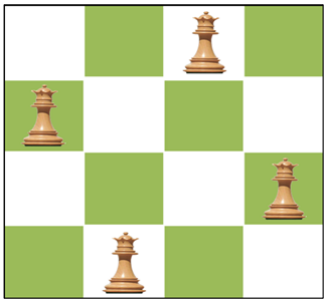
문제 2

n-queen

전산학에서 백트래킹 문제로 n-queen problem이 유명하다.
이 문제는 n*n체스 보드판에 n개의 queen을 서로 공격하지 못하도록 배치하는 방법을 찾아내는 문제이다.

아래 그림은 n이 4일 경우 queen을 서로 공격하지 못하게 배치한 한 예를 나타낸다.



체스판 크기 및 queen의 수를 나타내는 n을 입력받아서 서로 공격하지 못하도록 배치하는 총 방법의 수를 구하는 프로그램을 작성하시오.

입력

정수 n이 입력으로 들어온다.(3 <= n <= 9)

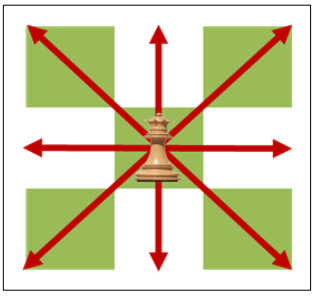
출력

서로 다른 총 경우의 수를 출력한다.

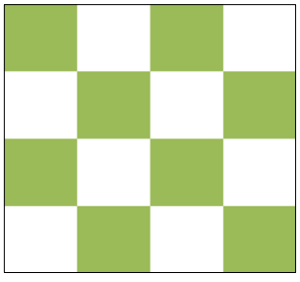
입력 예	출력 예
4	2

풀이

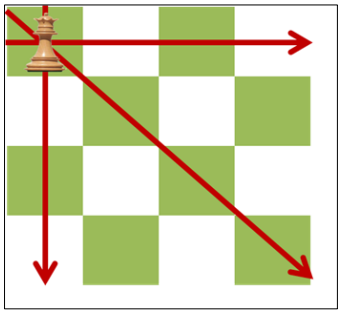
일단 이 문제를 풀기 위해서 퀸이 공격할 수 있는 위치에 대한 생각을 해야 한다. 일단 퀸이 공격할 수 있는 루트는 다음과 같다. (8방향으로 체스판의 마지막 칸까지 모두 공격 가능하다.)



이 문제를 해결하기 위하여 확실한 것은 한 행에 하나 이상의 퀸을 놓을 수 없다는 것이다. 4*4의 체스판을 살펴보자.



위 체스판에서 1행 1열에 하나의 퀸을 배치하면 공격범위는 아래 화살표와 같으며 화살표가 지나는 칸에는 퀸을 놓을 수 없다.



따라서 다음과 같은 방법을 활용할 수 있다.

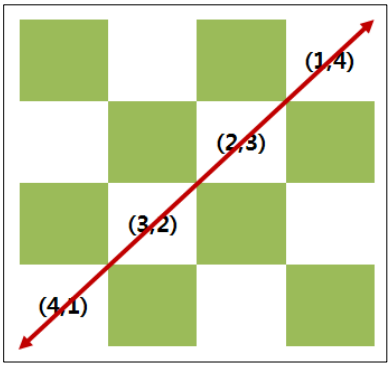
1. 첫 번째 행, 첫 번째 열에 퀸을 놓는다.
2. 다음 행에서 가능한 가장 왼쪽 열에 퀸을 놓는다.
3. n번째 열에 더 이상 퀸을 놓을 수 없다면 백트랙한다.
4. 마지막 행에 퀸을 놓으면 하나의 해를 구한 것이다.
5. 모든 경우를 조사할 때까지 백트래킹해가며 해들을 구한다.

위 방법으로 깊이우선탐색하며 해를 구할 때 마다 카운트하면 원하는 해를 구할 수 있다.

알고리즘 작성 시 주의할 점은 퀸을 놓을 수 있는지 없는지 판단하는 절차를 효율적으로 작성해야 한다.

이 풀이에서는 행은 검사할 필요가 없으므로, 열과 대각선만 검사하면 된다. 열을 검사하는 방법은 크기가 n인 체크배열을 만들어 k번째 열에 퀸을 놓았다면 배열의 k번째 위치를 체크한다. 체크하는 이유는 이후의 행에서는 체크된 열에 퀸을 놓지 않도록 하기 위함이다.

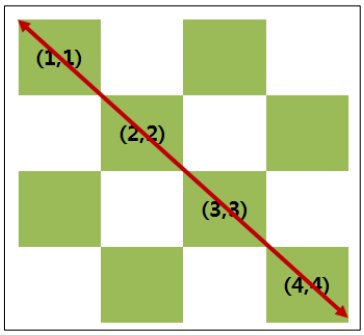
대각선은 기울기가 증가하는 대각선 부분과 기울기가 감소하는 부분의 2가지 대각선이 존재한다. 이 2가지 대각선에 대해서도 체크배열을 만들어서 활용할 수 있다. 기울기가 증가하는 대각선부터 살펴보면 다음과 같다.



위 대각선 상에 있는 칸의 특징을 보면 행+열의 값이 일정하다. n이 4일 경우 행+열의 최소값은 2이고 최댓값은 8이다.

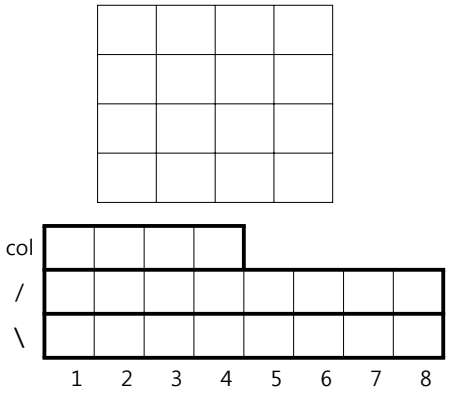
따라서 기울기가 증가하는 대각선은 체크배열의 행+열 위치에 체크하여 기울기가 증가하는 대각선 상에 퀸을 놓을 수 있는지 없는지를 쉽게 확인할 수 있다.

