게임 과제

💡 본 웹페이지는 게임 `카드짝맞추기` , `미로찾기` 게임을 진행할 수 있습니다.

https://github.com/jeongjinuk/game

♠ 카드 짝 맞추기

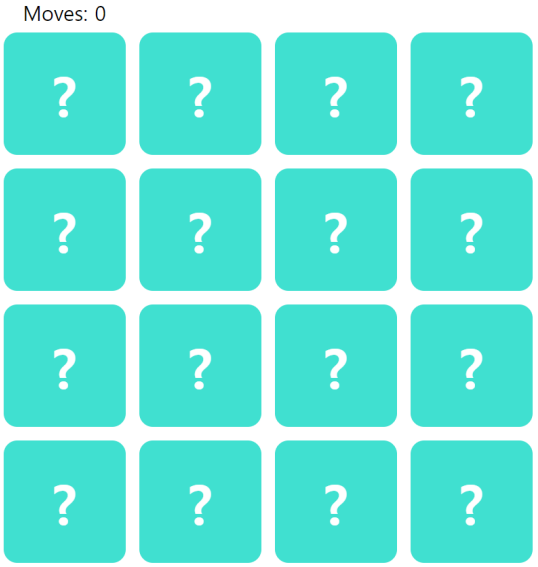
📍 미로 찾기

♠

카드 짝 맞추기

게임 화면 및 게임 진행 방식

💡 카드 짝 맞추기 게임은 같은 숫자를 맞추면 클리어 할 수 있는 게임입니다.
게임의 진행 방식은 첫 화면에서 힌트를 볼 시간(초)을 입력합니다.
입력한 시간 만큼 카드를 볼 수 있고 시간이 경과한 이후 모든 카드가 뒤집힙니다.
이후 같은 짝의 카드를 기억하여 클릭하면 됩니다.
단, 카드의 짝이 아닐 경우 두 번째 카드까지 보여지지 않습니다.



코드 설명

코드 수정

▼ `html`

```
<!DOCTYPE html>
<html lang="ko">
<head>
    <meta charset="UTF-8">
    <title>카드 짝 맞추기</title>
    <link href="css/card-style.css" rel="stylesheet">
    <!-- 게임 시작시 카드개수 받고 랜덤 배열 만들어줌  -->
    <!--  카드 추가 해줌  -->
    <script src="js/cardBuilder.js"></script>
</head>
<body>
<div id="score" style="display: none;">0</div>
<div id="moveCount">Moves: 0</div>
</section>
<div class="card__container">
    <div class="card__section">
        <!-- 카드 생성-->
        <script>
            let visible = prompt("힌트 보기(초)")
            let visibleTime = visible * 1000;
            new CardBuilder(8)
                .start()
                .build()
                .forEach(cardHtml => document.write(cardHtml));
        </script>
    </div>
</div>
</body>
<script src="js/card-event.js"></script>
</html>
```

▼ `css`

```
.card__container{
  font-family: sans-serif;
  justify-content: center;
  display: flex;
  justify-content: center;
  align-items: center;
  width: 100%;
}
.card__section{
  display: flex;
  justify-content: center;
  align-items: center;
  flex-wrap: wrap;
  width: 900px;
  height: auto;
}
.scene {
```

```
  display: inline-block;
  width: 180px;
  height: 180px;
  perspective: 600px;
  margin: 10px;
  padding: 0;
}
.card:hover {
  transform : scale(1.1);
  transition : 0.5s ease-in;
}

.card {
  position: relative;
  width: 100%;
  height: 100%;
  cursor: pointer;
  transform-style: preserve-3d;
  transform-origin: center right;
  transition: transform 1s;
}

.card.is-flipped {
  transform: translateX(-100%) rotateY(-180deg);
}

.card__face {
  border-radius: 20px;
  position: absolute;
  width: 100%;
  height: 100%;
  line-height: 180px;
  color: white;
  text-align: center;
  font-weight: bold;
  font-size: 80px;
  backface-visibility: hidden;
}

.card__face--front {
  background: #40e0d0;
}

.card__face--back {
  background: #ff0080;
  transform: rotateY(180deg);
}

#moveCount {
  font-family: sans-serif;
  font-size: 30px;
  text-align: center;
  margin-top: 50px;
  margin-right: 600px;
}
```

