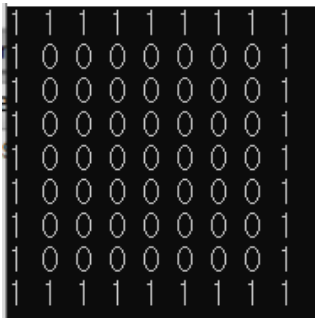


Algorithm

1. Class Maze (information을 저장하는 클래스)

```
1 //201624588 조영우 -> 알고리즘 과제3
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 class Maze {
7 private:
8     bool** check;
9     string input;
10    int size;
11    int numberOfPath;
12 public:
13     Maze(int _size) : size(_size), numberOfPath(0) {
14         check = new bool* [size + 2];
15         for (int i = 0; i < size + 2; i++) { check[i] = new bool[size + 2]{ false }; }
16
17         //init
18         for (int i = 0; i < size + 1; i++) {
19             check[0][i] = true;
20             check[i][size+1] = true;
21             check[i+1][0] = true;
22             check[size+1][i+1] = true;
23         }
24         cin >> input; //string 받기
25     }
26     ~Maze() {
27         for (int i = 0; i < size; ++i) { delete[] check[i]; }
28         delete[] check;
29     }
30
31     void print() const { cout << numberOfPath << endl; }
32
33     bool ladle(int i, int j); //뚝 : 곳자
34     bool promising(int i, int j, int step);
35     void escape(int i, int j, int step);
36 };
37
```

Class Maze에는 지나간 자리를 체크하는 이중배열 데이터가 존재하고, input을 받는 string 객체, maze의 size 그리고 총 Path의 개수를 세는 numberOfPath 멤버 변수가 존재한다. 과제의 알고리즘에 해당하는 escape, promising, ladle function의 내용은 뒷부분에 기재하였다.



Constructor에서 check 배열의 크기에서 벽면을 의미하는 데이터를 추가하여 메모리를 할당하고, 벽면에 해당하는 부분을 true로 채워준다. 이렇게 되면 index와 행렬 데이터가 동일하기에 좌표를 이해하기 쉽고 벽면에 따른 예외는 신경 쓰지 않아도 된다. string input data를 input에 cin을 사용하여 넣어준다. **왼쪽의 캡처와 같이 지나간 자리는 true로 표시하고, 지나가지 않은 자리는 false로 표시한다.**

2. Maze::Escape() : 탈출하는 함수

```
81 void Maze::escape(int i, int j, int step) {  
82     //step는 [0]번째 string을 읽어서 가동함을 의미함  
83     if (promising(i, j, step)) {  
84         check[i][j] = true;  
85         if (input[step] == '?') {  
86             escape(i - 1, j, step + 1); //Up  
87             escape(i, j + 1, step + 1); //Right  
88             escape(i + 1, j, step + 1); //Down  
89             escape(i, j - 1, step + 1); //Left  
90         }  
91         else if (input[step] == 'U') {  
92             escape(i - 1, j, step + 1); //Up  
93         }  
94         else if (input[step] == 'R') {  
95             escape(i, j + 1, step + 1); //Right  
96         }  
97         else if (input[step] == 'L') {  
98             escape(i, j - 1, step + 1); //Left  
99         }  
100        else if (input[step] == 'D') {  
101            escape(i + 1, j, step + 1); //Down  
102        }  
103        check[i][j] = false;  
104    }  
105    else if (step == 48 && i == 7 && j == 1) {  
106        numberOfPath++;  
107        return;  
108    }  
109 }  
110
```

Escape function의 경우 check[i][j]를 판단할 때 사용하는 i행, j열 그리고 input string data를 48개를 모두 읽고 (7,1)에 도달해야 하기에 이것이 promising(유망한, 그곳으로 갈 수 있는)한지 확인하고 갈 수 있다면 그 자리를 지나간 자리를 의미하는 true로 설정하고, 해당 input string data의 순서에 맞게 '?'라면 상우하좌 방향으로 재귀 호출하고, 'U', 'R', 'L', 'D' 라면 해당 방향에 맞게 escape function을 재귀 호출한다. 그리고 지나간 자리를 돌아갈 때에는 false로 BackTracking한다.

