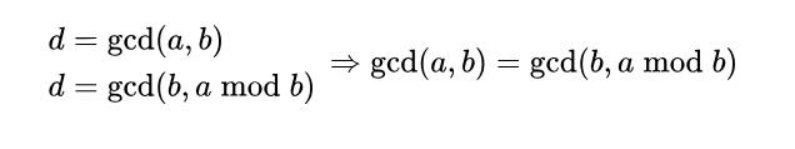
# 最大公约数与最小公倍数

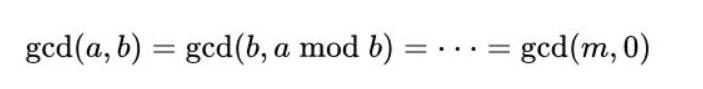
**问题：**最大公因数Greatest Common Divisor(GCD)，也称最大公约数、最大公因子，指两个或多个整数共有约数中最大的一个。a，b的最大公约数记为（a，b），同样的，a，b，c的最大公约数记为（a，b，c），多个整数的最大公约数也有同样的记号。求最大公约数有多种方法，常见的有质因数分解法、短除法、辗转相除法、更相减损法。与最大公约数相对应的概念是最小公倍数，a，b的最小公倍数记为[a，b]。

## 欧几里得算法（辗转相除法）

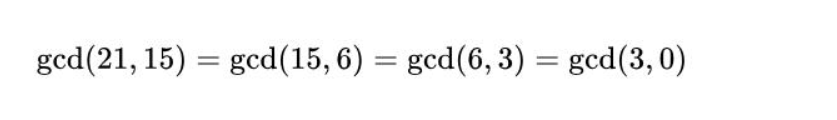
**欧几里得算法：** 欧几里得算法又称辗转相除法，是指用于计算两个非负整数a，b的最大公约数。应用领域有数学和计算机两个方面。计算公式gcd(a,b) = gcd(b,a mod b)。

**计算原理：** a可以表示成a = kb + r（a，b，k，r皆为正整数，且r<b），则r = a mod b。假设d是a,b的一个公约数，记作d|a,d|b，即a和b都可以被d整除。而r = a - kb，两边同时除以d，r/d=a/d-kb/d=m，由等式右边可知m为整数，因此d|r（r可以被d整除）。因此d也是b,a mod b的公约数。因(a,b)和(b,a mod b)的公约数相等，则其最大公约数也相等。

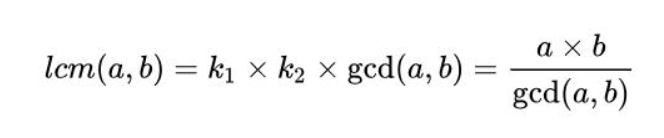
r=a mod b,上式就是一次辗转相除法，由于余数r式逐渐减少的，到某一时刻他会为零，即



很明显m就是a和b之间的最大公约数了，比如21和15的最大公约数如下：



**最小公倍数：**最小公倍数可以通过最大公约数求：最小公倍数 = 两数之积 / 最大公约数。



## 更相减损术

**更相减损法：**更相减损术， 出自于中国古代的《九章算术》，也是一种求最大公约数的算法。

1.先判断两个数的大小，如果两数相等，则这个数本身就 是就是它的最大公约数。

2.如果不相等，则用大数减去小数，然后用这个较小数与它们相减的结果相比较，如果相等，则这个差就是它们的最大公约数，而如果不相等，则继续执行2操作。

## Stein算法（结合两种算法优势，同时运用位运算）

**Stein算法：**众所周知，移位运算的性能非常快。对于给定的正整数a和b，不难得到如下的结论。其中gcb(a,b)的意思是求a,b的最大公约数的函数

1.当a和b均为偶数，gcb(a,b) = 2gcb(a/2, b/2) = 2gcb(a>>1, b>>1)

2.当a为偶数，b为奇数，gcb(a,b) = gcb(a/2, b) = gcb(a>>1, b)

3.当a为奇数，b为偶数，gcb(a,b) = gcb(a, b/2) = gcb(a, b>>1)