기울기가 감소하는 대각선도 아래와 같은 특징이 있다.



기울기가 감소하는 대각선 부분은 행과 열의 차가 일정하다. 범위는 n이 4일 경우 −3에서 3까지의 값을 지닌다. 음의 값을 양의 값으로 보정하기 위해 n을 더해주어 체크배열의 n+(행−열)의 위치에 체크하여, 퀸이 놓일 수 있는지 여부를 확인할 수 있다.

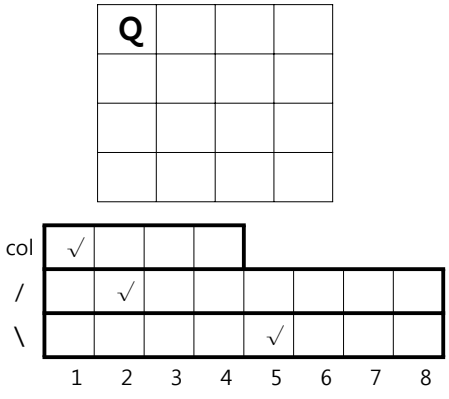
각 단계별 진행 과정은 다음과 같다.



[준비 상태]

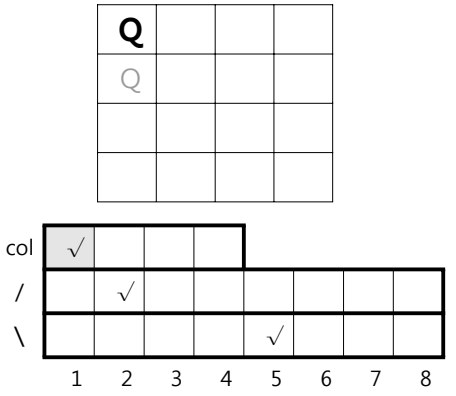
사실 위에 있는 4*4의 체스판은 실제로 구현하지 않는다. 실제로 체스판으로 구현할 수도 있지만 이 방법보다는 여기서 소개하는 방법이 훨씬 효율적이며 속도도 빠르다.

col : 행, / : 대각선 1, \ : 대각선 2의 상태를 나타낸다.



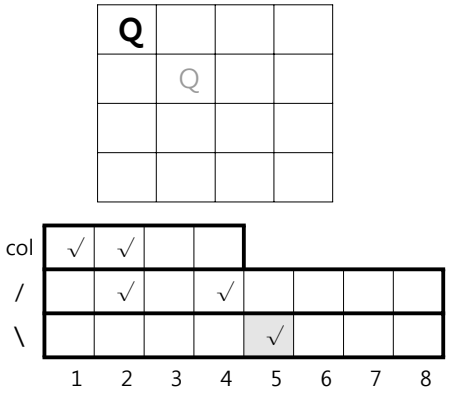
[1단계]
1행 1열에 퀸을 하나 놓고,

col에 1열을 사용했기 때문에 col[1] 체크
대각선 1은 inc[1+1]에 체크
대각선 2는 dec[4-(1-1)]에 체크 (n=4이므로)



[2개 놓기]

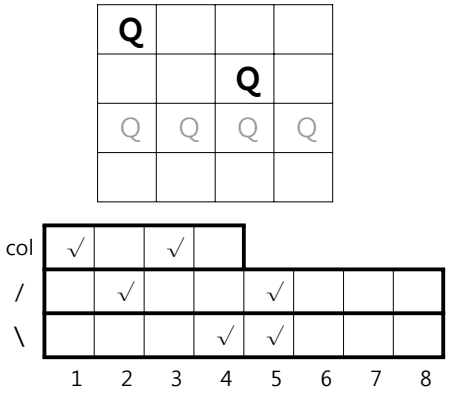
2행 1열에 퀸을 놓아보자.
col[1]이 이미 체크되어 있으므로 놓을 수 없다.



[2개 놓기]

다음으로 2행 2열에 퀸을 놓아보자.

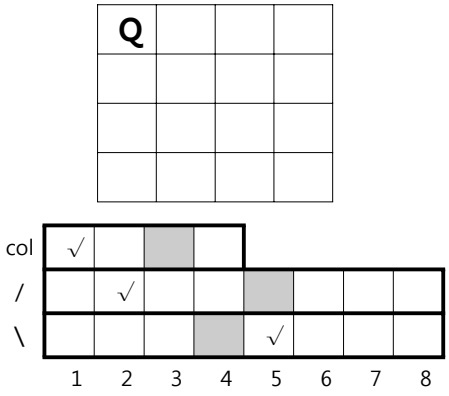
col[2]는 체크 안 되었으므로 OK!
inc[2+2]도 체크 안 되어 있으므로 OK!
dec[4-(2+2)+1]가 이미 체크되었음. 즉 기울기가 감소하는 대각선에 퀸이 있다는 의미이므로 불가!



[3개 놓기]

3행에는 1, 2, 3, 4열 모두 각각 체크 배열에 의해서 놓을 수 있는 위치가 없으므로 3행에는 퀸을 놓을 수 없다.

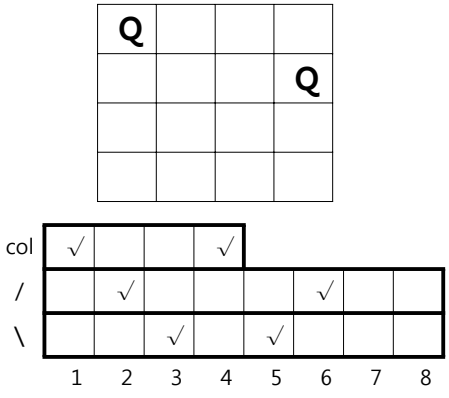
따라서 백트랙!!!



[백트랙]

백트랙 시에 가장 중요한 점은 체크배열에 기록해 두었던 체크를 모두 해제해야 한다는 점이다.

