풀이

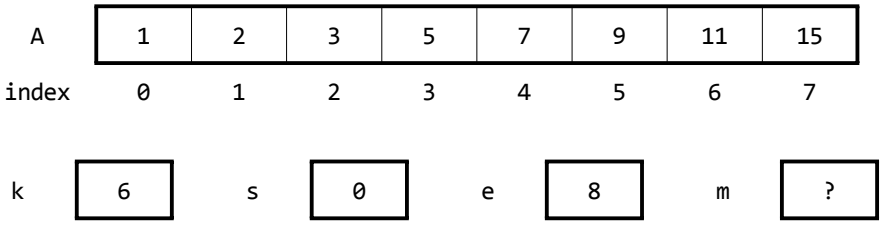
이 문제에서 다루는 lower bound는 대회에 자주 등장하는 방법이므로 꼭 익혀둘 수 있도록 한다. lower bound는 이분탐색을 이용하여 구할 수 있다. 이분탐색은 찾고자 하는 값이 없으면 탐색 실패가 되지만, lower bound는 찾고자하는 정확한 값이 없더라도 찾고자 하는 값보다 큰 가장 작은 정수 값을 찾으므로 차이가 있다.

lower bound인 경우에는 같은 원소가 여러 개 있더라도 항상 유일한 해를 구할 수 있기 때문에 알고리즘을 설계하는 것이 이분탐색 보다는 까다로우나 근본은 같으므로 잘 익혀 둘 수 있도록 한다.

먼저 구간을 [s, e]로 설정하고, 중간위치의 값을 m이라 하면, A[m-1] 〈k 이면서 A[m]〉= k인 최소 m을 찾는 문제가 된다. 이 때 m은 2이상인 값이다. 따라서 일반적인 이분탐색에서 A[m] == k인 부분을 다른 부분에 포함해야 한다는 점을 잘 확인해야 한다.

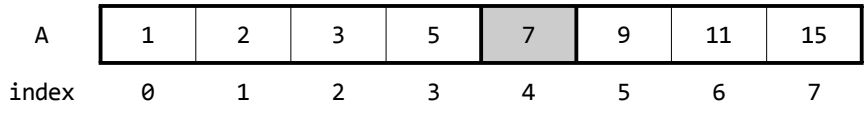
다음으로 모든 원소가 k보다 작을 때는 n+1을 출력해야 하므로 처음 구간을 잡을 때, [1, n]을 잡는 것이 아니라 [1, n+1]로 설정하여 시작한다는 점도 유의해야 한다.

준비 단계



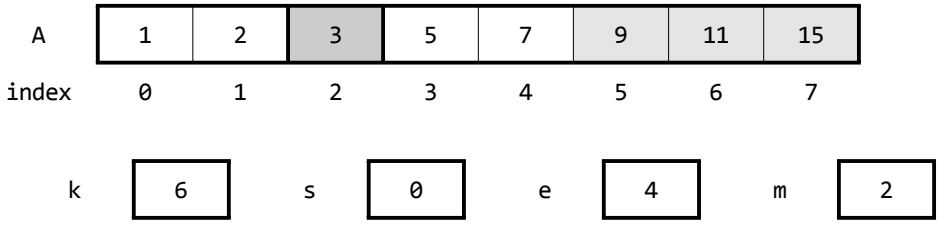
〈입력받은 상태에서 탐색 준비를 한다. 탐색 범위는 0~7〉

[1 단계]



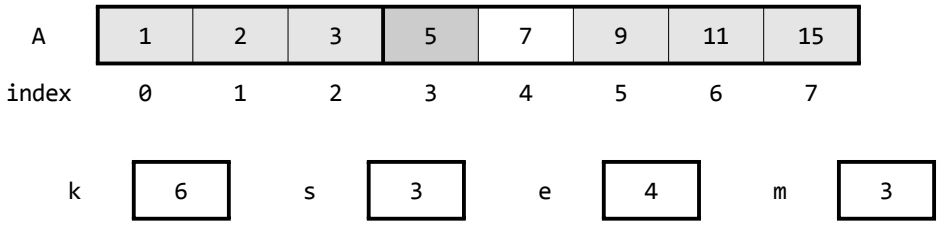
〈A[4]가 6보다 크므로 범위를 0~4로 한다. 만약 일반 이분탐색이었으면 0~3으로 범위를 좁혀야 하나 lower bound는 k 이상이 최솟값의 위치이므로 e까지 포함한다.〉

[2 단계]



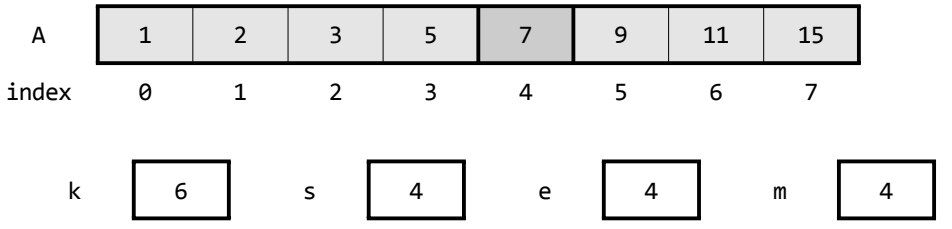
〈A[2]가 6보다 작으므로 범위를 3~4로 하고 재탐색을 시작한다.〉

[3 단계]



〈A[3]이 6보다 작으므로 범위를 4 ~ 4로 하고 재탐색을 시작한다.〉

[4 단계]



〈이제 더 이상 탐색할 원소가 없으므로 인덱스 4에 있는 원소가 k 이상인 최소 원소의 위치가 된다.〉

이와 같이 lower bound는 이분탐색과 유사하나 좀 더 엄밀하게 접근해야 한다. 그리고 매우 다양한 응용범위가 있으므로 잘 익힐 수 있도록 한다.

