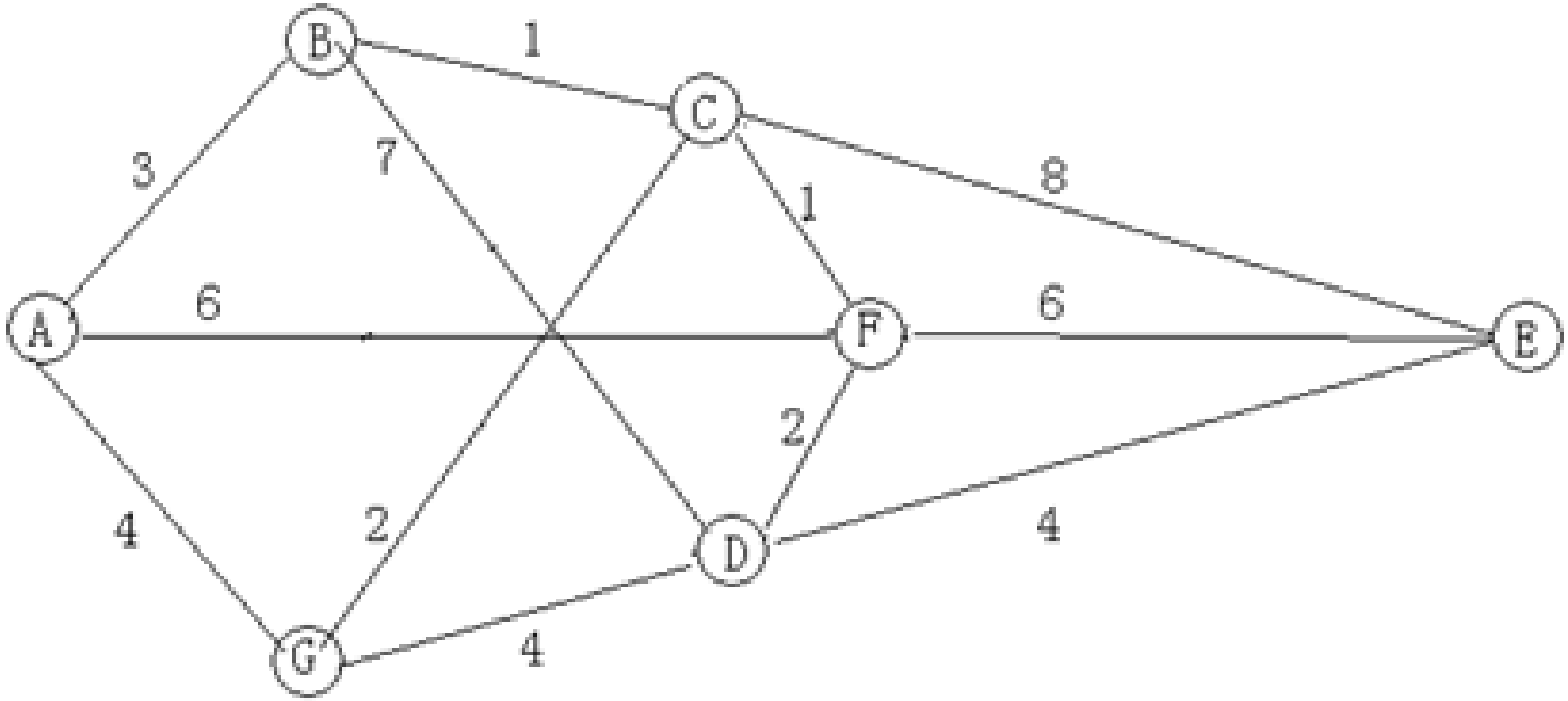
Day 5

图和图的存储结构

图

在数学上，一个图（Graph）是表示个体与个体之间的关系的方法，是图论的基本研究对象。一个图看起来是由一些小圆点（称为顶点或结点）和连结这些圆点的直线或曲线（称为边）组成的。

如果给图的每条边规定一个方向，那么得到的图称为有向图，其边也称为有向边。在有向图中，与一个节点相关联的边有出边和入边之分，而与一个有向边关联的两个点也有始点和终点之分。相反，边没有方向的图称为无向图。