그리고 모든 지점을 방문하여 (7,1)에 도달하였다면, 모든 check배열은 true로 가득 차 있고, 그 경우는 nonpromising하다. 그렇다면 해당 else if 문을 들어와 48step & (7,1)인지를 확인하게 된다. 맞다면 numberOfPath를 +1 증가시키고 함수를 return 하게 된다. 재귀호출의 break point중 하나다. 그리고 if문의 논리 특징을 이용하여 &&(and)로 묶인 데이터는 처음 비교문이 거짓이면 뒤의 비교문은 체크하지 않는 효율이 있기에, step==48 인 경우를 먼저 비교하여 조금이나마 효율을 높였다.

3. Maze::Promising() : 효율을 만드는 핵심 → 해당좌표는 Path를 찾기 적당한 좌표인가? (코드에 번호를 표시!)

```

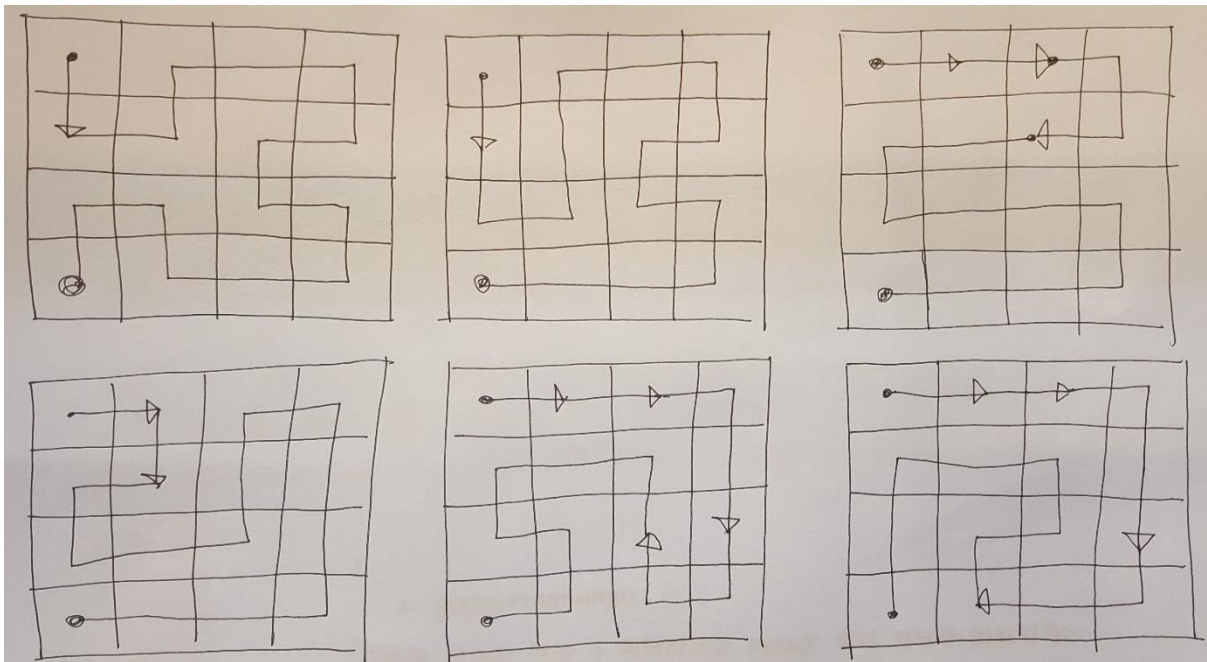
74 bool Maze::promising(int i, int j, int step) {
75     ① if (check[i][j] == true) { return false; }
76     ② if (check[i][j+1]==check[i][j-1] && check[i+1][j]==check[i-1][j]) { return false; }
77     ③ if (i == 7 && j == 1 && step != 48) { return false; }
78
79     ④ if(check[i-1][j-1] == 0){ return ladle(i-1, j-1); }
80     if(check[i-1][j+1] == 0){ return ladle(i-1, j+1); }
81     if(i!=6 && j!=2){
82         if(check[i+1][j-1] == 0){ return ladle(i+1, j-1); }
83     }
84     if(check[i+1][j+1] == 0){ return ladle(i+1, j+1); }
85
86     return true;
87 }

```

(1) 일단 해당 좌표가 갈 수 있는 좌표인지를 확인한다.