▼ `javascript 이벤트`

```
let flag = true;
let first = "";
let second = "";
let disableDeck = false;

let cards = document.querySelectorAll('.card');
let score = 0;
let movesCounter = 0;

// 무브 횟수를 증가시키는 함수
function increaseMoveCount() {
    movesCounter++;
}

// 스코어를 증가시키는 함수
function increaseScore() {
    score++;
}

// 스코어를 화면에 업데이트하는 함수
function updateScore() {
    // 업데이트된 스코어를 화면에 표시
    document.getElementById('score').innerText = score;
}

// 무브 횟수를 화면에 업데이트하는 함수
function updateMoveCount() {
    // 업데이트된 무브 횟수를 화면에 표시
    document.getElementById('moveCount').innerText = "Moves: " + movesCounter;
}

// 화면 초기화 클리어 시
function reset() {
    for (let card of document.querySelectorAll('.card')) {
        flag = flag && card.style.pointerEvents === "none";
    }
    if (flag) {
        alert(score - visible + "점 입니다.");
        location.reload();
        return;
    }
    flag = true;
}

// 카드 뒤집기 이벤트
function flipEvent(e){
    let currentTarget = e.currentTarget;
    if (currentTarget !== first && !disableDeck) {
        currentTarget.classList.add("is-flipped");
        if (!first) {
            first = currentTarget;
            return;
```

```
        }
        second = e.currentTarget;
        disableDeck = true;
        if (first.attributes.value.value - second.attributes.value.value === 0) {
            first.style.pointerEvents = "none";
            second.style.pointerEvents = "none";
        } else {
            first.classList.remove("is-flipped");
            second.classList.remove("is-flipped");
        }
        first = "";
        second = "";
        disableDeck = false;
    }
}

// 클릭 이벤트 발생 시 스코어 증가 및 업데이트
cards.forEach(card => {
    card.addEventListener('click', function () {
        // 클릭 이벤트에 따른 게임 로직 추가

        // 무브 횟수 증가
        increaseMoveCount();

        // 두 카드가 매치되지 않아도 클릭 시 스코어 증가
        increaseScore();

        // 스코어 및 무브 횟수 업데이트 함수 호출
        updateScore();
        updateMoveCount();
    });
});

// 초기 할당시 카드 뒤집기가능
cards.forEach(card => {
        card.classList.add("is-flipped");
        setTimeout(() => { // 시간 만큼 보여주기
            card.classList.remove("is-flipped");
            card.style.pointerEvents = "auto";
        }, visibleTime);
    });


// 리셋 및 뒤집기 이벤트 할당
cards.forEach(card => {
        card.addEventListener('click', flipEvent);
        card.addEventListener('click', reset);
    });
```