B
1
C
3
7
8
1
6
A
6
F
E
2
4
2
4
D
4
G

基本概念

端点和邻接点

无向图：若存在一条边(i，j)　　顶点i和顶点j为端点，它们互为邻接点。

有向图：若存在一条边<i，j>　　顶点i为起始端点（简称为起点），顶点j为终止端点（简称终点），它们互为邻接点。

顶点的度、入度和出度

无向图：以顶点i为端点的边数称为该顶点的度。

有向图：以顶点i为终点的入边的数目，称为该顶点的入度。以顶点i为始点的出边的数目，称为该顶点的出度。一个顶点的入度与出度的和为该顶点的度。

完全图

无向图：每两个顶点之间都存在着一条边，称为无向完全图，包含有 n(n-1)/2条边。

有向图：每两个顶点之间都存在着方向相反的两条边，称为有向完全图，包含有n(n-1)条边。

路径和路径长度

在一个图**G=(V，E)**中，从顶点***i***到顶点***j***的一条路径（***i***，i_1，i_2，…，i_m，***j***）。其中所有的**(i_x，i_y)∈E(G)**，或者**<i_x，i_y> ∈E(G)**，路径长度是指一条路径上经过的边的数目。

若一条路径上除开始点和结束点可以相同外，其余顶点均不相同，则称此路径为简单路径。

回路或环

若一条路径上的开始点与结束点为同一个顶点，则此路径被称为回路或环。开始点与结束点相同的简单路径被称为简单回路或简单环。

基本概念

若图中任意两个顶点都连通，则称为连通图，否则称为非连通图。无向图G中的极大连通子图称为G的连通分量。显然，任何连通图的连通分量只有一个，即本身，而非连通图有多个连通分量。

权和网

图中每一条边都可以附带有一个对应的数值，这种与边相关的数值称为权。权可以表示从一个顶点到另一个顶点的距离或花费的代价。

边上带有权的图称为带权图，也称作网。

稀疏图与稠密图

若一张图的边数远小于其点数的平方，那么它是一张稀疏图 (Sparse graph) 。

若一张图的边数接近其点数的平方，那么它是一张稠密图 (Dense graph) 。

稀疏图与稠密图的特征决定该使用什么存储结构与算法。

图的存储

邻接矩阵

开一个二维数组G。$G[i][j]$表示边(i,j)的权。如果边(i,j)不存在，就令$G[i][j]=\texttt{INF}$（当然，(i,i)不是一条边，$G[i][i]$也等于INF）。

邻接矩阵最大的缺点就是内存空间占用太大，内存浪费严重。

边目录

设置三个数组$u[M]$、$v[M]$、$w[M]$，分别表示起点、终点和权。

邻接表（链表）

用一个列表列出所有与现结点之间有边存在的结点名称。

```
struct edge {
    int v,w;
    edge *next;
} mem[M];          // mem相当于动态内存分配。
int size=-1;
#define NEW(p) p=&mem[++size]; p->next=NULL
edge *adj[N];          // adj[i]代表以i为起点的边。
……
memset(adj, 0, sizeof(adj));
for (int e=0; e<m; e++) {
    int u;
    edge *p;
    NEW(p);
    cin>>u>>(p->v)>>(p->w);
    p->next=adj[u];
    adj[u]=p;
}

//如果想检查从a出发的所有边，那么可以
for (edge *e=adj[a]; e!=NULL; e=e->next) {
    // e->u是起点，e->v是终点，e->w是权
}
```

链式前向星

```
struct edge {
    int v,w;
    int next;
} mem[M];   // mem相当于动态内存分配。
int size=-1;
int adj[N]; // adj[i]代表以i为起点的边。

      memset(adj,-1,sizeof(adj));

   void add(int u, int v,int w){
       mem[++size].w = w;
       mem[size].v = v;
       mem[size].next = adj[u];
       adj[u] = size;
   }
```

邻接表（静态数组）
注意，在这个“邻接表”里放置的元素是边的序号，不是点的序号。所以还要和边目录配合使用。

```
int first[N];                   // first[u]表示从u出发的第一条边的序号
int u[M],v[M],w[M], next[M];        // next[e]表示编号为e的下一条边的序号
......
memset(first, -1, sizeof(first));
for (int e=0; e<m; e++){
    cin>>u[e]>>v[e]>>w[e];
    next[e]=first[u[e]];            // 插入一条边
    first[u[e]]=e;
}
```

