줄	코드	참고
1	#include<stdio.h>	
2	int m[11][11];	
3	int col_check[11];	
4	int n, min_sol=0x7fffffff;	
5		
6	void input(void)	
7	{	
8	scanf("%d", &n);	
9	for(int i=0; i<n; i++)	
10	for(int j=0; j<n; j++)	
11	scanf("%d", &m[i][j]);	
12	}	
13		
14	void solve(int row, int score)	
15	{	
16	if(row==n)	
17	{	
18	if(score<min_sol)	
19	min_sol = score;	
20	return;	
21	}	
22	for(int i=0; i<n; i++)	
23	{	
24	if(col_check[i]==0)	
25	{	
26	col_check[i]=1;	
27	solve(row+1, score+m[row][i]);	
28	col_check[i]=0;	
29	}	
30	}	
31	return;	
32	}	
33		
34	int main()	
35	{	
36	input();	
37	solve(0, 0);	
38	printf("%d", min_sol);	
39	return 0;	
40	}	

문제 11

앱(S)

우리는 스마트폰을 사용하면서 여러 가지 앱 (App)을 실행하게 된다. 대개의 경우 화면에 보이는 '실행중'인 앱은 하나뿐이지만 보이지 않는 상태로 많은 앱이 '활성화' 되어 있다. 앱들이 활성화되어 있다는 것은 화면에 보이지 않더라도 메인메모리에 직전의 상태가 기록되어 있는 것을 말한다. 현재 실행중이 아니더라도 이렇게 메모리에 남겨두는 이유는 사용자가 이전에 실행하던 앱을 다시 불러올 때에 직전의 상태를 메인메모리로부터 읽어 들여 실행 준비를 빠르게 마치기 위해서이다.

하지만 스마트폰의 메모리는 제한적이기 때문에 한 번이라도 실행했던 모든 앱을 활성화된 채로 메인메모리에 남겨두다 보면 메모리 부족 상태가 되기 쉽다. 새로운 앱을 실행시키기 위해 필요한 메모리가 부족해지면 스마트폰의 운영체제는 활성화되어 있는 앱들 중 몇 개를 선택하여 메모리로부터 삭제하는 수밖에 없다. 이러한 과정을 앱의 '비활성화'라고 한다.

메모리 부족 상황에서 활성화되어있는 앱들을 무작위로 필요한 메모리만큼 비활성화하는 것은 좋은 방법이 아니다. 비활성화된 앱들을 재실행할 경우 그만큼 시간이 더 필요하기 때문이다. 여러분은 이러한 앱의 비활성화 문제를 스마트하게 해결하기 위한 프로그램을 작성해야 한다.

현재 n개의 앱,A1, .. , An 이 활성화되어 있다고 가정하자. 이들 앱Ai 는 각각 mi 바이트만큼의 메모리를 사용하고 있다. 또한, 앱 Ai 를 비활성화한 후에 다시 실행하고자 할 경우, 추가적으로 들어가는 비용(시간 등)을 수치화한 것을 ci라고 하자. 이러한 상황에서 사용자가 새로운 앱 B를 실행하고자 하여, 추가로 M 바이트의 메모리가 필요하다고 하자. 즉, 현재 활성화되어 있는 앱 A1,..,An 중에서 몇 개를 비활성화하여 M 바이트 이상의 메모리를 추가로 확보해야 하는 것이다. 여러분은 그 중에서 비활성화했을 경우의 비용 ci의 합을 최소화하여 필요한 메모리 M 바이트를 확보하는 방법을 찾아야 한다.

앱(S) (계속)

입력

첫 줄에는 정수 n과 M이 공백문자로 구분되어 주어지며,

둘째 줄과 셋째 줄에는 각각 n개의 정수가 공백문자로 구분되어 주어진다.

둘째 줄의 n개의 정수는 현재 활성화되어 있는 앱 A1,..., An이 사용 중인 메모리의 바이트 수인 m1, ..., mn을 의미하며,

셋째 줄의 n 개의 정수는 각 앱을 비활성화했을 경우의 비용 c1,..., cn을 의미한다.

[입력의 정의역]

$1 \le n \le 100$

$1 \le M \le 10{,}000{,}000$

$1 \le m1, \ldots, mn \le 10{,}000{,}000$

$0 \le c1, \ldots, cn \le 100$

출력

필요한 메모리 M 바이트를 확보하기 위한 앱 비활성화의 최소의 비용을 계산하여 한 줄에 출력해야 한다.

입력 예	출력 예
5 60 30 10 20 35 40 3 0 3 5 4	6

출처: 한국정보올림피아드(2013 지역본선 중고등부)

풀이

이 문제는 새로운 앱을 실행하기 위해 활성화되어 있는 앱들 중 몇 개를 비활성화해서 메모리 M 이상을 확보하는 데 드는 비용을 최소화하는 문제이다.