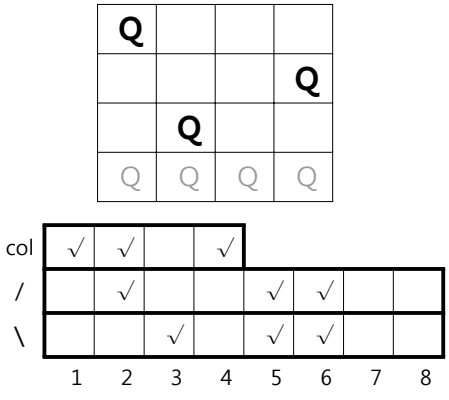
비선형구조의 탐색에서 복귀 시에 흔적을 지우는 것은 매우 중요한 요소이므로 익힐 수 있도록 한다.



[2개 놓기]

2행 3열까지는 아까 두었으므로, 2행 4열에 도전!!

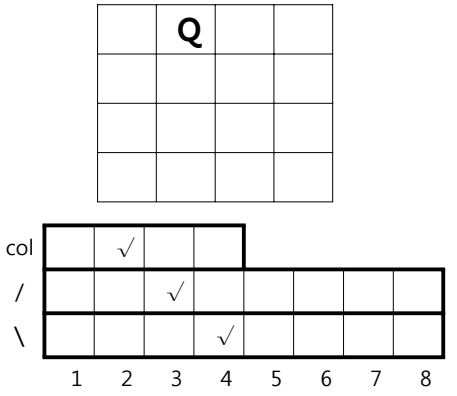
col[4], inc[2+4], dec[4-(2-4)+1] 모두 비었으므로 둘 수 있음.



[3개 놓기]

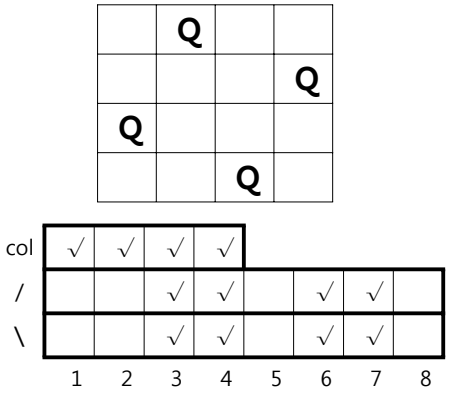
다음으로 3행 1열은 퀸을 놓을 수 없고, 3행 2열에 퀸을 놓을 수 있다.

마지막으로 4행에는 퀸을 놓을 수 있는 방법이 없으므로, 결국은 백트랙을 2번 하여 결국 1행 2열에 다시 놓게 된다.



[1개 놓기]

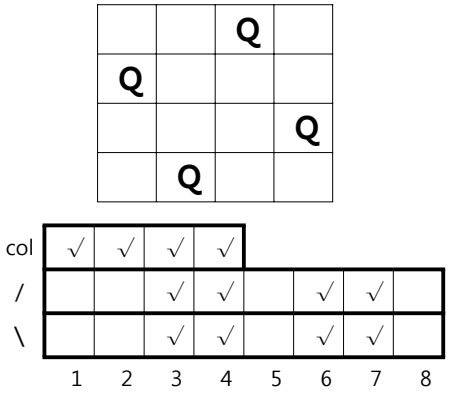
1행 1열에 두면 가능한 방법이 없으므로, 다시 모두 백트랙한 후, 1행 2열에 놓고 다시 진행을 시작한다.



[계속 놓기]

다음으로 연속으로 깊이우선탐색을 진행하면 2행 4열, 3행 1열, 4행 3열에 각각 하나씩 퀸을 놓을 수 있고 한 가지의 가능한 경우를 찾을 수 있다.

다시 다른 해를 찾기 위해서 다시 백트랙 하여 계속 진행한다.



[계속 놓기]

마지막으로 1행 3열, 2행 1열, 3행 4열, 4행 2열로 또 다른 해를 찾을 수 있다.

따라서 모두 2가지의 서로 다른 경우를 발견할 수 있다.

이 방법을 종합하여 깊이우선탐색으로 해결한 소스코드는 다음과 같다.

줄	코드	참고
1	#include<stdio.h>	9: 마지막 행까지 다 놓았으면 해를 추가 10: 백트랙 12: r행에 대해서 각 열에 놓기 시도 15: 체크 17: 백트랙 후 흔적 제거(매우 중요)
2		
3	int n, ans, col[10], inc[20], dec[20];	
4		
5	void solve(int r)	
6	{	
7	if(r>n)	
8	{	
9	ans++;	
10	return;	
11	}	
12	for(int i=1; i<=n; i++)	
13	if(!col[i] && !inc[r+i] && !dec[n+(r-i)+1])	
14	{	
15	col[i]=inc[r+i]=dec[n+(r-i)+1]=1;	
16	solve(r+1);	
17	col[i]=inc[r+i]=dec[n+(r-i)+1]=0;	
18	}	
19	}	
20		
21	int main()	
22	{	
23	scanf("%d", &n);	
24	solve(1);	
25	printf("%d", ans);	
26	}	

위 소스코드는 깊이우선탐색을 기반으로 퀸을 더 이상 못 놓는 상태라면 이전 상태로 백트랙하여 가능한 상태가 될 때까지 반복하는 것을 구현한 것이다.

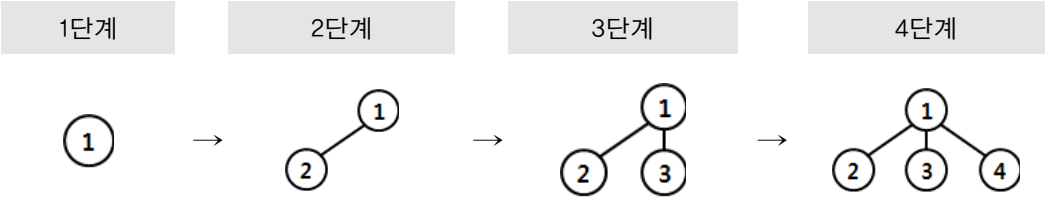
알고리즘의 효율을 높이기 위하여 퀸을 놓을 수 있는지 없는지를 $O(1)$만에 계산하기 위해, 현재 상태를 col, inc, dec라는 3개의 배열에 각각 열, 대각선 2가지의 상태를 저장하여 매우 빠른 속도로 처리할 수 있도록 하였다.

