1. 简单桶排序：用数组存储元素出现的次数，a[i]的值表示元素i出现的次数。

假设桶的数目为M，需要排序的元素有N个，那么算法时间复杂度为O(M+N)。

int main()

{

int n, num, book[1001] = {0};

scanf("%d", &n);

for(int i = 1; i <= n; i++)

{

scanf("%d", &num);

book[num]++;

}

for(int i = 0; i <= 1000; i++)

{

while(book[i] > 0)

{

printf("%d ", i);

book[i]--;

}

}

return 0;

}

1. 冒泡排序

基本思想：每次比较两个相邻的元素，如果它们的顺序错误就交换这两个元素。

时间复杂度O（N^2）

核心

for(int i = 1; i <= n-1; i++) //轮数

for(int j = 0; j < n-i; j++) //每1轮从第1个数到第n-i开始比较，下标0~n-i

{

if(a[j] > a[j+1]) //从小到大排序

{

int tmp = a[j];

a[j] = a[j+1];

a[j+1] = tmp;

}

}

1. 快速排序

基本思想：选择一个基准值（假设是第一个元素），从数组两端开始探测，从左往右探测比基准值大的元素（假设从小到大排序），从右往左探测比基准值小的元素，交换两个元素，直至两个探测路标相遇，交换左标和基准值的位置，完成基准值的归位。经过一轮处理，基准值左边都是小于基准值的数，右边都是大于基准值的数。对下面的子数组进行相同处理，类似二分。

平均时间复杂度O（NlogN），最差O(N^2)

void quicksort(int left, int right, int a[])

{

if(left > right)

return;

int srd = a[left]; //选择数组第一个数为基准值

int i = left, j = right;

while(i != j)

{ //顺序很重要，要先从右往左找

while(a[j] >= srd && i < j)c

j--;

//再从左往右

while(a[i] <= srd && i < j)

i++;

if(i < j)

{

//交换两个数

int tmp = a[i];

a[i] = a[j];

a[j] = tmp;

}

}

//基准值归位

a[left] = a[i];

a[i] = srd;

//对子数组进行相同的处理，这是个递归的过程

quicksort(left, i-1, a);

quicksort(i+1, right, a);

};

第二章：栈、队列、链表

1、队列（先进先出）

const int N = 1000+10;

struct Queue

{

int data[N]; //队列的主体，用来存储内容

int head; //队首

int tail; //队尾

};

head用来记录队列的队首（即第一位），tail用来记录队列的末尾（即最后一位）的下一个位置。

在队首删除一个元素的操作是head++，在队尾增加一个数（假设这个数是x）的操作是q[tail]=x; tail++，当队列中没有元素时，head = tail，表示队列为空。

2、栈（先进后出）

栈的实现只需要一个一维数组和一个指向栈顶的变量top就可以了，通过top来对栈进行插入和删除操作。

初始化栈：top = 0；

入栈，假设需要入栈的数是x：s[top] = x; top++;

6

2 4 1 2 5 6

1. 1 3 5 6 4

3、链表

每个结点都由两个部分组成，左边的部分用来存放具体的数值，右边的部分需要存储下一个结点的地址，可以用指针来实现（也成为后继指针）。

定义结构体：

struct node

{

int data;

node \*next;

};

这个结构体有两个成员，第一个成员是整型data，用来存储具体的数值；第二个成员是一个指针，用来存储下一个结点的地址。因为下一个结点的类型也是node。

构建链表：

for(int i = 0; i < n; i++)

{

scanf("%d", &num);

p = (node\*)malloc(sizeof(node));

p->data = num;

p->next = NULL;

if(head == NULL)

head = p;

else

q->next = p;

q = p;

}

遍历：

while(t != NULL)

{

printf("%d ", t->data);

t = t->next;

}

插入一个元素：

while(t != NULL)

{

if(t->next == NULL || t->next->data > num)

{

p = (node\*)malloc(sizeof(node));

p->data = num;

p->next = t->next;

t->next = p;

break;