위의 과정을 구현한 소스코드는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`int n, k, A[1000001];`	
3	`int solve(int s, int e)`	
4	`{`	
5	`  int m;`	
6	`  while(e-s>0)`	
7	`  {`	
8	`    m=(s+e)/2;`	
9	`    if(A[m]<k) s=m+1;`	
10	`    else e=m;`	
11	`  }`	
12	`  return e+1;`	
13	`}`	
14	`int main()`	
15	`{`	
16	`  scanf("%d",&n);`	
17	`  for(int i=0; i<n; i++)`	
18	`    scanf("%d", A+i);`	
19	`  scanf("%d",&k);`	
20	`  printf("%d\n", solve(0, n));`	
21	`  return 0;`	
22	`}`	

6행의 e-s 〉 0 은 e 〉 s와 같은 의미이다. while문을 탈출했을 때, 즉 11행에서는 s 〉= e가 된다는 사실을 잘 이해하고 있어야 한다. 따라서 12행에서 반환하는 값이 e+1이 된다는 것이 중요한 점이다.

만약 6행을 e-s 〉 1로 설정한다면 어떻게 될지도 생각해보면 실력향상에 많은 도움이 될 것이다. 이러한 부분들은 수학적으로 엄밀하게 접근하는 연습을 하는 데 많은 도움이 되니 꼭 실습해보기 바란다.

이번에는 이 문제를 해결하는 데 STL을 직접 활용하는 방법을 소개한다. 사실 실제 대회에서는 이렇게 lower bound를 작성하는 경우는 흔치 않으며, 대부분 std::lower_bound( )

함수를 활용하게 될 것이므로 이 std::lower_bound( )를 꼭 익힐 수 있도록 한다.

S라는 배열의 처음부터 n-1번째까지의 원소들 중 k의 low bound에 해당하는 원소의 위치를 반환하는 std::lower_bound( )의 기본적인 사용법은 아래와 같다.

```
std::lower_bound( S, S+n, k, [compare] );
```

여기서 compare 함수는 앞에 std::sort()에서 사용했던 compare와 같은 역할을 하는 함수로서 작성법도 동일하므로, 앞에 예를 참고하면 된다. 그리고 compare를 생략할 경우에는 오름차순이라고 가정하고 동작하게 된다.

다음 소스코드는 std::lower_bound()를 활용하여 문제를 해결한 것을 보여준다. 이 예는 자주 활용할 가능성이 크므로 반드시 익혀두기 바란다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#include <algorithm>`	
3		
4	`int n, k, A[1000001];`	
5		
6	`int main()`	
7	`{`	
8	`  scanf("%d",&n);`	
9	`  for(int i=0; i<n; i++)`	
10	`    scanf("%d", A+i);`	
11	`  scanf("%d",&k);`	
12	`  printf("%d\n", std::lower_bound(A,A+n,k)-A+1);`	
13	`}`	

이 소스코드에서는 12행의 내용을 이해하는 것이 중요하다.

std::lower_bound(A, A+n, k)의 의미는 배열 A[0]~A[n-1]이 오름차순으로 정렬되어 있을 때, k의 lower bound위치의 주소를 구한다.

따라서 그 주소에서 A를 빼면 k가 존재하는 배열 A의 인덱스가 되며, 배열의 인덱스는 0부터 시작하므로 1을 더해주면 우리가 원하는 해를 구할 수 있게 된다.

따라서 std::lower_bound(A, A+n, k)-A+1과 같이 활용할 수 있다.

문제 5

upper bound

n개로 이루어진 정수 집합에서 원하는 수 k보다 큰 수가 처음으로 등장하는 위치를 찾으시오.

단, 입력되는 집합은 오름차순으로 정렬되어 있으며, 같은 수가 여러 개 존재할 수 있다.

입력

첫째 줄에 한 정수 n, 둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다. 셋째 줄에는 찾고자 하는 값 k가 입력된다.
(단, 2 <= n <= 1,000,000, 각 원소의 크기는 100,000,000을 넘지 않는다.)

출력

찾고자 하는 원소의 위치를 출력한다. 만약 모든 원소가 k보다 작으면 n+1을 출력한다.

입력 예	출력 예
5 1 3 5 5 7 5	5
8 1 2 7 7 7 7 11 15 7	6
5 1 2 3 4 5 7	6
5 2 2 2 2 2 1	1

풀이

이 문제에서 다루는 upper bound 또한 대회에 자주 등장하는 방법이므로 꼭 익혀둘 수 있도록 한다. upper bound도 lower bound와 마찬가지로 이분탐색을 이용하여 구할 수 있다. upper bound는 k를 초과하는 가장 첫 번째 원소의 위치를 구하는 것이다.

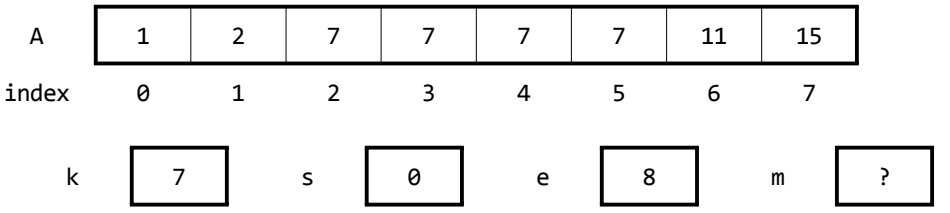
upper bound 도 lower bound와 같이 같은 원소가 여러 개 있더라도 항상 유일한 해를 구할 수 있기 때문에 알고리즘을 설계하는 것이 이분탐색 보다는 까다롭다.

하지만 upper bound와 lower bound를 함께 이용하면 다양한 문제를 접근할 수 있다. 예를 들어 정렬된 배열에 존재하는 k는 모두 몇 개인가? 와 같은 문제도 위 두 함수를 이용하면 쉽게 해결할 수 있으므로 이러한 문제에 대해서도 따로 연습해 둘 필요가 있다.

upper bound를 구하기 위해서는 먼저 구간을 [s, e]로 설정하고, 중간위치의 값을 m이라 하면, A[m-1] 〈= k 이면서 A[m] 〉 k인 최소 m을 찾는 문제가 된다. 이 때 m은 2이상인 값이다. 따라서 일반적인 이분탐색에서 A[m] == k인 부분을 다른 부분에 포함해야 한다는 점을 잘 확인해야 한다.