如果想检查从a出发的所有边，那么可以

```
for (int e=first[a]; e!=-1; e=next[e]){
    // u[e]是起点，v[e]是终点，w[e]是权
}
```

图的遍历（搜索）

广（宽）度优先遍历（BFS）

调用：

```
memset(visited,0,sizeof(visited));
for (int i=0;i<n;i++)
    if (!visited[i]) BFS(i);
```

广（宽）度优先遍历（Breadth First Search）：又称为广度优先搜索，简称BFS。

遍历的过程是先从顶点 V_0 出发，遍历与 V_0 直接连接而又未访问过的顶点V_1、V_2、V_3 等，再访问 与 V_1直接连接且未访问过的顶点。同样用一个数组来标记一个顶点是否已经访问过，用一个队列来存储待访问的顶点。

```
const int INF = 0x3f3f3f3f;
queue<int> q;
bool visited[N];
void BFS(int start){
    q.push(start); visited[start]=true;
do{
        int a=q.front(); q.pop();
        // 处理点a
        cout<<a<<' ';

        for (int i=0;i<n;i++)
            if  ((!visited[i]) && (G[a][i]!=INF)){
                q.push(i);
                visited[i]=true;
            }
    } while (!q.empty());
}
```

迷宫问题

定义一个矩阵：

0 1 0 0 0

0 1 0 1 0

0 0 0 0 0

0 1 1 1 0

0 0 0 1 0

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程序找出从左上角到右下角的最短路线。

先将起始位置入队列

每次从队列拿出一个元素，扩展其相邻的4个元素入队列(要判重），直到队头元素为终点为止。队列里的元素记录了指向父节点（上一步）的指针

广搜一般用于状态表示比较简单、求最优策略的问题

优点：是一种完备策略，即只要问题有解，它就一定可以找到解。并且，广度优先搜索找到的解，还一定是路径最短的解。

缺点：盲目性较大，尤其是当目标节点距初始节点较远时，将产生许多无用的节点，因此其搜索效率较低。需要保存所有扩展出的状态，占用的空间大。

深度优先遍历
DFS

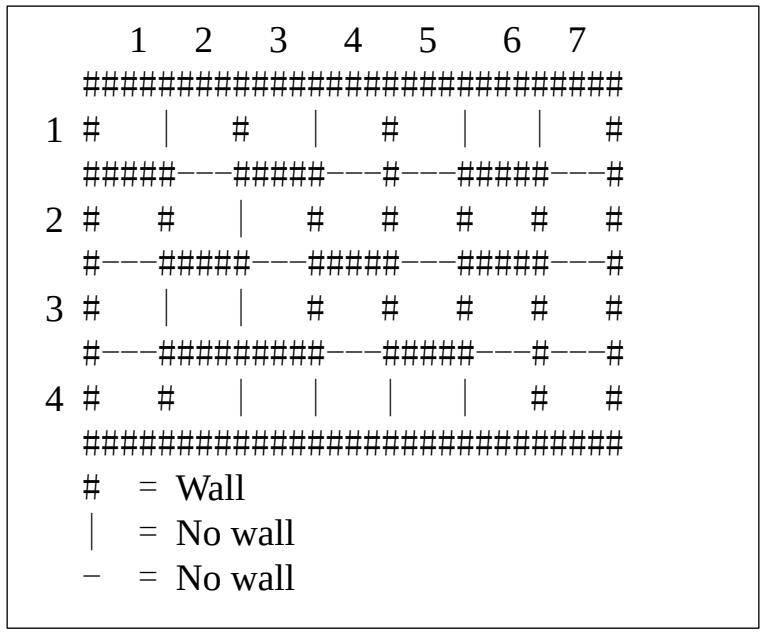
深度优先遍历（Depth First Search）：也称为深度优先搜索，DFS，深度优先搜索其实是一个递归过程，一条路先走到底有，点像二叉树的先序遍历。我们需要一个数组 visited 来标记已经访问过的顶点，`visited[i] = false` 表示未访问，true 表示已经访问过。

这是基于邻接矩阵的遍历。如果需要，可以改成基于邻接表的遍历。

```
bool visited[N];
void DFS(int start){
    visited[start]=true;
    // 处理点start
    cout<<start<<' ';

    for (int i=0; i<n; i++)
        if ((!visited[i]) && (G[start][i]!=INF))
            DFS(i);
}
调用:
memset(visited,0,sizeof(visited));
for (int i=0;i<n;i++)
    if (!visited[i]) DFS(i);
```