이 상황을 잘 생각해보면 배낭 문제(문제 17)와 유사해 보인다.

	배낭 문제		앱
N	배낭에 담을 수 있는 물건 개수	n	비활성화 할 수 있는 앱의 개수
W	배낭의 무게	M	확보해야할 메모리량
Wi	각 물건의 무게	mi	각 앱의 메모리 사용량
Vi	물건의 가치	ci	앱의 비활성화에 드는 비용

배낭 문제에서는 배낭의 무게 W를 넘지 않으면서 물건 가치를 최대로 높이는 경우를 찾는 것이고, 앱 문제에서는 메모리를 M 이상을 확보하면서 비활성화에 드는 최소 비용을 찾는 것이다.

문제에서 제시한 상황은 다음 표와 같다.

앱의 번호(i)	사용 중인 메모리(mi)	비활성화 비용(ci)
1	30	3
2	10	0
3	20	3
4	35	5
5	40	4

요구하는 메모리가 60이므로 비활성화 비용을 최소화하면서 60 이상의 메모리를 확보하기 위해서는 1, 2, 3번의 앱을 비활성화시켜야 한다.(30+10+20=60, 3+0+3=6)

배낭 문제와 유사하기 때문에 배낭 문제에서 사용한 알고리즘을 이 문제에 맞게 변형시켜보자. 배낭문제에서는 f(1, 0)을 호출해서 답을 구했지만, 이번에는 다른 방법으로 설계해도 똑같은 결과가 나온다는 것을 보여주기 위해 반대로 f(n, M)으로 설계하였다.

줄	코드	참고
1	#include <stdio.h>	8: 앱 번호 i, 남은 메모리 r
2	#define MAXV 999999	
3		
4	int M, n, i, m[101], c[101];	
5		
6	int min(int a, int b) { return a<b ? a:b;}	
7		
8	int f(int i, int r)	
9	{	
10	if(i==0)	
11	{	
12	if(r<=0) return 0;	
13	else return MAXV;	
14	}	
15	else if (r<0)	
16	return f(i-1, r);	
17	else	
18	return min(f(i-1,r), f(i-1,r-m[i])+c[i]);	
19	}	
20		
21	int main()	
22	{	
23	scanf("%d %d", &n, &M);	
24	for(i=1; i<=n; i++) scanf("%d", &m[i]);	
25	for(i=1; i<=n; i++) scanf("%d", &c[i]);	
26	printf("%d", f(n, M));	
27	return 0;	
28	}	

함수 f의 의미는 다음과 같다.

f(i, r) = 1~i번째 앱까지 고려했을 때, 메모리 r이상 확보하기 위한 최소 비용

설계 방법의 변경과 최솟값을 구하는 부분에서 약간의 소스가 변경되었다. 이 방법으로는 이 문제를 완벽하게 해결하기에는 시간이 부족하다. 고급편에서 이 알고리즘의 시간을 줄이는 방법에 대해서 다룬다.

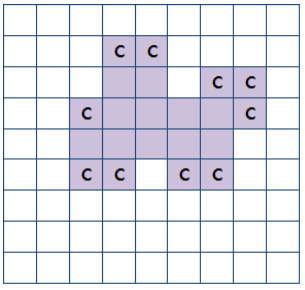
문제 12

치즈

n*m(5≤ n, m ≤100)의 모눈종이 위에 아주 얇은 치즈가 [그림 1]과 같이 표시되어 있다. 단, n은 세로 격자의 수이고, m은 가로 격자의 수이다. 이 치즈는 냉동 보관을 해야만 하는데 실내온도에 내어놓으면 공기와 접촉하여 천천히 녹는다.

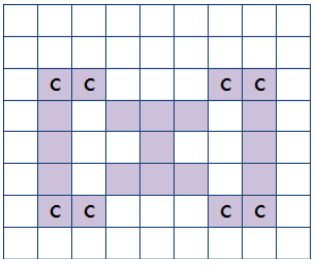
그런데 이러한 모눈종이 모양의 치즈에서 각 치즈 격자(작은 정사각형 모양)의 네 변 중에서 적어도 두 변 이상이 실내온도의 공기와 접촉한 것은 정확히 한 시간 만에 녹아 없어져 버린다.

따라서 아래 [그림 1] 모양과 같은 치즈(회색으로 표시된 부분)라면 C로 표시된 모든 치즈 격자는 한 시간 후에 사라진다.

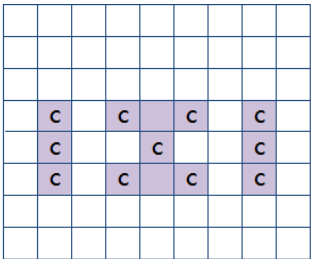


[그림 1]

[그림 2]와 같이 치즈 내부에 있는 공간은 치즈 외부 공기와 접촉하지 않는 것으로 가정한다. 그러므로 이 공간에 접촉한 치즈 격자는 녹지 않고 C로 표시된 치즈 격자만 사라진다. 그러나 한 시간 후, 이 공간으로 외부공기가 유입되면 [그림 3]에서와 같이 C로 표시된 치즈 격자들이 사라지게 된다.



