 노드와 연결되지 않았음을 나타냅니다.

**미로 찾기 역할:** 미로의 특정 지점을 나타내는 노드를 생성하는데 사용되며, DFS 탐색에서 경로를 표현하는 기본 단위로 사용됩니다.

popStack : popStack 함수는 스택에서 맨 위의 노드를 제거하고 반환합니다.

**작동 상세:**

* **스택 비어 있는지 확인:** isEmpty(s)를 호출하여 스택이 비어 있는지 확인합니다.
* **노드 제거 및 반환:** 스택이 비어 있지 않으면 s->top을 del에 저장하고, s->top을 top의 다음 노드(s->top->next)로 갱신합니다.
* **메모리 해제:** 제거된 노드의 메모리를 free(del)로 해제합니다.
* **노드 반환:** 제거된 노드를 복사한 ret을 반환합니다.

**미로 찾기 역할:** DFS 탐색 중 스택에서 현재 노드를 제거하여 이전 노드로 돌아갈 수 있게 하며, 탐색 경로를 역추적하는 기능을 제공합니다.

pushStack: pushStack 함수는 새로운 노드를 스택의 맨 위에 추가합니다.

**작동 상세:**

* **새로운 노드 추가:** newNode->next를 현재 s->top으로 설정하여 새로운 노드를 스택의 맨 위에 연결합니다.
* **갱신:** s->top을 newNode로 갱신하여 스택의 새로운 맨 위 노드로 설정합니다.
* **스택 길이 증가:** s->len을 증가시켜 스택의 크기를 업데이트합니다.

**미로 찾기 역할:** DFS 탐색 중 경로를 확장할 때 새로운 노드를 스택에 추가하여 탐색을 계속할 수 있게 합니다.

isEmpty : isEmpty 함수는 주어진 스택이 비어 있는지 확인합니다.

**작동 상세:**

* **길이 확인:** s->len == 0을 확인하여 스택의 길이가 0인지 검사합니다.
* **결과 반환:** 스택이 비어 있으면 true, 그렇지 않으면 false를 반환합니다.

**미로 찾기 역할:** DFS 탐색 중 스택이 비어 있는지 확인하여 탐색 종료 여부를 결정합니다.

makeStack : makeStack 함수는 새로운 스택을 생성합니다.

**작동 상세:**

* **스택 생성:** malloc(sizeof(MyStack))을 사용해 동적으로 MyStack 구조체를 할당합니다.
* **초기화:** 생성된 스택의 top을 NULL로 초기화하고, len을 0으로 설정합니다.

**미로 찾기 역할:** DFS 탐색에서 사용될 스택을 초기화하여, 경로를 저장하는 용도로 사용합니다.

makePath : makePath 함수는 allPath 구조체를 생성합니다.

**작동 상세:**

* **경로 생성:** malloc(sizeof(allPath))을 사용해 동적으로 allPath 구조체를 할당합니다.
* **초기화:** head를 NULL로 설정하여 경로 리스트의 초기 상태를 나타냅니다.

**미로 찾기 역할:** DFS 탐색에서 전체 경로를 저장할 수 있는 리스트를 생성합니다.

addPath : addPath 함수는 allPath 구조체에 새로운 경로 노드를 추가합니다.

**작동 상세:**

* **헤드 확인:** ap->head가 NULL이면 ap->head를 path로 설정합니다.
* **현재 경로의 끝을 찾기:** cur를 사용해 현재 경로 리스트의 마지막 노드를 찾습니다.
* **노드 연결:** 마지막 노드의 next를 새로운 경로 노드 path로 설정하여 리스트에 추가합니다.

**미로 찾기 역할:** DFS 탐색 중 모든 경로를 리스트에 저장하여 전체 탐색 경로를 기록합니다.

addShortestPath : addShortestPath 함수는 최단 경로를 sp에 추가합니다.