4.当a和b均为奇数，利用更相减损术运算一次，gcb(a,b) = gcb(b, a-b)， 此时a-b的结果必然是偶数，又可以继续进行移位运算。

# 最短路算法

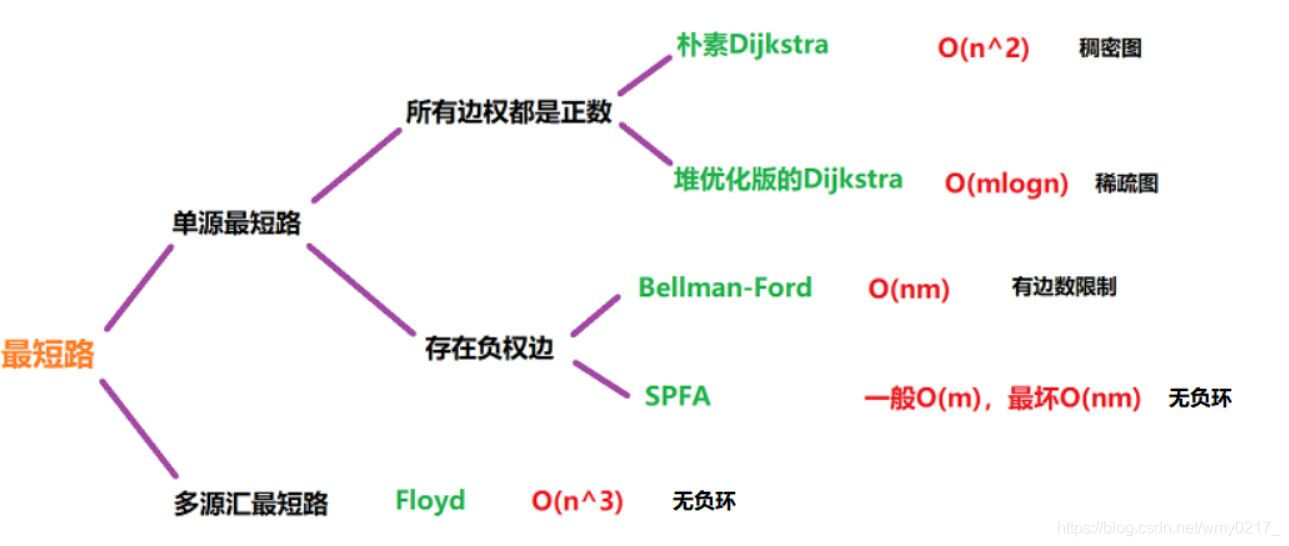
**问题：**从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径——最短路径。解决最短路的问题有以下算法，Dijkstra算法，Bellman-Ford算法，Floyd算法和SPFA算法，另外还有著名的启发式搜索算法A\*，其中Floyd算法可以求解任意两点间的最短路径的长度。笔者认为任意一个最短路算法都是基于这样一个事实：从任意节点A到任意节点B的最短路径不外乎2种可能，1是直接从A到B，2是从A经过若干个节点到B。

**单源最短路：** 求一个点到其他点的最短路

**多源最短路：** 求任意两个点的最短路

稠密图用邻接矩阵存，稀疏图用邻接表存储。

**稠密图：** m 和 n2 一个级别  
**稀疏图：** m 和 n 一个级别



## Floyd算法（求传递闭包,）

**Floyd算法/Floyd-Warshell算法（暴力枚举所有可能）：**从任意节点A到任意节点B的最短路径不外乎2种可能，1是直接从A到B，2是从A经过若干个节点到B，所以，假设dist(AB)为节点A到节点B的最短路径的距离，对于每一个节点K，我们检查dist(AK) + dist(KB) < dist(AB)是否成立，如果成立，证明从A到K再到B的路径比A直接到B的路径短，我们便设置 dist(AB) = dist(AK) + dist(KB)，这样一来，当我们遍历完所有节点K，dist(AB)中记录的便是A到B的最短路径的距离。

Floyd本质上是动态规划的思想。倘若现在我们想求i到j的最短路径长度，我们限制这条路径上除i和j之外只准经过前k个点（这样的路径称为k允许路径），我们在算法的最外层循环每次将k加1，那么当k等于点数时求得的结果便是最优的。