城堡问题



如图是一个城堡的地形图。请你编写一个程序，计算城堡一共有多少房间，最大的房间有多大。城堡被分割成m×n(m≤50， n≤50)个方块，每个方块可以有0~4面墙。

输入输出

输入

程序从标准输入中读入数据。

第一行是两个整数，分别是南北向、东西向的方块数。

在接下来的输入行里，每个方块用一个数字(0≤p≤50)描述。用一个数字表示方块周围的墙，1表示西墙，2表示北墙，4表示东墙，8表示南墙。每个方块用代表其周围墙的数字之和表示。城堡的内墙被计算两次，方块(1,1)的南墙同时也是方块(2,1)的北墙。

输入的数据保证城堡至少有两个房间。

输出

城堡的房间数、城堡中最大房间所包括的方块数。

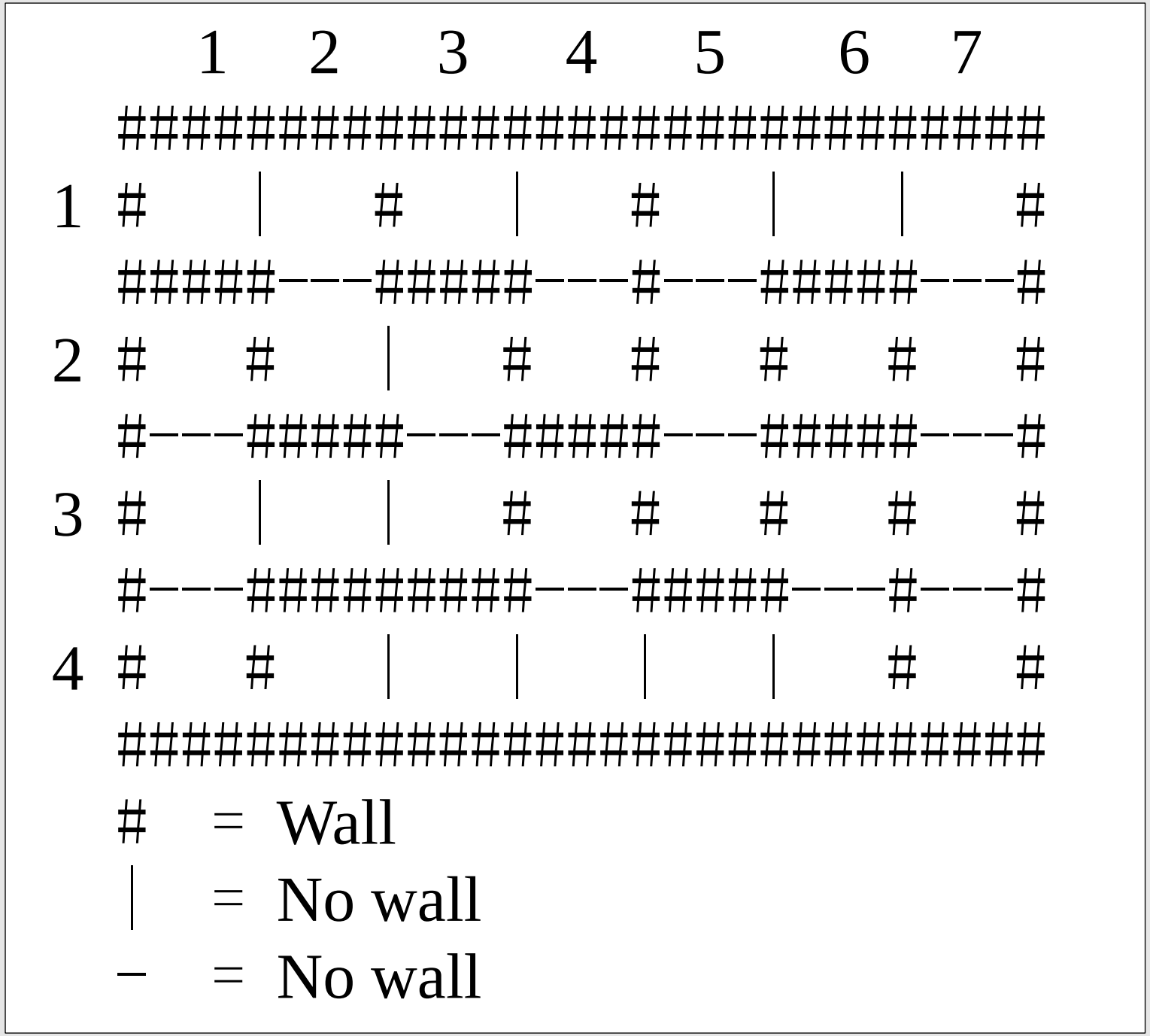
结果显示在标准输出上。

样例输入

```
4
7
11 6 11 6 3 10 6
7 9 6 13 5 15 5
1 10 12 7 13 7 5
13 11 10 8 10 12 13
```

样例输出

```
5 9
```



把方块看作是节点，相邻两个方块之间如果没有墙，则在方块之间连一条边，这样城堡就能转换成一个图。

求房间个数，实际上就是在求图中有多少个极大连通子图。

对每一个房间，深度优先搜索，从而给这个房间能够到达的所有位置染色。最后统计一共用了几种颜色，以及每种颜色的数量。

```cpp
#include <iostream>
#include <stack>
#include <cstring>
using namespace std;
int R,C;  //行列数
int rooms[60][60];
int color[60][60]; //方块是否染色过的标记
int maxRoomArea = 0, roomNum = 0;
int roomArea;

   void Dfs(int i,int k) {
       if( color[i][k] )
           return;
       ++ roomArea;
       color [i][k] = roomNum;
       if( (rooms[i][k] & 1) == 0 )
          Dfs(i,k-1);   //向西走
       if( (rooms[i][k] & 2) == 0 )
          Dfs(i-1,k);   //向北
       if( (rooms[i][k] & 4) == 0 )
          Dfs(i,k+1);  //向东
       if( (rooms[i][k] & 8) == 0 )
          Dfs(i+1,k);  //向南
    }

int main()  {
    cin >> R >> C;
    for( int i = 1;i <= R;++i)
        for ( int k = 1;k <= C; ++k)
            cin >> rooms[i][k];
    memset(color,0,sizeof(color));
