## 欧几里得算法求最大公约数

求a和b的最大公约数  
记 a mod b=c ,即a=kb+c   
设a b的最大公约数为d,则a=m\*d b=n\*d,m和n互质。  
c=a-kb=md-knd=(m-kn)d,m和n互质，则n和m-kn互质，c和b的最大公约数也是d  
所以： "a和b(a>b)的最大公约数等于b和a Mode b的最大公约数"，递归或迭代计算，直到余数为0，此时除数为最大公约数。

也可以这样理解：  
a=m\*d b=n\*d,m和n互质,c=a-b=(m-n)\*d

,m-n 和 n互质，  
即，"a 、b（a>b）的最大公约数等于 b 和 a-b 的最大公约数。"递归或迭代计算，直到两数相等，此时的值为最大公约数。

**package** day01;

**import** java.util.Scanner;

/\*

\* 欧几里得算法求解最大公约数

\*

\* 最大公约数最小公倍数

\*/

**public** **class** MainDemo1015 {

**public** **static** **void** main(String[] args) {

Scanner sc = **new** Scanner(System.*in*);

**int** x = sc.nextInt();

**int** y = sc.nextInt();

**int** gcd1 = *gcd1*(x,y);

**int** gcd2 = *gcd2*(x,y);

System.*out*.println("gcd1: "+gcd1+","+"gcd2: "+gcd2);

**int** lcm = *lcm*(x,y);

System.*out*.println(lcm);

}

**public** **static** **int** gcd1(**int** x, **int** y) {

//要求x>=y

**if** (x < y) {

**int** temp = y;

y = x;

x = temp;

}

**if**(y!=0){

**return** *gcd1*(y,x%y);

} **else** {

**return** x;

}

}

**public** **static** **int** gcd2(**int** x ,**int** y){

**if**(x<y){

**int** temp = y;

y=x;

x=temp;

}

**if**(y!=0){

**return** *gcd2*(y,x-y);

} **else** {

**return** x;

}

}

**public** **static** **int** lcm(**int** x ,**int** y){

**return** x\*y/*gcd1*(x,y);

}

}

## 快速排序

**package** day01;

/\*

\* 快速排序： 1 找到基准数

\* 2 从后向前遍历，如果小于基准数，就交换到前面，否则继续遍历。

\* 3 再从前向后遍历，如果大于基准数，就交换到后面，否则继续遍历。

\* 4 第一轮结束之后，数组会被划分为两个部分，

\* 左边部分都小于基准数，右边部分都大于基准数

\* 5 分别对左右两个部分使用递归，重复23步骤。

\*/

**public** **class** Test {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 87, 55, 10, 2, 3, 4, 11 };

*quick\_sort*(arr, 0, arr.length - 1);

**for** (Integer i : arr) {

System.*out*.print(i + " ");

}

}

**public** **static** **void** quick\_sort(**int**[] arr, **int** first, **int** last) {

**if** (first < last) {

// 基准值

**int** x = arr[first], i = first, j = last;

**while** (i < j) {

**while** (i < j && arr[j] > x) {

j--;

}

**if** (i < j) {

arr[i++] = arr[j];

}

**while** (i < j && arr[i] < x) {

i++;

}

**if** (i < j) {

arr[j--] = arr[i];

}

}

arr[i] = x;

*quick\_sort*(arr, first, i - 1);

*quick\_sort*(arr, i + 1, last);

}

}

}

## 求解吸血鬼数

**package** charper03;

**import** java.util.Arrays;

**public** **class** SearchforVampireDigit {

**public** **static** **void** main(String[] args) {

String[] targetNum = **null**;

String[] gunNum = **null**; // 目标数字和枪数字（即对比数字

**int** count = 0, sum = 0; // 有效判断次数

**for** (**int** i = 10; i < 100; i++) {

**for** (**int** j = i + 1; j < 100; j++) {// 没有哪个两位数满足ab\*ab=abab，所以这里j从i+1开始就可以了

**int** i\_target = i \* j;

**if** (i\_target < 1000 || i\_target > 9999 || i\_target % 100 == 0 || (i\_target - i - j) % 9 != 0) {

**continue**;

}

count++;

targetNum = String.*valueOf*(i\_target).split(""); // 转换为字符串数组

gunNum = (String.*valueOf*(i) + String.*valueOf*(j)).split("");