▼ `javascript 카드생성`

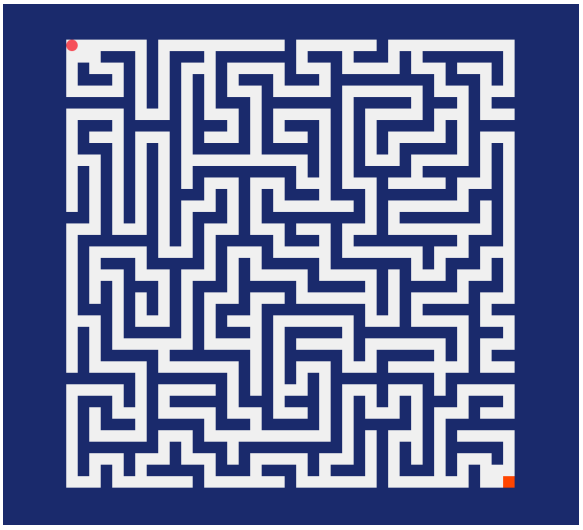
```
class CardBuilder {
    constructor(cardSize) {
```

```
        this.cardSize = cardSize;
    }
    // 카드 랜덤 생성
    // 랜덤하게 생성해도 괜찮긴한데
    // 그러면 최악의 경우 무한루프가 돌 수 있음
    // 그래서 미리 16개의 카드 배열을 만들고
    // 인덱스를 기준으로 랜덤하게 뽑음
    // 밑줄친 코드가 중요
    start(){
        this.cardList = new Array();
        let array = new Array();
        let pair = 2

        while (pair-- > 0){
            for (let i = 1; i <= this.cardSize; i++) {
                array.push(i);
            }
        }

        let pairCardSize = array.length;

        while (pairCardSize-- > 0){
            let rand = Math.floor(Math.random() * pairCardSize);
            this.cardList.push(array.splice(rand,1));
        }
        return this;
    }
    // 카드 html 템플릿
    #getCardHTML(value){
        return `<div class="scene scene--card">
            <div class="card"  value="${value}" style="pointer-events: none;">
                <div class="card__face card__face--front" >?</span></div>
                <div class="card__face card__face--back">${value}</div>
            </div>
        </div>`;
    }

    build() {
        let resultTemp = [];
        for (let i = 0; i < this.cardSize * 2; i++) {
            resultTemp.push(this.#getCardHTML(this.cardList[i]));
        }
        return resultTemp;
    }
}
```

미로 찾기

게임 화면 및 게임 진행 방식

💡 미로의 끝에 가장 짧은 거리로 도착하면 되는 게임입니다.
게임의 진행 방식은 접속과 동시에 미로가 생성됩니다.
키보드의 방향키를 움직여 미로에 표시된 도착지까지 가장 짧은 거리로 도달하면 됩니다.



코드 설명

▼ `html`

```
<!DOCTYPE html>
<html lang="ko">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>미로 찾기</title>
    <link href="css/maze-style.css" rel="stylesheet">
    <script src="js/MazeBuilder.js"></script>
    <script src="js/MazeCell.js"></script>
```

```
</head>
<body>
<div class="f">
    <div class="maze" id="maze">
        <div class="movable" id="player"></div>
        <!-- 미로의 벽 생성-->
        <script>
            new MazeBuilder(39)
                .build()
                .forEach(mazeRow =>
                    mazeRow.forEach(mazeCol =>
                        document.write(mazeCol.getHTML())));
            console.log(shortestRoute);
        </script>
    </div>
</div>
<script src="js/maze-event.js"></script>
</body>
</html>
```

▼ CSS

```
body {
    margin: 0;
    padding: 0;
    background-color: #1a2a6c;
}
.f {
    display: flex;
    height: 100vh;
    justify-content: center;
    align-items: center;
}
.maze {
    width: 780px;
    height: 780px;
    overflow: hidden;
    background-color: #f0f0f0;
}
.movable {
    position: absolute;
    width: 20px;
    height: 20px;
    background: #f64f59;
    border-radius: 50%;
    transition: transform 0.1s ease-out;
}

.wall {
    float: left;
    background-color: #1a2a6c;
    width: 20px;
    height: 20px;
}
.road {
    float: left;
    width: 20px;
    height: 20px;
}
.T {
    float: left;
    width: 20px;
    height: 20px;
```

```
    background: turquoise;
}
.end{
    background: orangered;
    float: left;
    width: 20px;
    height: 20px;
    z-index: 5;
}
```