    for( int i = 1;i <= R; ++i)
        for( int k = 1; k <= C; ++ k) {
            if( !color[i][k] ) {
                ++ roomNum ;   roomArea = 0;
                Dfs(i,k);
                maxRoomArea = max(roomArea,maxRoomArea);
            }
        }
    cout << roomNum << endl;
    cout << maxRoomArea << endl;
    return 0;
}
```

最短路径问题

每两点间最短路问题（APSP）

Floyd-Warshall

Floyd-Warshall算法是动态规划算法。

状态表示： $f(i,j)$ 表示从点i到点j的最短距离。

状态转移方程： $f(i,j)=\min\{f(i,k)+f(k,j)\}$ （i到k、k到j都是连通的）

时间复杂度： $O(n^3)$

也被称为Floyd算法, Roy–Warshall 算法, Roy–Floyd 算法或是 WFI 算法

```
const int INF = 0x3f3f3f3f;
int f[N][N], prev[N][N];
// 追踪prev可以得到最短路。
int len=INF;
// 最小环的长度
```

```
for (int i=0; i<k; i++)
    for (int j=i+1; j<k; j++)
        len = min(len, G[i][j]+f[i][k]+f[k][j]);
// G是无向图！len为最小环的和
```

```
void Floyd(){
    // 初始化——可以在读图时完成
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            f[i][j] = G[i][j];

    memset(prev, -1, sizeof(prev));
    /* len=INF; */

    // 计算。注意，k在最外面。
    for (int k=0; k<n; k++) {
        /* 如果求最小环，请将代码插入到这里。 */
        for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                if (f[i][k] + f[k][j] < f[i][j]){
                    f[i][j] = f[i][k] + f[k][j];
                    prev[i][j] = k;
                }
    }
}
// cout<<f[i][j]<<endl;
// 通过递归调用追踪prev，就可以得到最短路径上的结点。
```

