算法数据结构

浙江理工大学科技与艺术学院

计算机社团

标准竞赛模板

```
#include<cstdio>
#include<queue>
#include<cmath>
#include<cstring>
#include<algorithm>
#include<iostream>

#define LL long long
#define clr(a, b) memset(a, b, sizeof(a)
#define height(p) ( (p==NULL) ? -1 : (((Node *)(p))->height) )
#define max(a, b) ( (a) > (b) ? (a) : (b) )
const int MAXN = 100050;
const int HALF = 0x3f3f3f3f;
const int INF = (~(0x1<<31));
using namespace std;
```

本书由计算机社团整理 By Terrance
初稿 2017 年 5 月 6 日 23:04:35
更新 2017 年 6 月 30 日 11:48:19
更新 2017 年 8 月 2 日 13:48:33
更新 2017 年 9 月 7 日 18:27:09
更新 2017 年 9 月 15 日 12:37:59
更新 2017 年 10 月 25 日 21:16:00
更新 2017 年 10 月 30 日 12:47:53
=第一稿=

数据结构汇总

ACAutoman AC 自动机
BM 字符串模式匹配
BinarySearch 二叉搜索树
Dijkstra 最短路
FFT 傅里叶快速闭环
Floyed 最短路
HashTable 哈希
KMP 字符串模式匹配
Linklist 链表
LIS 最长非降子序列
MergeArray 归并
Mentecarlo 蒙特卡洛
Krusakal 最小生成树
Prim 最小生成树
Purfer 序列，树的生成树计数
NTT 快速数论变换
Radixsort 基数排序
RMQ 区间最值查询
RulerExtraction 尺取法
Segtree 线段树
SPFA 最短路
ThreeDevide 三分法
Data Structures 基本数据结构
Dictionaries 字典
Priority Queues 堆
Graph Data Structures 图
Set Data Structures 集合
Kd-Trees 线段树
Numerical Problems 数值问题
Solving Linear Equations 线性方程组
Bandwidth Reduction 带宽压缩
Matrix Multiplication 矩阵乘法
Determinants and Permanents 行列式
Linear Programming 线性规划
Random Number Generation 随机数生成
Discrete Fourier Transform 离散 Fourier 变换
Combinatorial Problems 组合问题
Sorting 排序
Searching 查找
Median and Selection 中位数
Generating Permutations 排列生成
Generating Subsets 子集生成
Generating Partitions 划分生成
Generating Graphs 图的生成
Calendrical Calculations 日期
Job Scheduling 工程安排
Satisfiability 可满足性
Connected Components 连通分支
Topological Sorting 拓扑排序
Minimum Spanning Tree 最小生成树
Shortest Path 最短路径
Transitive Closure and Reduction 传递闭包
Matching 匹配
Eulerian Cycle Chinese Postman Euler 回路中国邮路
Edge and Vertex Connectivity 割边割点
Network Flow 网络流
Drawing Graphs Nicely 图的描绘
Drawing Trees 树的描绘
Planarity Detection and Embedding 平面性检测和嵌入
Graph Problems -- hard 图论-NP 问题
Clique 最大团
Independent Set 独立集
Vertex Cover 点覆盖
Traveling Salesman Problem 旅行商问题
Hamiltonian Cycle Hamilton 回路
Graph Partition 图的划分
Vertex Coloring 点染色
Edge Coloring 边染色
Graph Isomorphism 同构
Steiner Tree Steiner 树
Feedback EdgeVertex Set 最大无环子图
Computational Geometry 计算几何
Convex Hull 凸包
Triangulation 三角剖分
Voronoi Diagrams Voronoi 图
Nearest Neighbor Search 最近点对查询
Range Search 范围查询
Point Location 位置查询
Intersection Detection 碰撞测试
Bin Packing 装箱问题
Medial-Axis Transformation 中轴变换
Polygon Partitioning 多边形分割

Simplifying Polygons 多边形化简
Shape Similarity 相似多边形
Motion Planning 运动规划
Maintaining Line Arrangements 平面分割
Minkowski Sum Minkowski 和
Set and String Problems 集合与串的问题
Set Cover 集合覆盖
Set Packing 集合配置
String Matching 模式匹配
Approximate String Matching 模糊匹配
Text Compression 压缩
Cryptography 密码
Longest Common Substring 最长公共子串
Shortest Common Superstring 最短公共父串
robustness 鲁棒性
rate of convergence 收敛速度

数据 Data
数据元素 Data element
数据项 Data item
数据结构 Data structure
逻辑结构 Logical structure
数据类型 Data type
指针 Pointer
顺序存储结构 Sequential storage structure
链状存储结构 Linked storage structure
稠密索引 Dense index
稀疏索引 Sparse index
抽象数据类型 Abstract DataType
算法 Algorithm
正确性 Correctness
可读性 Readability
健壮性 Robustness
频度 Frequency count
时间复杂度 Time complexity
空间复杂度 Space complexity
直接前驱 Immediate predecessor
直接后继 Immediate successor
线性表 Linear list
顺序表 Sequenatial list
单链表 Singly linked list
循环链表 Circylar linked lists
双向链表 Double linked lists
双向循环链表 Double circular linked list

栈 Stack
栈顶 Top
栈底 Botton
后进先出 Last In First Out
上溢 Overflow
下溢 Underflow
共享 Shared
队列 Queue
队尾 Rear
队头 Front
先进后出 First In Last Out
串 String
子串 Substring
模式匹配 Pattern matching
数组 Arrays
行为主序 Row major order
列为主序 Column major order
特殊矩阵 Special matrices
稀疏矩阵 Sparse matrices
三元组表 List of 3_tuples
十字链表 Orthogonal list
广义表 Generalized lists
树 Tree
二叉树 Binary tree
满二叉树 Full binary tree
完全二叉树 Complete binary tree
二叉排序树 Binary sort tree
二叉搜索树 Binary search tree
前序遍历 Preorder traversal
中序遍历 Inorder traversal
后序遍历 Postorder traversal
哈夫曼树 Huffman tree
回溯 Backtrackins
图 Graph
有向图 Directed graph (digraph)
无向图 Undirected graph (undigraph)
有向完全图 Undirected Complete Graph
无向完全图 directed complete graph
稀疏图 Sparse graph
稠密图 Dense graph
网点 Network
邻结点 Adjacent
度 Degree
出度 Outdegree

入度 Indegree
连通图 Connected graph
连通分支 Connected component
强连通图 Strong graph
生成树 Spanning tree
邻接矩阵 Adjacency lists
邻接表 Adjacency lists
邻接多重表 Adjacency multilists
深度优先索引 Depth-First Search
广度优先索引 Breath-First Search
最小生成树 Minimum spanning tree
最短路径 Shortest path
有向无环图 Directed acycline graph
拓扑排序 Topological sort
检索 Searching
关键字 Key
主关键字 Primary key
顺序检索 Sequential search
折半检索 Binary search
分块检索 Blocking search
平衡二叉树 Best wishes, alanced binary tree
平衡因子 Balanced factor
直接定址 Immediately allocate
除留余数法 Division method
数字分析法 Digit analysis method
折叠法 Folding method
线性探查 Linear probing
平方取中法 Mid-square method
开放定址法 Open addressing
链地址法 Chaining
排序 Sorting
直接插入排序 Straight insertion sort
希尔排序 Shells method
缩小增量排序 Diminishing increment sort
折半插入排序 Binary insertion sort
二路插入排序 2_way insertion sort
共享插入排序 Shared insertion sort
冒泡排序 Bubble sort
快速排序 Quick sort
选择排序 Selection sort
直接选择排序 Straight selection sort
树形选择排序 Tree selection sort
锦标赛排序 Tournament sort
堆排序 Heap sort
归并排序 Merging sort
二路归并 2_way merge
多路归并 Multi_way merge
基数排序 Radix sorting
最低位优先(LSD) Least Significant Digit First
最高位优先(MSD) Most Significant Digit First
文件 Files
顺序文件 Sequential file
索引文件 Indexed file
索引顺序存取方法 Indexed Sequential Access Method
虚拟存储存取方法 Virtual Storage Access Method
散列文件 Hashed file
多关键字文件 With more than one key
多重表文件 Multilist file
倒排文件 Inverted file