Arrays.*sort*(targetNum);

Arrays.*sort*(gunNum); //// 升序排列，因为只有排列了再比较才能保证不遗漏abcd=ba\*dc这样的情况

**if** (Arrays.*equals*(targetNum, gunNum)) {

sum++;

System.***out***.println("第" + sum + "个" + i\_target + "=" + i + "\*" + j);

}

}

}

System.***out***.println("共进行了" + count + "次判断找到" + sum + "吸血鬼数字");

}

}

吸血鬼数字：位数为偶数的数字，可以由一对数字相乘而得到，而这对数字各包含乘积的一半位数的数字，其中从最初的数字中选取的数字可以任意排序。

高效率做法:采用逆向思维。4位数字的吸血鬼数字只能拆分成两个2位数，因此遍历所有两个两位数相乘的情况，除去不符合的情况不用判断，其他挨个判断即可。

解释：

一、吸血鬼数字的两个因子不能同时以0结尾，故用条件i\_val % 100 == 0 先过滤一次。  
二、任何4位数i\_val都可以表示为1000a+100b+10c+d  
如果其是吸血鬼数字，其两个因子是i和j,那么i+j只能有以下六种情况：  
1.(i+j)=10a+b+10c+d  
2.(i+j)=10a+10b+c+d  
3.(i+j)=10a+b+c+10d  
4.(i+j)=a+10b+c+10d  
5.(i+j)=a+10b+10c+d  
6.(i+j)=a+b+10c+10d  
所以无论上述的哪种情况下i\_val-(i+j)都一定能被9整除，故不满足这个条件的一定不是吸血鬼数字，所以**通过条件(i\_val - i - j) % 9 != 0再**过滤一次。

## N&(N-1)

1、 判断一个数是否是2的方幂  
n > 0 && ((n & (n - 1)) == 0 )

解释((n & (n-1)) == 0)：

如果A&B==0，表示A与B的二进制形式没有在同一个位置都为1的时候。

那么本题到底啥意思？？

不妨先看下n-1是什么意思。

   令:n=1101011000(二进制,十进制也一样)，则

    n-1=1101010111。

n&(n-1)=1101010000

由此可以得出，n和n-1的低位不一样，直到有个转折点，就是借位的那个点，从这个点开始的高位，n和n-1都一样，如果高位一样这就造成一个问题，就是n和n-1在相同的位上可能会有同一个1，从而使((n & (n-1)) != 0),如果想要

((n & (n-1)) == 0)，则高位必须全为0，这样就没有相同的1。

所以n是2的幂或0

2. 求某一个数的二进制表示中1的个数  
while (n >0 ) {  
      count ++;  
      n &= (n-1);  
}

3. 计算N!的质因数2的个数。  
容易得出N!质因数2的个数 = [N / 2] + [N / 4] + [N / 8] + ....  
下面通过一个简单的例子来推导一下过程：N = 10101(二进制表示）  
现在我们跟踪最高位的1，不考虑其他位假定为0，  
则在  
[N / 2]    01000  
[N / 4]    00100  
[N / 8]    00010  
[N / 8]    00001  
则所有相加等于01111 = 10000 - 1  
由此推及其他位可得：(10101)!的质因数2的个数为10000 - 1 + 00100 - 1 + 00001 - 1 = 10101 - 3(二进制表示中1的个数)

推及一般N!的质因数2的个数为N - （N二进制表示中1的个数）

n&(-n)在树状数组中lowbit出现   用来求 t 中的因子中形如2^k的数为多少     用来取得n最右边的1，可以知道其因子中有几个2

10:  0000 1010

-10: 1111 0110

10&(-10)为 0010  =  2  所以10的因子中为2的有一个，2^k的形式的为2^1

8&(-8) = [1000] = 8   所以8的因子中为2的有3个，2^k的形式为2^3

# 背包问题（动态规划）

# 十进制转化为二进制

**package** com.wy.huawei;

**public** **class** test {

**public** **static** **void** main(String[] args) {

**int** num =11;//假设正十进制整数为11

String Str ="";//保存转换的结果

**while**(num !=0){//表示还有数字，还可以继续执行

Str =(num % 2) + Str;//保存结果

num =num/2;//改变num 内容

}

System.***out***.println(Str);//输出二进制数

}

}

**package** com.wy.huawei;

**public** **class** test {

**public** **static** **void** main(String[] args) {

*toBinary*(11);