**작동 상세:**

* **헤드 확인:** sp->head가 NULL이면 sp->head를 path로 설정합니다.
* **현재 경로의 끝을 찾기:** cur를 사용해 현재 경로 리스트의 마지막 노드를 찾습니다.
* **노드 연결:** 마지막 노드의 next를 새로운 경로 노드 path로 설정하여 리스트에 추가합니다.

**미로 찾기 역할:** DFS 탐색에서 최단 경로를 기록하여 탐색 결과를 저장합니다.

DFS : DFS 함수는 깊이 우선 탐색(DFS)을 사용하여 미로의 출발점에서 도착점까지의 경로를 찾습니다.

**작동 상세:**

* **초기화:** makeStack으로 스택을 생성하고, makePath로 경로를 초기화합니다.
* **시작점 설정:** makeNode(1, 1)으로 시작점을 생성하여 스택에 추가합니다.
* **탐색:** 스택이 비어 있지 않은 동안 다음을 반복합니다:
  + cur를 현재 스택의 맨 위 노드로 설정.
  + cur의 좌표가 도착점인지 확인하고, 도착점이면 didfind를 1로 설정.
  + dr와 dc 배열을 사용하여 네 방향(상, 하, 좌, 우)으로 이동 가능성을 검사.
  + 이동할 수 있는 경우, 새로운 노드를 생성하여 스택에 추가하고, 경로에 추가.
  + 이동할 수 없으면 스택에서 노드를 제거.
* **최단 경로 저장:** 탐색이 완료되면 스택에 남은 노드를 이용해 최단 경로를 sp에 저장.

**미로 찾기 역할:** 미로를 탐색하여 출발점에서 도착점까지의 경로를 찾고, 모든 경로와 최단 경로를 기록하는 기능을 합니다.

Dfsdraw: dfsdraw 함수는 DFS 결과를 그래픽으로 출력합니다.

**작동 상세:**

* **배경 설정:** ofBackground(255, 255, 255)로 배경색을 하얗게 설정합니다.
* **미로 그리기:** 이중 for문을 사용하여 input 배열을 순회하며 각 좌표의 벽을 검은색, 길을 하얀색으로 칠합니다.
* **경로 그리기:** ap에 저장된 모든 경로를 검은색으로, sp에 저장된 최단 경로를 빨간색으로 그립니다. ofDrawLine으로 경로를 선으로 그립니다.

**미로 찾기 역할:** 미로와 탐색 결과를 시각적으로 표현하여 사용자가 경로를 확인할 수 있게 합니다.

freeMemory : freeMemory 함수는 동적으로 할당된 메모리를 해제합니다.