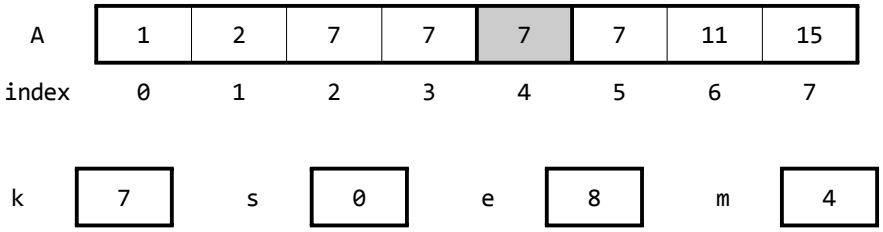
다음으로 모든 원소가 k보다 작을 때는 n+1을 출력해야 하므로 처음 구간을 잡을 때, [1, n]을 잡는 것이 아니라 [1, n+1]로 설정하여 시작한다는 점도 유의해야 한다. 다음은 upper bound의 과정을 나타낸다.

준비 단계



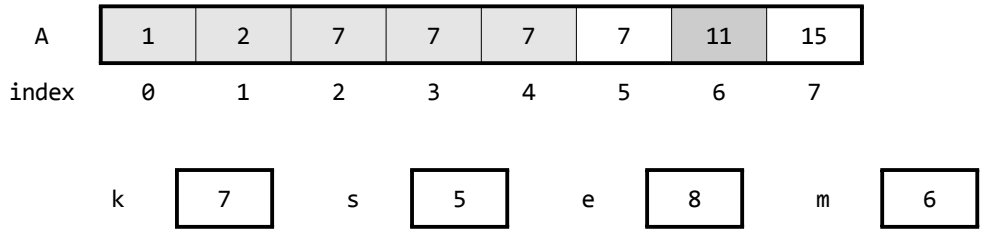
〈입력받은 상태에서 탐색 준비를 한다. 탐색 범위는 0~7〉

[1 단계]



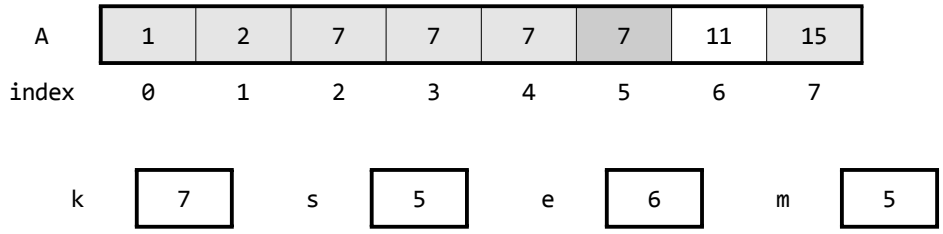
〈A[4]와 7이 같으므로 범위를 5~8로 설정한다. 만약 이분탐색이었으면 바로 탐색을 종료해야 하나 upper bound는 k를 초과하는 최솟값의 위치이므로 4를 포함할 필요가 없다.〉

[2 단계]



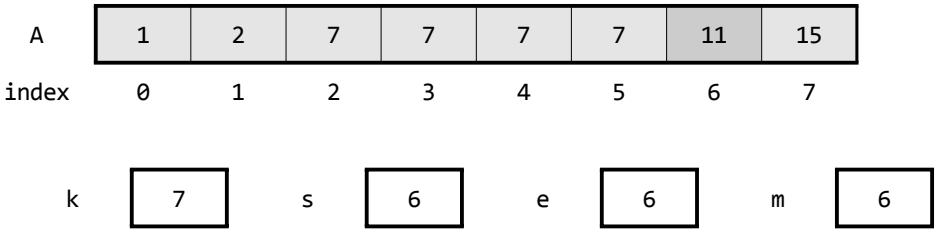
〈A[6]이 7보다 크므로 범위를 5~6까지로 하고 재탐색을 시작한다.〉

[3 단계]



〈A[5]와 7이 같으므로 범위를 6~6까지로 하고 재탐색을 시작한다.〉

[4 단계]



〈범위 상 더 이상 탐색을 할 필요가 없으므로 6번 인덱스가 조건을 만족하는 가장 작은 인덱스라는 사실을 확인할 수 있음.〉

이와 같이 upper bound는 lower bound와 유사하다. 자세한 내용은 다음 소스코드를 참고한다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`int n, k, A[1000001];`	
3	`int solve(int s, int e)`	
4	`{`	
5	`  int m;`	
6	`  while(e-s>0)`	
7	`  {`	
8	`    m = (s+e)/2;`	
9	`    if(A[m]<=k) s=m+1;`	
10	`    else e=m;`	
11	`  }`	
12	`  return e+1;`	
13	`}`	
14	`int main()`	
15	`{`	
16	`  scanf("%d",&n);`	
17	`  for(int i=0; i<n; i++)`	
18	`    scanf("%d", A+i);`	
19	`  scanf("%d",&k);`	
20	`  printf("%d\n", solve(0, n));`	
21	`}`	

lower bound를 구할 때와 차이점은 9행에서 A[m] 〈 k를 A[m] 〈= k로 바꾼 것뿐이다. 이 부분을 그냥 단순히 외우려 하지 말고, 왜 부등식이 위와 같이 바뀌면 upper

bound를 구할 수 있는지 수학적으로 엄밀히 분석해 두면 나중에 다른 문제들을 해결할 때 많은 도움이 될 것이다.

이 문제 또한 6행을 e-s 〉 1로 설정한다면 어떻게 될지도 생각해보면 실력향상에 많은 도움이 될 것이다.

upper bound도 lower bound와 마찬가지로 STL을 활용할 수 있는 방법이 있다.

S라는 배열의 처음부터 n-1번째까지의 원소들 중 k의 upper bound에 해당하는 원소의 주소를 반환하는 std::upper_bound()의 기본적인 사용법은 아래와 같다.

```
std::upper_bound( S, S+n, k, [compare] );
```

여기서 compare함수는 앞에 std::sort()에서 사용했던 compare와 같은 역할을 하는 함수고 작성법도 동일하므로, 앞의 예를 참고하면 된다. 그리고 compare를 생략할 경우에는 오름차순이라고 가정하고 동작하게 된다.