搜索入门

深度优先搜索 DFS

算法原理

深度优先搜索即 Depth First Search，是图遍历算法的一种。

DFS 的具体算法描述为选择一个起始点 v 作为当前结点，执行如下操作：

a. 访问 当前结点，并且标记该结点已被访问，然后跳转到 b；

b. 如果存在一个和 当前结点 相邻并且尚未被访问的结点 u，则将 u 设为 当前结点，继续执行 a；

c. 如果不存在这样的 u，则进行回溯，回溯的过程就是回退 当前结点；

上述所说的当前结点需要用一个栈来维护，每次访问到的结点入栈，回溯的时候出栈。

算法实现

深搜最简单的实现就是递归，写成伪代码如下：

```
def DFS(v):
    visited[v] = true
    dosomething(v)
    for u in adjcent_list[v]:
      if visited[u] is false:
DFS(u)
```

其中 dosomething 表示访问时具体要干的事情，根据情况而定，并且 DFS 是允许有返回值的。

基础应用

a. 求 N 的阶乘；

b. 求斐波那契数列的第 N 项；

c. 求 N 个数的全排列；

高级应用

a. 枚举：数据范围较小的的排列、组合的穷举；

b. 容斥原理：利用深搜计算一个公式，本质还是做枚举；

c. 基于状态压缩的动态规划：一般解决棋盘摆放问题，k 进制表示状态，然后利用深搜进行状态转移；

d.记忆化搜索：某个状态已经被计算出来，就将它 cache 住，下次要用的时候不需要重新求，此所谓记忆化。下面会详细讲到记忆化搜索的应用范围；

e.有向图强连通分量：经典的 Tarjan 算法；求解 2-sat 问题的基础；

f. 无向图割边割点和双连通分量：经典的 Tarjan 算法；

g. LCA：最近公共祖先递归求解；

h.博弈：利用深搜计算 SG 值；

i.二分图最大匹配：经典的匈牙利算法；最小顶点覆盖、最大独立集、最小值支配集 向二分图的转化；

j.欧拉回路：经典的圈套圈算法；

k. K 短路：依赖数据，数据不卡的话可以采用 2 分答案 + 深搜；也可以用广搜 + A*

l. 线段树：二分经典思想，配合深搜枚举左右子树；

m. 最大团：极大完全子图的优化算法。

n. 最大流：EK 算法求任意路径中有涉及。

o. 树形 DP：即树形动态规划，父结点的值由各个子结点计算得出。

剪枝原则

好的剪枝可以大大提升程序的运行效率

a. 正确性：剪掉的子树中如果存在可行解（或最优解），那么在其它的子树中很可能搜不到解导致搜索失败，所以剪枝的前提必须是要正确；

b. 准确性：剪枝要"准"。所谓"准"，就是要在保证在正确的前提下，尽可能多得剪枝。

c. 高效性：剪枝一般是通过一个函数来判断当前搜索空间是否是一个合法空间，在每个结点都会调用到这个函数，所以这个函数的效率很重要。

剪枝分类

可行性剪枝、最优性剪枝（上下界剪枝）。

1）可行性剪枝

可行性剪枝一般是处理可行解的问题，如一个迷宫，问能否从起点到达目标点之类的。

2）最优性剪枝（上下界剪枝）

即估价函数 h = k + s，那么当前情况下存在最优解的必要条件是 h < d，否则就可以剪枝了。最优性剪枝是不断优化解空间的过程。

基于 DFS 的 A*（迭代加深，IDA*）

在以上剪枝的过程中使用优先队列尽快的搜出可行解以保证最优性剪枝能够剪的尽量多

广度优先搜索

算法原理

广度优先搜索即 Breadth First Search，也是图遍历算法的一种。

BFS 的具体算法描述为选择一个起始点 v 放入一个先进先出的队列中，执行如下操作：

a. 如果队列不为空，弹出一个队列首元素，记为当前结点，执行 b；否则算法结束；

b. 将与 当前结点 相邻并且尚未被访问的结点的信息进行更新，并且全部放入队列中，继续执行 a；

维护广搜的数据结构是队列和 HASH，队列就是官方所说的 open-close 表，HASH 主要是用来标记状态的，比如某个状态并不是一个整数，可能是一个字符串，就需要用字符串映射到一个整数，可以自己写个散列 HASH 表，不建议用 STL 的 map，效率奇低。

广搜最基础的应用是用来求图的最短路。

算法实现

广搜一般用队列维护状态，写成伪代码如下：

```
def BFS(v):
    resetArray(visited,false)
    visited[v] = true
    queue.push(v)
    while not queue.empty():
 v = queue.getfront_and_pop()
 for u in adjcent_list[v]:
      if visited[u] is false:
   dosomething(u)
   queue.push(u)
```

基础应用

a. 最短路： bellman-ford 最短路的优化算法 SPFA，主体是利用 BFS 实现的。

b. 拓扑排序：首先找入度为 0 的点入队，弹出元素执行“减度”操作，继续将减完度后入度为 0 的点入队，循环操作，直到队列为空，经典 BFS 操作；

c. FloodFill：经典洪水灌溉算法；

高级应用

a. 差分约束：数形结合的经典算法，利用 SPFA 来求解不等式组。

b. 稳定婚姻：二分图的稳定匹配问题

c. AC 自动机：字典树 + KMP + BFS，在设定失败指针的时候需要用到 BFS。

d. 矩阵二分：矩阵乘法的状态转移图的构建可以采用 BFS；

e. 基于 k 进制的状态压缩搜索：这里的 k 一般为 2 的幂，状态压缩就是将原本多维的状态压缩到一个 k 进制的整数中，便于存储在一个一维数组中，往往可以大大地节省空间，又由于 k 为 2 的幂，所以状态转移可以采用位运算进行加速；

基于 BFS 的 A*

在搜索的时候，结点信息要用堆（优先队列）维护大小，即能更快到达目标的结点优先弹出。

K 短路问题

求初始结点到目标结点的第 K 短路，当 K=1 时，即最短路问题，K=2 时，则为次短路问题，当 K >= 3 时需要 A*求解。

双向广搜

1) 算法原理

初始状态 和 目标状态 都知道，求初始状态到目标状态的最短距离；

利用两个队列，初始化时初始状态在 1 号队列里，目标状态在 2 号队列里，并且记录这两个状态的层次都为 0，然后分别执行如下操作：

a.若 1 号队列已空，则结束搜索，否则从 1 号队列逐个弹出层次为 K(K >= 0)的状态；

i. 如果该状态在 2 号队列扩展状态时已经扩展到过，那么最短距离为两个队列扩展状态的层次加和，结束搜索；

ii. 否则和 BFS 一样扩展状态，放入 1 号队列，直到队列首元素的层次为 K+1 时执行 b；

b.若 2 号队列已空，则结束搜索，否则从 2 号队列逐个弹出层次为 K(K >= 0)的状态；

i. 如果该状态在 1 号队列扩展状态时已经扩展到过，那么最短距离为两个队列扩展状态的层次加和，结束搜索；

ii. 否则和 BFS 一样扩展状态，放入 2 号队列，直到队列首元素的层次为 K+1 时执行 a；

动态规划

状态和状态转移

在介绍递推和记忆化搜索的时候，都会涉及到一个词---状态，它表示了解决某一问题的中间结果，这是一个比较抽象的概念，无论是递推还是记忆化搜索，首先要设计出合适的状态，然后通过状态的特征建立状态转移方程（f[i] = f[i-1] + f[i-2] 就是一个简单的状态转移方程）。

最优化原理和最优子结构

如果问题的最优解包含的子问题的解也是最优的，就称该问题具有最有子结构，即满足最优化原理。

决策和无后效性

一个状态演变到另一个状态，往往是通过“决策”来进行的。有了“决策”，就会有状态转移。而无后效性，就是一旦某个状态确定后，它之前的状态无法对它之后的状态产生“效应”（影响）。

动态规划的经典模型

1、线性模型

线性模型的是动态规划中最常用的模型，上文讲到的最长单调子序列就是经典的线性模型，这里的线性指的是状态的排布是呈线性的。

2、区间模型

区间模型的状态表示一般为 d[i][j]，表示区间[i, j]上的最优解，然后通过状态转移计算出[i+1, j]或者[i, j+1]上的最优解，逐步扩大区间的范围，最终求得[1, len]的最优解。

3、背包模型

背包问题是动态规划中一个最典型的问题之一。

有 N 种物品（每种物品 1 件）和一个容量为 V 的背包。放入第 i 种物品耗费的空间是 Ci，得到的价值是 Wi。求解将哪些物品装入背包可使价值总和最大。f[i][v]表示前 i 种物品恰好放入一个容量为 v 的背包可以获得的最大价值。决策为第 i 个物品在前 i-1 个物品放置完毕后，是选择放还是不放，状态转移方程为：f[i][v] = max{ f[i-1][v], f[i-1][v - Ci] +Wi }时间复杂度 O(VN)，空间复杂度 O(VN) （空间复杂度可利用滚动数组进行优化达到 O(V)，下文会介绍滚动数组优化）。