▼ `javascript 이벤트`

`maze-event.js`

```
const container = document.querySelector("#maze"); // 미로 칸들을 담는 컨테이너
const movableElement = document.querySelector("#player"); // 플레이어 포인트
const walls = document.querySelectorAll(".wall"); // 벽
const ends = document.querySelectorAll(".end"); // 도착지
const step = 20; // 이동거리 px

let travel = 0; // 움직임 횟수
let isMoving = false; // 연속 키눌림 방지를 위해서

function arrived(x, y){ // 도착 확인
    let arrived = false;
    ends.forEach(end => {
        const  endRect = end.getBoundingClientRect();
        arrived = arrived || ( x ==  endRect.top && y ==  endRect.left);
    });
    return arrived;
}

function isCollision(x, y){ // 벽과 플레이어 좌표확인
    let isCollision = false;
    walls.forEach(wall => {
        const wallRect = wall.getBoundingClientRect();
        isCollision = isCollision || ( x ==  wallRect.top && y ==  wallRect.left);
    });
    return isCollision;
}

// 방향키 이벤트
function keyDown(e) {
    if (isMoving) return;
    isMoving = !isMoving;

    const playerRect = movableElement.getBoundingClientRect();
    let newX = playerRect.left - container.getBoundingClientRect().left; // 컨테이너 기준 좌표
    let newY = playerRect.top - container.getBoundingClientRect().top; // 컨테이너 기준 좌표
    let top = movableElement.getBoundingClientRect().top; // 브라우저 기준 좌표
    let left = movableElement.getBoundingClientRect().left; // 브라우저 기준 좌표
    // 위, 아래, 왼, 오
    switch (e.key) {
        case "ArrowUp":
            newY = Math.max(newY - step, 0);
            top = top - step;
            break;
        case "ArrowDown":
            // 컨테이너 내에서만 이동할 수 있도록 컨테이너 사이즈와 현재 내 좌표를 비교해야함
            newY = Math.min(newY + step, container.clientHeight - playerRect.height);
            top = top + step; // 화면 사이즈 왼쪽 상단 기준 좌표
            break;
        case "ArrowLeft":
            newX = Math.max(newX - step, 0);
```

```
            left = left - step;
            break;
        case "ArrowRight":
            newX = Math.min(newX + step, container.clientWidth - playerRect.width);
            left = left + step;
            break;
    }
    // 벽과 플레이어 포인트가 겹치는지 확인 후 겹치지 않는다면 새로운 좌표로 플레이어 위치 변경
    if (!isCollision(top, left)) {
        movableElement.style.transform = `translate(${newX}px, ${newY}px)`;
        travel++;
    }
    // 도착지 좌표와 플레이어 좌표가 동일하면 alert 및 브라우저 리로드
    if (arrived(top, left)){
        let format = `최단거리 : ${shortestRoute} \n당신의 기록 : ${travel}`
        alert(format);
        location.reload();
    }
    // 일정 시간(예: 100ms) 아니면 이벤트가 연속으로 발생해서 벽통과함
    setTimeout(() => {isMoving = !isMoving}, 140);}

document.addEventListener("keydown",  keyDown);
```

▼ `javascript 미로 생성`

mazeCell.js

```
class MazeCell {

    constructor(x, y) {
        this.type = false;
        this.routeList = [1,2,3,4]; // 경로 값 상하좌우 이동 1 = 2 row 이런식
        this.curPosition = [x,y]; // 0은 x, 1은 y
    }

    getHTML(){
        if(this.type == "end"){
            return `<div class='end'></div>`;
        }
        else if(this.type == "T"){
            return `<div class='T'></div>`;
        }
        return `<div class='${this.type ? "road" : "wall"}'></div>`;
    }

    // 현재 좌표에서 step 만큼 이동한 좌표 반환
    getNext(step, operator){
        let next = [0,0, !operator ? this.routeList.splice(Math.floor(Math.random() * this.routeLi
st.length), 1)[0] : operator];
        switch (next[2]){
            case 1:
                next[0] = this.curPosition[0] + step;
                next[1] = this.curPosition[1];
                break;
            case 2:
                next[0] = this.curPosition[0];
                next[1] = this.curPosition[1] + step;
                break;
            case 3:
                next[0] = this.curPosition[0] - step;
                next[1] = this.curPosition[1];
                break;
            case 4:
```

```
                next[0] = this.curPosition[0];
                next[1] = this.curPosition[1] - step;
                break;
        }
        return next;
    }
}
```