## 朴素Dijkstra算法（贪心,复杂度）

**朴素Dijkstra算法**：Dijkstra算法通常是求解单源最短路中最快的算法，但它无法处理存在负权边的情况（原因在正确性证明中）。Dijkstra本质上是一种贪心算法，通过不断调整每个点的“当前距离”最终得到最优结果。

首先把起点到所有点的距离存下来找个最短的，然后松弛一次再找出最短的，所谓的松弛操作就是，遍历一遍看通过刚刚找到的距离最短的点作为中转站会不会更近，如果更近了就更新距离，这样把所有的点找遍之后就存下了起点到其他所有点的最短距离。（用现在的最小路径去更新其他的路径）。

假设现在要求出从某一点s到其他所有点的最短距离，对于每个点v均维护一个“当前距离”（dist[v]）和“是否访问过”(visited[v])。首先将dist[s]初始化为0，将其他点的距离初始化为无穷，并将所有点初始化为未访问的。记u->v的边权为weight[u->v]。然后进行以下步骤：

1.初使时令 S={V0},T={其余顶点}，T中顶点对应的距离值， 若存在<V0,Vi>，为<V0,Vi>弧上的权值（和ＳＰＦＡ初始化方式不同），若不存在<V0,Vi>，为Inf。

2.从T中选取一个其距离值为最小的顶点W(贪心体现在此处)，加入S(注意不是直接从S集合中选取，理解这个对于理解vis数组的作用至关重要)，对T中顶点的距离值进行修改：若加进W作中间顶点，从V0到Vi的距离值比不加W的路径要短，则修改此距离值（上面两个并列for循环，使用最小点更新）。

2.5.补充：如果除了最短距离之外还想求出具体的路径，只需建立一个pre数组，在步骤2后添加操作：pre[v] = Vi（前提是dist[v]被更新）。

3.重复上述步骤，直到S中包含所有顶点，即S=V为止（说明最外层是除起点外的遍历）

0

2

3

4

5

6

1

322

7

17

2

6

30

5

8

1333

9

最短路径

<V0,V1>》>

<V0,V2>

<V0,V2,V3>

<V0,V2,V3,V4>

<V0,V2,V3,V4,V5>

<V0,V1,V6>

13

8

13

21

19

20

长度

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 终点 | 从V0到各终点的最短路径及其长度 | | | | |
| V1 | 13  <V0,V1> | 13  <V0,V1> | --- | --- | --- |
| V2 | 8  <V0,V2> | --- | --- | --- | --- |
| V3 | ∞ | 13  <V0,V2,V3> | 13  <V0,V2,V3> | --- | --- |
| V4 | 30  <V0,V4> | 30（2和4间为Inf）  <V0,V4> | 30  <V0,V4> | 19  <V0,V2,V3,V4> | --- |
| V5 | ∞ | ∞ | 22  <V0,V1,V5> | 22  <V0,V1,V5> | 22  <V0,V2,V3,V4,V5> |
| V6 | 32  <V0,V6> | 32  <V0,V6> | 20  <V0,V1,V6> | 20  <V0,V1,V6> | 20  <V0,V1,V6> |
| Vj | V2:8  <V0,V2> | V1:13  <V0,V1> | V3:13  <V0,V2,V3> | V4:19  <V0,V2,V3,V4> | V6:20  <V0,V1,V6> |

按列来看，第一列是初始化过程，最后一行是每次求得的next点。

## 堆优化Dijkstra算法（）

**堆优化Dijkstra算法：**对于**稀疏图**来说，由于边数与点数相差不大，一个点所能到达的点比较少，所以用**邻接表**来存储效率比较高。在确定一个还未确定最短距离的点中距离源点最近距离的点时朴素Dijkstra的方法是遍历所有的点通过比较找出最近的点，在这个地方可以使用优先队列来进行优化，通过优先队列优化后朴素Dijkstra算法就叫做堆优化版的Dijkstra算法。

首先将优先队列定义成小根堆，将源点的距离初始化为0加入到优先队列中，然后从这个点开始扩展。先将队列中的队头元素ver保存到一个临时变量中，并将队头元素出队，然后遍历这个点的所有出边所到达的点j，更新所有到达的点距离源点最近的距离。

如果源点直接到 j 点的距离比源点先到 ver 点再从 ver 点到 j 点的距离大，那么就更新 dist[j]，使 dist[j] 到源点的距离最短，并将该点到源点的距离以及该点的编号作为一个 pair 加入到优先队列中，然后将其标记，表示该点已经确定最短距离。因为是小根堆，所以会根据距离进行排序，距离最短的点总是位于队头。一直扩展下去，直到队列为空。