[그림 2]



[그림 3]

치즈 (계속)

모눈종이의 맨 가장자리에는 치즈가 놓이지 않는 것으로 가정한다. 입력으로 주어진 치즈가 모두 녹아 없어지는 데 걸리는 정확한 시간을 구하는 프로그램을 작성하시오.

입력

첫째 줄에는 모눈종이의 크기를 나타내는 두 개의 정수 n, m(5 ≤ n, m ≤100)이 주어진다. 그 다음 n개의 줄에는 모눈종이 위의 격자에 치즈가 있는 부분은 1로 표시되고, 치즈가 없는 부분은 0으로 표시된다. 또한, 각 0과 1은 하나의 공백으로 분리되어 있다.

출력

출력으로는 주어진 치즈가 모두 녹아 없어지는 데 걸리는 정확한 시간을 정수로 첫 줄에 출력한다.

입력 예	출력 예
8 9 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 0 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 1 1 1 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	4

출처: 한국정보올림피아드(2000 지역본선 초등부)

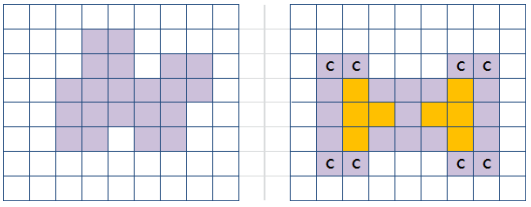
풀이

이 문제는 앞에서 다루었던 깊이우선탐색이나 너비우선탐색을 이용한 flood fill 기법과 다양한 문제해결 기법을 응용해야 되는 문제이다. 이 문제를 통해서 많은 것을 배울 수 있다.

치즈가 녹는 규칙과 치즈의 초기 상태가 주어졌을 때, 치즈가 다 녹는 데 걸리는 시간을 출력하는 문제이다.

시간이 진행되면서 치즈가 어떻게 녹는지를 시뮬레이션하는 방법으로 풀 수 있다.

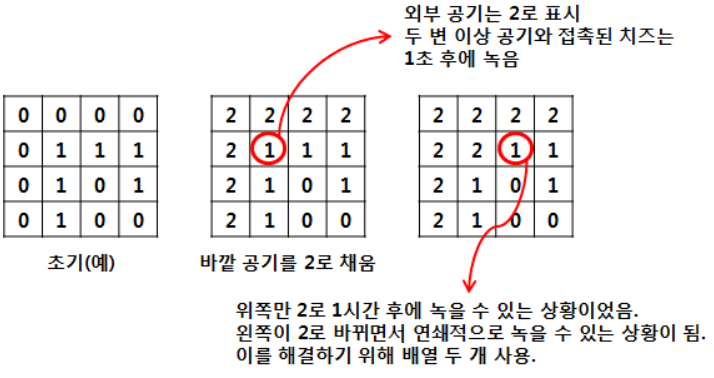
즉, 매 시간마다 치즈의 상태가 변해가는 모습을 기록한 후, 최종 상태까지 걸린 시간을 출력하면 된다.



주황 격자에 해당하는 내부 공간은 치즈를 녹이지 못하기 때문에 한 시간 후에는 C로 표시된 치즈만 사라진다.

바깥쪽 공기와 치즈의 안쪽 구멍을 구분하는 것이 이 문제의 핵심이다. 먼저 가장 쉽게 모든 것을 구현하며 풀어보는 방법을 알아보자. 기본 아이디어는 1시간마다 백트래킹으로 바깥 공기를 다시 체크하는 방법이다.

단, 녹을 치즈를 바로 공기로 바꾸면 영향을 받기 때문에 배열 두 개를 사용한다.



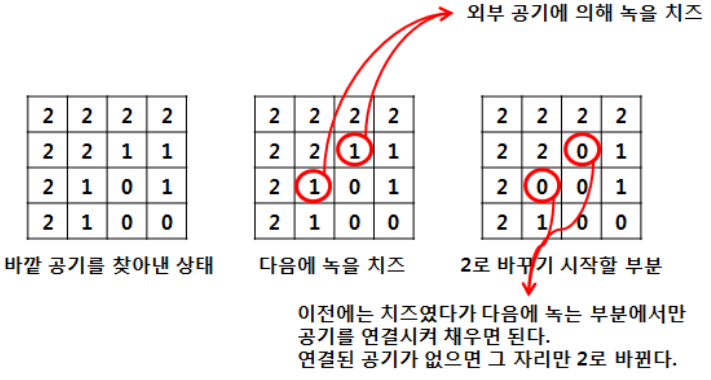
위 아이디어를 flood fill기법으로 해결한 소스코드는 다음과 같다.