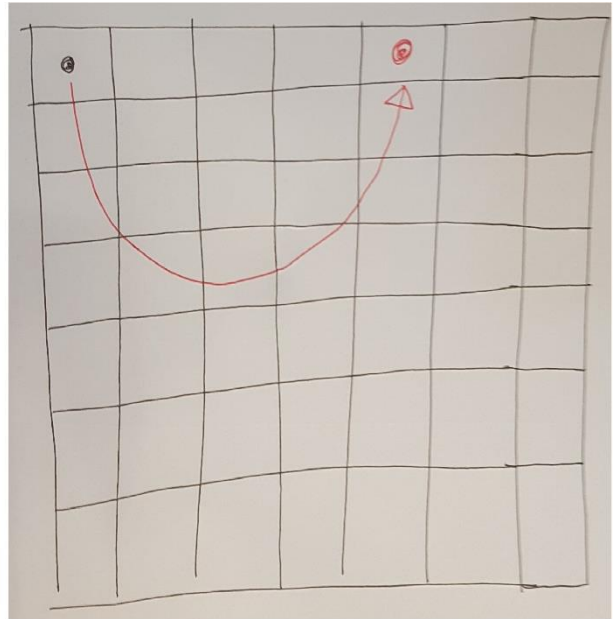
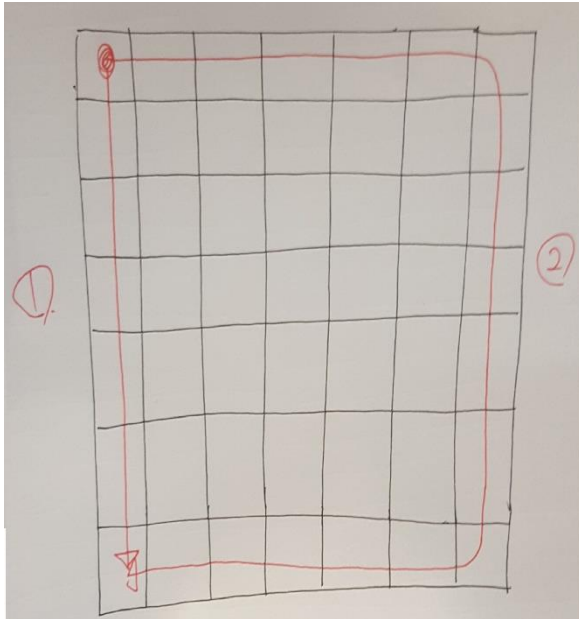
지금 도착한 좌표가 이미 지나간 자리이면 false를 return한다. 이렇게 되면 범위를 벗어난 경우 또한 true로 표시하였기 때문에 0행, 0열, 8행, 8열 또한 지나갈 수 없다.

(2) 첫번째 핵심이다. 알고리즘을 생각한 스토리 형식으로 서술하였다.



우선, 크기가 4인 matrix로 몇 가지 경로를 표시하였다. 표시함으로 보이는 경로의 특징이 존재하였다. 그것은 해당 matrix의 테두리를 의미하는 좌표가 sequential하게 true가 되어야 된다는 사실이다. 예를 들어, 크기가 7x7 matrix의 경우 (1,1)에서 (7,1)로 가는 모든 경로를 찾을 것이고, 아래 왼쪽그림에서의 ① → 여기서 1열에 해당하는 check배열의 경우 (1,1), (2,1), (3,1), ... (7,1) 순서로 지나가야 하는 것이다. 그리고 이와 별개로 밑의 왼쪽 그림의 ②는 어느 지점을 지나고 오더라도 테두리 좌표만 바라보았을 때 다음과 같은 순서대로 지나가야 한다는 것이다. 결국 (1,1)에서 시

작한 경로는 $(1,1) \rightarrow (1,7) \rightarrow (7,7) \rightarrow (7,1)$ 순서대로 경로를 지나가야 하고 그러기 위해서는 2가지 경우에 맞게 경로를 지나가야 한다는 것을 발견하였다.



이유는 오른쪽 그림과 같이 2가지 경우에 해당하지 않게 임의의 테두리 좌표에 도착하게 되면, 그 경로에 의해 공간이 두 공간으로 나누어 지게 되고, 이 나누어진 공간을 더 이상 가거나 돌아올 수 있는 경로가 존재하지 않게 되는 것이다.