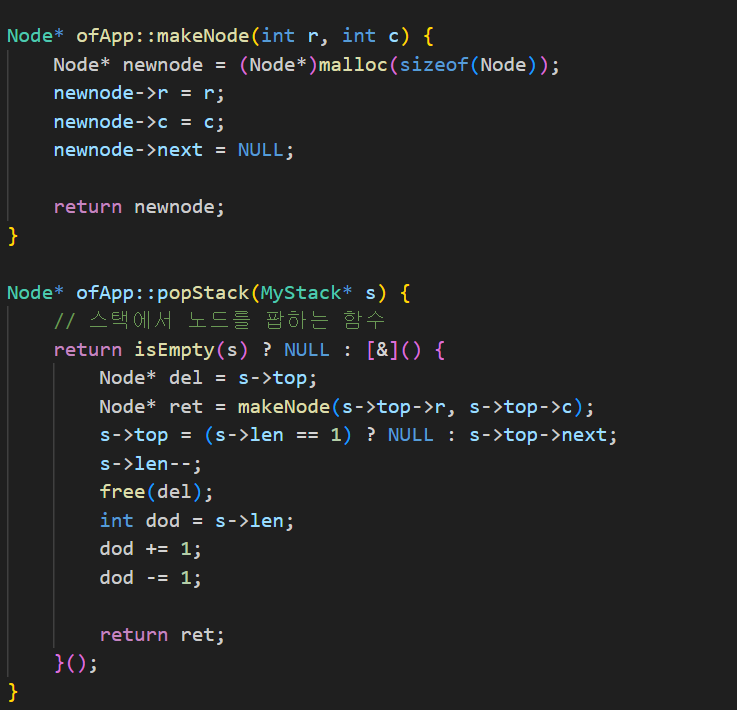
**작동 상세:**

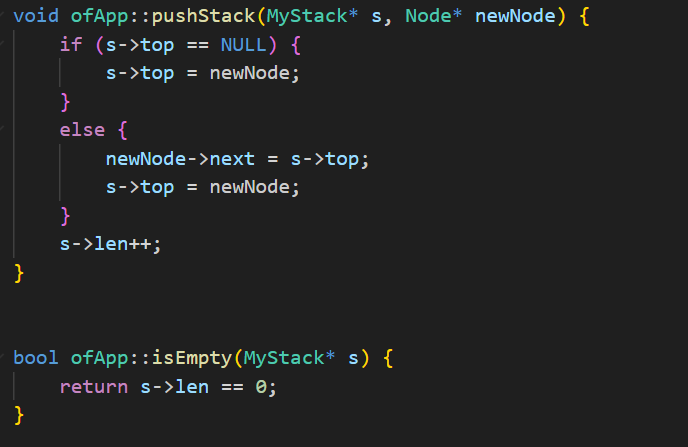
* **배열 메모리 해제:** input, map, visited, dist 배열의 각 행을 free로 해제하고, 배열 자체도 해제합니다.
* **경로 메모리 해제:** ap와 sp의 각 노드를 순회하며 메모리를 해제합니다.

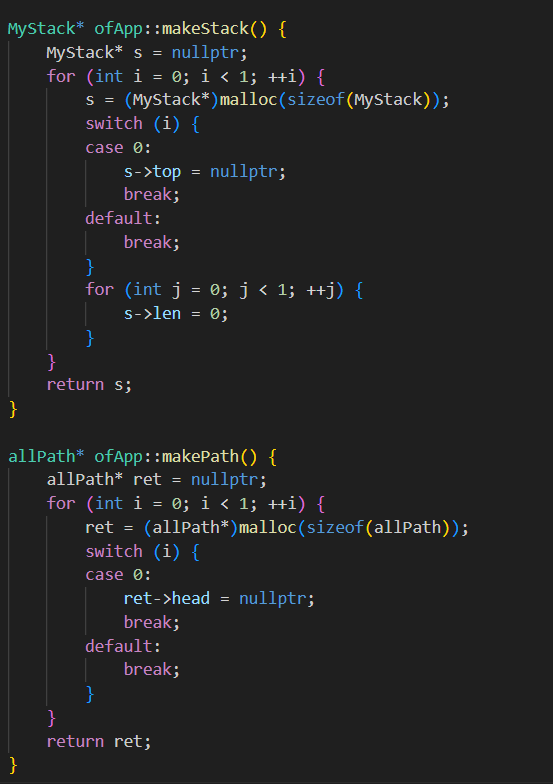
**미로 찾기 역할:** 프로그램 종료 시 동적으로 할당된 모든 메모리를 해제하여 메모리 누수를 방지합니다.