다음 소스코드는 std::upper_bound()를 활용하여 문제를 해결한 것을 보여준다. 이 예는 자주 활용할 가능성이 크므로 반드시 익혀두기 바란다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#include <algorithm>`	
3		
4	`int n, k, A[1000001];`	
5		
6	`int main()`	
7	`{`	
8	`  scanf("%d",&n);`	
9	`  for(int i=0; i<n; i++)`	
10	`    scanf("%d", A+i);`	
11	`  scanf("%d",&k);`	
12	`  printf("%d\n", std::upper_bound(A, A+n, k)-A+1);`	
13	`}`	

이 소스코드에서는 12행의 내용을 이해하는 것이 중요하다.

std::upper_bound(A, A+n, k)의 의미는 배열 A[0]~A[n-1]이 오름차순으로 정렬되어 있을 때, k의 upper bound위치의 주소를 구한다.

주소에 대한 설명은 lower bound에서 설명한 것과 같다. 따라서 std::upper_bound(A, A+n, k)-A+1 와 같이 활용할 수 있다.

나. 비선형구조의 탐색

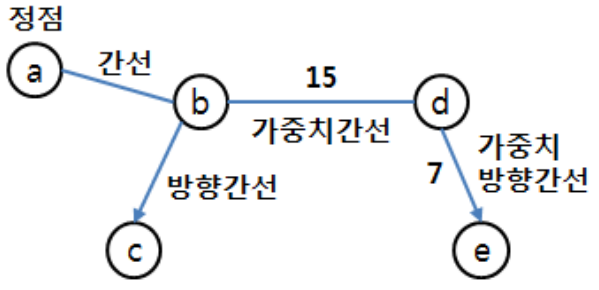
비선형구조란 i번째 원소를 탐색한 다음 그 원소와 연결된 다른 원소를 탐색하려고 할 때, 여러 개의 원소가 존재하는 탐색구조를 말한다. 일반적으로 자료가 트리나 그래프로 구성되어 있을 경우를 비선형구조라 하고 이러한 트리나 그래프의 모든 정점을 탐색하는 것을 비선형 탐색이라고 이해하면 된다.

비선형구조는 선형과 달리 자료가 순차적으로 구성되어 있지 않으므로 단순히 반복문을 이용하여 탐색하기에는 어려움이 있다. 그러므로 비선형구조는 스택이나 큐와 같은 자료구조를 활용하여 탐색하는 것이 일반적이다.

비선형구조의 탐색은 크게 깊이우선탐색(depth first search, dfs)과 너비우선탐색(breadth firth search, bfs)으로 나눌 수 있으며, 이 두 가지 탐색법을 활용한 다양한 응용이 있으나 이 교재에서는 기본적인 두 가지 탐색법에 대해서 익히도록 한다.

- 비선형구조

비선형구조의 탐색을 다루기 전에 그래프와 트리에 대해서 간단히 알아보자. 트리와 그래프를 이루는 기본 요소를 정점(vertex)과 간선(edge)이라고 한다.

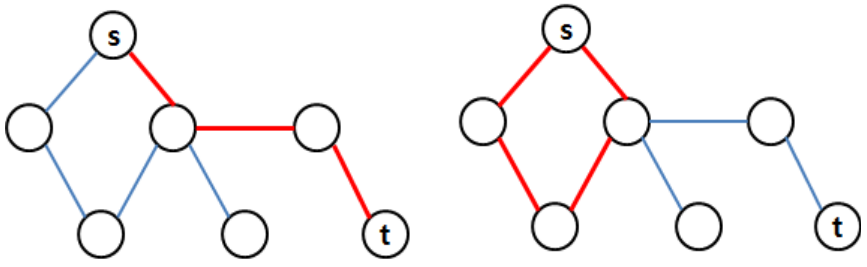


원은 정점, 선분은 간선을 나타내며, $\overline{ab}$는 보통간선, $\overrightarrow{bc}$는 방향간선, $\overline{bd}$는 가중치가 15인 양방향통행 간선, $\overrightarrow{de}$는 가중치가 7인 일방통행 간선(방향간선)을 나타낸다.

정점은 점 또는 원으로 표현하며, 일반적으로 상태나 위치를 표현한다. 간선은 정점들을 연결하는 선으로 표현하며, 정점들 간의 관계를 표현한다.

- 경로(path)와 회로(cycle)

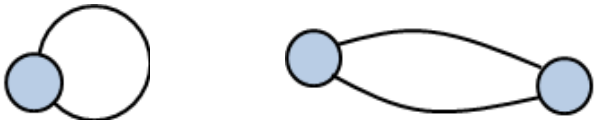
그래프에서 임의의 정점 s에서 임의의 정점 t로 이동할 때, s에서 t로 이동하는데 사용한 정점들을 연결하고 있는 간선들의 순서로 된 집합을 경로라고 한다. 회로는 그래프에서 임의의 정점 s에서 같은 정점 s로의 경로들을 말한다.



왼쪽의 빨간 간선들은 s에서 t로의 경로를 나타내고, 오른쪽의 빨간 간선들은 s에서 t로의 회로를 나타낸다.

- 자기간선(loop)과 다중간선(multi edge)

임의의 정점에서 자기 자신으로 연결하고 있는 간선을 자기간선, 임의의 정점에서 다른 정점으로 연결된 간선의 수가 2개 이상일 경우를 다중간선이라고 한다.

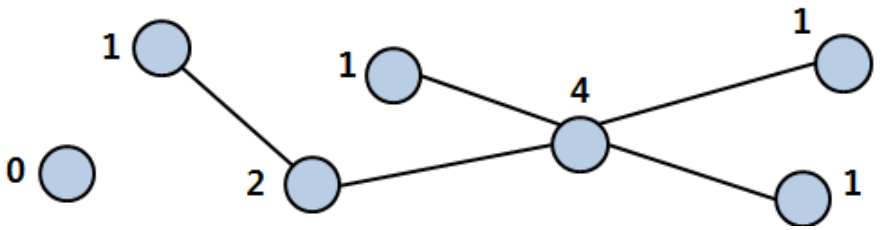


왼쪽은 자기간선 오른쪽은 다중간선을 나타낸다.

- 그래프의 차수(degree)

그래프의 임의의 한 정점에서 다른 정점으로 연결된 간선의 수를 차수라고 한다. 다음

그림은 각 정점에 대한 차수를 나타낸다.