그렇다면 이런 경우가 중간지점에서는 발견되지 않을까? 발견된다!! 위의 경우를 포함하며, 임의의 지점에서 다음과 같은 공간분리 현상을 nonpromising으로 가지치기를 하여서 나온 알고리즘 코드는 다음과 같다.

if (check[i][j-1]==check[i][j+1] && check[i-1][j]==check[i+1][j]) { return false; }

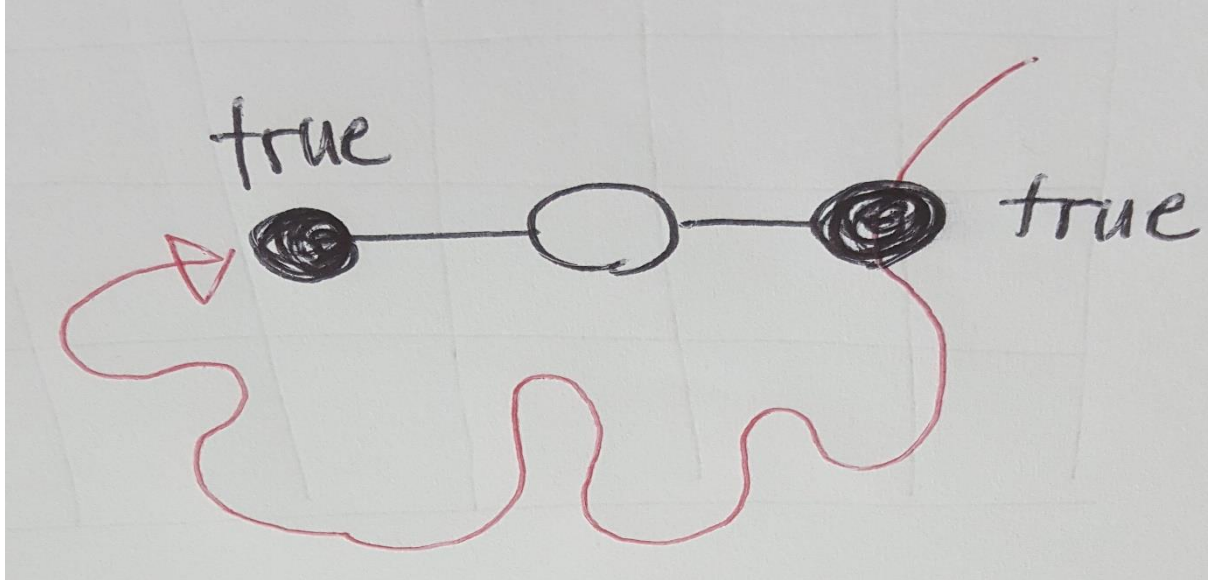
㉑ 내가 (i, j) 지점으로 도착하였다면, 동서남북의 좌표가 모두 0인 경우는 논리상 존재할 수 없다. 공중낙하는 허용하지 않기 때문이다.

㉒ 내가 서쪽에서 왔다면 서쪽은 true이고, 이때 동쪽이 true인 점을 판단할 것이다. 이때, 북쪽과 남쪽의 true, false여부가 같다면 논리적으로 nonpromising이다.

㉓ 만약 북쪽과 남쪽이 true이면, 그리고 그 지점이 정상적으로 오게 된 도착지점이라면 escape function에서 numberOfPath을 +1 증가시킨다. 그 경우가 아니라면 4개의 좌표 안에 갇힌 구조가 될 것이다. 그렇기에 nonpromising하다.

㉔ 북쪽과 남쪽이 false이면, 공간을 두 공간으로 나누게 되어 서로 다른 공간으로 들어가고 나올 수 없는 구조가 된다. **일종의 주머니를 형성하게 되고** 안으로 들어가면 나올 수 없고, 밖으로 나오면 안으로 들어갈 수 없게 되는 것이다. 시작지점에서 true가 된 양쪽 두 지점을 모두 지나가려

면 경로가 존재하고 그 경로는 경로 안 좌표와 경로 밖 좌표를 두 공간으로 나누게 되고, 내가 현재 서 있는 좌표의 양 옆의 두개의 true지점 사이에 있다면, 북쪽으로 가든 남쪽으로 가든 다른 공간의 좌표로는 더 이상 갈 수 없게 된다. 그렇기에 nonpromising이다. (아래의 그림을 보자!)