mazeBuilder.js

```
const x = [1, -1, 0, 0];
const y = [0, 0, 1, -1];
let size = 0;
const maze = new Array(size);
let shortestRoute = 0;

// 실행 함수 및 초기화 작업
class MazeBuilder {
    constructor(mazeSize) {
        size = mazeSize;
        for (let i = 0; i < size; i++) {
            maze[i] = [];
            for (let j = 0; j < size; j++) {
                maze[i][j] = new MazeCell(i, j);
            }
        }
    }
    build() {
        this.#createdMaze(maze[Math.floor(Math.random() * size)][Math.floor(Math.random() * size)]);
        this.#setStartAndEndPoint();
        this.#increasedDifficulty();
        this.#setShortestRoute();
        return maze;
    }

    // 미로 범위
    #isMazeRouteRange(x, y) {
        return x >= 0 && y >= 0 && x <= size - 1 && y <= size - 1;
    }

    // 미로생성
    #createdMaze(mazeCell){
        mazeCell.type = true; // 시작을 길로 바꿔줌
        while (mazeCell.routeList.length > 0){ // 갈 수 있는 경로수가 0이면 종료
            let next = mazeCell.getNext(2, 0); // 경로를 랜덤으로 들고옴
            //만약 미로 사이즈를 넘거나, 방문했던 곳이면 지나감
            if (!this.#isMazeRouteRange(next[0], next[1]) || maze[next[0]][next[1]].type){
                continue;
            }
            // 아니라면 다음 경로를 길로 바꿔주고, 벽도 길로 바꿔줌
            let cur = maze[next[0]][next[1]];
            let wall = mazeCell.getNext(1, next[2]);
            maze[wall[0]][wall[1]].type = true;
            this.#createdMaze(cur); // 다음 경로를 다시 재귀호출
        }
    }

    // 밑에도 Any를 이용하고 싶은데 이게 또 만들어줘야함
    // 가장 최단 경로 검색
    #setShortestRoute() {
        const copy = JSON.parse(JSON.stringify(maze));
        const q = [];
        q.push([0, 0, 1]); // x, y, 이동했는 횟수
```