여기서 특별히 중요한 점은 다음 전체탐색을 위한 백트랙을 진행하면서 이전 전체탐색의 흔적을 지워야 한다는 것이다. 이 코드에서는 17행이 그 일을 하고 있다. 이 부분은 문

제의 특성에 따라 매우 중요할 수 있으므로 이 소스를 반드시 이해하여 활용할 수 있도록 해야 한다.

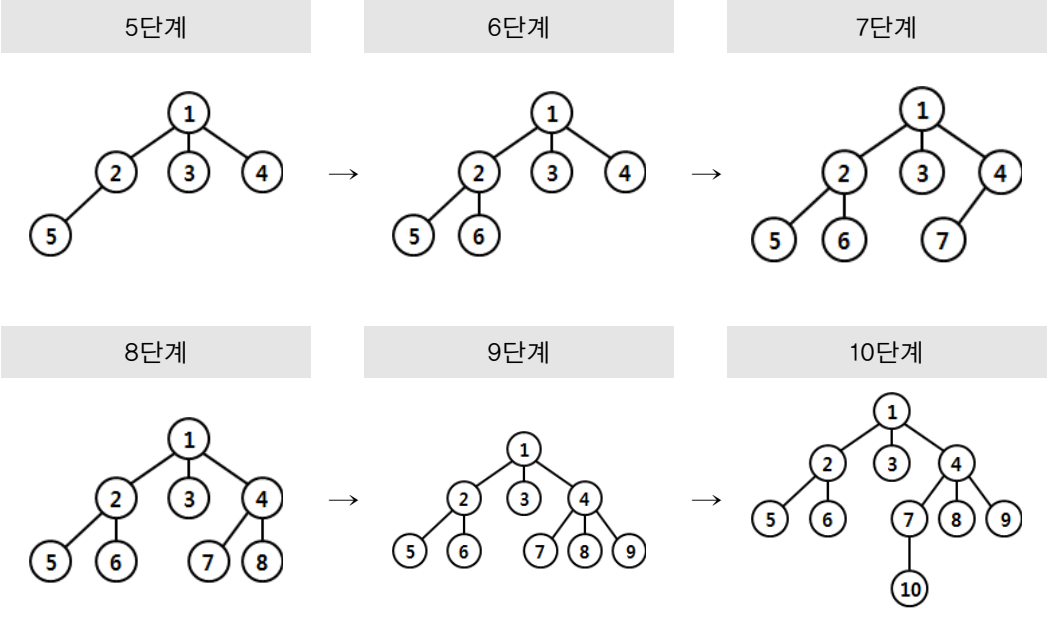
- 너비우선탐색(bfs)

너비우선탐색은 깊이우선탐색과는 달리 현재 정점에서 깊이가 1인 정점을 모두 탐색한 뒤 깊이를 늘려가는 방식이다. 73페이지의 '10개의 정점과 9개의 간선을 가진 트리'를 통해서 너비우선탐색을 살펴보자.



너비우선 1 ~ 4 단계

먼저 1단계부터 4단계까지를 살펴보면 1에서 출발하여 깊이가 1인 세 정점을 모두 순차적으로 방문한다. 계속해서 너비우선탐색의 결과를 살펴보면 다음과 같다.



너비우선 5 ~ 10 단계

너비우선탐색은 백트랙을 하지 않는다. 대신에 현재 정점에서 깊이가 1인 정점을 모두 방문해야 하므로 큐(queue)라는 선입선출(FIFO) 자료구조를 활용하여 현재 정점에서 깊이가 1 더 깊은 모든 정점을 순차적으로 큐에 저장하여 탐색에 활용한다. 따라서 STL에서 제공하는 std::queue()를 활용하는 방법을 익힐 필요가 있다.

너비우선탐색 알고리즘은 다음과 같다.

줄	코드	참고
1	`#include <queue>`	1: std::queue를 이용하기 위함 2: 방문했는지 체크해 두는 배열 5: Queue를 선언 6: 출발 정점을 Queue에 삽입 7: Queue가 빌 때까지 반복 9: Queue에서 하나 삭제 10: 연결된 정점 모두 검사 11: 아직 방문하지 않았으면, 13: 체크 후 Queue에 추가
2	`bool visited[101];`	
3	`void bfs(int k)`	
4	`{`	
5	`  std::queue<int> Q;`	
6	`  Q.push(k), visited[k]=1;`	
7	`  while(!Q.empty())`	
8	`  {`	
9	`    int current=Q.front(); Q.pop();`	
10	`    for(int i=0; i<G[current].size(); i++)`	
11	`      if(!visited[G[current][i]])`	
12	`      {`	
13	`        visited[G[current][i]]=1;`	
14	`        Q.push(G[current][i]);`	
15	`      }`	
16	`  }`	
17	`}`	

이 방법은 그래프를 인접리스트에 저장했을 경우에 활용할 수 있으며, 전체를 탐색하는데 있어서 반복문의 실행횟수는 모두 m 번이 된다. 따라서 일반적으로 속도가 더 빠르기 때문에 자주 활용된다. 만약 인접행렬로 그래프를 저장했다면 다음과 같이 작성하면 된다.

하지만 표준 라이브러리(standard library)에 정의된 자료구조인 스택, 큐 등은 C++에서 쉽게 활용할 수 있지만 직접 구현하는 것보다 속도가 느리기 때문에 문제의 특성에 따라서 직접 구현하여 활용하는 것이 좋을 수도 있다.