}

t = t->next;

}

1、全排列

void dfs(int step)

{

if(step == n)

{

for(int i = 0; i < n; i++)

printf("%d ", a[i]);

printf("\n");

return;

}

for(int i = 1; i <= n; i++)

{

if(book[i] == 0)

{

a[step] = i;

book[i] = 1;

dfs(step+1);

book[i] = 0;

}

}

}

void dfs(int step)

{

判断边界

尝试每一种可能for(int i = 1; i <= n; i++)

{

继续下一步dfs(step+1);

}

返回

}

每一种尝试都是一种“扩展”

2、走迷宫（dfs）

使用递归，思路：从起始点出发，一直走，直至不能继续走下去，再去尝试别的路径。

void dfs(int x, int y, int step)

{

判断是否达到终止条件

尝试每一种可能for(int i = 0; i < 4; i++) //几个能走的方向

{

判断是否越界

判断是是否为障碍物或者已经在路径中

继续走下一步

}

返回

}

1. 走迷宫（层层递进-bfs）

使用队列结构，思路：从起始点出发，将通过该点能够到达的结点加入队列，扩展结点，然后对下一个结点（head++）进行处理，直至能够扩展到目标结点。

结构体

struct node //队列

{

int x; //横坐标

int y; //纵坐标

int s; //步数

};

最短路径

1、多源最短路Floyd-Warshall

思路：如果要让任意两点i、j之间的距离变短，只能引入第三个点k，通过这个顶点k中转即i->k->j，才可能缩短i到j的路程。Floyd算法就是每次增加一个允许通过的中转点，来求所有顶点的最短距离。

核心代码：

for(int k = 1; k <= n; k++)

for(int i = 1; i <= n; i++)

for(int j = 1; j <= n; j++)

if(a[i][k] < INF && a[k][j] < INF && a[i][k]+a[k][j] < a[i][j])

a[i][j] = a[i][k] + a[k][j];

这段代码的基本思想是：最开始只允许经过1号顶点进行中转，接下来只允许经过1号和2号顶点进行中转……允许经过1~n号所有顶点进行中转，求任意两点之间的最短距离。

时间复杂度O(N^3)，可以处理有负权边的图，不能处理负权回路。

2、单源最短路径 Dijkstra

用来求指定一个点（源点）到其他顶点的距离。

算法基本思想：每次找到离源点最近的一个顶点，然后以该顶点为中心进行扩展，最终得到源点到其余所有点的路径。步骤如下：

1. 将所有的顶点分为两部分：已知最短路径的顶点集合P和未知最短路径的顶点集合Q。最开始，已知最短路径的顶点集合P中只有源点一个顶点。用一个book数组来记录哪些点在集合P中，为1表示在P中，为0表示在Q中。
2. 设置源点s到自己的最短路径为0，即dis[s] = 0，若存在有源点能直接到达的顶点i，则把dis[i]设为e[s][i]，同时把所有其他（源点不能直接到达的）顶点的最短路径设为∞。
3. 在集合Q的所有顶点中选择一个离源点s最近的顶点u（即dis[u]最小）加入到集合P，并考察所有以点u为起点的边，对每一条边进行松弛操作。

例如，存在一条从u到v的边，那么可以通过将边u->v添加到尾部来拓展一条从s到v的路径，这条路径的长度是dis[u]+e[u][v]。如果这个值比目前已知的dis[v]的值要小，就可以更新dis[v]=dis[u]+e[u][v]

1. 重复第3步，如果集合Q为空，算法结束，最终dis数组中的值就是源点到所有顶点的最短路径。

算法时间复杂度O(N^2)，找最近的那个顶点的时间复杂度O(N)，使用堆优化可以把这部分降到O(logN)，不能处理负权边，使用邻接表存储可以优化。

void dijkstra()

{

//需要把集合Q中的n-1个顶点拿到P中

for(int i = 1; i <= n-1; i++)

{

int minx = INF;

int u;

for(int j = 1; j <= n; j++)