줄	코드	참고
1	#include <stdio.h>	
2	int a1[101][101], a2[101][101];	
3	int n, m;	
4	void copy()	
5	{	
6	int i, j;	
7	for(i=1; i<=n; i++)	
8	for(j=1; j<=m; j++)	
9	a1[i][j]=a2[i][j];	
10	}	
11	void fill1(int x, int y)	
12	{	
13	if(x<1 || y<1 || x>n || y>m) return;	
14	if(a1[x][y]==0)	
15	{	
16	a1[x][y]=2;	
17	fill1(x+1,y);	
18	fill1(x-1,y);	
19	fill1(x,y+1);	
20	fill1(x,y-1);	
21	}	
22	}	
23	int check(int x, int y)	
24	{	
25	int t=0;	
26	if(a1[x+1][y]==2) t++;	
27	if(a1[x-1][y]==2) t++;	
28	if(a1[x][y+1]==2) t++;	
29	if(a1[x][y-1]==2) t++;	
30	return t;	
31	}	
32	int main()	
33	{	
34	int i, j, hour=0, count;	
35	scanf("%d %d",&n, &m);	
36	for(i=1; i<=n; i++)	
37	for(j=1; j<=m; j++)	
38	{	
39	scanf("%d",&a1[i][j]);	
40	a2[i][j]=a1[i][j];	

줄	코드	참고
41	} while(1)	
42	{	
43	fill1(1,1);	
44	count=0;	
45	for(i=1; i<=n; i++)	
46	for(j=1; j<=m; j++)	
47	{	
48	if(a1[i][j]==1 && check(i,j)>=2)	
49	{	
50	a2[i][j]=0;	
51	count++;	
52	}	
53	}	
54	if(count==0)	
55	{	
56	printf("%d", hour);	
57	break;	
58	}	
59	hour++;	
60	copy();	
61	}	
62	return 0;	
63	}	
64		

위 알고리즘에서 조금 더 개선하는 방법을 생각해보자. 한 시간마다 바깥 공기 덩어리를 다시 체크하지 않고, 녹았을 때만 그 자리에서 다시 연결된 공기를 체크한다.

이렇게 하면 조금 더 빠르게 해결할 수 있다.



위 알고리즘으로 fill을 fill1과 fill2로 만들 수 있다.

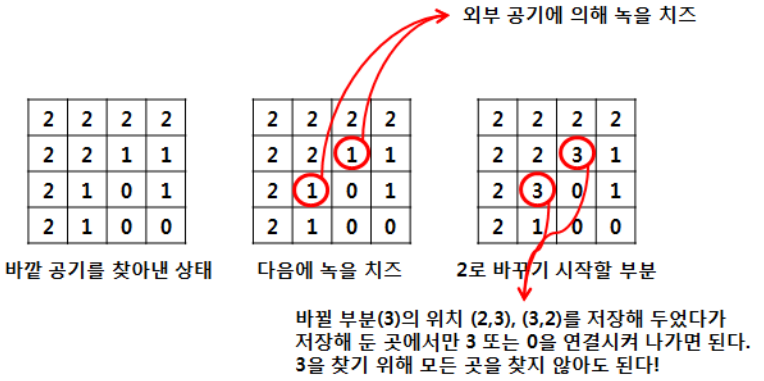
줄	코드	참고
1	void fill1(int x, int y)	
2	{	
3	if(x<1 || y<1 || x>n || y>m) return;	
4	if(a1[x][y]==0)	
5	{	
6	a1[x][y]=2;	
7	fill1(x+1,y);	
8	fill1(x-1,y);	
9	fill1(x,y+1);	
10	fill1(x,y-1);	
11	}	
12	}	
13		
14	void fill2(int x, int y)	
15	{	
16	if(x<1 || y<1 || x>n || y>m) return;	
17	if(a2[x][y]==0)	
18	{	
19	a2[x][y]=2;	
20	fill2(x+1,y);	
21	fill2(x-1,y);	
22	fill2(x,y+1);	
23	fill2(x,y-1);	
24	}	
25	}	

이와 같이 작성하면 main()은 다음과 같이 수정된다.

줄	코드	참고
1	int main()	
2	{	
3	int i, j, hour=0, count;	
4	scanf("%d %d",&n, &m);	
5	for(i=1; i<=n; i++)	
6	for(j=1; j<=m; j++)	
7	{	
8	scanf("%d",&a1[i][j]);	
9	a2[i][j]=a1[i][j];	
10	}	
11	fill1(1,1);	

줄	코드	참고
12	fill2(1,1);	
13	while(1)	
14	{	
15	count=0;	
16	for(i=1; i<=n; i++)	
17	for(j=1; j<=m; j++)	
18	{	
19	if(a1[i][j]==1 && check(i,j)>=2)	
20	{	
21	a2[i][j]=0;	
22	count++;	
23	}	
24	}	
25	if(count==0)	
26	{	
27	printf("%d", hour); break;	
28	}	
29	for(i=1; i<=n; i++)	
30	for(j=1; j<=m; j++)	
31	if(a1[i][j]==1 && a2[i][j]==0)	
32	fill2(i,j);	
33	hour++;	
34	copy();	
35	}	
36	return 0;	
37	}	

위의 알고리즘들을 조금 더 개선해 보자. 바뀔 부분을 찾기 위해 모두 검색하는 것이 아니라, 저장해 두었다가 처리한다.