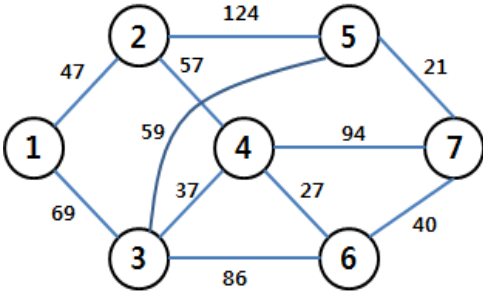
각 정점에서의 차수

- 그래프의 구현

그래프를 구현하는 방법은 인접행렬(adjacency matrix)과 인접리스트(adjacency list)로 크게 나눌 수 있다.

일반적으로 정보올림피아드를 비롯한 프로그래밍 대회에서 그래프는 정점의 수, 간선의 수, 각 간선들이 연결하고 있는 정점 2개로 이루어진 정보가 주어지는 경우가 대부분이다.

다음은 실제로 그래프가 주어질 때, 이를 저장하는 2가지 방법을 보여준다.



7개의 정점과 11개의 간선을 가지는 가중치 그래프의 예

이러한 그래프의 경우 일반적인 입력데이터의 형식은 다음과 같다.

[표-2] 그래프의 대표적인 입력형식과 입력데이터의 예

입력 형식	입력데이터의 예
첫 번째 줄에 정점의 수 n과 간선의 수 m이 공백으로 구분되어 입력된다. 두 번째 줄부터 m개의 줄에 걸쳐서 간선으로 연결된 두 정점의 번호와 가중치가 공백으로 구분되어 입력된다.	7 11 1 2 47 1 3 69 2 4 57 2 5 124 3 4 37 3 5 59 3 6 86 4 6 27 4 7 94 5 7 21 6 7 40

- 인접행렬의 구현

[표-2]의 입력예시를 인접행렬로 받기 위해서는 2차원 배열을 이용한다. 먼저 최대 정점의 수에 맞추어 2차원 배열을 선언하고 각 배열의 칸에 연결된 정보를 저장한다. 앞 그래프를 2차원 행렬을 이용하여 다음과 같이 저장한다.

	1	2	3	4	5	6	7
1		1	1				
2	1			1	1		
3	1			1	1	1	
4		1	1			1	1
5		1	1				1
6			1	1			1
7				1	1	1	

	1	2	3	4	5	6	7
1		47	69				
2	47			57	124		
3	69			37	59	86	
4		57	37			27	94
5		124	59				21
6			86	27			40
7				94	21	40	

왼쪽은 가중치가 없는 표현, 오른쪽은 가중치가 있는 표현이다. 예를 들어, 3행 4열의 경우 왼쪽은 1, 오른쪽은 37이 기록되어 있다. 왼쪽의 1은 간선이 있음을 의미하고, 오른쪽은 간선이 있을 때 가중치를 저장한다.

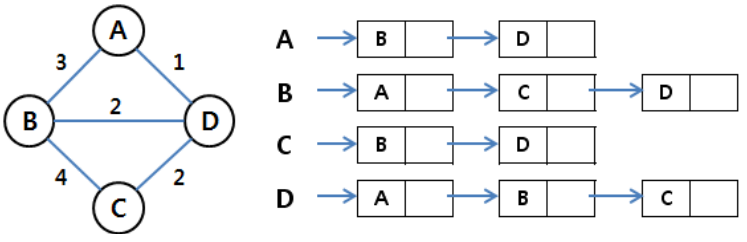
2차원 행렬을 이용하여 저장하는 소스코드는 다음과 같다. 단 최대 정점의 수는 100개로 가정한다.

줄	코드	참고
1	`#include <stdio.h>`	10: 정점 a, b를 연결하는 간선, w는 가중치 12: 만약 가중치가 없다면 1,방향 간선이면G[a][b]만 저장.
2		
3	`int n, m, G[101][101];`	
4		
5	`int main()`	
6	`{`	
7	`  scanf("%d %d",&n,&m);`	
8	`  for(int i=0; i<m; i++)`	
9	`  {`	
10	`    int a, b, w;`	
11	`    scanf("%d %d %d", &a, &b, &w);`	
12	`    G[a][b]=G[b][a]=w;`	
13	`  }`	
14	`}`	

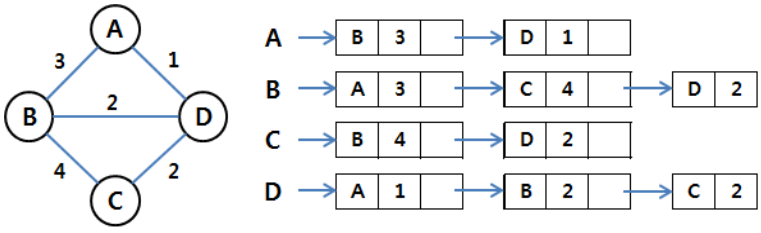
- 인접리스트의 구현

인접행렬로 표현할 때에는 연결되지 않았던 부분까지 모두 표현이 된다. 즉, 각 칸에 0이라고 기록된 부분은 연결이 되지 않은 부분을 의미한다. 사실 일반적인 그래프에서 행렬상에서 0이라고 표현되는 부분이 많을 가능성이 크다.

알고리즘을 구현할 때에도 이 0이라고 표시된 부분까지 모두 조사를 해야 하므로 효율이 떨어지는 경우가 많다. 이러한 단점을 극복하기 위하여 제안된 방법이 인접리스트이고 이 방법은 인접행렬에서 0으로 표시된 부분은 저장하지 않으므로 효율을 높이고 있다.



[그림-10] 그래프의 인접리스트 표현



[그림-11] 가중치 그래프의 인접리스트 표현

[표-2]의 입력 예시를 인접리스트로 구현하기 위해서는 [그림-10], [그림-11]과 같이 연결리스트로 구현할 수 있지만 STL에서 제공하는 std::vector()를 이용하여 간단하게 구현할 수 있다. [표-2]의 입력예시를 인접리스트로 구현하면 다음과 같은 그림이 된다.