{

if(book[j] == 0 && dis[j] < minx)

{

u = j;

minx = dis[j];

}

}

book[u] = 1;

for(int v = 1; v <= n; v++)

{

if(dis[u] + a[u][v] < dis[v])

dis[v] = dis[u] + a[u][v];

}

}

}

1. 解决负权边-Bellman-Ford

核心代码只有四行，并可以完美解决带有负权边的图，还可以用来检测是否有负权回路

for(int k = 1; k <= n-1; k++) //进行n-1轮松弛

for(int i = 1; i <= m; i++) //枚举每一条边

if(dis[u[i]] != INF && w[i] != INF && dis[u[i]] + w[i] < dis[v[i]]) //尝试对每一条边进行松弛

dis[v[i]] = dis[u[i]] + w[i];

最多进行n-1轮松弛。因为在一个含有n个顶点的图中，任意两点之间的最短路最多包含n-1条边。

时间复杂度O(NM)

Bellman-Ford算法还可以用来检测一个图是否含有负权回路，如果在进行n-1轮松弛之后，仍然存在

if(dis[u[i]] != INF && w[i] != INF && dis[u[i]] + w[i] < dis[v[i]]) //尝试对每一条边进行松弛

dis[v[i]] = dis[u[i]] + w[i];

的情况，说明在n-1轮松弛之后仍然可以继续松弛成功，那么此图必然存在负权回路。检测代码如下：

for(int k = 1; k <= n-1; k++)

for(int i = 1; i <= m; i++)

if(dis[u[i]] + w[i] < dis[v[i]])

dis[v[i]] = dis[u[i]] + w[i];

int flag = 0;

for(int i = 1; i <= m; i++)

{

if(dis[u[i]] + w[i] < dis[v[i]])

{

flag = 1;

break;

}

}

if(flag)

printf("此图含有负权回路\n");

优化：如果已经松弛完毕，提前跳出循环，使用check变量来标记本轮是否发生了变化

for(int k = 1; k <= n-1; k++)

{

int check = 0;

for(int i = 1; i <= m; i++)

{

if(dis[u[i]] + w[i] < dis[v[i]])

{

dis[v[i]] = dis[u[i]] + w[i];

check = 1;

}

}

if(!check)

break;

}

1. Bellman-Ford的队列优化

在每实施一次松弛操作后，就会有一些顶点已经求得其最短路，此后这些顶点的最短路的估计值就会一直保持不变，不再受后续松弛操作的影响，但是上面代码还会继续判断是否需要松弛，这里浪费了时间。

这启发：每次仅对最短路估计值发生变化的顶点的所有出边执行松弛操作。

思路：每次选取队首顶点u，对顶点u的所有出边进行松弛操作，例如有一条u->v的边，如果dis[u]+e[u][v]<dis[v]，且顶点v不在当前队列中，就将顶点v放入队尾。需要注意的是，同一个顶点同时在队列中出现多次毫无意义，所以需要一个数组来判重（判断哪些顶点已经在队列中）。在对顶点u的所有边松弛完毕后，就将顶点u出队。家下来不断从队列中取出新的队首顶点再进行如上操作，直至队列为空。

借助队列实现，代码如下（这里使用数组实现的邻接表）：

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <cstring>

using namespace std;

const int N = 51;

const int INF = 99;

int u[N], v[N], w[N], book[N];

int head, tail, dis[N], n, m;

int Queue[N], next[N], first[N];

void bellman\_ford\_()

{

while(head < tail)

{

int t = Queue[head];

int k = first[t];

while(k != -1)

{

if(dis[u[k]] + w[k] < dis[v[k]])

{

dis[v[k]] = dis[t] + w[k];

if(book[v[k]] == 0)

{

Queue[tail] = v[k];

tail++;

book[v[k]] = 1;

}

}

k = next[k];

}

book[Queue[head]] = 0;

head++;

}

}

int main()