}

**public** **static** **void** toBinary(**int** num){

**if**(num/2==0){//此时已经计算到了结束

System.***out***.print(num%2);

} **else**{

*toBinary*(num/2);//向下继续计算

System.***out***.print(num%2);

}

}

}

# Morris算法遍历二叉树

遍历二叉树常用的方法是使用递归或者堆栈迭代实现 堆栈实现遍历的时间复杂度是O(n) 空间复杂度为O(logn) 使用递归算法时，空间复杂度为O(logn) 在节点数目较多时，递归的开销很大，占用资源大，因为需要不断的开辟新的栈桢，并保护现场。

使用Morris算法遍历二叉树时，时间复杂度O(n) 空间复杂度O(1)。

Morris算法的原理是线索二叉树。

在线索二叉树中，使用线索来保存前驱和后继的信息，而这些线索是利用叶节点的空指针域来保存，知道了树中每个节点的前驱和后继的位置就可以有效降低遍历过程中对空间的需求，但是使用线索二叉树先通过一次二叉树遍历算法，为二叉树建立线索。

Morri算法使用二叉树叶节点的right指针域来保存后面将要访问节点的信息（线索），当这个right指针域使用完毕之后再重新置为null。但是在访问过程中有些节点会被访问两次。

中序遍历

算法步骤：

1 如果当前节点的左子节点为空，则输出当前节点，并将当前节点置为该节点的右子节点。

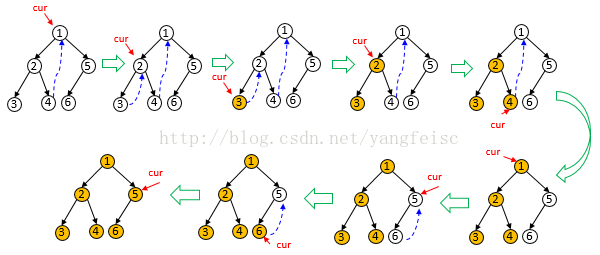
2如果当前节点的左子节点不为空时，则找到当前节点左子树的最右节点（该节点为当前节点中序遍历的前驱节点）

A 如果最右节点的右指针为空，则将最右节点的右指针指向当前节点，并把当前节点置为其左子节点。

B 如果最右节点的右指针不为空，则**输出当前节点**，并将最右节点的右指针重新置为空，将当前节点置为其右子节点。

3 重复1和2，直至当前节点为空。

（建议自己画图并结合程序）



//中序遍历

**public** List<Integer> morrisInOrder(TreeNode root) {

List<Integer> result = **new** ArrayList<Integer>();

**if**(root==**null**) **return** result; //返回[]

TreeNode cur = root;

**while**(cur!=**null**) {

**if**(cur.left==**null**) {

result.add(cur.val);

cur=cur.right;

} **else** {

TreeNode tmp = cur.left;

**while**(tmp.right!=**null** && tmp.right!=cur) {

tmp = tmp.right; //找到左子树的最右节点

}

**if**(tmp.right==**null**) {

tmp.right=cur;

cur=cur.left;

} **else** {

result.add(cur.val);

tmp.right=**null**;

cur=cur.right;

}

}

}

**return** result;

}

前序遍历

算法步骤：

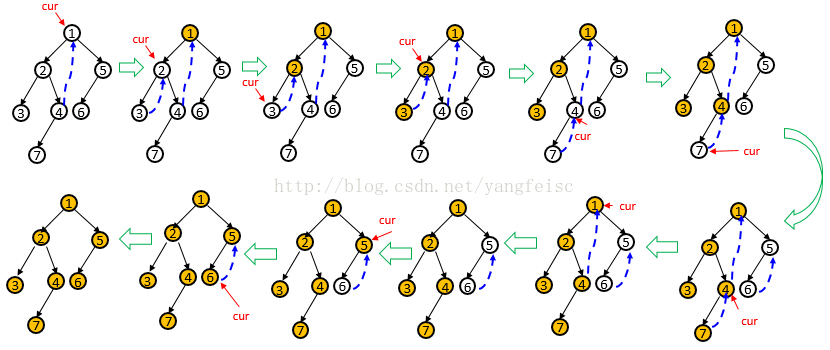
1如果当前节点的左子节点为空，则输出当前节点，并将当前节点置为该节点的右子节点。