因为有重边的缘故，所以该点可能会有冗余数据，即如果在扩展的时候，第一次遍历到的点是2号点，距离源点的距离为10，此时 dist[2] = 0x3f3f3f3f > dist[1] + distance[1 -> 2] = 0 + 10 = 10 所以 dist[2] 会被更新为 10，此时会将 {10, 2} 入队。但是很不巧从 源点 到 2 号点有一个距离为 6 的重边，当遍历到这个重边时，由于 dist[2] = 10 > dist[1] + distance[1 -> 2] = 0 + 6 = 6,所以 {6, 2} 也入队了，入队之后由于是 小根堆 所以 {6, 2} 会排在 {10, 2} 前面，所以 {6, 2} 会先出队，出队之后会被标记。所以当下一次再遇到已经被标记的 2 号点时，直接 continue 忽略掉冗余数据继续扩展下一个点即可。

## Bellman-Ford算法()

**Bellman-Ford算法(一种基于松弛（relax）操作的最短路算法，可以求出有负权的图的最短路，并可以对最短路不存在的情况进行判断。):**对于一个不包含负权环的V个点的图，任意两点之间的最短路径至多包含V-1条边。为了能够求解边上带有负值的单源最短路径问题，Bellman(贝尔曼，动态规划提出者)和Ford(福特)提出了从源点逐次绕过其他顶点，以缩短到达终点的最短路径长度的方法。 Bellman-ford算法是求含负权图的单源最短路径算法，效率很低，但代码很容易写。即进行不停地松弛，每次松弛把每条边都更新一下，若n-1次松弛后还能更新，则说明图中有负环，无法得出结果，否则就成功完成。Bellman-ford算法有一个小优化：每次松弛先设一个flag，初值为FALSE，若有边更新则赋值为TRUE，最终如果还是FALSE则直接成功退出。

递推公式(求顶点u到源点v的最短路径)：

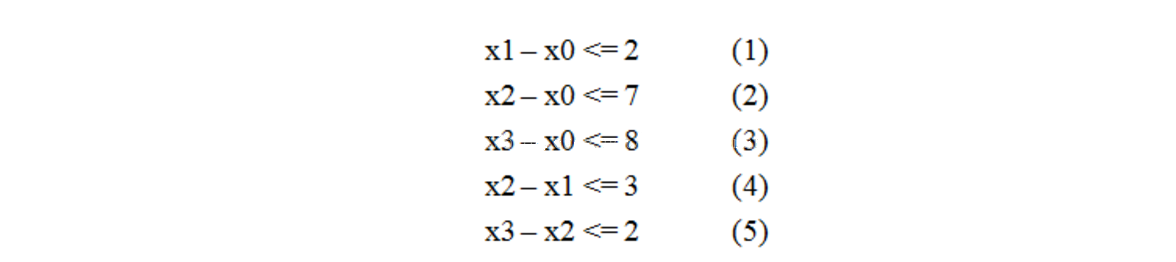
dist 1 [u] = Edge[v][u]

dist k [u] = min{ dist k-1 [u], min{ dist k-1 [j] +Edge[j][u] } }, j=0,1,…,n-1,j≠u

*Dijkstra算法和Bellman算法思想有很大的区别：Dijkstra算法在求解过程中，源点到集合S内各顶点的最短路径一旦求出，则之后不变了，修改 的仅仅是源点到T集合中各顶点的最短路径长度。Bellman算法在求解过程中，每次循环都要修改所有顶点的dist[ ]，也就是说源点到各顶点最短路径长度一直要到Bellman算法结束才确定下来。*

**适用条件：**单源最短路径(从源点s到其它所有顶点v)，有向图&无向图(无向图可以看作(u,v),(v,u)同属于边集E的有向图)，边权可正可负(如有负权回路输出错误提示)，**差分约束系统**。

**差分约束系统（system of difference constraints）：**求解关于一组变数的特殊不等式组之方法。如果一个系统由n个变量和m个约束条件组成，其中每个约束条件形如xj-xi<=bk(i,j∈[1,n],k∈[1,m]),则称其为差分约束系统(system of difference constraints)。亦即，差分约束系统是求解关于一组变量的特殊不等式组的方法。通俗一点地说，差分约束系统就是一些不等式的组，而我们的目标是通过给定的约束不等式组求出最大值或者最小值或者差分约束系统是否有解。