{

memset(book, 0, sizeof(book));

scanf("%d%d", &n, &m);

for(int i = 1; i <= n; i++)

{

dis[i] = INF;

first[i] = -1;

}

for(int i = 1; i <= m; i++)

{

scanf("%d%d%d", &u[i], &v[i], &w[i]);

next[i] = first[u[i]];

first[u[i]] = i;

}

dis[1] = 0;

head = tail = 0;

Queue[tail++] = 1;

book[1] = 1;

bellman\_ford\_();

for(int i = 1; i <= n; i++)

printf("%d ", dis[i]);

return 0;

}

/\*

5 7

1 2 2

1 5 10

2 3 3

2 5 7

3 4 4

4 5 5

5 3 6

\*/

需要说明：

1. book[Queue[head]] = 0;通常一个顶点出队后不会再重新进入队列，而这里一个顶点很可能在出队列之后再次放入队列，也就是当一个顶点的最短路程估计值变小之后，需要对其所有的出边进行松弛，但是如果这个顶点的最短路程估计值再次变小，仍需要对其所有的出边再次进行松弛，这样才能保证相邻顶点的最短路程估计值同步更新。
2. Bellman-Ford队列优化如何判断一个图是否有负环？

某个点进入队列的次数超过n次。

树

1、树之旅

树是指任意两个结点间有且只有一条路径的无向图（只要是没有回路的连通无向图就是树）。

深度是从根到这个结点的层数（根为第一层）

2、二叉树

二叉树是一种特殊的树，每个结点最多有两个儿子。

更严格的递归定义：二叉树要么为空，要么由根节点、左子树和右子树组成，而左子树和右子树分别是一棵二叉树。

满二叉树：如果二叉树的每个节点都有两个儿子，这样的树叫满二叉树（深度为h且结点数为2^h-1的二叉树）。

完全二叉树：若设二叉树的高度为h，除第h层外，其他各层（1~h-1）的结点数都达到最大个数，第h层从右向左连续缺若干结点，则这个二叉树就是完全二叉树。

将完全二叉树从上到下、从左到右进行编号，如果完全二叉树的一个父节点编号为k，那么左儿子编号2k，右儿子编号2k+1。如果已知儿子（左二子或右儿子）的编号是x，那么父节点的编号就是x/2。如果一棵完全二叉树有N个结点，那么它的高度就是logN（底数为2）。

二叉树最典型应用就是堆。

3、堆

最小堆：所有子节点比父节点大。

最大堆：所有子节点比父节点小。

场景：删除数组中最小的数，并增加一个数

解决方法：建立最小堆，删除堆顶元素，将新增加的数放置到堆顶，如果此时不符合最小堆的特性，则需要将这个数向下调整，直到找到合适的位置为止，使其重新符合最小堆的特性。

时间复杂度O(N)

场景：只新增一个数，如何在原有的堆上直接插入一个元素呢？

解决方法：直接将新元素插入到末尾，再根据情况判断新元素是否需要上移，如果堆的大小为N，则插入一个新元素所需要的时间为O(logN)。

如何建立这个堆？

可以从空的堆开始，然后依次往堆中插入每一个元素，直到所有数都被插入（转移到堆中）为止。每次插入一个元素所用的时间是O(logi)，插入所有元素的整体时间复杂度是O(NlogN)。

如何更快地建立这个堆？

直接把所有元素放入一个完全二叉树中（用一个一维数组来存储完全二叉树），在这棵完全二叉树中，从最后一个结点开始，依次判断以这个结点为根的子树是否符合最小堆的特性。如果所有的子树都符合最小堆特性，那么整棵树就是最小堆了。

小结：把n个元素建立一个堆。首先可以将这n个结点以自顶向下、从左到右的方式从1到n编码，这样就可以把这n个结点转化成一棵完全二叉树。紧接着从最后一个非叶结点（结点编号n/2）开始到根节点（结点编号为1），逐个扫描所有的结点，根据需要将当前节点向下调整，直到以当前节点为根结点的子树符合堆的特性。