4、状态压缩模型

状态压缩的动态规划，一般处理的是数据规模较小的问题，将状态压缩成 k 进制的整数，k 取 2 时最为常见。

5、树状模型

树形动态规划（树形 DP），是指状态图是一棵树，状态转移也发生在树上，父结点的值通过所有子结点计算完毕后得出。给定一颗树，和树上每个结点的权值，求一颗非空子树，使得权和最大。

用 d[1][i] 表示 i 这个结点选中的情况下，以 i 为根的子树的权和最大值;

用 d[0][i]表示 i 这个结点不选中的情况下，以 i 为根的子树的权和最大值;

d[1][i] = v[i] + sum{ d[1][v] | v 是 i 的直接子结点 && d[1][v] > 0

d[0][i] = max(0, max{ max(d[0][v], d[1][v]) | v 是 i 的直接子结点 })

这样，构造一个以 1 为根结点的树，然后就可以通过 dfs 求解了。

这题题目要求求出的树为非空树，所以当所有权值都为负数的情况下需要特殊处理，选择所有权值中最大的那个作为答案。

动态规划算法三要素

（摘自黑书，总结的很好，很有概括性）:

①所有不同的子问题组成的表

②解决问题的依赖关系可以看成是一个图

③填充子问题的顺序（即对②的图进行拓扑排序，填充的过程称为状态转移）;

则如果子问题的数目为 O(nt)，每个子问题需要用到 O(ne)个子问题的结果，那么我们称它为 tD/eD 的问题，于是可以总结出四类常用的动态规划方程：下面会把 opt 作为取最优值的函数（一般取 min 或 max），w(j, i)为一个实函数，其它变量都可以在常数时间计算出来

1、1D/1D

d[i] = opt{ d[j] + w(j, i) | 0 <= i < j } (1 <= i <= n)

2、2D/0D

d[i][j] = opt{ d[i-1][j] + xi, d[i][j-1] + yj, d[i-1][j-1] + zij }　　(1<= i, j <= n)

3、2D/1D

d[i][j] = w(i, j) + opt{ d[i][k-1] + d[k][j] }, (1 <= i < j <= n)

区间模型常用方程。

另外一种常用的 2D/1D 的方程为：

d[i][j] = opt{ d[i-1][k] + w(i, j, k) | k < j }　　(1<= i <= n, 1 <= j <= m)

4、2D/2D

d[i][j] = opt{ d[i'][j'] + w(i', j', i, j) |　0 <= i' < i, 0 <= j' < j}

常见于二维的迷宫问题，由于复杂度比较大，所以一般配合数据结构优化，如线段树、树状数组等。

对于一个 tD/eD 的动态规划问题，在不经过任何优化的情况下，可以粗略得到一个时间复杂度是 O(nt+e)，空间复杂度是 O(nt)的算法，大多数情况下空间复杂度是很容易优化的，难点在于时间复杂度，下一章我们将详细讲解各种情况下的动态规划优化算法。

动态规划和数据结构结合的常用优化

1、滚动数组

我们发现将 d[i][j]理解成一个二维的矩阵，i 表示行，j 表示列，那么第 i 行的结果只取决于第 i+1 和第 i 行的情况，对于第 i+2 行它表示并不关心，那么我们只要用一个 d[2][N]的数组就能保存状态了，其中 d[0][N]为奇数行的状态值，d[1][N]为偶数行的状态值，当前需要计算的状态行数为奇数时，会利用到 d[1][N]的部分状态，奇数行计算完毕，d[1][N]整行状态都没用了，可以用于下一行状态的保存，类似“传送带”的滚动来循环利用空间资源，美其名曰 - 滚动数组。

这是个 2D/0D 问题，理论的空间复杂度是 O(n2)，利用滚动数组可以将空间降掉一维，变成 O(n)。背包问题的几个状态转移方程同样可以用滚动数组进行空间优化。

2、最长单调子序列

d[i] = max{ d[j] | j < i && a[j] < a[i] } + 1;

那个问题的状态转移方程如下：

最长递增子序列的 N 变成 100000，其余不变。

首先明确决策的概念，我们认为 j 和 k (j < i, k < i)都是在计算 d[i]时的两个决策。那么假设他们满足 a[j] < a[k]（它们的状态对应就是 d[j] 和 d[k]），如果 a[i] > a[k]，则必然有 a[i] > a[j]，能够选 k 做决策的也必然能够选 j 做决策，那么如若此时 d[j] >= d[k]，显然 k 不可能是最优决策（j 的决策始终比它优，以 j 做决策，a[j]的值小但状态值却更大），所以 d[k]是不需要保存的。基于以上理论，我们可以采用二分枚举，维护一个值（这里的值指的是 a[i]）递增的决策序列，不断扩大决策序列，最后决策的数目就是最长递增子序列的长度。具体做法是：

枚举 i，如果 a[i]比决策序列中最大的元素的值还大，则将 i 插入到决策序列的尾部；否则二分枚举决策序列，找出其中值最小的一个决策 k，并且满足 a[k] > a[i]，然后用决策 i 替换决策 k。

这是个 1D/1D 问题，理论的时间复杂度是 O(n2)，利用单调性优化后可以将复杂度将至 O(nlogn)。

给定 n 个元素(n <= 100000)的序列，将序列的所有数分成 x 堆，每堆都是单调不增的，求 x 的最小值。

结论：可以转化成求原序列的最长递增子序列。

证明：因为这 x 堆中每堆元素都是单调不增的，所以原序列的最长递增子序列的每个元素在分出来的每堆元素中一定只出现最多一个，那么最长递增子序列的长度 L 的最大值为 x，所以 x >= L。而我们要求的是 x 的最小值，于是这个最小值就是 L 了。

3、矩阵优化

n 的范围比较大，虽然是几个简单的加法方程，但是一眼看下去也不知道有什么规律可循。我们把状态转移用另外一种形式表现出来，存储图的连通信息的一种方法就是矩阵。令这个矩阵为 A，Aij 表示从 i 号岛到 j 号岛是否连通，连通标 1，不连通标 0，它还有另外一个含义，就是经过 1 天，从 i 岛到 j 岛的方案数，利用矩阵的传递性，A2 的第 i 行的第 j 列则表示经过 2 天，从 i 岛到 j 岛的方案数，同样的，An 则表示了经过 n 天，从 i 岛到 j 岛的方案数，那么问题就转化成了求 An MOD 100000007 的值了。An 当 n 为偶数的时候等于(An/2)*(An/2)；当 n 为奇数的时候，等于(An/2)*(An/2)*A，这样求解矩阵 An 就可以在 O(logn)的时间内完成了，加法和乘法对 MOD 操作都是可交换的（即 “先加再模” 和 “先模再加再模” 等价），所以可以在矩阵乘法求解的时候，一旦超出模数，就执行取模操作。

最后求得的矩阵 T = An MOD 100000007，那么 T[1][1]就是我们要求的解了。

4、斜率优化

那么可以用单调队列来维护一个决策队列的单调性，单调队列存的是决策序列。

一开始队列里只有一个决策，就是 0 这个点（虚拟出的初始决策），根据第一个结论，如果队列里面决策数目大于 1，则判断 slope(Q[front], Q[front+1]) < 2*s[i]是否成立，如果成立，Q[front]是个无用决策，front ++，如果不成立那么 Q[front]必定是当前 i 的最优决策，通过状态转移方程计算 f[i]的值，然后再根据第二个结论，判断 slope(Q[rear-2], Q[rear-1]) > slope(Q[rear-1], i)是否成立，如果成立，那么 Q[rear-1]必定是个无用决策，rear --，如果不成立，则将 i 作为当前决策 插入到队列尾， 即 Q[rear++] = i。

这题需要注意，斜率计算的时候，分母有可能为 0 的情况。

5、树状数组优化

树状数组是一种数据结构，它支持两种操作：

1、对点更新，即在给你(x, v)的情况下，在下标 x 的位置累加一个和 v（耗时 O(log(n)) ）。

函数表示 void add(x, v);

2、成端求和，给定一段区间[x, y]，求区间内数据的和（这些数据就是第一个操作累加的数据，耗时 O(log(n)) ）。

函数表示 int sum(x, y);