1	2	47	3	69				
2	1	47	4	57	5	124		
3	1	69	4	37	5	59	6	86
4	2	57	3	37	6	27	7	94
5	2	124	3	59	7	21		
6	3	86	4	27	7	40		
7	4	94	5	21	6	40		

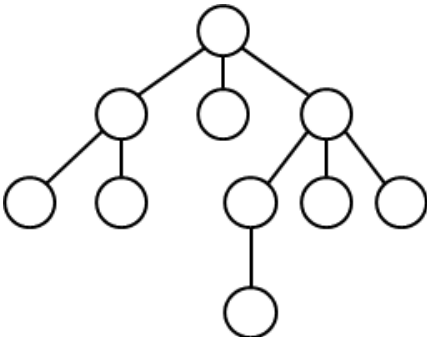
인접리스트에서는 정점과 가중치의 쌍으로 간선이 있는 것만 연결한다.
예를 들어 1행의 경우 1-2로의 간선의 가중치는 47이고 1-3으로의 간선의
가중치는 69라는 의미이다.

std::vector()를 이용한다면 위와 같이 인접행렬로 구현하는 것보다 공간을 적게 사용한다. 따라서 전체탐색법을 구현할 때, 당연히 탐색시간도 줄일 수 있다. 계산량으로 표현하자면, 인접행렬로 모든 정점을 탐색하는데 $O(nm)$의 시간이 드는데 반해, 인접리스트로 표현하면 $O(n+m)$의 시간이 든다.

여러 가지 장점이 있기 때문에 대회에서는 주로 인접리스트를 이용한 방법을 활용하는 경우가 많으므로 반드시 익혀둘 수 있도록 한다.

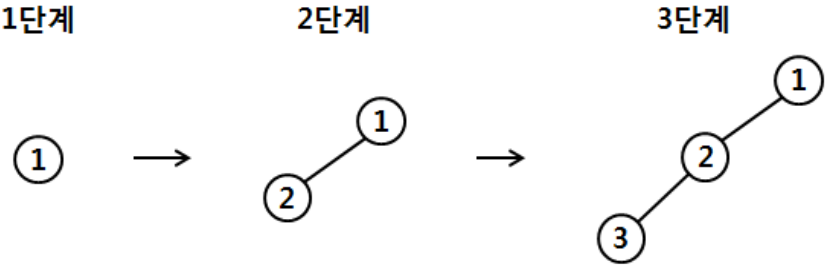
- 깊이우선탐색(dfs)

그래프 중 회로(cycle)가 없는 그래프를 트리라고 한다. [그림-13]은 트리를 나타낸다. 이 트리의 가장 위에 있는 정점에서 출발하여 모든 정점들을 깊이우선으로 탐색하며, 탐색하는 순서를 알아보자.



10개의 정점(vertex)과 9개의 간선(edge)을 가진 트리

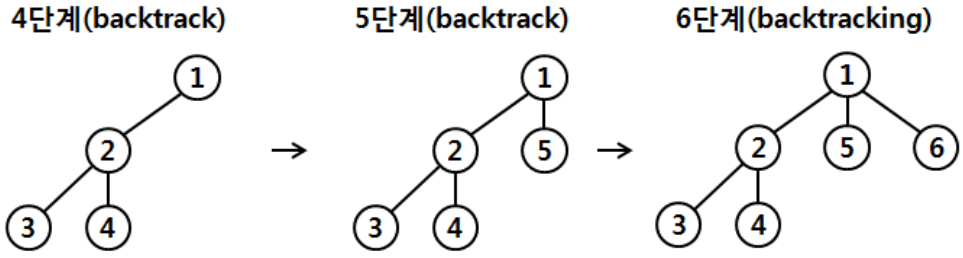
출발 정점을 트리의 가장 위에 있는 정점으로 하고, 한 정점에서 이동 가능한 정점이 여러 개 있을 경우 왼쪽의 정점부터 방문한다고 가정하면, 단계별 탐색 과정은 다음과 같다.



깊이우선 1~3 단계

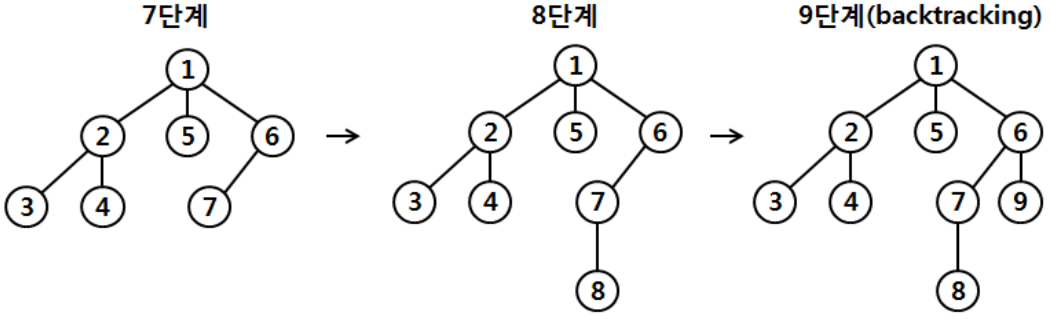
깊이우선탐색과정에서 3단계 이후 더 이상 진행할 수 있는 정점이 없다. 그 이유는 간선으로 연결된 정점들 중 아직 방문하지 않은 정점을 방문하기 때문이다.

이처럼 더 이상 진행할 수 없을 때는 다시 이전 정점으로 되돌아가는 과정이 필요하다. 일반적으로 이 과정을 백트랙(backtrack)이라고 한다. 백트랙은 비선형구조의 탐색에서 매우 중요하다. 백트랙은 스택(stack)이나 재귀함수(recursion)를 이용하면 쉽게 구현할 수 있다.



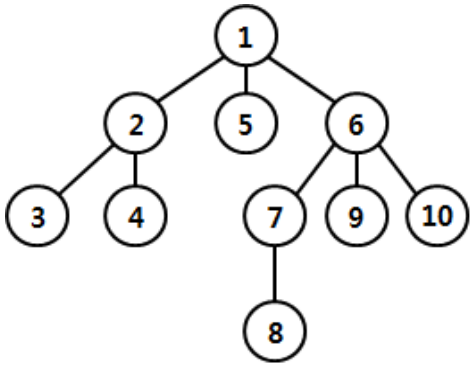
깊이우선 4 ~ 6 단계

4, 5, 6단계는 연속으로 백트랙이 발생한다. 이는 더 이상 진행할 수 없는 정점까지 도달했다는 것을 의미한다. 계속 해서 다음 단계로 진행하는 과정은 다음과 같다.



깊이우선 7~9 단계

위 단계에서 마지막 정점을 방문하면 깊이우선탐색이 완료된다.