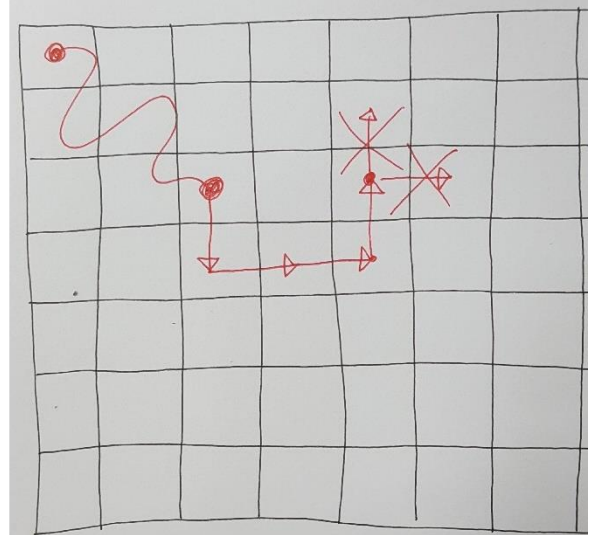
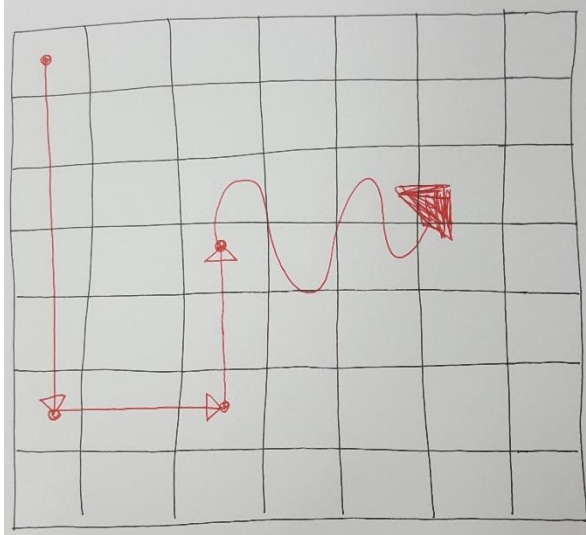


㉔ 이렇게 되면 동서남북 어느 방향에서 오는 위의 알고리즘은 동일하게 nonpromising하고 그것을 잡아낼 수 있다. 그리고 테두리를 넘어선 벽면에 해당하는 좌표 또한 메모리를 할당하고 그곳에 true로 표시하여 테두리에 해당하는 순차적 접근 알고리즘 또한 포함하고 있다.

(3) 그리고 마지막 도착지점인 (7,1)에 도착했는데, 그것이 마지막까지 데이터를 읽어서 step이 48이 된 것이 아니라면 모든 좌표를 path한 것이 아니기 때문에 nonpromising이다.

(4)두번째 핵심이다. 가지치기 핵심 → 국자모양은 갈 필요가 없다!

위의 (1), (2), (3)의 조건만으로는 쓸데없는 전진을 계속 이어가게 된다.



왼쪽 그림을 보면 국자형태로 (5, 2)는 이제 절대 갈 수 없고, 이 경로로는 단 한 경우도 나올 수 없다는 것을 논리적으로 알 수 있다. 하지만 (1),(2),(3) 조건만으로는 여기서 R과 U을 하게 되고 이로부터 수많은 경로를 지나게 되는 것을 발견하였다.