时间复杂度O(logN)

堆排序

时间复杂度O(NlogN)，

思路：如果现在要进行从小到大排序，可以先建立最小堆，然后每次删除顶部元素并将顶部元素输出或者放入一个新的数组中，直到堆为空为止。最终输出的或者存放在新数组中的数就是排好序的了。

堆排序的另外一种思路：从小到大排序的时候不建立最小堆而是建立最大堆，最大堆建立好后，最大的元素在h[1]，因为需求是从小到大排序，希望最大的放在最后，因此将h[1]和h[n]进行交换，此时h[n]就是数组中的最大元素。OK，最大的元素归位后，将堆的大小减1，并将交换后的h[1]向下调整以保持堆的特性。如此反复，直到堆的大小变成1为止。此时，数组h中的数就已经是排好序的了。

小结：

1. Dijkstra算法中每次找离源点最近的一个顶点也可以用堆来优化，使算法时间复杂度降到O((M+N)logN)
2. 求第K大的数，只需要建立一个大小为K的最小堆，堆顶就是第K大的数（举个例子，假设有10个数，要求第3大的数。第一步选取任意三个数比如前三个，将这三个数建成最小堆，然后从第4个数开始，与堆顶的元素比较，如果比堆顶小就不要，如果比他大，就舍弃当前堆顶将这个新的数作为堆顶，并再去维护堆，用同样的方法处理第5~10个数）

代码尝试：

//求第K大，建立大小为K的最小堆

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <cstring>

using namespace std;

const int N = 110;

int h[N];

int n, k;

void swap(int i, int j)

{

int t = h[i];

h[i] = h[j];

h[j] = t;

}

void siftdown(int i)

{

int flag = 0, t;

while(i\*2 <= k && flag == 0)

{

if(h[i] > h[i\*2])

t = i\*2;

else

t = i;

if(i\*2 + 1 <= n)

{

if(h[t] < h[i\*2+1])

t = i\*2+1;

}

if(t != i)

{

swap(i, t);

i = t;

}

else

flag = 1;

}

return;

}

void setup()

{

for(int i = k/2; k >= 1; k--)

siftdown(i);

}

int main()

{

scanf("%d%d", &n, &k);

int tmp;

for(int i = 1; i <= n; i++)

{

scanf("%d", &tmp);

if(i <= k)

siftdown(i);

else

{

if(tmp > h[1])

{

h[1] = tmp;

siftdown(1);

}

else

continue;

}

}

printf("%d", h[1]);

return 0;

}

1. 求第K小的数，建立大小为K的最大堆
2. 求前K大和前K小的数

并查集

也称为不相交集数据结构，

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <cstring>

#include <cmath>

using namespace std;

const int N = 110;

int n, m, a[N];

//找爹函数

int getpapa(int t)

{

if(a[t] == t)

return t;

else

{//路径压缩，每次在函数返回的时候，顺带把路上遇到的人的“boss“改为最后找到的祖宗编号

a[t] = getpapa(a[t]);

return a[t];

}

}

int main()

{

scanf("%d%d", &n, &m);

for(int i = 1; i <= n; i++)

a[i] = i;

int u, v;

for(int i = 1; i <= m; i++)

{

scanf("%d%d", &u, &v);

int upa = getpapa(u);

int vpa = getpapa(v);

if(upa != vpa) //向左边归顺

{

a[vpa] = a[upa];

}

}

int cnt = 0;

for(int i = 1; i <= n; i++)

{

printf("%d-%d\n", i, a[i]);

if(i == a[i])

cnt++;

}

printf("%d\n", cnt);

}

/\*

10 9

1 2

3 4

5 2

4 6

2 6

8 7

9 7

1 6

2 4

\*/

最后

1、图的最小生成树 Kruskal

让边的总长度之和最短，首先选择最短的边，其次选择次短的……，直到选择了n-1条边为止。

难实现的是：判断两个顶点是否已经连通。（可以使用广搜或者深搜，但这样效率低）可以使用已经学习的并查集，把所有的顶点放到并查集中，判断两个顶点是否连通，只需判断两个顶点是否在同一个集合（即是否有共同的祖先）即可，这样时间复杂度为O(logN)。