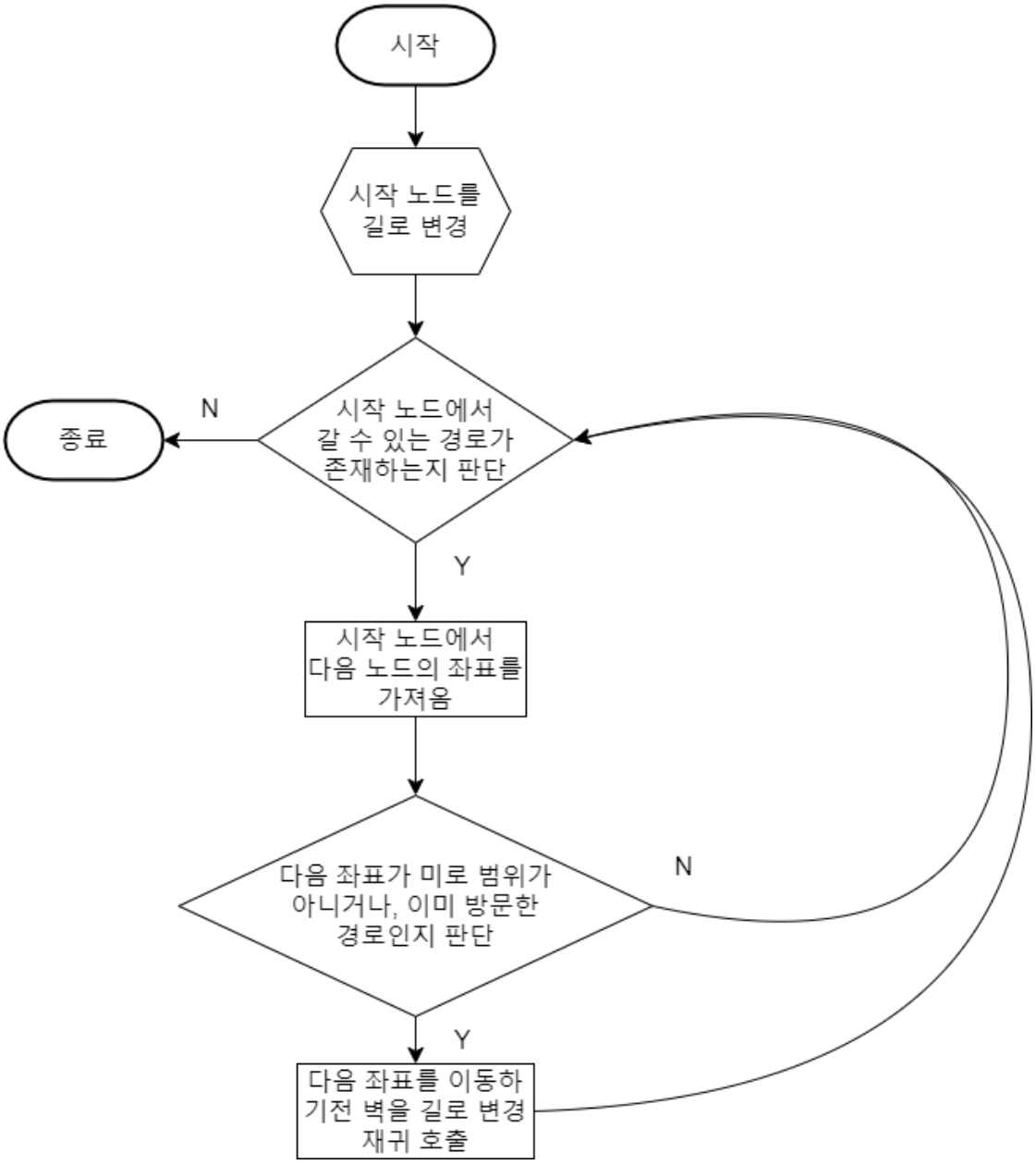
```
        while (q.length && !(q[0][0] == size - 1 && q[0][1] == size - 1)) {
            let pop = q.shift();
            copy[pop[0]][pop[1]].type = false;
            for (let i = 0; i < 4; i++) {
                let nx = pop[0] + x[i];
                let ny = pop[1] + y[i];
                if (this.#isMazeRouteRange(nx, ny) && copy[nx][ny].type) {
                    q.push([nx, ny, pop[2] + 1]);
                }
            }
        }
        shortestRoute = q[0][2];
    }

    // 끝지정 시작 지정
    #setStartAndEndPoint() {
        let row0 = maze[0];
        let row1 = maze[1];
        let row2 = maze[size - 1];
        let row3 = maze[size - 2];

        row0[0].type = true;
        row0[1].type = true;
        row1[0].type = true;
        row1[1].type = true;

        row2[size - 1].type = "end";
        row2[size - 2].type = true;
        row3[size - 1].type = true;
        row3[size - 2].type = true;
    }

    // 미로 난이도 올리기
    #increasedDifficulty() {
        const walls = [];
        const copy = JSON.parse(JSON.stringify(maze));
        for (let i = 0; i < size; i++) {
            for (let j = 0; j < size; j++) {
                if (!copy[i][j].type) {
                    const coordinate = [];
                    this.#wallSearch(i, j, copy, coordinate);
                    walls.push(coordinate);
                }
            }
        }

        this.#spliceWalls(walls);
    }

    // 벽 좌표 탐색 및 벽길이 알 수 있게
    #wallSearch(i, j, copy, coor) {
        if (this.#isMazeRouteRange(i, j) && !copy[i][j].type) {
            copy[i][j].type = true;
            coor.push([i, j]);
            for (let k = 0; k < 4; k++) {
                this.#wallSearch(i + x[k], j + y[k], copy, coor);
            }
        }
    }

    // 벽 자르기 여러 루트가 생길 수 있도록
    // 미로를 더 복잡하게 만들기 위해서
    #spliceWalls(walls) {
        const setRoad = (coors) => coors.forEach((coor) => maze[coor[0], coor[1]].type = true);
```

```
        walls
            .filter(value => value.length > 5) // 벽길이가 5 넘어갈때만 벽을 경로로 변경
            .forEach(wall => {
                let start = wall.splice(0, wall.length / 2).splice(wall.length / 2 / 2, 3);
                let mid = wall.splice(wall.length / 2, 3);
                setRoad(start);
                setRoad(mid);
            });
    }
}
```