줄	코드	참고
1	void fill3(int x, int y)	
2	{	
3	if(x<1 || y<1 || x>n || y>m) return;	
4	if(a1[x][y]==3 || a1[x][y]==0)	
5	{	
6	a1[x][y]=2;	
7	fill3(x+1,y);	
8	fill3(x-1,y);	
9	fill3(x,y+1);	
10	fill3(x,y-1);	
11	}	
12	}	
13		
14	int main()	
15	{	
16	int i, j, hour=0, count;	
17	scanf("%d %d",&n, &m);	
18	for(i=1; i<=n; i++)	
19	for(j=1; j<=m; j++)	
20	{	
21	scanf("%d",&a1[i][j]);	
22	a2[i][j]=a1[i][j];	
23	}	
24	fill3(1,1);	
25	while(1)	
26	{	
27	count=0;	
28	for(i=1; i<=n; i++)	
29	for(j=1; j<=m; j++)	
30	{	
31	if(a1[i][j]==1 && check(i,j)>=2)	
32	{	
33	a2[i][j]=0;	
34	count++;	
35	}	
36	}	
37	if(count==0)	
38	{	
39	printf("%d", hour); break;	
40	}	

줄	코드	참고
41	for(i=1; i<=n; i++)	
42	for(j=1; j<=m; j++)	
43	if(a1[i][j]==1 && a2[i][j]==0)	
44	fill2(i,j);	
45	hour++;	
46	copy();	
47	}	
48	return 0;	
49	}	

마지막으로 위 아이디어를 조금 더 효율적으로 접근한 소스코드를 소개한다. 이 소스코드에는 지금까지 배웠던 다양한 기법들이 적용되었기 때문에, 자세히 분석해서 익힐 수 있도록 한다.

줄	코드	참고
1	#include <cstdio>	
2	int dx[4]={1,0,-1,0}, dy[4]={0,1,0,-1}, h, w, S[110][110],	
3	res;	
4	bool inside(int a, int b)	
5	{	
6	return ((0<=a && a<h) && (0<=b && b<w));	
7	}	
8	bool done(void)	
9	{	
10	int cnt=0;	
11	for(int i=0; i<h; i++) for(int j=0; j<w; j++)	
12	if(S[i][j]==-1 || S[i][j]>2) S[i][j]=0;	
13	else if(S[i][j]==2 || S[i][j]==1)	
14	S[i][j]=1, cnt++;	
15	return cnt==0;	
16	}	
17	int solve(int a, int b)	
18	{	
19	S[a][b]=-1;	
20	for(int i=0; i<4; i++)	
21	if(inside(a+dx[i], b+dy[i]))	
22	{	
23	if(S[a+dx[i]][b+dy[i]]==0)	

줄	코드	참고
24	solve(a+dx[i],b+dy[i]);	
25	else if(S[a+dx[i]][b+dy[i]]>0)	
26	S[a+dx[i]][b+dy[i]]++;	
27	}	
28	}	
29	int main()	
30	{	
31	scanf("%d %d",&h,&w);	
32	for(int i=0; i<h; i++) for(int j=0; j<w; j++)	
33	scanf("%d", &S[i][j]);	
34	for(res=0; !done(); res++) solve(0,0);	
35	printf("%d", res);	
36	return 0;	
37	}	

문제 13

두 색 칠하기 (bicoloring)

평면 위에 지도가 있을 때, 각 영역을 인접한 다른 영역과 구분할 수 있게 서로 다른 색으로 칠하고자 한다면, 네 가지 색만 있으면 된다는 4색 정리라는 것이 있다. 이 정리는 100년이 넘게 증명되지 않은 채로 남아 있다가 1976년에서야 컴퓨터의 도움을 받아서 증명될 수 있었다.

이 문제는 그래프의 정점을 칠하는 문제로 구조화하여 풀 수 있다. 어떤 연결 그래프가 주어졌을 때 그 그래프를 두 색으로 칠할 수 있는지, 즉 모든 정점을 빨간색 또는 검은색으로 칠할 때 인접한 정점이 같은 색으로 칠해지지 않게 할 수 있는지 알아보자.

문제를 단순하게 하기 위해 그래프가 연결 그래프이고 무방향 그래프이며 자체 루프가 없다고 가정하자. 0부터 n-1까지의 n개의 정점과 간선의 수 m이 입력될 때, 2가지 색깔로 칠할 수 있는지 결정하는 프로그램을 작성하시오.