Kruskal：首先按照边的权值进行从小到大排序，每次从剩余的边中选择权值较小且边的两个顶点不在同一个集合的边（就是不会产生回路的边），加入到生成树中，直到加入了n-1条边为止。

时间复杂度分析：对顶点进行快排的时间复杂度是O(logN)，对边进行快排的时间复杂度是O(MlogM)，在m条边中找到n-1条边的时间复杂度是O(MlogN)，所以Kruskal算法的时间复杂度是O(MlogM+MlogN)，因为M通常比N大，所以最终时间复杂度是O(MlogM)。

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <cmath>

using namespace std;

struct edge

{

int u;

int v;

int w;

};

const int N = 110;

int n, m, a[N];

edge node[N];

void init()

{

memset(a, 0, sizeof(a));

for(int i = 1; i <= n; i++)

a[i] = i;

}

int getfather(int i)

{

if(a[i] == i)

return i;

else

{

a[i] = getfather(a[i]);

return a[i];

}

}

int mergeuv(int u, int v)

{

int ufa = getfather(u);

int vfa = getfather(v);

if(ufa != vfa)

{

a[vfa] = a[ufa];

return 0;

}

return 1;

}

void quicksort(int left, int right)

{

if(left >= right)

return;

edge temp = node[left];

int i = left;

int j = right;

while(i != j)

{

while(node[j].w >= node[left].w && i < j)

j--;

while(node[i].w <= node[left].w && i < j)

i++;

if(i < j)

{

edge tmp = node[i];

node[i] = node[j];

node[j] = tmp;

}

}

node[left] = node[i];

node[i] = temp;

quicksort(left, i-1);

quicksort(i+1, right);

return;

}

int main()

{

scanf("%d%d", &n, &m);

init();

for(int i = 1; i <= m; i++)

{

scanf("%d%d%d", &node[i].u, &node[i].v, &node[i].w);

}

quicksort(1, m);

// for(int i = 1; i <= m; i++)

// {

// printf("%d %d %d\n", node[i].u, node[i].v, node[i].w);

// }

int cnt = 0;

int sum = 0;

for(int i = 1; i <= m; i++)

{

int t = mergeuv(node[i].u, node[i].v);

if(t == 0 && cnt != n-1)

{

cnt++;

sum += node[i].w;

printf("%d %d\n", node[i].u, node[i].v);

}

else

continue;

}

printf("%d", sum);

}

/\*

6 9

2 4 11

3 5 13

4 6 3

5 6 4

2 3 6

4 5 7

1 2 1

3 4 9

1 3 2

\*/

2、再谈-图的最小生成树 Prim

算法流程：

1. 从任意一个顶点开始构造生成树，假设从1号开始。首先将顶点1号加入生成树中，用一个一维数组book来标记哪些顶点已经加入了生成树。
2. 用数组dis记录生成树到各个顶点的距离。最初生成树中只有1号顶点，有直连边的时候，数组dis存储的就是1号顶点到该顶点的边的权值，没有直连边的时候就是无穷大，即初始化数组。
3. 从数组中选择离生成树最近的顶点（假设这个顶点为j）加入到生成树中（即在dis数组中找最小值）。再以j为中间点，更新生成树到每个非树顶点的距离（松弛操作），即如果dis[k] > e[j][k]则更新dis[k]=e[j][k]。
4. 重复第三步，直到生成树中有n个顶点为止。

时间复杂度O(N^2)

#include <iostream>

#include <cstdio>

#include <cstdlib>

#include <cstring>

#include <cmath>

using namespace std;

const int N = 110;

int n, m;

const int INF = 99999;

int a[N][N], book[N];

int dis[N];

int main()