줄	코드	참고
1	`#include <queue>`	1: std::queue()를 이용하기 위함 2: 방문했는지 체크해 두는 배열 5: Queue를 선언 6: 출발 정점을 Queue에 삽입 7: Queue가 빌 때까지 반복 9: Queue에서 하나 삭제 10: 모든 정점에 대해 검사 11:검사하는 정점이 현재 정점과 연결되어 있고, 아직 방문하지 않았으면 13: 체크 후 Queue에 추가
2	`bool visited[101];`	
3	`void bfs(int k)`	
4	`{`	
5	`  std::queue<int> Q;`	
6	`  Q.push(k), visited[k]=1;`	
7	`  while(!Q.empty())`	
8	`  {`	
9	`    int current=Q.front(); Q.pop();`	
10	`    for(int i=1; i<=n; i++)`	
11	`      if(G[current][i] && !visited[G[current][i]])`	
12	`      {`	
13	`        visited[G[current][i]]=1;`	
14	`        Q.push(G[current][i]);`	
15	`      }`	
16	`  }`	
17	`}`	

이 방법은 전체를 탐색하는 데 있어서 반복문을 n^2번 실행하게 된다. 따라서 평균적으로 인접리스트보다 느리지만 구현이 간편하므로, n값이 크지 않은 문제라면 충분히 적용할 가치가 있다.

문제 3

두더지 굴(L)

정올이는 땅속의 굴이 모두 연결되어 있으면 이 굴은 한 마리의 두더지가 사는 집이라는 사실을 발견하였다.

정올이는 뒷산에 사는 두더지가 모두 몇 마리인지 궁금해졌다. 정올이는 특수 장비를 이용하여 뒷산의 두더지 굴을 모두 나타낸 지도를 만들 수 있었다.

이 지도는 직사각형이고 가로 세로 영역을 0또는 1로 표현한다. 0은 땅이고 1은 두더지 굴을 나타낸다. 1이 상하좌우로 연결되어 있으면 한 마리의 두더지가 사는 집으로 정의할 수 있다.

0	1	1	0	1	0	0
0	1	1	0	1	0	1
1	1	1	0	1	0	1
0	0	0	0	1	1	1
0	1	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	0	0	0

[그림 1]

0	1	1	0	2	0	0
0	1	1	0	2	0	2
1	1	1	0	2	0	2
0	0	0	0	2	2	2
0	3	0	0	0	0	0
0	3	3	3	3	3	0
0	3	3	3	0	0	0

[그림 2]

[그림 2]는 [그림 1]을 두더지 굴로 번호를 붙인 것이다. 특수촬영 사진 데이터를 입력받아 두더지 굴의 수를 출력하고, 각 두더지 굴의 크기를 오름차순으로 정렬하여 출력하는 프로그램을 작성하시오.

입력

첫 번째 줄에 가로, 세로의 크기를 나타내는 n이 입력된다. n은 30 이하의 자연수 두 번째 줄부터 n줄에 걸쳐서 n개의 0과 1이 공백으로 구분되어 입력된다.

출력

첫째 줄에 두더지 굴의 수를 출력한다. 둘째 줄부터 각 두더지 굴의 크기를 내림차순으로 한 줄에 하나씩 출력한다.

입력 예	출력 예
7 0 1 1 0 1 0 0 0 1 1 0 1 0 1 1 1 1 0 1 0 1 0 0 0 0 1 1 1 0 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0	3 9 8 7

풀이

이 문제는 깊이우선탐색으로 해결했던 문제이다. 하지만 너무 깊은 깊이에 대한 깊이우선탐색의 단점인 runtime error를 방지하기 위해서는 너비우선탐색을 적용할 수 있다. 이번 풀이는 너비우선탐색을 적용하여 이 문제를 해결한다.

기본적인 입력에 대한 그래프 처리 및 문제해결의 전반적인 방법은 앞의 문제를 참고하고, 연결된 정점을 처리하는 방법은 너비우선탐색을 이용한다.

이번 풀이에서는 flood fill을 너비우선탐색으로 처리하는 방법에 대해서 익혀보자. 소스코드는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#include <algorithm>`	
3	`#include <queue>`	
4		
5	`struct VERTEX{ int a, b; };`	
6	`int n, A[101][101], cnt, Size[101];`	
7	`int dx[4]={1,0,-1,0}, dy[4]={0,1,0,-1};`	
8		
9	`int main()`	
10	`{`	
11	`  input();`	
12	`  solve();`	
13	`  output();`	
14	`  return 0;`	
15	`}`	

기본적인 변수와 main()함수 부분이다. 깊이우선탐색 때와의 차이점은 VERTEX라는 구조체를 선언한 부분과 queue를 삽입한 것이 차이가 난다. 이 부분은 queue를 활용하여 너비우선탐색을 하기 위하여 추가된 부분이다.

줄	코드	참고
1	`bool safe(int a, int b)`	
2	`{`	
3	`  return (0<=a && a<n) && (0<=b && b<n);`	
4	`}`	
5	`bool cmp(int a, int b)`	
6	`{`	
7	`  return a > b;`	
8	`}`	

이 부분은 깊이우선탐색 때와 변함이 없다.

줄	코드	참고
1	`void bfs(int a, int b, int c)`	
2	`{`	
3	`  std::queue<VERTEX> Q;`	
4	`  Q.push( (VERTEX){a, b} ), A[a][b]=c;`	
5	`  while(!Q.empty())`	
6	`  {`	
7	`    VERTEX curr = Q.front(); Q.pop();`	
8	`    for(int i=0; i<4; i++)`	
9	`      if(safe(curr.a+dx[i], curr.b+dy[i]) &&`	
10	`A[curr.a+dx[i]] [curr.b+dy[i]]==1)`	
11	`      {`	
12	`        A[curr.a+dx[i]][curr.b+dy[i]]=c;`	
13	`        Q.push((VERTEX){curr.a+dx[i], curr.b+dy[i]});`	
14	`      }`	
15	`  }`	
16	`}`	
17		
18	`void solve()`	
19	`{`	
20	`  for(int i=0; i<n; i++)`	
21	`    for(int j=0; j<n; j++)`	
22	`      if(A[i][j]==1)`	
23	`      {`	
24	`        cnt++;`	
25	`        bfs(i,j,cnt+1);`	
26	`      }`	
27	`  for(int i=0; i<n; i++)`	
28	`    for(int j=0; j<n; j++)`	
29	`      if(A[i][j])`	
30	`        Size[A[i][j]-2]++;`	
31	`  std::sort(Size, Size+cnt, cmp);`	
32	`}`	

bfs 함수의 핵심적인 부분이다. 일단 solve 함수에서 A배열의 (0, 0)부터 (n-1, n-1)까지 차례로 검사하면서 만약 굴의 일부가 발견되면, 그 부분으로부터 시작하여 bfs로 연결된 굴을 모두 검사한다.

bfs(a, b, c) : (a, b)의 정점과 연결된 모든 정점들을 c로 칠한다.