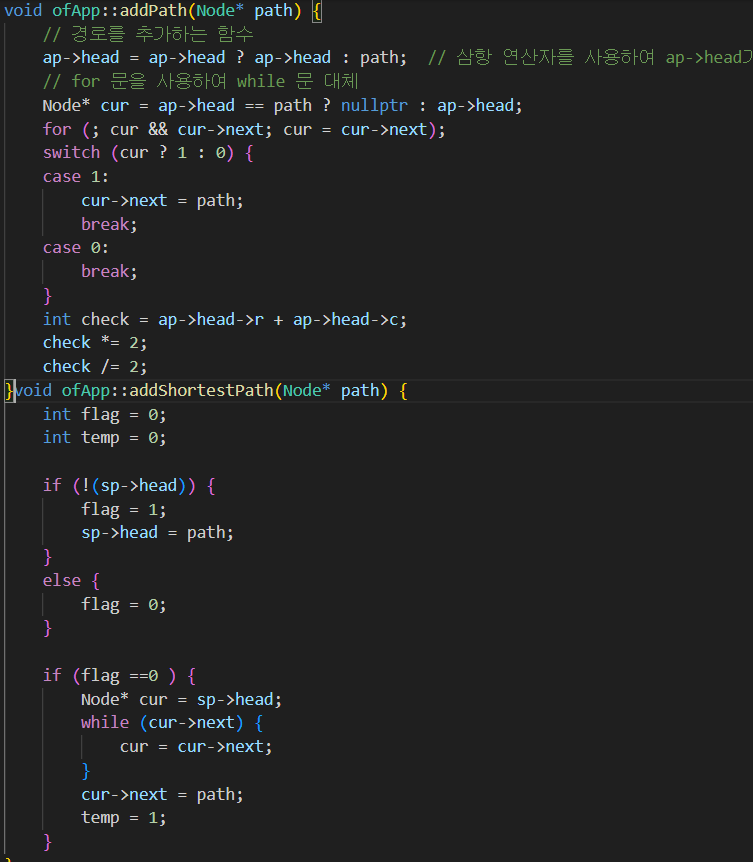
직사각형, 스크린샷, 도표, 폰트이(가) 표시된 사진

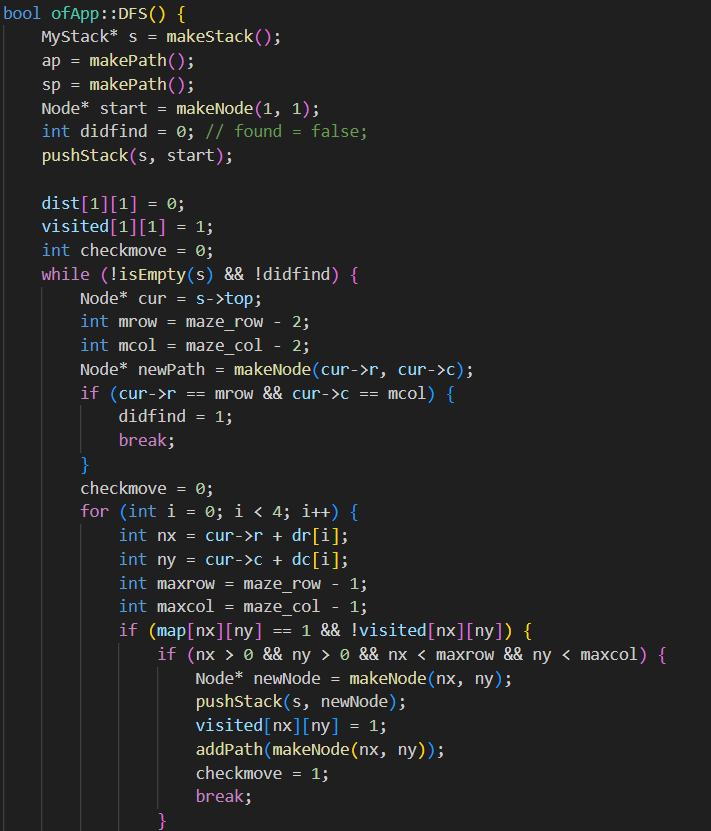
자동 생성된 설명

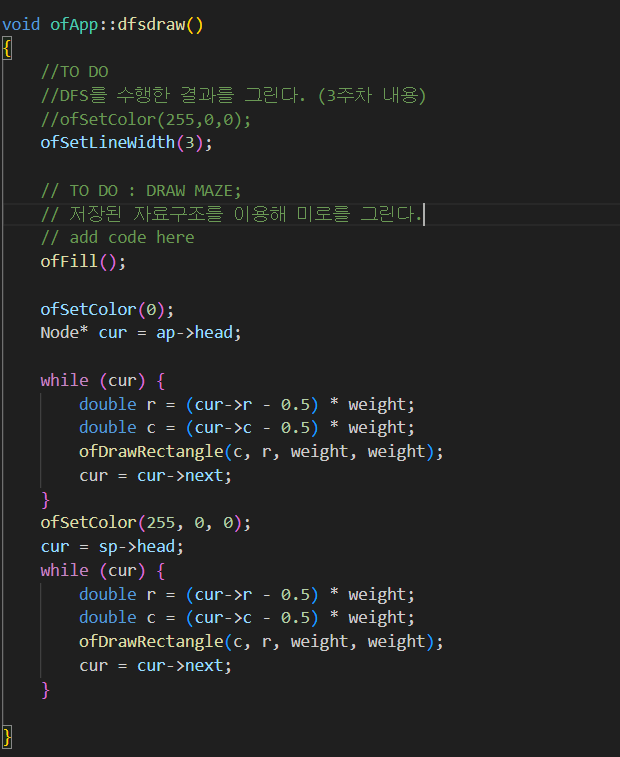


****

****

****

****

****

#### makeQueue 함수 : Queue\* makeQueue()

이 함수는 빈 큐를 생성하는 함수입니다. 새로운 Queue 구조체를 동적으로 할당하고, 그 초기값을 설정합니다.

* q->front와 q->rear는 각각 큐의 앞과 뒤를 가리킵니다.
* q->size는 큐의 크기를 나타냅니다.

#### Isempty 함수 : bool isEmpty(Queue\* q)

이 함수는 큐가 비어 있는지 확인하는 함수입니다. q->size가 0이면 큐가 비어있다는 것을 의미합니다.

* 큐가 비어있으면 true를 반환하고, 그렇지 않으면 false를 반환합니다.