用其它数据结构也是可以实现上述操作的，例如线段树（可以认为它是一种轻量级的线段树，但是线段树能解决的问题更加普遍，而树状数组只能处理求和问题），但是树状数组的实现方便、常数复杂度低，使得它在解决对点更新成端求和问题上成为首选。这里并不会讲它的具体实现，有兴趣请参见树状数组。

6、线段树优化

线段树是一种完全二叉树，它支持区间求和、区间最值等一系列区间问题，这里为了将问题简化，直接给出求值函数而暂时不去讨论它的具体实现，有兴趣的可以自行寻找资料。线段树可以扩展到二维，二维线段树是一棵四叉树，一般用于解决平面统计问题，参见二维线段树。

7、其他优化

a.散列 HASH 状态表示

b.结合数论优化

c.结合计算几何优化

d.四边形不等式优化

e.等等

目录

树状数组

数据结构的设计

我们现在假设有这样一种数据结构，可以支持以下三种操作：

1、插入(Insert)，将一个数字插入到该数据结构中；

2、删除(Delete)，将某个数字从该数据结构中删除；

3、询问(Query)，询问该数据结构中存在数字的中位数；

如果这三个操作都能在 O(log(n))或者 O(1)的时间内完成，那么这个问题就可以完美解决了。具体做法是：

首先将 a[1...2r+1]这些元素都插入到该数据结构中，然后询问中位数替换掉 a[r+1]，再删除 a[1]，插入 a[2r+2]，询问中位数替换掉 a[r+2]，以此类推，直到计算完第 n-r 个元素。所有操作都在 O(log(n))时间内完成的话，总的时间复杂度就是 O(nlogn)。

我们来看什么样的数据结构可以满足这三条操作都在 O(log(n))的时间内完成，考虑每个数字的范围是[0, 255] ,如果我们将这些数字映射到一个线性表中(即 HASH 表) ,插入和删除操作都可以做到 O(1)。

树状数组华丽登场

这里引入一种数据结构 - 树状数组（Binary Indexed Tree，BIT，二分索引树 ），它只有两种基本操作，并且都是操作线性表的数据的：

1、add(i, 1)　(1<=i<=n)　对第 i 个元素的值自增 1　　O(logn)

2、sum(i)　　(1<=i<=n)　统计[1...i]元素值的和　　O(logn)

有了这两种操作，我们需要将它们转化成之前设计的数据结构的那三种操作，首先：

1、插入(Insert)，对应的是 add(i, 1)，时间复杂度 O(logn)

2、删除(Delete)，对应的是 add(i, -1)，时间复杂度 O(logn)

3、询问(Query)，由于 sum(i)能够统计[1...i]元素值的和，换言之，它能够得到我们之前插入的数据中小于等于 i 的数的个数，那么如果能够知道 sum(i) >= r + 1 的最小的 i，那这个 i 就是所有插入数据的中位数了(因为根据上文的条件，插入的数据时刻保证有 2r+1 个)。因为 sum(i)是关于 i 的递增函数，所以基于单调性我们可以二分枚举 i (1 <= i <= n)，得到最小的 i 满足 sum(i) >= r + 1，每次的询问复杂度就是 O(logn * logn)。 一个 logn 是二分枚举的复杂度，另一个 logn 是 sum 函数的复杂度。

这样一来，一维的 Median Filter 模型的整体时间复杂度就降到了 O(n * logn * logn)，已经是比较高效的算法了。

细说树状数组

1、树 or 数组 ？

名曰树状数组，那么究竟它是树还是数组呢？数组在物理空间上是连续的，而树是通过父子关系关联起来的，而树状数组正是这两种关系的结合，首先在存储空间上它是以数组的形式存储的，即下标连续；其次，对于两个数组下标 x，y(x < y)，如果 x + 2^k = y (k 等于 x 的二进制表示中末尾 0 的个数)，那么定义(y, x)为一组树上的父子关系，其中 y 为父结点，x 为子结点。

2、结点的含义

然后我们来看树状数组上的结点 Ci 具体表示什么，这时候就需要利用树的递归性质了。我们定义 Ci 的值为它的所有子结点的值 和 Ai 的总和，之前提到当 i 为奇数时 Ci 一定为叶子结点，所以有 Ci = Ai (i 为奇数)

3、求和操作

明白了 Ci 的含义后，我们需要通过它来求 sum{ A[j] | 1 <= j <= i }，也就是之前提到的 sum(i)函数。为了简化问题，用一个函数 lowbit(i)来表示 2^k (k 等于 i 的二进制表示中末尾 0 的个数)。那么：

sum(i) = sum{ A[j] | 1 <= j <= i }

= A[1] + A[2] + ... + A[i]

= A[1] + A[2] + A[i-2^k] + A[i-2^k+1] + ... + A[i]
= A[1] + A[2] + A[i-2^k] + C[i]
= sum(i - 2^k) + C[i]
= sum(i - lowbit(i)) + C[i]

由于 C[i]已知，所以 sum(i)可以通过递归求解，递归出口为当 i = 0 时，返回 0。sum(i)函数的函数主体只需要一行代码：

```
int sum(int x){
    return x ? C[x]+ sum( x - lowbit(x)):0;
 }
```

观察 i - lowbit(i)，其实就是将 i 的二进制表示的最后一个 1 去掉，最多只有 log(i)个 1，所以求 sum(n)的最坏时间复杂度为 O(logn)。由于递归的时候常数开销比较大，所以一般写成迭代的形式更好。写成迭代代码如下：

```
int sum(int x){
    int s =0;
    for(int i = x; i ; i -= lowbit(i)){
        s += c[i][j];
    }
    return s;
}
```

4、更新操作

更新操作就是之前提到的 add(i, 1) 和 add(i, -1)，更加具体得，可以推广到 add(i, v)，表示的其实就是 A[i] = A[i] + v。但是我们不能在原数组 A 上操作，而是要像求和操作一样，在树状数组 C 上进行操作。递归的时候常数开销比较大，所以一般写成迭代的形式更好。写成迭代形式的代码如下

```
void add(int x,int v){
    for(int i = x; i <= n; i += lowbit(i)){
        C[i] += v;
    }
}
```

5、lowbit 函数 O(1)实现

上文提到的两个函数 sum(x)和 add(x, v)都是用递归实现的，并且都用到了一个函数叫 lowbit(x)，表示的是 2k，其中 k 为 x 的二进制表示末尾 0 的个数，那么最简单的实现办法就是通过位运算的右移，循环判断最后一位是 0 还是 1，从而统计末尾 0 的个数，一旦发现 1 后统计完毕，计数器保存的值就是 k，当然这样的做法总的复杂度为 O(logn)，一个 32 位的整数最多可能进行 31 次判断（这里讨论整数的情况，所以符号位不算）。

这里介绍一种 O(1)的方法计算 2k 的方法。由于&的优先级低于-，所以代码可以这样写：

```
int lowbit(int x) {
    return x & -x;
}
```

6、小结

至此，树状数组的基础内容就到此结束了，三个函数就诠释了树状数组的所有内容，并且都只需要一行代码实现，单次操作的时间复杂度为 O(log(n))，空间复杂度为 O(n)，所以它是一种性价比非常高的轻量级数据结构。

树状数组解决的基本问题是 单点更新，成端求和。上文中的 sum(x)求的是[1, x]的和，如果需要求[x, y]的和，只需要求两次 sum 函数，然后相减得到，即 sum(y) - sum(x-1)。

下面一节会通过一些例子来具体阐述树状数组的应用场景。

树状数组的经典模型

1、PUIQ 模型

【例题 1】一个长度为 n(n <= 500000)的元素序列，一开始都为 0，现给出三种操作：

1. add x v： 给第 x 个元素的值加上 v； (a[x] += v)
2. sub x v： 给第 x 个元素的值减去 v； (a[x] -= v)
3. sum x y： 询问第 x 到第 y 个元素的和； (print sum{ a[i] | x <= i <= y })

这是树状数组最基础的模型，1 和 2 的操作就是对应的单点更新，3 的操作就对应了成端求和。

具体得，1 和 2 只要分别调用 add(x, v)和 add(x, -v)，而 3 则是输出 sum(y) - sum(x-1)的值。

我把这类问题叫做 PUIQ 模型(Point Update Interval Query 点更新，段求和)。

2、IUPQ 模型

【例题 2】一个长度为 n(n <= 500000)的元素序列，一开始都为 0，现给出两种操作：

1. add x y v： 给第 x 个元素到第 y 个元素的值都加上 v； (a[i] += v, 其中 x <= i <= y)
2. get x： 询问第 x 个元素的值； (print a[x])