탐색종료

깊이우선탐색을 정리하여 설명하면 먼저 시작 정점에서 간선을 하나 선택하여 진행할 수 있을 때까지 진행하고 더 이상 진행할 수 없다면 백트랙하여 다시 다른 정점으로 진행하여 더 이상 진행할 정점이 없을 때까지 이 과정을 반복하는 탐색법으로, 간선으로 연결된 모든 정점을 방문할 수 있는 탐색법이다.

깊이우선탐색의 알고리즘은 다음과 같다. 이 탐색법은 백트래킹(backtracking)이라는 알고리즘 설계 기법의 중심이 되며 백트래킹 기법은 모든 문제를 해결할 수 있는 가장 기본적인 방법이므로 꼭 익혀둘 필요가 있다.

줄	코드	참고
1	`bool visited[101];`	1: 방문했는지 체크해 두는 배열 4: 정점 k와 연결된 모든 정점 방문 5: 만약 아직 방문하지 않았으면 7: 방문했다고 체크하고 8: 깊이우선탐색 진행 10: 더 이상 갈 길이 없으면 backtrack
2	`void dfs(int k)`	
3	`{`	
4	`  for(int i=0; i<G[i].size(); i++)`	
5	`    if(!visited[G[k][i].to])`	
6	`    {`	
7	`      visited[G[k][i].to]=true;`	
8	`      dfs(G[k][i]);`	
9	`    }`	
10	`  return;`	
11	`}`	

이 방법은 그래프를 인접리스트에 저장했을 경우에 활용할 수 있다. 이 방법은 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.

줄	코드	참고
1	`bool visited[101];`	1: 방문했는지 체크해 두는 배열 4: 모든 정점에 대해서 검사 5: k에 연결되어 있으면서, 아직 방문하지 않았으면 7: 방문했다고 체크하고 8: 깊이우선탐색 진행 10: 더 이상 갈 길이 없으면 backtrack
2	`void dfs(int k)`	
3	`{`	
4	`  for(int i=1; i<=n; i++)`	
5	`    if( G[k][i] && !visited[G[k][i]] )`	
6	`    {`	
7	`       visited[G[k][i]] = true;`	
8	`       dfs(G[k][i]);`	
9	`    }`	
10	`  return;`	
11	`}`	

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

문제 1

두더지 굴(S)

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 0 또는 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 30이하의 자연수) 두 번째 줄부터 n줄에 걸쳐서 n개의 0과 1이 공백으로 구분되어 입력된다.

출력

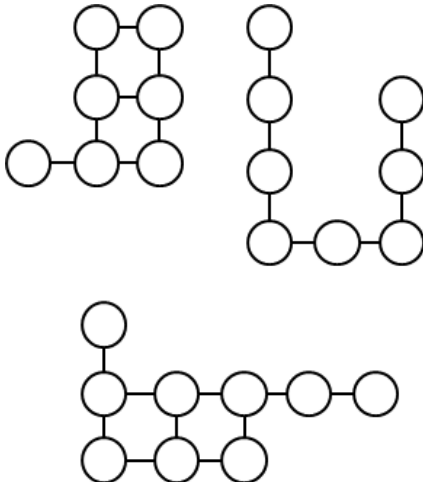
첫째 줄에 두더지 굴의 수를 출력한다. 둘째 줄부터 각 두더지 굴의 크기를 내림차순으로 한 줄에 하나씩 출력한다.

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 그냥 보기에는 비선형탐색, 즉 그래프의 문제로 보이지 않는다. 하지만 지도에서 각 칸을 정점으로 생각하고 각 칸 중 1인 칸을 중심으로 상, 하, 좌, 우 중 1이 있다면 이 부분에 간선이 있는 것으로 생각하면 그래프로 볼 수 있다.

문제에서 주어진 입력 예를 그래프로 나타내면 다음과 같다.

입력 예	대응되는 그래프
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	

이와 같이 그래프로 만든 다음 배열의 (0, 0)부터 순차탐색을 진행하면서 (a, b)의 값이 만약 1이라면 이 점을 시작으로 하여 깊이우선탐색을 이용하여 모든 연결된 점을 방문하고 특정 값으로 체크한다.

나머지 점들도 모두 순차탐색하면서 마지막 까지 깊이우선탐색을 실행하고 알고리즘을 종료한다. 마지막에 깊이우선탐색을 실행한 횟수가 두더지의 수가 되고, 각 두더지 굴의 크기는 다른 배열에 저장해 둔 다음 마지막에 std::sort()를 이용하여 정렬한 후 내림차순으로 출력하면 된다.

여기에 사용되는 알고리즘은 지뢰찾기, 뿌요뿌요 등의 게임에 많이 활용되는 방법으로서, flood fill이라고도 한다. 자주 등장하는 방법이므로 익혀두면 활용가치가 크다.

이 알고리즘에서는 재귀함수를 이용하여 깊이우선탐색을 구현한다. 이때 가장 조심해야 할 점은 재귀의 깊이가 너무 커지면 runtime error가 발생할 수도 있다는 것이다. 일반적으로 release 모드라면 재귀의 깊이는 대략 10만 내외가 된다. 이 문제에서는 관계없지만 깊이가 너무 크다고 판단되면 다음 절에서 배울 너비우선탐색으로 처리하거나, 재귀 대신 스택을 이용해도 된다.

다음은 위 알고리즘의 실행 과정을 나타낸다.

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

Size	0	0	0	0	0	0
index	0	1	2	3	4	5

[두더지 = 2]

두더지의 값은 2부터 시작 1은 주인 없는 굴이므로 2부터 증가함.

(0, 0)에서 탐색을 시작함.
(0, 0)의 원소가 0이므로 통과.

Size배열은 각 두더지 집의 크기를 저장할 배열

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

(0, 1)을 탐색, 원소의 값이 1이므로 dfs를 이용하여 상, 하, 좌, 우로 연결된 그래프를 모두 탐색하여 2로 수정함.

방문한 정점의 수인 7을 Size[2]에 기록하여 크기를 저장하고, 두더지 값 1 증가

Size	0	0	0	0	0	0
index	0	1	2	3	4	5

0	2	**2**	0	1	0	0
0	2	2	0	1	0	1
2	2	2	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

Size	0	0	7	0	0	0
index	0	1	2	3	4	5

[두더지 = 3]

(0, 2)를 탐색, 원소의 값이 원래 1이었으나 (0, 1)에 의해 2로 바뀌었으므로, 이미 다른 두더지의 굴에 포함되었음.