다른 부분은 모두 깊이우선탐색과 동일하나 bfs 함수의 내용을 잘 익혀둘 필요가 있다. 일단 시작정점을 큐에 삽입하고, 이 정점에서 4방향으로 연결된 모든 정점을 큐에 저장해 나간다. 이 때, 이미 큐에 들어있는 정점은 다시 넣지 않는다. 이 부분은 큐를 다루는 알고리즘에서 효율에 매우 큰 영향을 미치므로 반드시 익혀둘 수 있도록 한다.

큐에서 구조체를 이용하는 것도 활용도가 높으므로 구조체를 처리하는 부분의 코드들은 익혀두었다가 언제든지 활용할 수 있도록 연습하는 것이 중요하다.

줄	코드	참고
1	`void input()`	
2	`{`	
3	`  scanf("%d", &n);`	
4	`  for(int i=0; i<n; i++)`	
5	`    for(int j=0; j<n; j++)`	
6	`      scanf("%d", &A[i][j]);`	
7	`}`	
8	`void output()`	
9	`{`	
10	`  printf("%d\n", cnt);`	
11	`  for(int i=0; i<cnt; i++)`	
12	`    printf("%d\n", Size[i]);`	
13	`}`	

각 값을 차례로 입력받는 input함수이다. 만약 입력 자료가 공백으로 구분되어 있지 않고 연속적으로 입력된다면 문자열 형태로 받을 수도 있지만 scanf("%1d",&A[i][j]) 로 입력받으면 처리할 수 있다.

출력하는 부분은 먼저 굴의 수를 출력하고, 크기가 큰 굴부터 하나씩 출력한다.

문제 4

미로 찾기

크기가 h*w인 미로가 있다.

이 미로는 길과 벽으로 구성되어 있으며, 길은 ".", 벽은 "#"으로 구성되어 있으며, 시작위치 "S"와 도착위치 "G"가 존재한다.

위에서 제시한 각 정보가 주어질 때, S위치로부터 G위치까지의 최단 거리를 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에 h와 w가 공백으로 구분되어 입력된다.
(단, h, w는 5 이상 100 이하의 자연수이다.)
두 번째 줄부터 h줄에 걸쳐서 w개로 이루어진 문자열이 입력된다.
문자열은 길은 ".", 벽은 "#", 출발점은 "S", 도착점은 "G"로 표시된다. 그리고 S와 G의 위치는 서로 다르다

출력

출발지로부터 도착지까지의 최단거리를 출력한다.
단, 도달할 수 없는 미로일 경우에는 -1을 출력한다.

입력 예	출력 예
5 5 #S### #...# #.#.# #.... ###G#	6

풀이

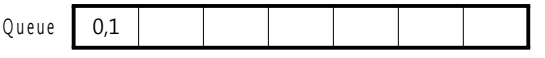
최단경로의 길이 즉, 최단거리를 찾는 문제는 너비우선탐색으로 해결할 수 있는 대표적인 예이다. 특히 이 문제의 경우에는 특별히 가중치 없이 이동하는 칸의 수가 최단거리 이므로 너비우선탐색을 적용하면 쉽게 해결할 수 있는 문제이다.

따라서 S로부터 출발하여 G까지 모두 6번의 이동으로 도착하는 것이 최소이다. 너비우선탐색은 출발정점에서 가까운 정점들로부터 탐색해나가기 때문에 도착정점까지의 최단거리를 더 쉽게 찾을 수 있다. 일단 주어진 예제를 이용하여 너비우선탐색을 진행해 나가는 과정을 살펴보면 다음과 같다.

```
#S###
#...#
#.#.#
#....
###G#
```

S의 위치 (0, 1)을 먼저 큐에 넣고 탐색을 시작한다.

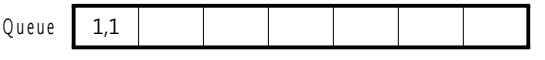
미로는 현재 원래의 입력과 다름없다.




```
#S###
#1..#
#.#.#
#....
###G#
```

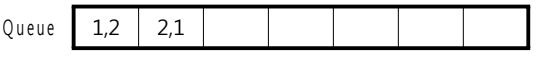
큐에서 자료를 하나 뺀다. 뺀 좌표가 (0, 1)이므로 이 좌표와 상하좌우에 위치한 칸들 중 이동가능한 모든 칸은 맵 상에 1을 기록하고 큐에 넣는다.

(1,1)만 이동 가능하므로 (1,1)만 큐에 삽입된다.



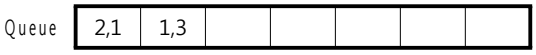

```
#S###
#12.#
#2#.#
#....
###G#
```

큐에서 자료를 하나 뺀다. 삭제된 좌표가 (1, 1)이다.
이 좌표와 상하좌우로 인접한 좌표 중 아직 방문하지 않았으면서 이동가능한 모든 위치의 맵의 (1, 1) 위치의 값 + 1을 기록하고, 모두 큐에 삽입한다.
이때는 (1, 2)와 (2, 1)이 삽입된다.



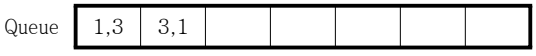
```
#S###
#123#
#2#.#
#....
###G#
```

이번에 큐에서 빠진 좌표는 (1, 2)이다. 따라서 여기서 이동가능한 곳은 (1, 3) 뿐이므로 이곳에 3이 기록되고 큐에 입력된다.