입력

첫째 줄에는 정점의 개수 n($1 \le n \le 200$)과 간선의 수 m이 입력된다.

둘째 줄부터 m줄에 걸쳐서 각 간선이 연결하는 정점의 번호가 공백으로 구분되어 입력된다.

출력

입력된 그래프가 두 색으로 칠할 수 있는 그래프인지를 판단하고 아래 예에 나온 형식에 맞게 결과를 출력하라.

입력 예	출력 예
3 3 0 1 1 2 2 0	IMPOSSIBLE
9 8 0 1 0 2 0 3 0 4 0 5 0 6 0 7 0 8	OK

풀이

그래프이기 때문에 비선형 전체탐색으로 해결할 수 있다.

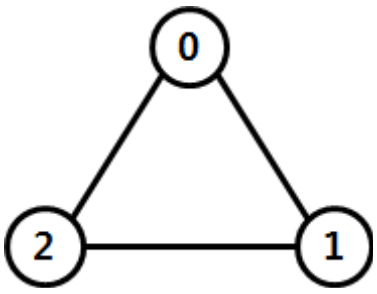
임의의 정점에 어떤 색이 칠해졌는지는 관계없다. 주변의 정점들과의 색깔만 다르면 되기 때문에 시작 정점을 검은색, 빨간색 중 아무거나 칠해도 관계없다.

시작 정점에서 임의의 색깔로 출발하여 인접한 정점에는 다른 색깔을 칠하도록 하자. 이 과정에서 깊이우선탐색이나 너비우선탐색 등의 방법은 모두 가능하다.

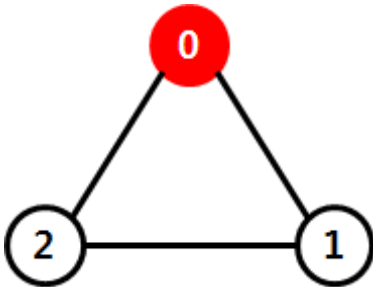
여기서는 깊이우선탐색을 기반으로 한 다음 알고리즘으로 색깔을 칠해보자. 1은 검은색, 2는 빨간색을 의미한다.

1. 정점 0을 1로 칠하고 정점 0에 연결된 임의의 정점으로 탐색을 진행.
2. 현재 정점을 1로 칠해보고 불가능하면 현재 정점을 2로 칠해보고 탐색을 진행.
3. 모든 정점에 색깔을 칠할 수 있으면 OK, 아니면 IMPOSSIBLE.

입력 예시1 로 주어진 그래프를 칠해가는 과정은 다음과 같다.

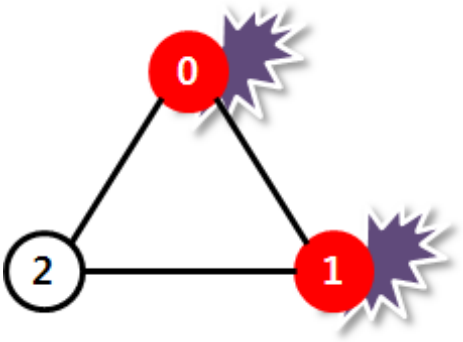


초기 그래프 상태
3개의 정점과, 3개의 간선을 가지는 그래프이다.

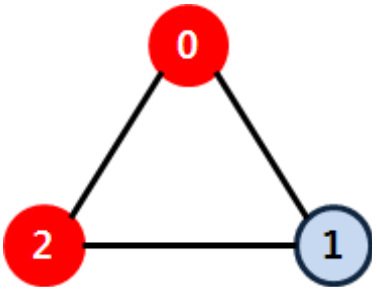


출발정점을 빨간색으로 칠하고 다음으로 1번 정점으로 탐색을 옮김

출발정점을 검은색으로 칠해도 의미는 같음.

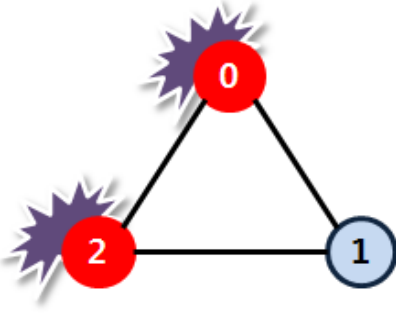


먼저 1번 정점을 빨간색으로 칠해본다. 하지만 정점 0과 같은 색깔이므로 사용 불가 백트랙!!

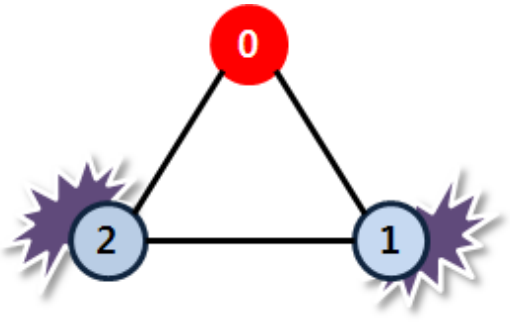


다음으로 검은색으로 칠해본다. 주변의 정점들과 색깔이 다르기 때문에 가능.