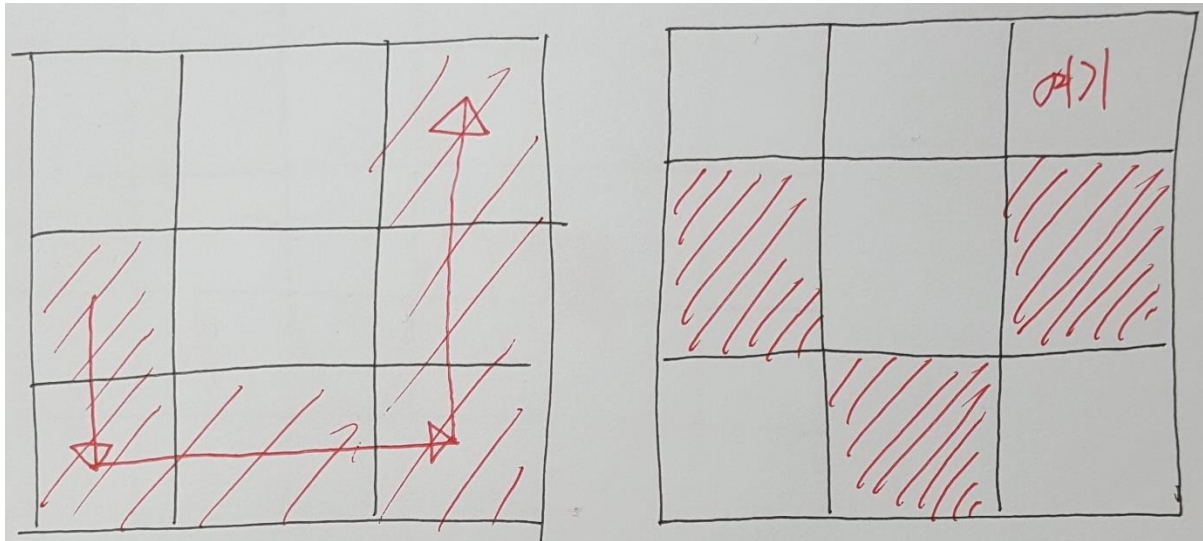
여기서 핵심은 오른쪽 그림과 같이 어떤 점에 도착하였을 때 국자형태를 만들면 더 이상 갈 필요 없이 **nonpromising**를 선언하는 것이다.

여기서 나는 왼쪽그림과 같이 (4,3) 지점에 와있다고 하면, 나는 어떤 좌표로부터 지금 좌표로 오게 되었는지 모른다. 그렇기에 각 대각선에 위치한 (3,2), (3,4), (5,2), (5,4) 4개 중에서 국자안에 갇힌 좌표가 있다면 **nonpromising**을 선언할 것이다.

그렇기에 위의 코드와 같이 (i, j)좌표가 promising한지를 체크할 때 우선 각 대각선에 위치한 좌표가 지나갈 수 있는 **true**인지를 확인하고 **false**이면 신경 갇힌 구조가 아니기에 if문을 탈출하고 **true**이면 **ladle(국자) function**을 호출한다.

Ladle function은 국자안에 갇힌 형태가 되면 false를 호출하는 함수이다.

국자안에 갇힌 구조는 어떤 구조인가?



왼쪽그림과 같이 중간에 갇힌 좌표가 존재하고 대각선의 좌표를 진행중이면 국자모양이다. 여기서 중간좌표로 갈 수 없는 이유는 좌, 우, 하가 막혀 있기 때문이다.

그리고 더 나아가 좌, 우, 하가 막혀 있으면 나는 "여기"를 포함하여, 대각선 4곳 어디에 있든 무조건 nonpromising이다. 그렇기에 symmetric하거나 상하좌우 반전이 되어 도 nonpromising은 똑같이 만족한다.

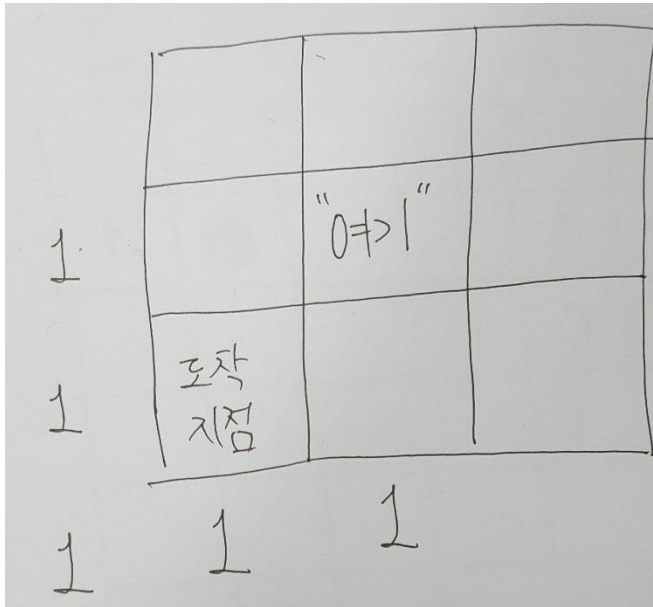
```
50 bool Maze::ladle(int i, int j){
51     if(check[i][j-1] == check[i][j+1] && check[i][j+1] == true){
52         if(check[i-1][j] != check[i+1][j]) {return false;}
53         return true;
54     }
55     else if(check[i-1][j] == check[i+1][j] && check[i+1][j] == true){
56         if(check[i][j-1] != check[i][j+1]) {return false;}
57         return true;
58     }
59     return true;
60 }
```