这类问题对树状数组稍微进行了一个转化，但是还是可以用 add 和 sum 这两个函数来解决，对于操作 1 我们只需要执行两个操作，即 add(x, v)和 add(y+1, -v)；而操作 2 则是输出 sum(x)的值。

这样就把区间更新转化成了单点更新，单点求值转化成了区间求和。

我把这类问题叫做 IUPQ 模型(Interval Update Point Query 段更新，点求值)。

3、逆序模型

【例题 3】给定一个长度为 n(n <= 500000)的排列 a[i]，求它的逆序对对数。1 5 2 4 3 的逆序对为(5,2)(5,3)(5,4)(4,3)，所以答案为 4。

朴素算法，枚举任意两个数，判断他们的大小关系进行统计，时间复杂度 O(n2)。不推荐！

来看一个给定 n 个元素的排列 X0 X1 X2 … Xn-2 Xn-1,对于某个 Xi 元素，如果想知道以它为"首"的逆序对的对数(形如(XiXj) 的逆序对)，就是需要知道 Xi+1 … Xn-2 Xn-1 这个子序列中小于 Xi 的元素的个数。

那么我们只需要对这个排列从后往前枚举，每次枚举到 Xi 元素时，执行 cnt += sum(Xi-1)，然后再执行 add(Xi, 1) ,n 个元素枚举完毕 ,得到的 cnt 值就是我们要求的逆序数了。总的时间复杂度 O(nlogn)。

这个模型和之前的区别在于它不是将原数组的下标作为树状数组的下标，而是将元素本身作为树状数组的下标。逆序模型作为树状数组的一个经典思想有着非常广泛的应用。

4、二分模型

【例题 5】给定 N(N <= 100000)个编号为 1-N 的球，将它们乱序丢入一个“神奇的容器”中，作者会在丢的同时询问其中编号第 K 大的那个球，“神奇的容器”都能够从容作答 ,并且将那个球给吐出来，然后下次又可以继续往里丢。

现在要你来模拟这个神奇的容器的功能。可以抽象成两种操作：

1. put x 向容器中放入一个编号为 x 的球；
2. query K 询问容器中第 K 大的那个球，并且将那个球从容器中去除 (保证 K<容器中球的总数)；

这个问题其实就是一维 Median Filter 的原型了，只是那个问题的 K = r+1，而这里的 K 是用户输入的一个常量。所谓二分模型就是在求和的过程中，利用求和函数的单调性进行二分枚举。

对于操作 1，只是单纯地执行 add(x, 1)即可；而对于操作 2，我们要看第 K 大的数满足什么性质，由于这里的数字不会有重复，所以一个显而易见的性质就是一定有 K-1 个数大于它，假设它的值为 x，那么必然满足下面的等式：sum(N) - sum(x) == K-1，然而，满足这个等式的 x 可能并不止一个。灰色的格子表示容器中的球，分别为 2、3、7、8，然后我们需要求第 3 大的球，理论的球编号为 3，但是满足上

面等式的球的编号为 3、4、5、6。所以我们需要再加一个限制条件，即满足上面等式的最小的 x。于是我们二分枚举 x，当满足 sum(N) - sum(x) <= K-1 时，将右区间缩小（说明找的数 x 偏大，继续往小的找），否则左区间增大(说明找的数 x 偏小，继续往大的找)，直到找到满足条件的最小的 x 为止。单次操作的时间复杂度为 O(logn * logn)。

5、再说 Median Filter

基于二分模型的一维 Median Filter 问题已经圆满解决了，那么最后让我们回到二维的 Median Filter 问题上来。

6、多维树状数组模型

给定一个 N*N(N <= 1000)的矩形区域，执行两种操作：

1. add x y v　　在(x, y)加上一个值 v；
2. sum x1 y1 x2 y2　　统计矩形(x1, y1) - (x2, y2)中的值的和；

PUIQ 模型的二维版本。我们设计两种基本操作：

1. add(x, y, v)　　在(x, y)这个格子加上一个值 v；
2. sum(x, y)　　求矩形区域(1, 1) - (x, y)内的值的和，那么(x1,y1)-(x2,y2)区域内的和可以通过四个求和操作获得，即 sum(x2, y2) - sum(x2, y1 - 1) - sum(x1 - 1, y2) + sum(x1 - 1, y1 - 1)。（利用容斥原理的基本思想）

add(x, y, v)和 sum(x, y)可以利用二维树状数组实现，二维树状数组可以理解成每个 C 结点上又是一棵树状数组（可以从二维数组的概念去理解，即数组的每个元素都是一个数组），具体代码如下：

```
void add(int x,int y,int v){
    for(int i = x; i <= n; i += lowbit(i)){
        for(int j = y; j <= n; j += lowbit(j)){
            c[i][j]+= v;
        }
    }
}

int sum(int x,int y){
    int s =0;
    for(int i = x; i ; i -= lowbit(i)){
        for(int j = y; j ; j -= lowbit(j)){
            s += c[i][j];
        }
    }
    return s;
 }
```

仔细观察即可发现，二维树状数组的实现和一维的实现极其相似，二维仅仅比一维多了一个循环，并且数据用二维数组实现。那么同样地，对于三维的情况，也只是在数组的维度上再增加一维，更新和求和时都各加一个循环而已。

差分约束

引例

一类不等式组的解

给定 n 个变量和 m 个不等式，每个不等式形如 x[i] - x[j] <= a[k] (0 <= i, j < n, 0 <= k < m，a[k]已知)，求 x[n-1] - x[0] 的最大值。例如当 n = 4，m = 5，不等式组如图一-1-1 所示的情况，求 x3 - x0 的最大值。

$$x1 - x0 <= 2 \qquad (1)$$
$$x2 - x0 <= 7 \qquad (2)$$
$$x3 - x0 <= 8 \qquad (3)$$
$$x2 - x1 <= 3 \qquad (4)$$
$$x3 - x2 <= 2 \qquad (5)$$

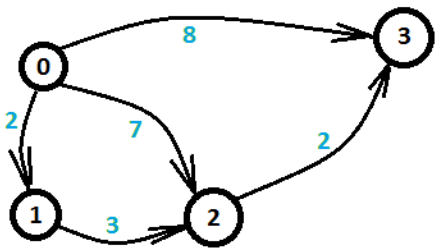
观察 x3 - x0 的性质，我们如果可以通过不等式的两两加和得到 c 个形如 x3 - x0 <= Ti 的不等式，那么 min{ Ti | 0 <= i < c } 就是我们要求的 x3 - x0 的最大值。

整理出以下三个不等式：

1. (3) x3 - x0 <= 8
2. (2) + (5) x3 - x0 <= 9
3. (1) + (4) + (5) x3 - x0 <= 7

这里的 T 等于{8， 9， 7}，所以 min{ T } = 7，答案就是 7。

这种方法即使做出来了还是带有问号的，不能确定正确与否，如何系统地解决这类问题呢？



给定四个小岛以及小岛之间的有向距离，问从第 0 个岛到第 3 个岛的最短距离。箭头指向的线段代表两个小岛之间的有向边，蓝色数字代表距离权值。

发现总共三条路线，如下：

1. 0 -> 3 长度为 8
2. 0 -> 2 -> 3 长度为 7+2 = 9
3. 0 -> 1 -> 2 -> 3 长度为 2 + 3 + 2 = 7

最短路为三条线路中的长度的最小值即 7，所以最短路的长度就是 7。

最短路

1、Dijkstra

对于一个有向图或无向图，所有边权为正（边用邻接矩阵的形式给出），给定 a 和 b，求 a 到 b 的最短路，保证 a 一定能够到达 b。这条最短路一定存在。相反，最长路就不一定了，由于边权为正，如果遇到有环的时候，可以一直在这个环上走，因为要找最长的，这样就使得路径越变越长，永无止境，所以对

于正权图，在可达的情况下最短路一定存在，最长路则不一定存在。这里只讨论正权图的最短路问题。最短路满足最优子结构性质，所以是一个动态规划问题。最短路的最优子结构可以描述为：

D(s, t) = {Vs ... Vi ... Vj ... Vt}表示 s 到 t 的最短路，其中 i 和 j 是这条路径上的两个中间结点，那么 D(i, j)必定是 i 到 j 的最短路，这个性质是显然的，可以用反证法证明。基于上面的最优子结构性质，如果存在这样一条最短路 D(s, t) = {Vs ... Vi Vt}，其中 i 和 t 是最短路上相邻的点，那么 D(s, i) = {Vs ... Vi} 必定是 s 到 i 的最短路。Dijkstra 算法就是基于这样一个性质，通过最短路径长度递增，逐渐生成最短路。