差分约束系统可以转化为图论来解决，对应于上面的不等式组，如果要求出x3-x0的最大值的话，叠加不等式可以推导出x3-x0<=7,最大值即为7，我们可以通过建立一个图，包含6个顶点，对每个xj-xi<=bk，建立一条i到j的有向边，权值为bk。通过求出这个图的x0到x3的最短路可以知道也为7。

之所以差分约束系统可以通过图论的最短路来解，是因为xj-xi<=bk，会发现它类似最短路中的三角不等式d[v] <=d[u]+w[u,v]，即d[v]-d[u]<=w[u,v]。而求取最大值的过程类似于最短路算法中的松弛过程。

三角不等式：

B - A <= c

C - B <= a

C - A <= b

如果要求C-A的最大值，可以知道max（C-A）= min(b,a+c),而这正对应了下图中C到A的最短路。

B

c

a

b

A

C

因此，对三角不等式加以推广，变量n个，不等式m个，要求xn-x1的最大值，便就是求取建图后的最短路。

同样地，如果要求取差分约束系统中xn-x1的最小值，便是求取建图后的最长路。最长路可以通过spfa求出来，只需要改下松弛的方向即可，即if(d[v] < d[u] + dist(u,v)) d[v] = d[u] + dist(u,v)。当然我们可以把图中所有的边权取负，求取最短路，两者是等价的。

最后一点，建图后不一定存在最短路/最长路，因为可能存在无限减小/增大的负环/正环，一般会对应于不同的输出。判断差分约束系统是否存在解一般判环即可。

**Bellman-Ford算法计算：**

1.初始化：将除源点外的所有顶点的最短距离估计值 d[v] ←+∞, d[s] ←0。

2.迭代求解：反复对边集E中的每条边进行松弛操作，使得顶点集V中的每个顶点v的最短距离估计值逐步逼近其最短距离；（运行|v|-1次，看下面的描述性证明(当做树)）。

3.检验负权回路：判断边集E中的每一条边的两个端点是否收敛。如果存在未收敛的顶点，则算法返回false，表明问题无解；否则算法返回true，并且从源点可达的顶点v的最短距离保存在d[v]中。

**描述性证明：**

首先指出，图的任意一条最短路径既不能包含负权回路，也不会包含正权回路，因此它最多包含|v|-1条边。

其次，从源点s可达的所有顶点如果存在最短路径，则这些最短路径构成一个以s为根的最短路径树。Bellman-Ford算法的迭代松弛操作，实际上就是按顶点距离s的层次，逐层生成这棵最短路径树的过程。

在对每条边进行1遍松弛的时候，生成了从s出发，层次至多为1的那些树枝。也就是说，找到了与s至多有1条边相联的那些顶点的最短路径；对每条边进行第2遍松弛的时候，生成了第2层次的树枝，就是说找到了经过2条边相连的那些顶点的最短路径……。因为最短路径最多只包含|v|-1条边，所以，只需要循环|v|-1 次。

每实施一次松弛操作，最短路径树上就会有一层顶点达到其最短距离，此后这层顶点的最短距离值就会一直保持不变，不再受后续松弛操作的影响。（但是，每次还要判断松弛，这里浪费了大量的时间，这就是Bellman-Ford算法效率底下的原因，也正是SPFA优化的所在）。

5

D

C

3

6

4

A

B

如图，若是B和C的最短路径不更新，那么点D的最短路径肯定也无法更新，这就是优化所在。

如果没有负权回路，由于最短路径树的高度最多只能是|v|-1，所以最多经过|v|-1遍松弛操作后，所有从s可达的顶点必将求出最短距离。如果 d[v]仍保持 +∞，则表明从s到v不可达。

如果有负权回路，那么第 |v|-1 遍松弛操作仍然会成功，这时，负权回路上的顶点不会收敛。

**需要注意的是，以S点为源点跑 Bellman-Ford 算法时，如果没有给出存在负环的结果，只能说明从 S点出发不能抵达一个负环，而不能说明图上不存在负环。**

**因此如果需要判断整个图上是否存在负环，最严谨的做法是建立一个超级源点，向图上每个节点连一条权值为 0 的边，然后以超级源点为起点执行 Bellman-Ford 算法。**

## SPFA算法()

**SPFA算法：**bellmon-ford算法是带着一定的盲目性的，作为对它的优化，spfa采用类似bfs的思想，使用一个队列，只松弛那些可能更新点的距离的边。用一个队列来进行维护。

初始时将源加入队列。每次从队列中取出一个元素，并对所有与他相邻的点进行松弛，若某个相邻的点松弛成功，则将其入队。直到队列为空时算法结束；这个算法，简单的说就是队列优化的bellman-ford，利用了每个点不会更新次数太多的特点发明的此算法。