2 如果当前节点的左子节点不为空，则找到当前节点左子树的最右节点（该节点为当前节点中序遍历的前驱节点）

A 如果最右节点的右指针为空，则**输出当前节点**，并将最右节点的右指针指向当前节点，当前节点置为其左子节点

B 如果最右节点的右指针不为空，则将最右节点的右指针重新置为空，并将当前节点置为其右子节点。

3 重复1和2，直至当前节点为空



//前序遍历

**public** List<Integer> morrisPreOrder(TreeNode root) {

List<Integer> result = **new** ArrayList<Integer>();

**if**(root==**null**) **return** result; //返回[]

TreeNode cur = root;

**while**(cur!=**null**) {

**if**(cur.left==**null**) {

result.add(cur.val); //输出当前节点

cur=cur.right;

} **else** {

TreeNode tmp = cur.left;

//找左子树的最右节点

**while**(tmp.right!=**null** && tmp.right!=cur) {

tmp=tmp.right;

}

**if**(tmp.right==**null**){

result.add(cur.val); //输出当前节点

tmp.right = cur;

cur=cur.left;

} **else** {

tmp.right=**null**; //恢复二叉树

cur=cur.right;

}

}

}

**return** result;

}

后序遍历：需要建立一个临时节点，并令该节点的左子节点为root，并且需要一个子过程，倒序输出某两个节点之间路径上的各个节点。

算法步骤：

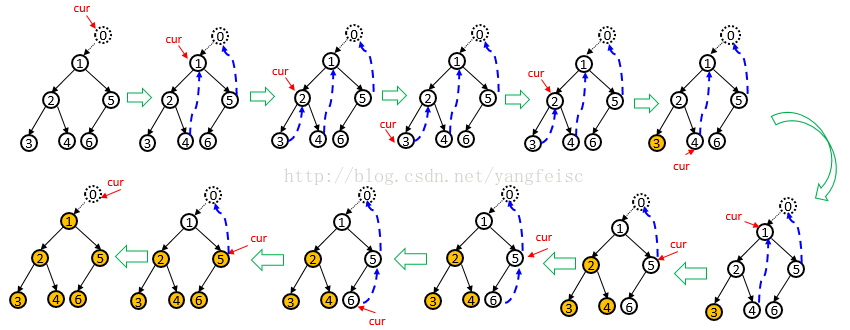
1 如果当前节点的左子节点为空，则将其右子节点作为当前节点。

2 如果当前节点的左子节点不为空，则找到当前节点左子树的最右节点（该节点为中序遍历的前驱节点）

A 如果最右节点的右指针为空，则将最右节点的右指针指向当前节点，当前节点置为其左子节点。

B 如果最右节点的右指针不为空，则将最右节点的右指针重新置为空，倒序输出从当前节点的左子节点到该最右节点路径上的所有节点，并将当前节点置为其右节点。

3 重复1和2，直至当前节点为空。



//后序遍历

**public** List<Integer> morrisPostOrder(TreeNode root) {

List<Integer> result = **new** ArrayList<Integer>();

**if**(root == **null**) **return** result;

TreeNode tmpNode = **new** TreeNode(-1); //建立一个临时节点

tmpNode.left = root;//临时节点的左子节点指向root

TreeNode cur = tmpNode;

**while**(cur!=**null**) {

**if**(cur.left==**null**) {

cur = cur.right;

} **else** {

TreeNode tmp = cur.left;

**while**(tmp.right!=**null** && tmp.right!=cur)

tmp = tmp.right;

**if**(tmp.right==**null**) {

tmp.right=cur;

cur=cur.left;

} **else** {

tmp.right=**null**;

TreeNode t = cur.left;

List<Integer> tmpList = **new** ArrayList<>();

**while**(t!=**null**) { //倒序输出

tmpList.add(0,t.val); //在指定位置上插入元素，向右移动该位置上原先的元素及后续元素

t = t.right;

}

result.addAll(tmpList);

cur=cur.right;