单源最短路问题（SSSP）

从start到点i的距离为d[i]。如果d[i]==INF，说明start和i不连通。

Dijkstra

Dijkstra算法是贪心算法。

它只适用于**所有边的权都大于0的图。**

基本思想是：设置一个顶点的集合S，并不断地扩充这个集合，当且仅当从源点到某个点的路径已求出时它才属于集合S。

开始时S中仅有源点，调整非S中点的最短路径长度，找当前最短路径点，将其加入到集合S，直到所有的点都在S中。

时间复杂度：$O(n^2)$

邻接矩阵

```
bool visited[N]; // 是否被标号
int d[N];   // 从起点到某点的最短路径长度
int prev[N];
// 通过追踪prev可以得到具体的最短路径（注意这里是逆序的）
```

```
void Dijkstra(int start){
    // 初始化：d[start]=0，且所有点都未被标号
    memset(visited, 0, sizeof(visited));
    for (int i=0; i<n; i++) d[i]=INF;
    d[start]=0;
    for (int i=0; i<n; i++) {// 计算n次
        int x, min=INF;
        // 在所有未标号的结点中，选择一个d值最小的点x。
        for (int a=0; a<n; a++)
            if (!visited[a] && d[a]<min) min=d[x=a];
        visited[x]=true; // 标记这个点x。
        // 对于从x出发的所有边（x，y），更新一下d[y]。
        for (int y=0; y<n; y++)
            if (d[y] > d[x]+G[x][y]){
                d[y] = d[x]+G[x][y];
                prev[y] = x;    // y这个最短路径是从x走到y。
            }
    }
}
```

使用优先队列、邻接表

朴素的Dijkstra算法在选d值最小的点时要浪费很多时间，所以可以用优先队列（最小堆）来优化。
时间复杂度：$O[(n+m)\log m]$，最差情况（密集图）为$O(n^2 \log m)$

```
#include <queue>
#include <vector>
#include <utility>
typedef pair<int, int> pii;
// 将终点和最短路径长度“捆绑”的类型
priority_queue < pii,
       vector<pii>, greater<pii> > q;
// 定义一个优先队列，d值最小的先出列
int d[N], prev[N];
```

```
void Dijkstra(int start){
    for (int i=0; i<n; i++) d[i]=INF;
    d[start]=0;
    q.push (make_pair(d[start], start));
    while (!q.empty()){
        // 在所有未标号的结点中，选择一个d值最小的点x。
        pii u=q.top(); q.pop();
        int x=u.second;

        if (u.first!=d[x]) continue;   // 已经计算完

        for (edge *e=adj[x]; e!=NULL; e=e->next){
            int &v=e->v, &w=e->w;
            if (d[v] > d[x]+ w) {
                d[v] = d[x]+ w;          // 松弛
                prev[v]=x;
                q.push(make_pair(d[v], v));
            }
        }
    }
}
```

Bellman-Ford

Bellman-Ford算法是迭代法，它不停地调整图中的顶点值（源点到该点的最短路径值），直到没有点的值调整了为止。

该算法除了能计算最短路，还可以检查负环（一个每条边的权都小于0的环）。如果图中有负环，那么这个图不存在最短路。

```cpp
bool Ford(int start){  // 有负环则返回false
    // 初始化
    for (int i=0; i<n; i++) d[i]=INF;
    d[start]=0;
    for (int k=0;k<n-1;k++)      // 迭代n次
        for (int i=0; i<m; i++) {   // 检查每条边
            int &x=u[i], &y=v[i];
            if (d[x]<INF) d[y]=min(d[y],d[x]+w[i]);
        }
    // 下面的代码用于检查负环——如果全部松弛之后还能松弛，说明一定有负环
    for (int i=0; i<m; i++){    // 再次检查每条边
        int &x=u[i], &y=v[i];
        if (d[y]>d[x]+w[i]) return false;
    }
    return true;
}
```

SPFA

SPFA是使用队列实现的Bellman-Ford算法。操作步骤如下：

① 初始队列和标记数组。

② 源点入队。

③ 对队首点出发的所有边进行松弛操作（即更新最小值）。

④ 将不在队列中的尾结点入队。

⑤ 队首点更新完其所有的边后出队。

时间复杂度：O(|V||E|)

```cpp
bool SPFA(int start) {// 有负环则返回false
    for (int i=0; i<n; i++) d[i]=INF; // 初始队列和标记数组
    d[start]=0;
    memset(cnt,0,sizeof(cnt));
    q.push(start);  // 源点入队
    cnt[start]++;
    while (!q.empty()) {
        int x=q.front(); q.pop();
        inqueue[x]=false;
        for (edge *e=adj[x];e!=NULL;e=e->next) {// 对队首点出发的所有边进行松弛操作（即更新最小值）
            int &v=e->v, &w=e->w;
            if (d[v]>d[x]+w) {
                d[v] = d[x]+w;
                if (!inqueue[v]) {// 将不在队列中的尾结点入队
                    inqueue[v]=true;
                    q.push(v);
                    if (++cnt[v]>n) return false;// 有负环
                }
            }
        }
    }
    return true;
}
```

```cpp
queue<int> q;
bool inqueue[N]; // 是否在队列中
int cnt[N];
// 检查负环时使用：结点进队次数。如果超过n说明有负环。
```