在Bellman-Ford算法中，如果某个点未被更新过，我们还是会用这个点去更新其他点，其实，该操作是不必要的，我们只需要拿更新过后的点去更新其他的点，因为只有用被更新过的点更新其他结点x，x的距离才可能变小。

# 强连通图算法

**问题：**在静态分析技术中, 常会将代码转成抽象语法树(AST), 然后采用深度遍历（DFS）来完成对语法树的遍历和查询，找到潜在的问题缺陷。

对于语义的分析，我们采用的控制流和数据流也都无一例外的采用了以图为基础的算法, 通过图的可达性, 来完成变量、表达式的可达分析, 以及变量的依赖分析、值流图等等。

图的算法是进行静态分析的基础数据算法，如何提高图的分析效率，就需要对图的算法有进一步的认识。

Tarjan 算法是图论中非常实用 / 常用的算法之一，能解决强连通分量，双连通分量，割点和桥，求最近公共祖先（LCA）等问题。

**关联（incident）：**点为边的端点;

**邻接（adjacent）：**点与点关联同一条边，或边与边关联同一顶点；

**子图：**图G'的点和边都是图G的子集，则G'为G的子图;

**道路：**从点v到点u的路径；

**简单道路：**没有重复边的道路；

**回路：**起点与终点相同的道路；

**简单回路：**没有重复边的回路；

**连通：**两顶点间有道路；

**强连通：**有向图u→v与v→u都有道路；

**连通图：**任意两顶点间都有道路（若有向图除去方向后连通，则称有向图连通）；

**简单图：**没有重复边和自环的图；

**完全图：**任意两顶点间有一条边到达的简单图（有向完全图与无向完全图）；

**强连通（strongly connected）:** 在有向图G 中，如果两个顶点间至少存在一条路径，称两个顶点强连通（strongly connected）；

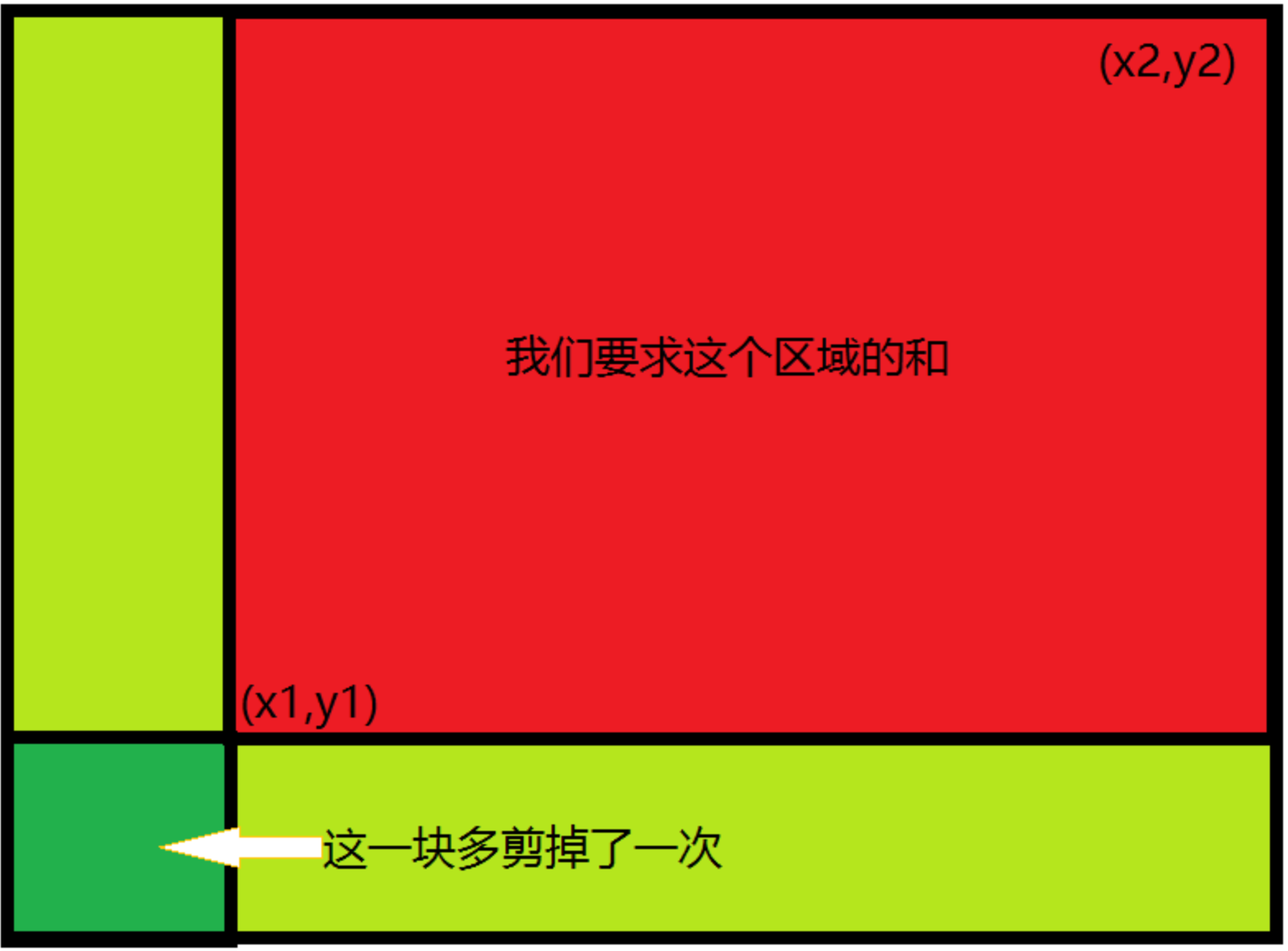