계속하여 2번 정점으로 탐색을 이동.



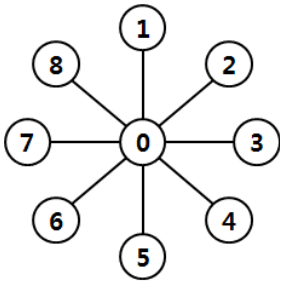
2번 정점을 먼저 빨간색으로 칠해본다. 하지만 0번 정점과 색깔이 같으므로 사용 불가 백트랙!!



2번 정점을 다시 검은색으로 칠해본다. 역시 1번 정점과 색깔이 같으므로 사용 불가!! 백트랙!!

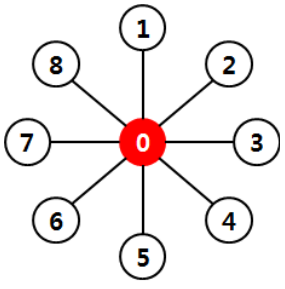
이와 같은 방법으로 끝까지 백트랙을 해도 채울 수 있는 방법이 없으므로 결과는 IMPOSSIBLE이 된다.

다음으로 2번째 예제의 경우를 살펴보자. 2번째 예제는 다음과 같은 방법으로 칠할 수 있으므로 처리할 수 있다.

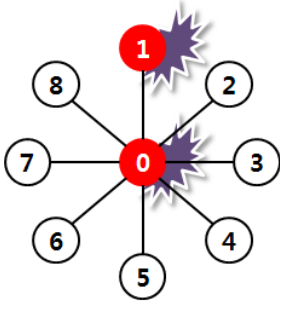


초기 상태는 다음과 같다.

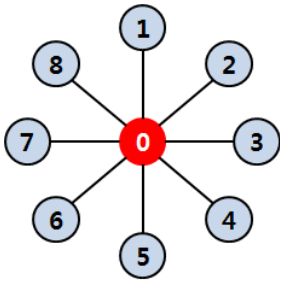
9개의 정점과 8개의 간선을 가지는 트리 형태의 그래프이다. (사실 트리는 항상 2가지 색깔로 칠할 수 있다. 이는 수학적으로 증명된다.)



출발 정점을 빨간색으로 칠하고 1번 정점으로 탐색을 이동한다.



1번을 빨간색으로 칠하면 0번과 색깔이 같으므로 불능!! 백트랙!!



검은 색으로 칠하면 이상 없음...

이와 같은 과정으로 나머지 모든 정점도 검은색으로 처리할 수 있으므로 2가지 색깔로 처리할 수 있다.

위의 과정을 작성한 소스코드는 다음과 같다.

줄	코드	참고
1	`#include <stdio.h>`	
2		
3	`int n, m, G[200][200], visited[200];`	
4		
5	`void solve(int v, int c)`	
6	`{`	
7	`  visited[v]=c;`	
8	`  int can=1;`	
9	`  for(int i=0; i<n; i++)`	
10	`    if(G[v][i] && visited[i]==c) can=0;`	
11	`  if(!can)`	
12	`  {`	
13	`    visited[v]=0;`	
14	`    return;`	
15	`  }`	
16	`  for(int i=0; i<n; i++)`	
17	`  {`	
18	`    if(!visited[i] && G[v][i])`	
19	`    {`	
20	`      solve(i, 1);`	
21	`      solve(i, 2);`	
22	`    }`	
23	`  }`	
24	`}`	
25		
26	`int main()`	
27	`{`	
28	`  scanf("%d %d", &n, &m);`	
29	`  for(int i=0; i<m; i++)`	
30	`  {`	
31	`    int s, e;`	
32	`    scanf("%d%d",&s,&e);`	
33	`    G[s][e]=G[e][s]=1;`	
34	`  }`	
35	`  solve(0, 1);`	
36	`  for(int i=0; i<n; i++)`	
37	`    if(visited[i]==0)`	
38	`    {`	

줄	코드	참고
39	puts("IMPOSSIBLE");	
40	return 0;	
41	}	
42	printf("OK");	
43	return 0;	
44	}	

위 알고리즘은 그래프를 인접행렬로 표현한 것이다. 이를 인접리스트로 표현하면 속도가 더 빨라진다. 이 문제의 경우에는 정점의 수가 적어서 인접행렬로도 해결이 되지만 인접리스트로도 작성하는 연습을 하는 것이 좋다.

인접리스트로 작성한 소스코드는 다음과 같다. 실전에서는 인접리스트 표현이 더 자주 활용되므로 익혀두기 바란다.