💡 미로 생성을 위해서 사용한 알고리즘
1. 생성 - 깊이우선탐색 응용
2. 최단거리 - 너비우선탐색

깊이 우선 탐색 응용

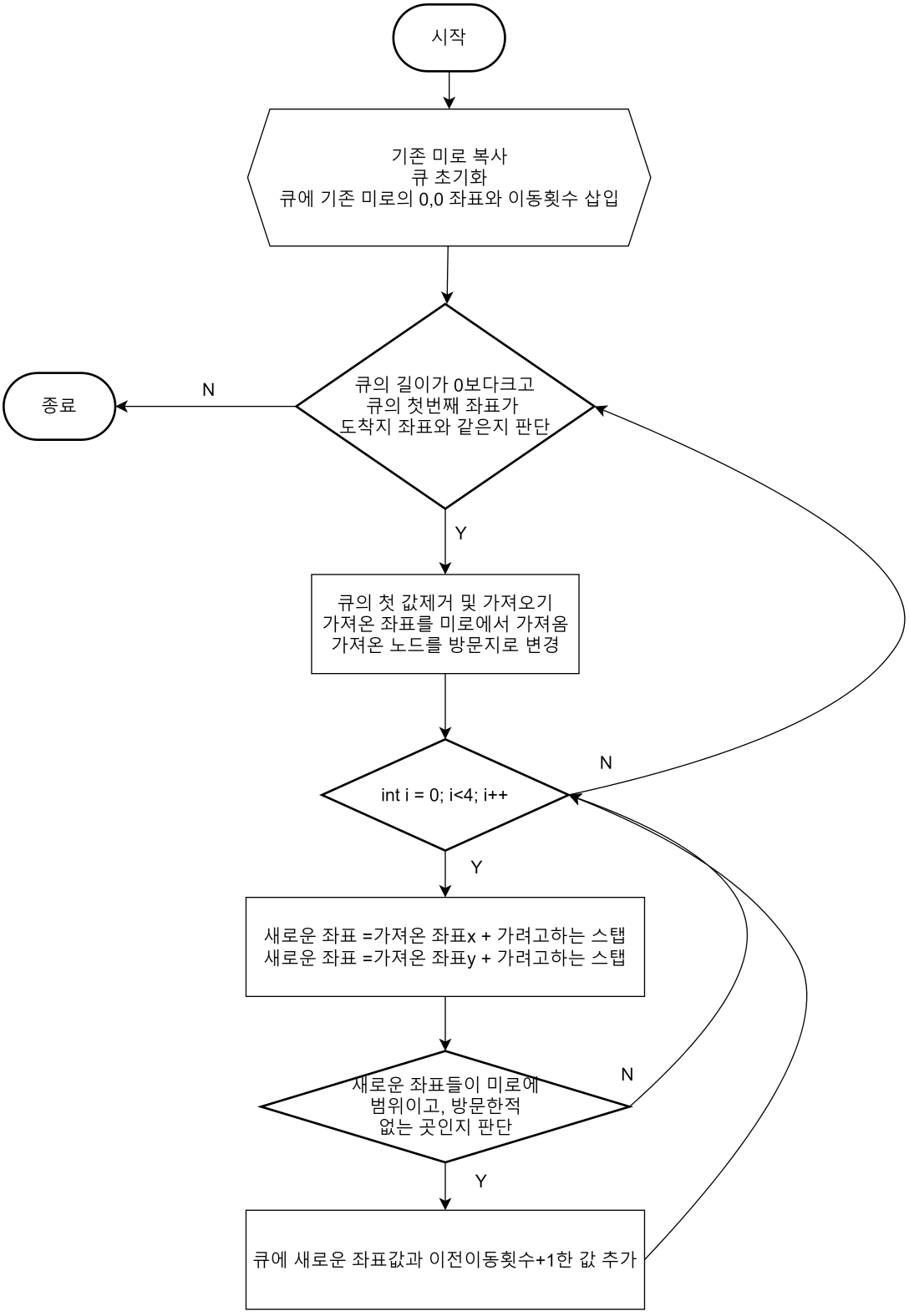
```
// 미로생성
#createdMaze(mazeCell){
    mazeCell.type = true; // 시작을 길로 바꿔줌
    while (mazeCell.routeList.length > 0){ // 갈 수 있는 경로수가 0이면 종료
        let next = mazeCell.getNext(2, 0); // 경로를 랜덤으로 들고옴
        //만약 미로 사이즈를 넘거나, 방문했던 곳이면 지나감
        if (!this.#isMazeRouteRange(next[0], next[1]) || maze[next[0]][next[1]].type){
            continue;
        }
        // 아니라면 다음 경로를 길로 바꿔주고, 벽도 길로 바꿔줌
        let cur = maze[next[0]][next[1]];
        cur.type = true;
        let wall = mazeCell.getNext(1, next[2]);
        maze[wall[0]][wall[1]].type = true;
        this.#createdMaze(cur); // 다음 경로를 다시 재귀호출
    }
}
```