Dijkstra 算法是最经典的最短路算法，用于计算正权图的单源最短路 SSSP(Single Source Shortest Path，源点给定，通过该算法可以求出起点到所有点的最短路)，它是基于这样一个事实：如果源点到 x 点的最短路已经求出，并且保存在 d[x] (可以将它理解为 D(s, x))上，那么可以利用 x 去更新 x 能够直接到达的点 的最短路。即：

d[y] = min{ d[y], d[x] + w(x, y) }　　y 为 x 能够直接到达的点，w(x, y) 则表示 x->y 这条有向边的边权

具体算法描述如下：对于图 G = <V, E>，源点为 s，d[i]表示 s 到 i 的最短路，visit[i]表示 d[i]是否已经确定(布尔值)。

1) 初始化 所有顶点 d[i] = INF, visit[i] = false，令 d[s] = 0；

2) 从所有 visit[i]为 false 的顶点中找到一个 d[i]值最小的，令 x = i; 如果找不到，算法结束；

3) 标记 visit[x] = true, 更新和 x 直接相邻的所有顶点 y 的最短路： d[y] = min{ d[y], d[x] + w(x, y) }

(第三步中如果 y 和 x 并不是直接相邻，则令 w(x, y) = INF)

2、图的存储

以上算法的时间复杂度为 O(n^2)，n 为结点个数，即每次找一个 d[i]值最小的，总共 n 次，每次找到后对其它所有顶点进行更新，更新 n 次。由于算法复杂度是和点有关，并且平方级别的，所以还是需要考虑一下点数较多而边数较少的情况。

邻接矩阵是直接利用一个二维数组对边的关系进行存储，矩阵的第 i 行第 j 列的值 表示 i -> j 这条边的权值；特殊的，如果不存在这条边，用一个特殊标记来表示；如果 i == j，则权值为 0。它的优点是实现非常简单，而且很容易理解；缺点也很明显，如果这个图是一个非常稀疏的图，图中边很少，但是点很多，就会造成非常大的内存浪费，点数过大的时候根本就无法存储。

邻接表是图中常用的存储结构之一，每个顶点都有一个链表，这个链表的数据表示和当前顶点直接相邻的顶点 (如果边有权值，还需要保存边权信息)。邻接表的优点是对于稀疏图不会有数据浪费，缺点就是实现相对麻烦，需要自己实现链表，动态分配内存。

前向星是以存储边的方式来存储图，先将边读入并存储在连续的数组中，然后按照边的起点进行排序，这样数组中起点相等的边就能够在数组中进行连续访问了。它的优点是实现简单，容易理解，缺点是需要在所有边都读入完毕的情况下对所有边进行一次排序，带来了时间开销，实用性也较差，只适合离线算法。

3、链式前向星

链式前向星和邻接表类似，也是链式结构和线性结构的结合，每个结点 i 都有一个链表，链表的所有数据是从 i 出发的所有边的集合 (对比邻接表存的是顶点集合)，边的表示为一个四元组(u, v, w, next)，其中(u, v)代表该条边的有向顶点对，w 代表边上的权值，next 指向下一条边。

具体的，我们需要一个边的结构体数组 edge[MAXM]，MAXM 表示边的总数，所有边都存储在这个结构体数组中，并且用 head[i]来指向 i 结点的第一条边。

边的结构体声明如下：

```
struct EDGE {
    int u, v, w, next;
    EDGE() {}
```

```
        EDGE(int _u, int _v, int _w, int _next) {
            u = _u, v = _v, w = _w, next = _next;
        }
    }edge[MAXM];
```

初始化所有的 head[i] = INF，当前边总数 edgeCount = 0

每读入一条边，调用 addEdge(u, v, w)，具体函数的实现如下：

```
void addEdge(int u, int v, int w) {
    edge[ edgeCount ] = EDGE(u, v, w, head[u]);
    head[u] = edgeCount ++;
}
```

这个函数的含义是每加入一条边(u, v)，就在原有的链表结构的首部插入这条边，使得每次插入的时间复杂度为 O(1)，所以链表的边的顺序和读入顺序正好是逆序的。这种结构在无论是稠密的还是稀疏的图上都有非常好的表现，空间上没有浪费，时间上也是最小开销。

调用的时候只要通过 head[i]就能访问到由 i 出发的第一条边的编号，通过编号到 edge 数组进行索引可以得到边的具体信息，然后根据这条边的 next 域可以得到第二条边的编号，以此类推，直到 next 域为 INF（这里的 INF 即 head 数组初始化的那个值，一般取-1 即可）。

4、Dijkstra + 优先队列(最小二叉堆)

有了链式前向星，再来看 Dijkstra 算法，我们关注算法的第 3)步，对和 x 直接相邻的点进行更新的时候，不再需要遍历所有的点，而是只更新和 x 直接相邻的点，这样总的更新次数就和顶点数 n 无关了，总更新次数就是总边数 m，算法的复杂度变成了 O(n^2 + m)，之前的复杂度是 O(n^2)，但是有两个 n^2 的操作，而这里是一个，原因在于找 d 值最小的顶点的时候还是一个 O(n)的轮询，总共 n 次查找。那么查找 d 值最小有什么好办法呢？

数据结构中有一种树，它能够在 O(log(n))的时间内插入和删除数据，并且在 O(1)的时间内得到当前数据的最小值。 在 C++中，可以利用 STL 的优先队列(priority_queue)来实现获取最小值的操作。

用 n 表示点数，m 表示边数，那么优先队列中最多可能存在的点数有多少？因为我们在把顶点插入队列的时候并没有判断队列中有没有这个点，而且也不能进行这样的判断，因为新插入的点一定会取代之前的点（距离更短才会执行插入），所以同一时间队列中的点有可能重复，插入操作的上限是 m 次，所以最多有 m 个点，那么一次插入和删除的操作的平摊复杂度就是 O(logm)，但是每次取距离最小的点，对于有多个相同点的情况，如果那个点已经出过一次队列了，下次同一个点出队列的时候它对应的距离一定比之前的大，不需要用它去更新其它点，因为一定不可能更新成功，所以真正执行更新操作的点的个数其实只有 n 个，所以总体下来的平均复杂度为 O((m+n)log m)，而这个只是理论上界，一般问题中都是很快就能找到最短路的，所以实际复杂度会比这个小很多，相比 O(n^2)的算法已经优化了很多了。

Dijkstra 算法求的是正权图的单源最短路问题，对于权值有负数的情况就不能用 Dijkstra 求解了，因为如果图中存在负环，Dijkstra 带优先队列优化的算法就会进入一个死循环，因为可以从起点走到负环处一直将权值变小 。对于带负权的图的最短路问题就需要用到 Bellman-Ford 算法了。

5、Bellman-Ford

Bellman-Ford 算法可以在最短路存在的情况下求出最短路，并且在存在负权圈的情况下告诉你最短路不存在，前提是起点能够到达这个负权圈，因为即使图中有负权圈，但是起点到不了负权圈，最短路还是有可能存在的。它是基于这样一个事实：一个图的最短路如果存在，那么最短路中必定不存在圈，所以最短路的顶点数除了起点外最多只有 n-1 个。

Bellman-Ford 同样也是利用了最短路的最优子结构性质，用 d[i]表示起点 s 到 i 的最短路，那么边数上限为 j 的最短路可以通过边数上限为 j-1 的最短路 加入一条边 得到，通过 n-1 次迭代，最后求得 s 到所有点的最短路。

具体算法描述如下：对于图 G = <V, E>，源点为 s，d[i]表示 s 到 i 的最短路。

1) 初始化 所有顶点 d[i] = INF, 令 d[s] = 0，计数器 j = 0；

2) 枚举每条边(u, v)，如果 d[u]不等于 INF 并且 d[u] + w(u, v) < d[v]，则令 d[v] = d[u] + w(u, v)；

3) 计数器 j + +，当 j = n - 1 时算法结束，否则继续重复 2)的步骤；

第 2)步的一次更新称为边的“松弛”操作。

以上算法并没有考虑到负权圈的问题，如果存在负圈权，那么第 2)步操作的更新会永无止境，所以**判定负权圈的算法也就出来了，只需要在第 n 次继续进行第 2)步的松弛操作，如果有至少一条边能够被更新，那么必定存在负权圈。**