}

}

}

**return** result;

}

# 希尔排序的时间性能优于插入排序

1 当文件初态基本有序时直接插入排序所需的比较和移动次数均较少。

2 当n值较小时，n和 n²的差别也较小，即直接插入排序的最好时间复杂度O(n)和最坏时间复杂度0(n²)差别不大。

3 在希尔排序开始时增量较大，分组较多，每组的记录数目少，故各组内直接插入较快，后来增量di逐渐缩小，分组数逐渐减少，而各组的记录数目逐渐增多，但由于已经按di-1作为距离排过序，使文件较接近于有序状态，所以新的一趟排序过程也较快。

因此，希尔排序在效率上较直接插入排序有较大的改进。

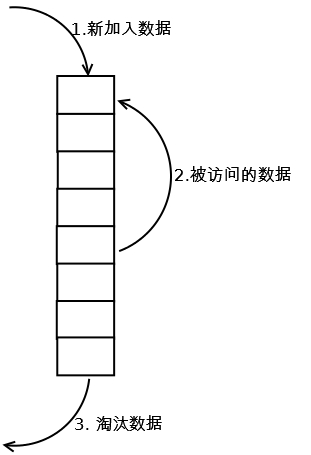
插入排序是稳定的 希尔排序不稳定

# LRU算法

<http://flychao88.iteye.com/blog/1977653>

Least recently used，最近最少使用算法。是根据数据的历史访问记录来进行淘汰数据，核心思想是如果数据最近被访问过，那么将来被访问的几率也很高。

常见的是使用一个链表保存缓存数据



步骤：1 新数据插入到链表头部

2 每当缓存命中（即缓存数据被访问时），则将数据移到链表头部。

3 当链表满的时候，将链表尾部的数据丢弃。

# Kmp算法

<http://www.cnblogs.com/yjiyjige/p/3263858.html>

<http://www.cnblogs.com/tangzhengyue/p/4315393.html>

kmp算法主要是解决主串中的模式定位问题。

主串T 模式串P

如果模式串在主串中出现，就返回模式串具体的位置，否则返回-1

暴力解法：设置两个指针分别遍历主串和模式串，从左到右一个个匹配，如果匹配过程中，某个字符不匹配，则主串的指针回溯到上次开始遍历时的下一位（i-j+1），模式串的指针回溯到位置0处。

上述的解法并不高效。

KMP算法利用已经部分匹配的有效信息，保持主串的指针不回溯，只通过修改模式串中j的指针，使模式串移动到有效的位置。

通过实例发现，如果在模式串中，当前位置j的不匹配时，那么j应该重新回溯到的位置是——在模式串中，最前面k个字符和j之前最后k个字符是一样，则j回溯到k+1的位置。

也就是找到j位置之前的字符串中最长公共前后缀的长度。

如何确定这个j回溯的位置呢？这里通过一个next数组，保存模式串中每一位置上的元素应该回溯的位置。即next[j]=k，意味着当T[i]!=P[j]时，j指针应该回溯到位置k上。

public int getNext(String ps) {

char[] p = ps.toCharArray();

int[] next = new int[p.length];

//初始化 j为遍历模式串的指针 k为对应的j应该回溯到的位置

int j=0;

int k=-1;

next[0]=-1;

while(j<p.length-1){

if(k==-1 || p[j]==p[k]) {

next[++j] = next[++k];

} else {

k=next[k];  
}

}

}

得到next数组之后，就可以写出kmp算法了

主要部分：

While(i<t.length && j<p.length) {

If(j==-1 || t[i]=p[j]) {

i++;j++;

} else {

j=next[j];

}

}

# 最短路径算法

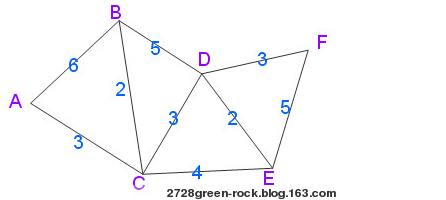
## Dijkstra

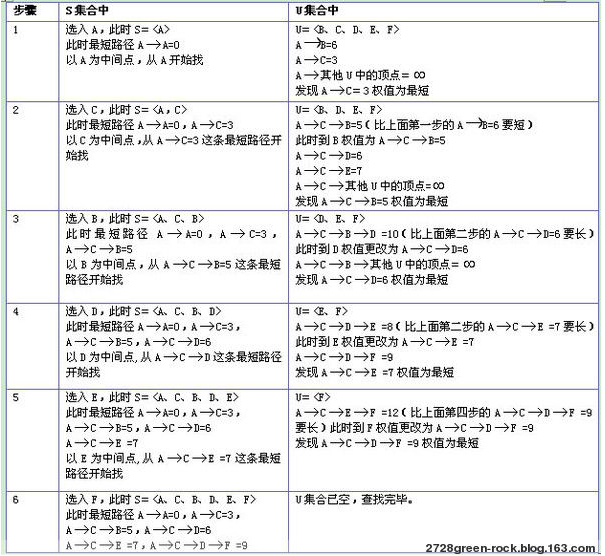
**Dijkstra迪杰斯特拉算法是单源最短路径算法，是用于计算一个节点到其他所有节点的最短路径。**