줄	코드	참고
1	#include <stdio.h>	
2	#include <vector>	
3		
4	int n, m, visited[200];	
5	std::vector<int> G[200];	
6		
7	void solve(int v, int c)	
8	{	
9	visited[v]=c;	
10	int can=1;	
11	for(int i=0; i<G[v].size(); i++)	
12	if(visited[G[v][i]]==c) can=0;	
13	if(!can)	
14	{	
15	visited[v]=0;	
16	return;	
17	}	
18	for(int i=0; i<G[v].size(); i++)	
19	{	
20	if(!visited[G[v][i]])	
21	{	
22	solve(G[v][i], 1);	
23	solve(G[v][i], 2);	
24	}	

줄	코드	참고
25	} }	
26		
27		
28	int main()	
29	{	
30	scanf("%d %d", &n, &m);	
31	for(int i=0; i<m; i++)	
32	{	
33	int s, e;	
34	scanf("%d %d",&s,&e);	
35	G[s].push_back(e);	
36	G[e].push_back(s);	
37	}	
38	solve(0, 1);	
39	for(int i=0; i<n; i++)	
40	if(visited[i]==0)	
41	{	
42	puts("IMPOSSIBLE");	
43	return 0;	
44	}	
45	printf("OK");	
46	return 0;	
47	}	

빨간색으로 표시된 코드는 인접행렬로 구현했을 때와의 차이가 나는 부분을 의미한다.

문제 14

maximum sum(S)

n개의 원소로 이루어진 집합이 있다. 이 집합에서 최대로 가능한 부분합을 구하는 것이 문제이다.

부분합이란 n개의 원소 중 i번째 원소로부터 j번째 원소까지의 연속적인 합을 의미한다(단, 1 < i <= j <= n). 만약 다음과 같이 6개의 원소로 이루어진 집합이 있다고 가정하자.

6 -7 3 -1 5 2

이 집합에서 만들어지는 부분합 중 최댓값은 3번째 원소부터 6번째 원소까지의 합인 9이다.

입력

첫 줄에 원소의 수를 의미하는 정수 n이 입력되고, 둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다.
(단, 2 <= n <= 100 , 각 원소의 크기는 -1000부터 1000 사이의 정수이다.)

출력

주어진 집합에서 얻을 수 있는 최대 부분합을 출력한다.

입력 예	출력 예
6 6 -7 3 -1 5 2	9

풀이

이 문제는 n의 값이 클 경우에는 고민해야 할 부분 많은데 이 경우는 n의 최댓값이 100이기 때문에 $O(n^3)$으로도 처리할 수 있다. 따라서 단순히 선형으로 전체탐색하면 해를 구할 수 있다.

기본적인 알고리즘은 다음과 같다.

1. 구간합을 구할 구간의 시작점 s를 정한다(모든 값에 대하여).
2. 구간합을 구할 구간의 끝점 e를 정한다(모든 값에 대하여).
3. [s, e] 구간의 합을 구한다. 만약 지금까지 구한 합보다 더 크면 갱신한다.
4. 마지막까지 진행하고, 가장 큰 합을 출력한다.

먼저 각 구간의 시작과 끝을 구하기 위하여 다음과 같은 코드를 작성할 수 있다. 단, 구간의 끝은 시작보다 같거나 커야 한다.

줄	코드	참고
1	int count=1;	
2	for(int s=0; s<n; s++)	
3	{	
4	for(int e=s; e<n; e++)	
5	{	
6	printf("[%d~%d] ", s, e);	
7	if(count%7==0) puts("");	
8	count++;	
9	}	
10	}	

위의 코드는 모든 구간을 설정할 수 있는 코드이다. 따라서 $O(n^2)$에 모든 구간을 설정할 수 있다. 각 구간을 출력한 결과는 다음과 같다.

```
[0~0] [0~1] [0~2] [0~3] [0~4] [0~5] [1~1]
[1~2] [1~3] [1~4] [1~5] [2~2] [2~3] [2~4]
[2~5] [3~3] [3~4] [3~5] [4~4] [4~5] [5~5]
```

위 코드에서 정한 구간의 합을 정하면 된다. 이를 정리한 소스코드는 다음과 같다.

줄	코드	참고
1	#include <stdio.h>	
2		
3	int n, A[110], ans;	
4		
5	int main()	
6	{	
7	scanf("%d", &n);	
8	for(int i=0; i<n; i++)	
9	scanf("%d", A+i);	
10	for(int s=0; s<n; s++)	
11	{	
12	for(int e=s, sum; e<n; e++)	
13	{	
14	sum=0;	
15	for(int k=s; k<=e; k++)	
16	sum+=A[k];	
17	ans=ans<sum ? sum:ans;	
18	}	
19	}	
20	printf("%d\n", ans);	
21	return 0;	
22	}	

이 문제는 아이디어에 따라서 훨씬 효율적인 방법들이 많이 있으므로, 다양한 생각들을 해보기 바란다.