这个算法的时间复杂度为 O(nm)，n 为点数，m 为边数。

这里有一个小优化，我们可以注意到第 2)步操作，每次迭代第 2)步操作都是做同一件事情，也就是说如果第 k(k <= n-1)次迭代的时候没有任何的最短路发生更新，即所有的 d[i]值都未发生变化，那么第 k+1 次必定也不会发生变化了，也就是说这个算法提前结束了。所以可以在第 2)操作开始的时候记录一个标志，标志初始为 false，如果有一条边发生了松弛，那么标志置为 true，所有边枚举完毕如果标志还是 false 则提前结束算法。

这个优化在一般情况下很有效，因为往往最短路在前几次迭代就已经找到最优解了，但是也不排除上文提到的负权圈的情况，会一直更新，使得整个算法的时间复杂度达到上限 O(nm)，那么如何改善这个算法的效率呢？接下来介绍改进版的 Bellman-Ford —— SPFA。

6、SPFA

SPFA(Shortest Path Faster Algorithm)是基于 Bellman-Ford 的思想，采用先进先出(FIFO)队列进行优化的一个计算单源最短路的快速算法。

类似 Bellman-Ford 的做法，我们用数组 d 记录每个结点的最短路径估计值，并用链式前向星来存储图 G。利用一个先进先出的队列用来保存待松弛的结点，每次取出队首结点 u，并且枚举从 u 出发的所有边(u, v)，如果 d[u] + w(u, v) < d[v]，则更新 d[v] = d[u] + w(u, v)，然后判断 v 点在不在队列中，如果不在就将 v 点放入队尾。这样不断从队列中取出结点来进行松弛操作，直至队列空为止。

只要最短路径存在，SPFA 算法必定能求出最小值。因为每次将点放入队尾，都是经过松弛操作达到的。即每次入队的点 v 对应的最短路径估计值 d[v]都在变小。所以算法的执行会使 d 越来越小。由于我们假定最短路一定存在，即图中没有负权圈，所以每个结点都有最短路径值。因此，算法不会无限执行下去，随着 d 值的逐渐变小，直到到达最短路径值时，算法结束，这时的最短路径估计值就是对应结点的最短路径值。

那么最短路径不存在呢？如果存在负权圈，并且起点可以通过一些顶点到达负权圈，那么利用 SPFA 算法会进入一个死循环，因为 d 值会越来越小，并且没有下限，使得最短路不存在。那么我们假设不存在负权圈，则任何最短路上的点必定小于等于 n 个（没有圈），换言之，用一个数组 c[i]来记录 i 这个点入队的次数，所有的 c[i]必定都小于等于 n，所以一旦有一个 c[i] > n，则表明这个图中存在负权圈。

接下来给出 SPFA 更加直观的理解，假设图中所有边的边权都为 1，那么 SPFA 其实就是一个 BFS（Breadth First Search，广度优先搜索）BFS 首先到达的顶点所经历的路径一定是最短路(也就是经过的最少顶点数)，所以此时利用数组记录节点访问可以使每个顶点只进队一次，但在至少有一条边的边权不为 1 的带权图中，最先到达的顶点的路径不一定是最短路，这就是为什么要用 d 数组来记录当前最短路估计值的原因了。

SPFA 算法的最坏时间复杂度为 O(nm)，其中 n 为点数，m 为边数，但是一般不会达到这个上界，一般的期望时间复杂度为 O(km)， k 为常数，m 为边数（这个时间复杂度只是估计值，具体和图的结构有很大关系，而且很难证明，不过可以肯定的是至少比传统的 Bellman-Ford 高效很多，所以一般采用 SPFA 来求解带负权圈的最短路问题）。

7、Floyd-Warshall

最后介绍一个求任意两点最短路的算法，很显然，我们可以求 n 次单源最短路（枚举起点），但是下

面这种方法更加容易编码，而且很巧妙，它也是基于动态规划的思想。

令 d[i][j][k]为只允许经过结点[0, k]的情况下，i 到 j 的最短路。那么利用最优子结构性质，有两种情况：

a. 如果最短路经过 k 点，则 d[i][j][k] = d[i][k][k-1] + d[k][j][k-1];

b. 如果最短路不经过 k 点，则 d[i][j][k] = d[i][j][k-1];

于是有状态转移方程： d[i][j][k] = min{ d[i][j][k-1], d[i][k][k-1] + d[k][j][k-1] } (0 <= i, j, k < n)

这是一个 3D/0D 问题，只需要按照 k 递增的顺序进行枚举，就能在 O(n^3)的时间内求解，又第三维的状态可以采用滚动数组进行优化，所以空间复杂度为 O(n^2)。

差分约束

1、数形结合

如若一个系统由 n 个变量和 m 个不等式组成，并且这 m 个不等式对应的系数矩阵中每一行有且仅有一个 1 和-1，其它的都为 0，这样的系统称为差分约束(difference constraints)系统。引例中的不等式组可以表示成如图三-1-1 的系数矩阵。

然后继续回到单个不等式上来，观察 x[i] - x[j] <= a[k]， 将这个不等式稍稍变形，将 x[j]移到不等式右边，则有 x[i] <= x[j] + a[k]，然后我们令 a[k] = w(j, i)，再将不等式中的 i 和 j 变量替换掉，i = v，j = u，将 x 数组的名字改成 d (以上都是等价变换，不会改变原有不等式的性质)，则原先的不等式变成了以下形式：d[u] + w(u, v) >= d[v]。

这时候联想到 SPFA 中的一个松弛操作：

```
if(d[u] + w(u, v) < d[v]) {
    d[v] = d[u] + w(u, v);
}
```

对比上面的不等式，两个不等式的不等号正好相反，但是再仔细一想，其实它们的逻辑是一致的，因为 SPFA 的松弛操作是在满足小于的情况下进行松弛，力求达到 d[u] + w(u, v) >= d[v]，而我们之前令 a[k] = w(j, i)，所以我们可以将每个不等式转化成图上的有向边：

对于每个不等式 x[i] - x[j] <= a[k]，对结点 j 和 i 建立一条 j -> i 的有向边，边权为 a[k]，求 x[n-1] - x[0] 的最大值就是求 0 到 n-1 的最短路。

2、三角不等式

如果还没有完全理解，我们可以先来看一个简单的情况，如下三个不等式：

B - A <= c (1)

C - B <= a (2)

C - A <= b (3)

我们想要知道 C - A 的最大值，通过(1) + (2)，可以得到 C - A <= a + c，所以这个问题其实就是求 min{b, a+c}。我们发现 min{b, a+c}正好对应了 A 到 C 的最短路，而这三个不等式就是著名的三角不等式。将三个不等式推广到 m 个，变量推广到 n 个，就变成了 n 个点 m 条边的最短路问题了。

3、解的存在性

上文提到最短路的时候，会出现负权圈或者根本就不可达的情况，所以在不等式组转化的图上也有可能出现上述情况，先来看负权圈的情况，如图三-3-1，下图为 5 个变量 5 个不等式转化后的图，需要求得是 X[t] - X[s]的最大值，可以转化成求 s 到 t 的最短路，但是路径中出现负权圈，则表示最短路无限小，即不存在最短路，那么在不等式上的表现即 X[t] - X[s] <= T 中的 T 无限小，得出的结论就是 X[t] - X[s] 的最大值 不存在。

再来看另一种情况，即从起点 s 无法到达 t 的情况，如图三-3-2，表明 X[t]和 X[s]之间并没有约束关系，这种情况下 X[t] - X[s]的最大值是无限大，这就表明了 X[t]和 X[s]的取值有无限多种。

在实际问题中这两种情况会让你给出不同的输出。

综上所述，差分约束系统的解有三种情况：1、有解；2、无解；3、无限多解；

4、最大值 => 最小值

然后，我们将问题进行一个简单的转化，将原先的"<="变成">="，转化后的不等式如下：

B - A >= c (1)

C - B >= a (2)

C - A >= b (3)

然后求 C - A 的最小值，类比之前的方法，需要求的其实是 max{b, c+a}，于是对应的是图三-2-1 从 A 到 C 的最长路。同样可以推广到 n 个变量 m 个不等式的情况。

5、不等式标准化

如果给出的不等式有"<="也有">="，又该如何解决呢？很明显，首先需要关注最后的问题是什么，如果需要求的是两个变量差的最大值，那么需要将所有不等式转变成"<="的形式，建图后求最短路；相反，如果需要求的是两个变量差的最小值，那么需要将所有不等式转化成">="，建图后求最长路。

如果有形如：A - B = c 这样的等式呢？我们可以将它转化成以下两个不等式：

A - B >= c (1)