{

scanf("%d%d", &n, &m);

int u, v, w;

memset(book, 0, sizeof(book));

for(int i = 1; i <= n; i++)

dis[i] = INF;

for(int i = 1; i <= n; i++)

{

for(int j = 1; j <= n; j++)

{

a[i][j] = INF;

}

}

for(int i = 1; i <= m; i++)

{

scanf("%d%d%d", &u, &v, &w);

a[u][v] = w;

a[v][u] = w;

a[i][i] = 0;

}

for(int i = 1; i <= n; i++)

dis[i] = a[1][i];

book[1] = 1;

//Prim

int sum = 0;

for(int k = 1; k <= n-1; k++)

{

int t, minx = INF;

for(int i = 1; i <= n; i++)

{

if(book[i] == 0 && dis[i] < minx)

{

minx = dis[i];

t = i;

}

}

book[t] = 1;

sum += dis[t];

for(int i = 1; i <= n; i++)

{

if(a[t][i] < dis[i])

{

dis[i] = a[t][i];

}

}

}

printf("%d\n", sum);

return 0;

}

/\*

6 9

2 4 11

3 5 13

4 6 3

5 6 4

2 3 6

4 5 7

1 2 1

3 4 9

1 3 2

\*/

采用邻接表，最小堆，可以降低复杂度。

2、图的割点

抽象问题：在一个无向连通图中，如果删除某个顶点后，图不再连通（即任意两点之间不能互相到达），称这样的顶点为割点（割顶）。

如何求割点？

这里提供了一种时间戳的思路，顶点v在不经过父节点u的情况下不能回到祖先。num[i]表示当前顶点i的时间戳（即深搜的次序），low[i]表示顶点i在不经过父节点时，能够回到的最小时间戳。如果low[v] >= num[u]，表示不能回到祖先。

3、图的割边

抽象问题：在一个无向连通图中，如果删除某条边后，图不再连通，则这样的边是割边。

如何求割边？只需要将求割点的算法修改一个符号，改成low[v] > num[u]。因为>=代表点v不可能在不经过父亲结点u而回到祖先的（包括父亲），所以顶点u是割点，如果=表示还有可能回到祖先，>则表示连父亲都回不到了。

倘若顶点v不能回到祖先，也没有另外一条路能回到父亲，那么u-v这条边就是割边。

4、二分图

如果一个图的所有顶点可以被分为X和Y两个集合，并且所有边的两个顶点恰好一个属于集合X，一个属于集合Y，即每个集合内的顶点没有边相连，那么此图就是二分图。

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <cstring>

#include <cmath>

using namespace std;

const int N = 110;

int a[N][N];

int book[N], n, m, match[N];

int dfs(int cur)

{

for(int i = 1; i <= n; i++)

{

if(a[cur][i] == 1 && book[i] == 0)

{

book[i] = 1; //标记点i已经访问过

/\*

如果点i未被配对或者找到了新的配对

\*/

if(match[i] == 0 || dfs(match[i]))

{

//更新配对关系

match[i] = cur;

return 1;

}

}

}

return 0;

}

int main()

{

scanf("%d%d", &n, &m);

int u, v;

for(int i = 1; i <= m; i++)

{

scanf("%d%d", &u, &v);

a[u][v] = 1;

}

memset(match, 0, sizeof(match));

int cnt = 0;

for(int i = 1; i <= n; i++)

{

memset(book, 0, sizeof(book));

if(dfs(i)) //寻找增广路

cnt++;

}

printf("\n");

for(int i = 1; i <= n; i++)

{

if(match[i])

printf("%d->%d\n", match[i], i);

}

return 0;

}

/\*

3 5

1 1

1 2

2 2

2 3

3 1

match[v] = u;

\*/

5、附加考题（主元素问题）

假如现在有一个序列，已知其中有一个数出现的次数超过50%，请找出这个数。

提示：这样的序列有一个，即在原序列中去除两个不一样的数，那么在原序列中出现次数超过了50%的数，在新序列中的出现次数也一定会超过50%。