따라서 그냥 통과!

0	2	2	0	**1**	0	0
0	2	2	0	1	0	1
2	2	2	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

Size	0	0	7	0	0	0
index	0	1	2	3	4	5

[두더지 = 3]

(0, 3), (0, 4)는 모두 패스, (0, 5)에서 다시 1이 등장하므로 이 점을 기준으로 dfs로 flood fill을 수행하면 상, 하, 좌, 우의 모든 칸들이 3으로 바뀜.

Size[3]을 방문한 정점의 수인 8로 채우고, 두더지의 값 1 증가

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

Size	0	0	7	8	0	0
index	0	1	2	3	4	5

[두더지 = 4]

(0, 6)부터 (4, 0)까지는 1이 하나도 없으므로 모두 패스. (4, 1)에서 1이 등장하므로 이 칸으로부터 dfs로 모든 영역을 4로 채움.
그리고 방문한 정점의 수를 Size[4]에 기록함, 두더지 값은 5가 됨.

0	2	2	0	3	0	0
0	2	2	0	3	0	3
2	2	2	0	3	0	3
0	0	0	0	3	3	3
0	4	0	0	0	0	0
0	4	4	4	4	4	0
0	4	4	4	0	0	**0**

Size	0	0	7	8	9	0
index	0	1	2	3	4	5

[두더지 = 5]

마지막 칸까지 1의 값이 없으므로 모든 작업 종료.

세 마리 두더지가 있었고, 각 굴의 크기는 7, 8, 9임을 알 수 있음.

이 풀이에서는 특히 깊이우선탐색과 std::sort()를 내림차순 정렬하는 과정도 포함하고 있으므로, 잘 익혀두면 많은 도움이 될 것이다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#include <algorithm>`	
3		
4	`int n, A[101][101], cnt, Size[101];`	
5		
6	`int main()`	
7	`{`	
8	`  input();`	
9	`  solve();`	
10	`  output();`	
11	`}`	

기본적인 변수와 main 함수 부분이다. 입력, 풀이, 출력으로 따로 호출하고 있으며, 각 변수에 대한 설명은 다음과 같다.

배열 A는 전체 지도를 저장할 배열 (0은 땅, 1은 굴), 배열 Size는 각 두더지 굴의 크기를 저장할 배열, 배열 dx, dy는 현재 지점과 연결된 4곳의 x, y축 이동 양을 저장하는 배열이며, cnt는 총 두더지 굴의 수를 저장할 변수이다.

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	}	

safe 함수는 이동해야할 장소가 지도의 경계를 넘었는지 검사하는 판정 함수, 지도를 벗어나는 곳이라면 false를 반환한다.

cmp는 정수를 기준으로 내림차순으로 정렬하기 위한 비교 함수

줄	코드	참고
1	void dfs(int a, int b, int c)	
2	{	
3	A[a][b]=c;	
4	if(safe(a+1, b) && A[a+1][b]==1)	
5	dfs(a+1, b, c);	
6	if(safe(a-1,b) && A[a-1][b]==1)	
7	dfs(a-1, b, c);	
8	if(safe(a,b+1) && A[a][b+1]==1)	
9	dfs(a, b+1, c);	
10	if(safe(a,b-1) && A[a][b-1]==1)	
11	dfs(a, b-1, c);	
12	}	
13	void solve()	
14	{	
15	for(int i=0; i<n; i++)	
16	for(int j=0; j<n; j++)	
17	if(A[i][j]==1)	
18	{	
19	cnt++;	
20	dfs(i,j,cnt+1);	
21	}	
22	for(int i=0; i<n; i++)	
23	for(int j=0; j<n; j++)	
24	if(A[i][j])	
25	Size[A[i][j]-2]++;	
26	std::sort(Size, Size+cnt, cmp);	
27	}	

A배열의 (0, 0)부터 (n-1, n-1)까지 차례로 검사하면서 만약 굴의 일부가 발견되면, 그 부분으로부터 시작하여 dfs로 연결된 굴을 모두 검사한다.

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

dfs 함수 부분의 4방향 탐색을 dx, dy를 이용하여 다음과 같이 편리하게 작성할 수 있다.

줄	코드	참고
1	`int dx[4]={1,0,-1,0}, dy[4]={0,1,0,-1};`	1: 4방향의 성분을 미리 설정한다.
2		
3	`void dfs(int a, int b, int c)`	
4	`{`	
5	`  A[a][b] = c;`	
6	`  for(int i=0; i<4; i++)`	
7	`    if(safe(a+dx[i],b+dy[i]) && A[a+dx[i]][b+dy[i]]==1)`	
8	`      dfs(a+dx[i], b+dy[i], c);`	
9	`}`	

이 방법은 앞으로도 이런 패턴의 문제에 다양하게 활용할 수 있으므로 활용법을 익힐 수 있도록 한다.

4방향으로 모두 탐색하며 탐색한 곳은 2이상의 값으로 바꾼다. 따라서 굴을 탐색하는 과정에서 1이 나오면 아직 확인하지 않은 두더지 굴이고 2이상의 값이 있다면 한 마리의 두더지의 굴로 확인했다는 의미로 해석할 수 있다.

22~25행은 각 굴의 크기를 Size배열에 채우는 과정을 나타낸다. 이 아이디어도 자주 활용하는 방법이므로 잘 익혀둘 수 있도록 한다. 26행은 Size의 내용을 내림차순으로 정렬하는 부분이다.

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 다음과 같이 처리할 수도 있다.

```
scanf("%1d",&A[i][j]);
```

위와 같이 입력받으면, 연속된 문자열로부터 1자씩 정수형으로 입력받는 것이 가능하다. 예를 들어 다음과 같이 주민등록 번호로부터 생년월일, 성별 등을 알고자 할 때, 다음과 같이 입력받으면 매우 편리하다.

```
scanf("%2d%2d%2d-%1d%d", &year, &mon, &day, &gender, &etc);
```

위와 같이 입력문을 사용하고 입력은 단순히 문자열 형태로 처리할 수 있다.

마지막으로 출력을 담당하는 함수인 output()은 먼저 굴의 수를 출력하고, 큰 굴부터 하나씩 출력한다.