**强连通图:** 如果有向图G 的每两个顶点都强连通，称G 是一个强连通图；

**强连通分量(strongly connected components):** 非强连通图有向图的极大强连通子图，称为强连通分量(strongly connected components)。

# 线段树和他的儿子们-前缀和差分与树状数组

**前缀和：**可以理解为数学上数列的前n项和（对于一维数组的前缀和），定义对于一个数组a的前缀和数组s，s[i] = a[1] + a[2] + … + a[i].（前缀和一般用来进行求区间和：如：在对一个一维数组a，要求进行m次询问，每次询问下标j到k的和。M次查询要进行m次\*(j-i）的总复杂度，而使用前缀数组，只进行M次减法就可以完成，求出每个位置的前缀和，然后用s[k]-s[j],计算区间和）。

**二维前缀和：**与一维前缀和类似，设s[i][j]表示所有a[I’][j’]的和（1<=i’<=I, 1<=j’<=j）类似于矩形面积，将一整块区域的值加起来。



假设在这个矩阵（二维数组）中，要求和的是上图红色区域，此时已经处理完所有点的前缀和，此时查询以（x1,y1），（x2,y2）以两点连线为对角线的子矩阵数值之和。将大矩阵减去黄色部分再加一次绿色部分。

故对于一次查询答案应该等于

s[x2][y2]-s[x2][y1-1]-s[x1-1][y2]+s[x1-1][y1-1]

此二维前缀也称为：**二维差分序列。**

**一维差分：**对场景：

（1）给定一个长度为n的一维数组a[]，数组内每个元素有初始值。

（2）修改操作：做m次区间修改，每次修改对区间内所有元素做相同的加减操作。例如第i次修改，把[Li,Ri]内所有元素加上di。

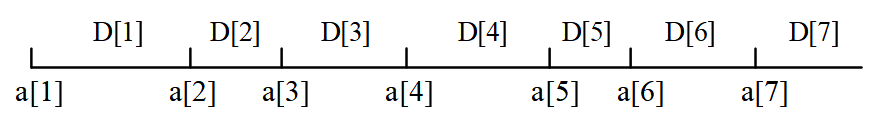
（3）查询操作：查询一个元素的新值是多少。

如果简单直接进行修改，每次修改复杂度为O(n)，m次修改共O（mn）采用差分法减少至O（m+n）.