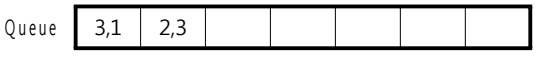

```
#S###
#123#
#2#.#
#3...
###G#
```

다음으로 큐에서 (2, 1)을 삭제하고, 이 좌표에서 이동 가능한 좌표인 (3, 1)에 3을 기록하고 다시 큐에 삽입한다.



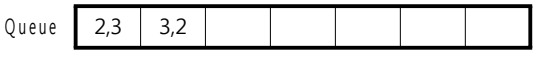

```
#S###
#123#
#2#4#
#3...
###G#
```

다음으로 (1, 3)이 큐에서 제거되고, 제거된 좌표에서 이동 가능한 (2, 3)에 4를 기록하고 큐에 삽입



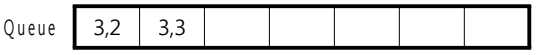

```
#S###
#123#
#2#4#
#34..
###G#
```

다음으로 (3, 1)이 큐에서 제거되고, 제거된 좌표로부터 이동 가능한 (3, 2)에 4를 기록하고, 큐에 삽입



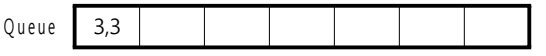
```
#S###
#123#
#2#4#
#345.
###G#
```

다음으로 (2, 3)이 큐에서 제거되고, 제거된 좌표로부터 이동 가능한 (3, 3)에 5를 기록하고, 큐에 삽입



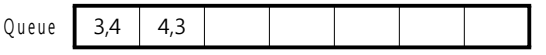

```
#S###
#123#
#2#4#
#345.
###G#
```

다음으로 (3, 2)가 큐에서 제거되고, 제거된 좌표로부터 아직 방문하지 않았거나 이동 가능한 정점이 없으므로 그냥 패스!




```
#S###
#123#
#2#4#
#3456
###6#
```

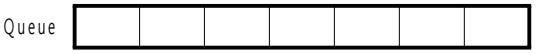
다음으로 (3, 3)이 큐에서 제거되고, 제거된 좌표로부터 이동 가능한 (3, 4)와 (4, 3)을 모두 큐에 삽입함.




```
#S###
#123#
#2#4#
#3456
###6#
```

큐에서 (3, 4)를 제거하고 이 좌표로부터 더 이상 이동 가능한 좌표가 없으므로, 다시 큐에서 (4, 3)을 제거한다.

(4, 3)은 목표지점의 좌표이므로, 더 이상 탐색을 진행할 필요가 없다. 알고리즘은 종료되고, 출발지로부터 목적지까지의 최단길이는 6임을 알 수 있다.



이 문제를 풀 때, 입력 자료를 문자열의 형태로 받아야하므로 주의해야 하며 큐를 이용하여 너비우선탐색을 구현하는 방법으로 해결해보자.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#include <queue>`	
3		
4	`struct VERTEX{ int a, b; };`	
5	`int h, w, Sa, Sb, Ga, Gb, visited[101][101];`	
6	`int dx[4]={1,0,-1,0}, dy[4]={0,1,0,-1};`	
7	`char M[101][101];`	
8		
9	`bool safe(int a, int b)`	
10	`{`	
11	`  return (0<=a && a<h) && (0<=b && b<w);`	
12	`}`	
13		
14	`int main()`	
15	`{`	
16	`  input();`	
17	`  printf("%d\n", solve());`	
18	`}`	

각 변수 h, w는 전체 미로의 높이와 폭을 가지는 변수이고, Sa, Sb는 출발점의 좌표, Ga, Gb는 도착점의 좌표이며, visited는 각 정점까지의 거리를 기록하면서 현재 정점을 방문한지 안한지를 체크하는 용도로 사용되며, dx, dy는 이동 가능한 4방향을 설정한다.

구조체 VERTEX는 미로의 한 칸을 나타내는 구조체로 좌표값을 가진다. M은 미로의 각 칸이 어떤 값으로 구성되었는지 저장하는 배열로 활용된다.

safe는 이동하려고 하는 정점이 실제 미로의 내부인지 아닌지 판단하는 역할을 하는 함수이다.

줄	코드	참고
1	void input(void)	
2	{	
3	scanf("%d %d", &h, &w);	
4	for(int i=0; i<h; i++)	
5	{	
6	scanf("%s", M[i]);	
7	for(int j=0; j<w; j++)	
8	if(M[i][j]=='S') Sa=i, Sb=j;	
9	else if(M[i][j]=='G') Ga=i, Gb=j, M[i][j]='.';	
10	}	
11	}	
12		
13	int solve(void)	
14	{	
15	std::queue<VERTEX> Q;	
16	Q.push((VERTEX){Sa, Sb}), visited[Sa][Sb] = 0;	
17	while(!Q.empty())	
18	{	
19	VERTEX cur=Q.front(); Q.pop();	
20	if(cur.a==Ga && cur.b==Gb) break;	
21		
22	for(int i=0; i<4; i++)	
23	{	
24	int a=cur.a+dx[i], b=cur.b+dy[i];	
25	if(safe(a, b) && !visited[a][b] && M[a][b]=='.')	
26	{	
27	visited[a][b]=visited[cur.a][cur.b]+1;	
28	Q.push((VERTEX){a, b});	
29	}	
30	}	
31	}	
32	return visited[Ga][Gb];	
	}	

위 소스코드는 핵심적인 부분이다. 먼저 입력부에서 중요한 점은 도착점의 값을 'G'에서 '.'로 바꾼다. 마지막 도착점 또한 이동 가능한 상태로 두어야 더 쉬운 코딩이 가능하기 때문이며, 도착 여부의 판단은 좌표를 이용하면 된다.

28행에서 구조체에 자료를 입력하는 부분의 코드가 익숙하지 않을 수 있다. 일반적으로는 28행의 내용을 처리하는 코드는 다음과 같다.