**问题描述**：在无向图 G=(V,E) 中，假设每条边 E[i] 的长度为 w[i]，找到由顶点 V0 到其余各点的最短路径。（单源最短路径）

**算法思想**：设G=(V,E)是一个带权有向图，把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合（用S表示，初始时S中只有一个源点，以后每求得一条最短路径 , 就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了），第二组为其余未确定最短路径的顶点集合（用U表示），按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

**例子**





## Floyd

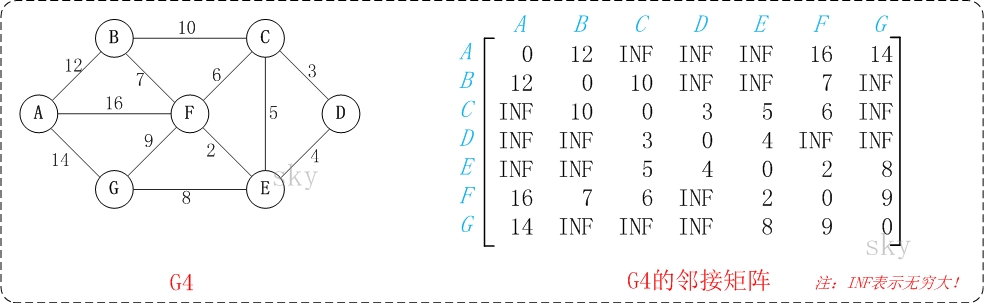
弗洛伊德算法是用于寻找给定的加权图中顶点间最短路径的算法。和dijkstra不同的是，floyd算法是求解图中各个顶点之间的最短路径。

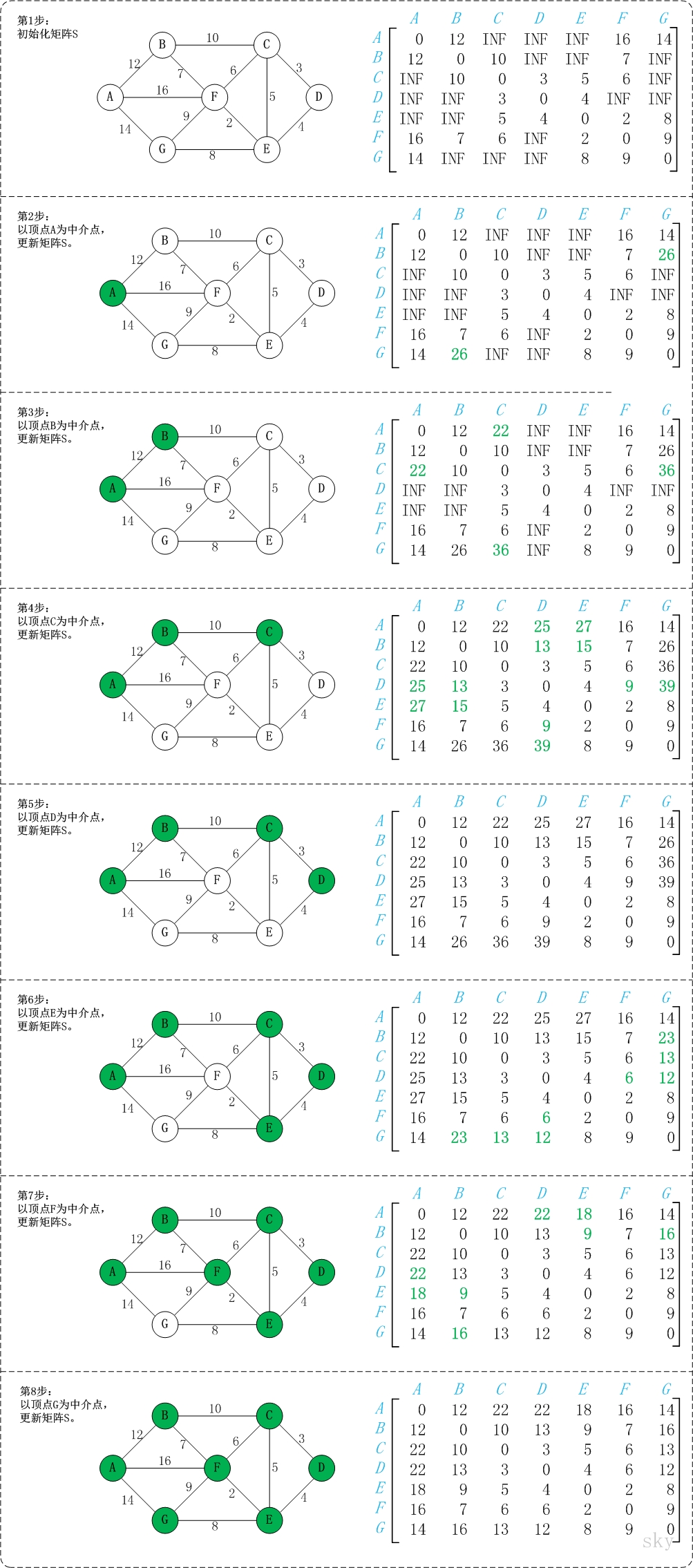
需要两个矩阵（二维数组），path[i][j]表示顶点i到j的最短路径经过节点path[i][j]。dist[i][j]表示顶点i到j的最短路径。