差分法中用到两个数组，原数组a[]，差分数组D[]。

差分数组D[]的定义为：D[k] = a[k] – a[k-1],即原数组a[]相邻元素之间的差。a[k] = D[1] + D[2] + … + D[k].即a[]为D[]的前缀和。此公式揭示了差分与前缀的关系：**差分是前缀和的逆运算。**

为加深对前缀和的理解，可以把每个D[]看成一条直线上的小线段，它的两端是相邻的a[]。这些小线段相加，就得到了从起点开始的长线段a[]。

   注意， a[]和D[]的值都可能为负，下面图中所有的D[]都是长度为正的线段，只是为了方便图示。

把区间[L,R]内每个元素加上d，对应的D[]做以下操作：

1. 把D[L]加上d **D[L] += d;**
2. 把D[R+1]减去d **D[R+1] -= d;**

每次操作只需要修改区间[L,R]的两个端点的D[]值，复杂度是O(1)的。经过这种操作后，原来直接在a[]上做的复杂度为O(n)的区间修改操作，就变成了在D[]上做的复杂度为O(1)的端点操作。

利用D[]，能精确地实现只修改区间内元素的目的，而不会修改区间外的a[]值。因为前缀a[x] = D[1] + D[2] + … + D[x]**(位于L-R之间的所有a[i]均增长了d)**,有：

  （1）1 ≤ x < L，前缀和a[x]不变；

  （2）L ≤ x ≤ R，前缀和a[x]增加了d；

  （3）R < x ≤ N，前缀和a[x]不变，因为被D[R+1]中减去的d抵消了。

完成区间修改并得到D[]后，最后用D[]计算a[]，复杂度是O(n)的。m次区间修改和1次查询，总复杂度为O(m+n)，比暴力法的O(mn)好多了。

# 附录（算法实现）

## 欧几里得算法

C语言常规实现：

#include <stdio.h>

int main(){

int m,n,i;

scanf(“%d”,&m);

scanf(“%d”,&n);  
 for(i=m;i>=1;i--)

if(m%i==0 && n%i==0)

break;

printf(“%d\n”,i);

return 0;

}

C语言递归算法：

#include <stdio.h>

int gcd(int a,int b){

if(b==0) return a;

return gcd(b,a%b);

}

int main(){

int a,b;

while(scanf(“%d %d”,&a,&b)!=EOF){

printf(“%d”,gcd(a,b));

};

return 0;

}

C++库函数实现:

#include <bits/stdc++.h>

Using namespace std;

int main(){

int m,n;

cin>>m>>n;

cout<<\_\_gcd(m,n);

}

C++递归实现：

#include <cstdio>

int GCD(int a,int b){

return a%b?GCD(b,a%b):b;

}

int main(){

int x,y;

scanf(“%d%d”,&x,&y);

printf(“%d”,GCD(x,y));

return 0;

}

Java实现：

import java.util.Scanner;

public class Six{

public static void main(String[] args){

System.out.print(“输入a，b”);

Scanner scan=new Scanner(System.in);

int a=scan.nextInt();

int b=scan.nextInt();

int middle1,middle2,middle3;

middle1=a;

middle2=b;

middle3=0;

for(int i=0;i<i+1;i++){

middle3=middle1%middle2;

if(middle3==0)

break;

else{

middle1=middle2;

middle2=middle3;

}

}

System.out.println(“最大公约数为：”+middle2);

}

}

## 更相减损术算法

int gcd(int a,int b){

while(true){

if(a>b){

a-=b;

}

else if(a<b){

b-=a;

}

else{

return a;

}

}

}

int main(){

int a=12,b=18;

int res=gcd(a,b);

cout<<a<<“和”<<b<<“最大公约数是”<<res<<endl;

return 0;

}

## Stein算法

int gcd(int a,int b){

if(a == 0) return b;

if(b == 0) return a;

if(a % 2 == 0 && b % 2 == 0) return 2 \* gcd(a >> 1, b >> 1);

else if(a % 2 == 0) return gcd(a >> 1, b);

else if(b % 2 == 0) return gcd(a, b >> 1);

else return gcd(abs(a - b), min(a, b));

}

int main(){

int a = 12,b = 18;

int res = gcd(a,b);

cout << a << "和"<< b << "的最大公约数是" << res<<endl;

return 0;

}

4,srensiu=un