그렇기에 해당 대각선의 좌표로 ladle함수를 호출하게 되면 왼쪽과 오른쪽이 지나간 자리(true) 이고 위와 아래가 서로 다른 true, false 상태이면 오른쪽 그림과 같은 상태이고 각 대각선 4곳의 어디에서 ladle를 호출하든 nonpromising가 되기에 false를 반환하며 함수를 종료한다.

Else if의 경우는 90도 회전시킨 상태로서 위와 아래가 true로 같고, 왼쪽과 오른쪽이 서로 다르게 true, false 상태로 있는 것이다.

하지만 여기에는 치명적인 오류가 존재한다!!!!!!

그것은 (6,2)지점에서 promising function을 호출하면 국자안에 중앙에 갇히는 형태 중 정상적으로 도착지점에 가는 형태가 nonpromising로 도착하기 전에 가지치기가 되는 것이다.



현재 "여기"좌표에 있는 상태이고 도착지점(7, 1)은 아직 도착하지 않았기에 false이기에 `ladle(7,1)`이 호출된다. 여기서 "여기"의 왼쪽과 아래가 모두 false상태였으면 문제가 되지 않지만, 둘 중에 하나가 true이라면 정상적으로 가는 길이 가지치기가 된다.

예를 들어 "여기" 왼쪽이 지나간 true자리이면 "여기"에서 아래로 간 뒤 왼쪽으로 가는 정상적인 경로가 가지치기가 되는 것이다.

그렇기에 (6행2열)에서 도착지점이 국자 중앙이 되는 `ladle(i+1, j-1)` 호출은 이루어 지면 안된다. 그렇기에 조건문을 사용하여 막아 놓았다.


4. Main 함수


```
39 int main() {  
40     Maze maze(7);  
41     maze.escape(1, 1, 0);  
42     maze.print();  
43  
44     return 0;  
45 }  
46
```


Class로 캡슐화를 상승하였기에, maze객체를 생성하고, escape function을 (1,1)에서 step0로 시작 하라는 의미의 parameter를 전달하고 numberOfPath를 출력하는 main function이다.


5. 실행 결과 (개발 환경: CodeBlock)


test_data.txt의 데이터를 각각 입력하여 실행한 결과이다. 모든 input data에 대하여 1초도 채 걸리지 않는 빠른 시간안에 결과 값을 내게 되는 효율적인 알고리즘 코드이다.


(1)  C:\Users\whdud\Documents\algo3.exe
????D???L??D?L????????????R??D????????D??RD
0

(2)  C:\Users\whdud\Documents\algo3.exe
????????????????UL????U?L??L?D???RR????????
1

(3)  C:\Users\whdud\Documents\algo3.exe
??????R??????U????????????????????LD????D?
201

(4)  C:\Users\whdud\Documents\algo3.exe
??R?
0

(5)  C:\Users\whdud\Documents\algo3.exe
???????????L????????????????????????????
13048

(6)  C:\Users\whdud\Documents\algo3.exe
????U????????D????????????????????
6665

시간 테스트 출력

(#include <ctime> 로부터 알고리즘의 시간을 clock(ms)로 측정해보았다.)

```
C:\Users\whdud\Documents\algo3.exe
????????????????????????????????????????????????????????
648ms
88418

C:\Users\whdud\Documents\algo3.exe
????????????????????????????????????????????????????????R?
643ms
0

C:\Users\whdud\Documents\algo3.exe
?????????????L????????????????????????????????????????
105ms
13048
```

모든 입력 데이터는 0.7초도 걸리지 않는다.