假设图中的顶点个数为N，则需要对矩阵dist进行N次更新。

初始时，dist[i][j]为顶点i到j的权值，如果i和j不相邻，则为无穷大。然后对dist进行N次更新，第1次更新时，如果"dist[i][j]的距离" > " dist [i][0]+ dist [0][j]"( dist [i][0]+ dist [0][j]表示"i与j之间经过第1个顶点的距离")，则更新dist [i][j]为" dist [i][0]+ dist [0][j]"。 同理，第k次更新时，如果" dist [i][j]的距离" > " dist [i][k]+ dist [k][j]"，则更新dist [i][j]为" dist [i][k]+ dist [k][j]"。更新N次之后，操作完成！

例子





程序实现

**基本定义**

public class MatrixUDG {

private int mEdgNum; // 边的数量

private char[] mVexs; // 顶点集合

private int[][] mMatrix; // 邻接矩阵

private static final int INF = Integer.MAX\_VALUE; // 最大值

...

}

/\*

\* floyd最短路径。

\* 即，统计图中各个顶点间的最短路径。

\*

\* 参数说明：

\* path -- 路径。path[i][j]=k表示，"顶点i"到"顶点j"的最短路径会经过顶点k。

\* dist -- 长度数组。即，dist[i][j]=sum表示，"顶点i"到"顶点j"的最短路径的长度是sum。

\*/

public void floyd(int[][] path, int[][] dist) {

// 初始化

for (int i = 0; i < mVexs.length; i++) {

for (int j = 0; j < mVexs.length; j++) {

dist[i][j] = mMatrix[i][j]; // "顶点i"到"顶点j"的路径长度为"i到j的权值"。

path[i][j] = j; // "顶点i"到"顶点j"的最短路径是经过顶点j。

}

}

// 计算最短路径

for (int k = 0; k < mVexs.length; k++) {

for (int i = 0; i < mVexs.length; i++) {

for (int j = 0; j < mVexs.length; j++) {

// 如果经过下标为k顶点路径比原两点间路径更短，则更新dist[i][j]和path[i][j]

int tmp = (dist[i][k]==INF || dist[k][j]==INF) ? INF : (dist[i][k] + dist[k][j]);

if (dist[i][j] > tmp) {

// "i到j最短路径"对应的值设，为更小的一个(即经过k)

dist[i][j] = tmp;

// "i到j最短路径"对应的路径，经过k

path[i][j] = path[i][k];

}

}

}

}

// 打印floyd最短路径的结果

System.out.printf("floyd: \n");

for (int i = 0; i < mVexs.length; i++) {

for (int j = 0; j < mVexs.length; j++)

System.out.printf("%2d ", dist[i][j]);

System.out.printf("\n");

}

}