#### Enque 함수: void enqueue(Queue\* q, Node\* newNode)

이 함수는 큐에 새로운 노드를 추가하는 함수입니다.

* 만약 큐가 비어있으면, front와 rear를 newNode로 설정합니다.
* 그렇지 않으면, rear의 next를 newNode로 설정하고, rear를 newNode로 업데이트합니다.
* 큐의 크기를 1 증가시킵니다.

#### Deque : Node\* dequeue(Queue\* q)

이 함수는 큐에서 노드를 제거하고 반환하는 함수입니다.

* 만약 큐가 비어있으면 nullptr를 반환합니다.
* 그렇지 않으면 front 노드를 제거하고, front를 그 다음 노드로 업데이트합니다.
* 만약 front가 nullptr가 되면, rear도 nullptr로 설정합니다.
* 큐의 크기를 1 감소시킵니다.

#### Bfs 함수 : bool ofApp::BFS()

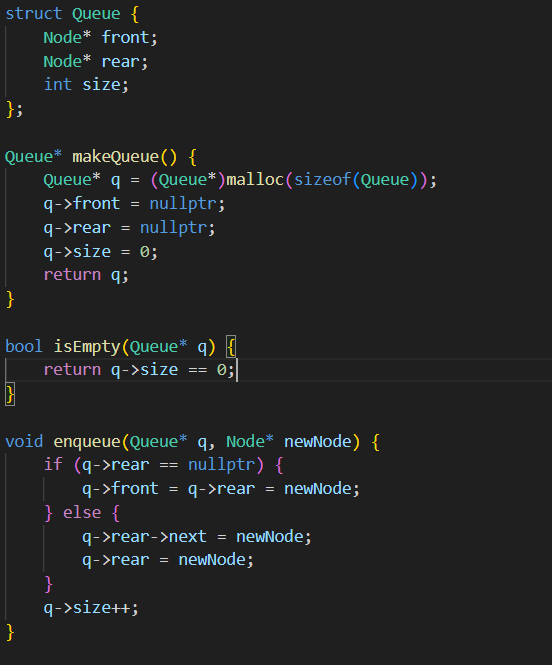
이 함수는 미로를 BFS 알고리즘을 사용하여 탐색하는 함수입니다.