让我们回到

每两点间最短路问题（APSP）

为什么要回到？
Floyd-WarShall算法太慢了

优化方法1
（对于无负权边的图）

每个点跑一次**堆优化的**Dijkstra算法

优化方法2
（对于无负环的图）

每个点跑一次Bellman-Ford算法

每个点跑一次SPFA算法

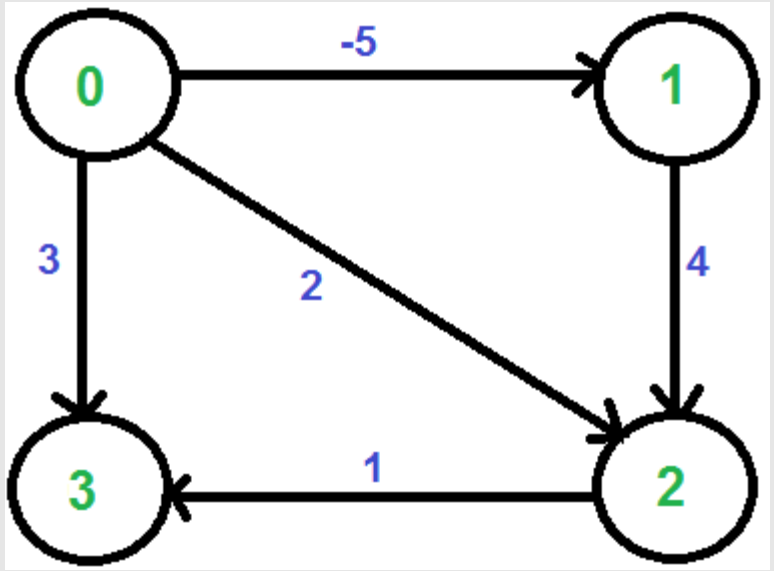
Johnson算法

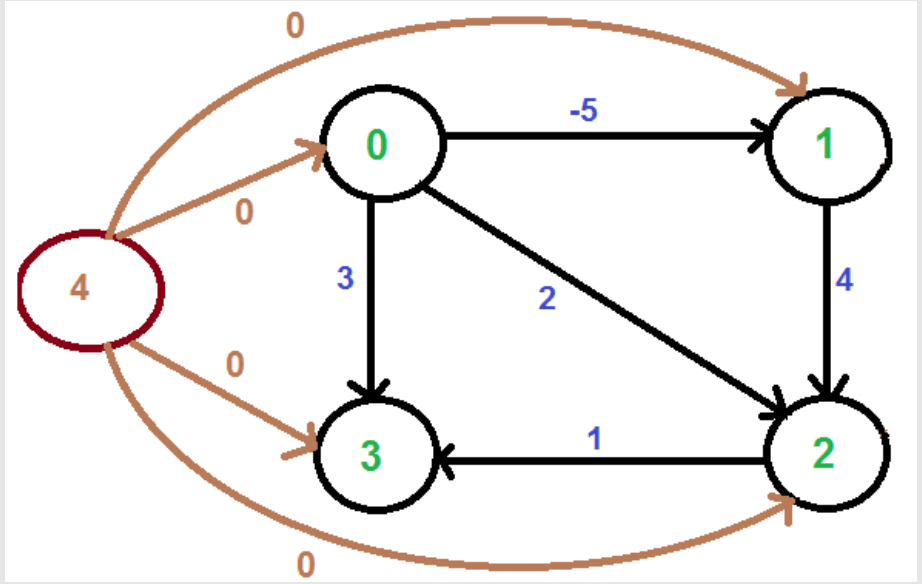
Johnson算法

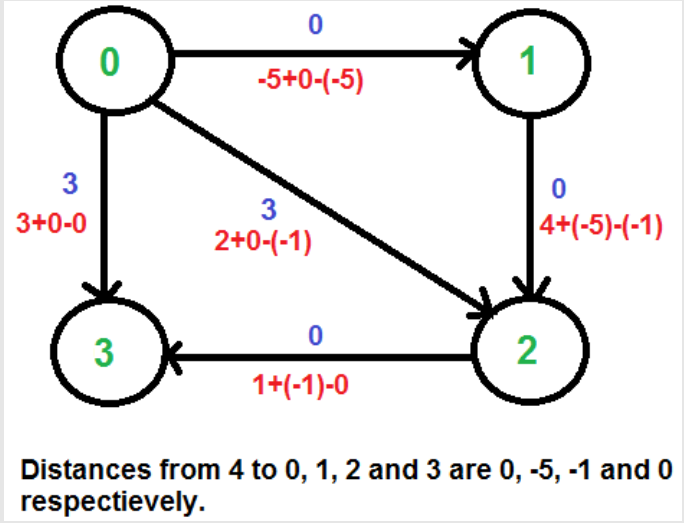
该算法在 1977 年由 Donald B. Johnson 提出。

$O(V^2 \log V + VE)$。

1. 给定图 G = (V, E)，增加一个新的顶点 s，使 s 指向图 G 中的所有顶点都建立连接，设新的图为 G’；
2. 对图 G’ 中顶点 s 使用 Bellman-Ford 算法计算单源最短路径，得到结果 h[] = {h[0], h[1], .. h[V-1]}；
3. 对原图 G 中的所有边进行 "re-weight"，即对于每个边 (u, v)，其新的权值为 w(u, v) + (h[u] - h[v])；
4. 移除新增的顶点 s，对每个顶点运行 Dijkstra 算法求得最短路径；







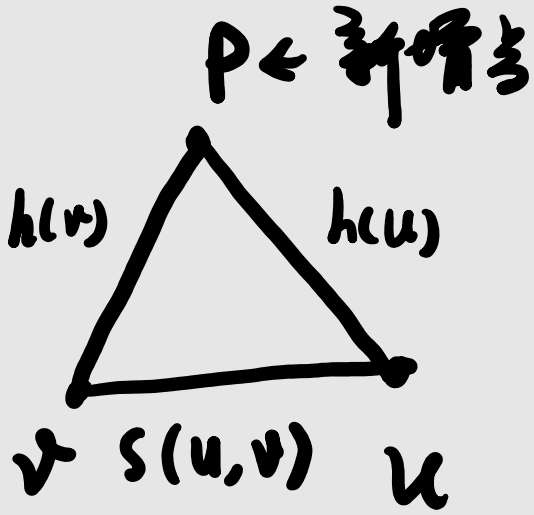
证明1：增加后，最短路径的值不会发生**不可复原的**改变