줄	코드	참고
1	if(safe(a, b) && !visited[a][b] && M[a][b]=='.')	
2	{	
3	VERTEX temp;	
4	temp.a=a;	
5	temp.b=b;	
6	visited[a][b]=visited[cur.a][cur.b]+1;	
7	Q.push(temp);	
8	}	

하지만 구조체에 값을 원소나열법과 같이 순서대로 나열하고 "{ }" 로 묶어서 대입하면 구조체로 처리할 수 있다. 그리고 형 변환을 해주면 보다 확실하게 처리할 수 있다. 따라서 다음과 같은 코드로 변경 가능하다.

줄	코드	참고
1	if(safe(a, b) && !visited[a][b] && M[a][b]=='.')	
2	{	
3	VERTEX temp=(VERTEX){ a, b };	
4	visited[a][b]=visited[cur.a][cur.b]+1;	
5	Q.push(temp);	
6	}	

마지막을 VERTEX의 선언 없이 직접 대입으로 28행과 같이 처리할 수 있다.

줄	코드	참고
1	if(safe(a, b) && !visited[a][b] && M[a][b]=='.')	
2	{	
3	visited[a][b]=visited[cur.a][cur.b]+1;	
4	Q.push((VERTEX){ a, b });	
5	}	

다음으로 13행부터 31행까지는 너비우선탐색을 구현한 부분이다. 먼저 시작점을 큐에 삽입하고, 큐가 빌 때까지 아직까지 방문하지 않은 정점들을 차례로 큐에 삽입한다. 큐의 특성 상, 출발점에서 가까운 정점들이 먼저 큐에 삽입된다.

여기서 중요한 점은 visited라는 배열에는 출발점과의 거리가 기록되도록 코딩한다는 점

이다. 방문했으면 1, 아니면 0으로 기록할 수도 있지만, 이 문제의 경우 방문하지 않았으면 0, 방문했으면 이동거리를 저장하는 아이디어를 이용하여 문제를 해결하고 있다.

이와 같이 다양한 아이디어를 이용하여 문제를 해결할 수 있기 때문에 평소에 다양한 관점에서 문제를 접근하는 연습을 한다면 창의적인 문제해결력이 향상된다.

5 전체탐색법

전체탐색법은 모든 문제해결의 기초가 되는 가장 중요한 설계법 중 하나라고 할 수 있다. 주어진 문제에서 해가 될 수 있는 모든 가능성을 검사하여 해를 구하기 때문에 항상 정확한 해를 구할 수 있다는 점이 장점이다. 하지만 탐색해야할 내용이 너무 많으면 문제에서 제시한 시간 이내에 해결할 수 없다는 점을 유의해야 한다.

하지만 전체탐색을 기반으로 한 다양한 응용들이 있으며, 이러한 응용들을 통하여 탐색해야할 공간을 배제해 나가면서 시간을 줄일 수 있는 다양한 방법들이 존재하기 때문에 잘 응용하면 많은 문제를 해결할 수 있는 강력한 도구가 될 수 있다. 따라서 전체탐색법을 잘 익혀두면 다른 알고리즘 설계법을 학습하는데 많은 도움이 된다.

전체탐색법은 앞 단원들에서 공부한 선형구조의 탐색, 비선형구조의 탐색을 기반으로 하여 문제를 해결한다.

가. 선형구조와 비선형구조의 전체탐색

선형구조의 전체탐색은 앞에서 배운 대로 주로 반복문을 이용하여 접근할 수 있다. 1차원 뿐만 아니라 2차원 이상의 다차원 구조에 대해서도 선형구조로 탐색할 수 있다.

비선형구조의 전체탐색은 문제해결의 가장 기본이 되는 알고리즘 설계법인 백트래킹이다. 백트래킹 기법은 재귀함수를 이용하여 간단하게 구현할 수 있고, 다양한 문제를 해결하는데 많이 응용되는 방법이므로 반드시 익혀둘 필요가 있다.

주어진 문제들을 통하여 선형구조, 비선형구조의 전체탐색법을 익힐 수 있도록 하자.

문제 1

약수의 합 구하기 1

한 정수 n을 입력받아서 n의 모든 약수의 합을 구하는 프로그램을 작성하시오.

예를 들어 10의 약수는 1, 2, 5, 10이므로 이 값들의 합인 18이 10의 약수의 합이 된다.

입력

첫 번째 줄에 정수 n이 입력된다.
(단, 1 <= n <= 100,000)

출력

n의 약수의 합을 출력한다.

입력 예	출력 예
10	18

풀이

이 문제는 기본적으로 수학적인 아이디어를 이용하여 해결할 수 있는 문제이지만 이 단원에서는 전체탐색법을 다루는 단원이므로 전체탐색법으로 해결해보자.

일단 n을 입력받으면 1부터 n까지의 모든 수를 차례로 반복문을 이용하여 선형으로 탐색하면서 n의 약수들을 검사한다. 만약 현재 탐색 중인 수가 n의 약수라면 누적하여 구할 수 있다. 이렇게 구한다면 계산량은 $O(n)$이 된다. 이 문제에서는 n의 최댓값이 100,000이므로 충분히 해결할 수 있는 문제가 된다.

어떤 수 x가 n의 약수라면 다음 조건을 이용해 구할 수 있다.

```
n % x == 0
```

이를 이용하여 문제를 해결한 소스코드는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2		
3	`int n;`	
4		
5	`int solve()`	
6	`{`	
7	`  int ans = 0;`	
8	`  for(int i=1; i<=n; i++ )`	
9	`    if(n%i==0)`	
10	`      ans+=i;`	
11	`  return ans;`	
12	`}`	
13		
14	`int main()`	
15	`{`	
16	`  scanf("%d", &n);`	
17	`  printf("%d\n", solve());`	
18	`}`	

이 문제는 이와 같은 방법으로 쉽게 해결할 수 있으나, n이 10억 이상의 값으로 커질 때는 다른 방법을 생각해야 한다. 나중에 다루게 될 것이므로 한 번 생각해보자.