너비 우선 탐색

```
#setShortestRoute() {
    const copy = JSON.parse(JSON.stringify(maze)); // 배열 복사(깊은 복사)
    const q = []; // 큐
    q.push([0, 0, 1]); // x, y, 이동했는 횟수

    while (q.length) { // 큐의 길이가 0이면 정지
        let pop = q.shift(); // 큐의 가장 앞 값 가져오기
        if (pop[0] == size - 1 && pop[1] == size - 1) {// 꺼낸 값이 도착지랑 동일한지 비교
            shortestRoute = pop[2];  // 동일하면 지금까지의 경로를 오기위해 거친 이동횟수 반환
            break; // 정지
        }
        copy[pop[0]][pop[1]].type = false; // 지나온 길 표시 배열을 깊은복사해서 변경해도됨
        for (let i = 0; i < 4; i++) { // 양방향 탐색 위 아래 왼 오
            let nx = pop[0] + x[i]; // x, y 배열 미리 정의해둠
            let ny = pop[1] + y[i]; // 동일
            // isMazeRouteRange [2차원 배열 사이즈를 넘어가는지 확인하는 메서드]
```

```
            // type이 true 면 갈 수있음
            if (this.#isMazeRouteRange(nx, ny) && copy[nx][ny].type) {
                q.push([nx, ny, pop[2] + 1]); //다음 좌표와 현재 이동횟수 + 1 해서 큐 삽입
            }
        }
    }
}
```


시작
기존 미로 복사
큐 초기화
큐에 기존 미로의 0,0 좌표와 이동횟수 삽입
큐의 길이가 0보다크고
큐의 첫번째 좌표가
도착지 좌표와 같은지 판단
N
종료
Y
큐의 첫 값제거 및 가져오기
가져온 좌표를 미로에서 가져옴
가져온 노드를 방문지로 변경
int i = 0; i<4; i++
N
Y
새로운 좌표 =가져온 좌표x + 가려고하는 스탭
새로운 좌표 =가져온 좌표y + 가려고하는 스탭
새로운 좌표들이 미로에
범위이고, 방문한적
없는 곳인지 판단
N
Y
큐에 새로운 좌표값과 이전이동횟수+1한 값 추가