对于一个图G = (V, E)，其中的一条路径s(u,v)

$$s(u,v) = d(u) + h[0] - h[u] + d(0) + h[1] - h[0] \dots d[v] + h[v] - h[n]$$

$$s(u,v) = d(u) + d(0) + \dots + d(v) + h[v] - h[u]$$

证明2：增加后每一边都变成**非负的**



$$s(u,v) + h(v) \geq h(u)$$
$$s(u,v) + \big(h(v) - h(u)\big) \geq 0$$

与树相关的内容

无根树

对于n个节点的无向图，如果满足以下条件中的任意一条，就称为无根树：

- 连通，且恰好有n-1条边
- 连通，且没有环
- 任意两点之间恰好有一条简单路径

有根树

有一个节点为根
除根以外，每个节点有一个父节点

所有节点都能通过父亲的关系走到根
父节点不会形成环

树都是图

无根树→无向图
有根树→有向图
　父亲指向孩子，或者孩子指向父亲

所有关于图的算法都能在树上直接用！

无根树转有根树

选取一个点作为根

- 一般情况下可以任取一个点，例如1号点
- 但有些题目可能选取某些特殊的点更加方便

从根开始DFS/BFS，即可得出其他所有点的父亲

树上动态规划

如果是无根树，先转为有根树

需要按照自底向上或者自顶向下的顺序来处理

一般情况下在DFS/BFS的时候顺便处理就可以了

- 通常**DFS**更好写

如果对于任何一个度为d的点，由它所有孩子的答案计算它的答案需要O(d T)的时间，那么总的时间就是O(n T)