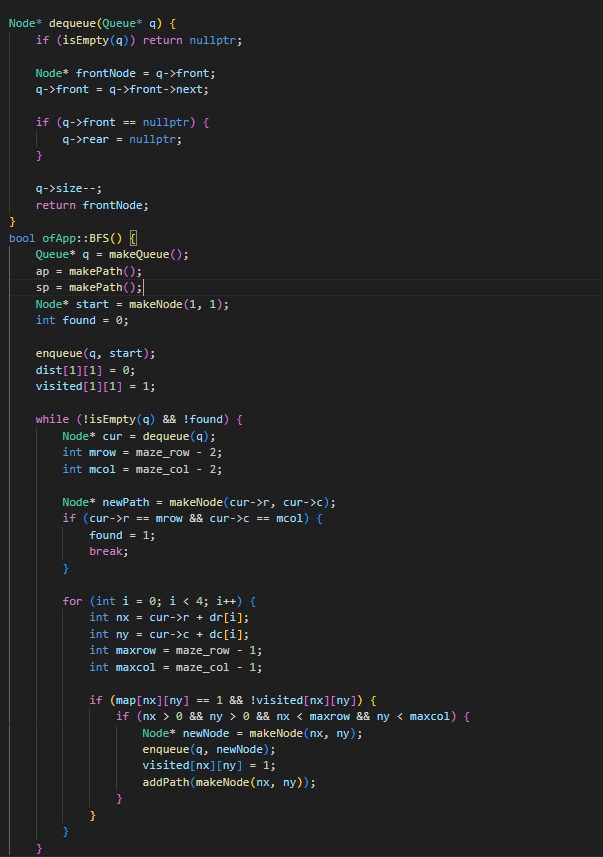
* 큐를 생성하고, 시작 노드를 큐에 추가합니다.
* BFS 탐색을 통해 목표 노드에 도달할 때까지 인접 노드를 큐에 추가하고 방문합니다.
* 목표 노드에 도달하면 found를 1로 설정하고 반복문을 종료합니다.
* 탐색이 완료되면 큐를 해제하고, 목표 노드에 도달했는지 여부를 반환합니다.

#### Bfsdraw 함수 : void ofApp::bfsdraw()

이 함수는 BFS 탐색 결과를 화면에 시각적으로 표시하는 함수입니다.

* BFS() 함수를 호출하여 BFS 탐색을 수행합니다.
* BFS 탐색이 성공적으로 수행되면, 최단 경로의 각 노드를 빨간색으로 표시합니다.
* BFS 탐색이 실패한 경우, 경로를 찾을 수 없음을 표시합니다.

****

****

**Dfs,bfs 모두시간 복잡도와 공간복잡도가 같습니다. 시간 복잡도는 O(v+e), 공간 복잡도는 O(v)가 됩니다. 여기서 v,e는 각각 정점의 개수와 간선의 개수를 의미합니다. 이전에 생각했던 각각의 알고리즘이 지금과 크게 달라진 것은 없습니다. 하지만, c++언어를 이용하기 때문에 구현 시 구조체와 연결리스트 등을 쓰는 것이 평소 파이썬을 많이 쓰는 저로서는 조금 힘들었습니다.**

**2.**

.......................

DFS는 깊이 우선으로 탐색하여 한 번의 경로를 끝까지 탐색하고, BFS는 너비 우선으로 탐색하여 최단 경로를 찾습니다.

* DFS: 깊이 우선 탐색으로 특정 경로를 먼저 탐색하여 빠르게 답을 찾을 수 있지만, 최적 경로는 아닐 수 있습니다.
* BFS: 너비 우선 탐색으로 모든 경로를 탐색하여 최단 경로를 보장합니다.

. DFS는 메모리가 적게 필요하고, BFS는 최단 경로를 보장하지만 메모리를 많이 사용합니다. 최악의 경우에는 DFS는 계속 길을 찾는 과정이 지연될 수가 있으니 BFS가 메모리를 많이 차지해도 더 적합합니다. 또한, 해당 미로 찾기는 왼쪽 위에서 오른쪽 아래로 가는 거라 오른쪽 혹은 아래쪽 이동이 주요한 이동 방향입니다. 그래서 BFS의 메모리가 그리 많이 필요하지 않습니다.

.........................