A - B <= c (2)

再通过上面的方法将其中一种不等号反向，建图即可。

最后，如果这些变量都是整数域上的，那么遇到 A - B < c 这样的不带等号的不等式，我们需要将它转化成"<="或者">="的形式，即 A - B <= c - 1。

差分约束的经典应用

1、线性约束

线性约束一般是在一维空间中给出一些变量（一般定义位置），然后告诉你某两个变量的约束关系，求两个变量 a 和 b 的差值的最大值或最小值。

【例题 1】N 个人编号为 1-N，并且按照编号顺序排成一条直线，任何两个人的位置不重合，然后给定一些约束条件。

X(X <= 100000)组约束 Ax Bx Cx(1 <= Ax < Bx <= N)，表示 Ax 和 Bx 的距离不能大于 Cx。

Y(X <= 100000)组约束 Ay By Cy(1 <= Ay < By <= N)，表示 Ay 和 By 的距离不能小于 Cy。

如果这样的排列存在，输出 1-N 这两个人的最长可能距离，如果不存在，输出-1，如果无限长输出-2。

像这类问题，N 个人的位置在一条直线上呈线性排列，某两个人的位置满足某些约束条件，最后要求第一个人和最后一个人的最长可能距离，这种是最直白的差分约束问题，因为可以用距离作为变量列出不等式组，然后再转化成图求最短路。

令第 x 个人的位置为 d[x]（不妨设 d[x]为 x 的递增函数，即随着 x 的增大，d[x]的位置朝着 x 正方向延伸）。

那么我们可以列出一些约束条件如下：

1、对于所有的 Ax Bx Cx，有 d[Bx] - d[Ax] <= Cx；

2、对于所有的 Ay By Cy，有 d[By] - d[Ay] >= Cy；

3、然后根据我们的设定，有 d[x] >= d[x-1] + 1 (1 < x <= N) （这个条件是表示任何两个人的位置不重合）

而我们需要求的是 d[N] - d[1]的最大值，即表示成 d[N] - d[1] <= T，要求的就是这个 T。于是我们将所有的不等式都转化成 d[x] - d[y] <= z 的形式，如下：

1、d[Bx] - d[Ax] <= Cx

2、d[Ay] - d[By] <= -Cy

3、d[x-1] - d[x] <= -1

对于 d[x] - d[y] <= z，令 z = w(y, x)，那么有 d[x] <= d[y] + w(y, x)，所以当 d[x] > d[y] + w(y,

x)，我们需要更新 d[x]的值，这对应了最短路的松弛操作，于是问题转化成了求 1 到 N 的最短路。

对于所有满足 d[x] - d[y] <= z 的不等式，从 y 向 x 建立一条权值为 z 的有向边。

然后从起点 1 出发，利用 SPFA 求到各个点的最短路，如果 1 到 N 不可达，说明最短路(即上文中的 T)无限长，输出-2。如果某个点进入队列大于等于 N 次，则必定存在一条负环，即没有最短路，输出-1。否则 T 就等于 1 到 N 的最短路。

2、区间约束

【例题 2】给定 n（n <= 50000）个整点闭区间和这个区间中至少有多少整点需要被选中，每个区间的范围为[ai, bi]，并且至少有 ci 个点需要被选中，其中 0 <= ai <= bi <= 50000，问[0, 50000]至少需要有多少点被选中。

例如 3 6 2 表示[3, 6]这个区间至少需要选择 2 个点，可以是 3,4 也可以是 4,6（总情况有 C(4, 2)种 ）。

这类问题就没有线性约束那么明显，需要将问题进行一下转化，考虑到最后需要求的是一个完整区间内至少有多少点被选中，试着用 d[i]表示[0, i]这个区间至少有多少点能被选中，根据定义，可以抽象出 d[-1] = 0，对于每个区间描述，可以表示成 d[bi] - d[ai - 1] >= ci，而我们的目标要求的是 d[50000] - d[-1] >= T 这个不等式中的 T，将所有区间描述转化成图后求-1 到 50000 的最长路。

这里忽略了一些要素，因为 d[i]描述了一个求和函数，所以对于 d[i]和 d[i-1]其实是有自身限制的，考虑到每个点有选和不选两种状态，所以 d[i]和 d[i-1]需要满足以下不等式： 0 <= d[i] - d[i-1] <= 1（即第 i 个数选还是不选）

这样一来，还需要加入 50000*2 = 100000 条边，由于边数和点数都是万级别的，所以不能采用单纯的 Bellman-Ford ，需要利用 SPFA 进行优化，由于-1 不能映射到小标，所以可以将所有点都向 x 轴正方向偏移 1 个单位（即所有数+1 ）。

3、未知条件约束

未知条件约束是指在不等式的右边不一定是个常数，可能是个未知数，可以通过枚举这个未知数，然后对不等式转化成差分约束进行求解。

【例题 3】在一家超市里，每个时刻都需要有营业员看管，R(i) (0 <= i < 24)表示从 i 时刻开始到 i+1 时刻结束需要的营业员的数目，现在有 N(N <= 1000)个申请人申请这项工作，并且每个申请者都有一个起始工作时间 ti，如果第 i 个申请者被录用，那么他会连续工作 8 小时。现在要求选择一些申请者进行录用，使得任何一个时刻 i，营业员数目都能大于等于 R(i)。

i = 0 1 2 3 4 5 6 ... 20 21 22 23 23，分别对应时刻 [i, i+1)，特殊的，23 表示的是[23, 0)，并且有些申请者的工作时间可能会“跨天”。

a[i] 表示在第 i 时刻开始工作的人数，是个未知量

b[i] 表示在第 i 时刻能够开始工作人数的上限， 是个已知量

R[i] 表示在第 i 时刻必须值班的人数，也是已知量

观察申请者的数量，当 i 个申请者能够满足条件的时候，i+1 个申请者必定可以满足条件，所以申请者的数量是满足单调性的，可以对 T 进行二分枚举，将枚举复杂度从 O(N)降为 O(logN)。

初等数论

数论基本概念

1、整除性

若 a 和 b 都为整数，a 整除 b 是指 b 是 a 的倍数，a 是 b 的约数（因数、因子），记为 a|b。整除的大部分性质都是显而易见的，为了阐述方便，我给这些性质都随便起了个名字。

i) 任意性，若 a|b，则对于任意非零整数 m，有 am|bm。

ii) 传递性，若 a|b，且 b|c，则 a|c。

iii) 可消性，若 a|bc，且 a 和 c 互质(互质的概念下文会讲到)，则 a|b。

iv) 组合性，若 c|a，且 c|b，则对于任意整数 m、n，有 c|(ma+nb)。

【例题 1】(公元 1987 年初二数学竞赛题) x，y，z 均为整数，若 11｜（7x+2y-5z），求证：11｜（3x-7y+12z）。

非常典型的一个问题，为了描述方便，令 a = （7x+2y-5z），b = （3x-7y+12z），通过构造可以得到一个等式：4a + 3b = 11（3x-2y+3z），则 3b = 11（3x-2y+3z） - 4a。

任意性+组合性，得出 11 |（11（3x-2y+3z） - 4a） = 11|3b。

可消性，由于 11 和 3 互质，得出 11 | b，证明完毕。

2、素数

a.素数与合数

素数又称质数，素数首先满足条件是要大于等于 2，并且除了 1 和它本身外，不能被其它任何自然数整除；其它的数称为合数；而 1 既非素数也非合数。

b.素数判定

如何判定一个数是否为素数？

i) 对 n 做[2, n)范围内的余数判定，如果有至少一个数用 n 取余后为 0，则表明 n 为合数；**如果所有数都不能整除 n，则 n 为素数，算法复杂度 O(n)。**

ii) 假设一个数能整除 n，即 a|n，那么 n/a 也必定能整除 n，不妨设 a <= n/a，则有 a^2 <= n，即 a <= sqrt(n)（sqrt 表示对 n 开根号），所以在用 i)的方法进行取余的时候，**范围可以缩小到 sqrt(n)，所以算法复杂度降为 O(sqrt(n))。**

iii) 如果 n 是合数，那么它必然有一个小于等于 sqrt(n)的素因子，只需要对 sqrt(n)内的素数进行测试即可，需要预处理求出 sqrt(n)中的素数，假设该范围内素数的个数为 s，**那么复杂度降为 O(s)。**

c.素数定理

当 x 很大时，小于 x 的素数的个数近似等于 x/ln(x)，其中 ln(x)表示 x 的自然对数

从这个定理可以发现，程序中进行素数判定的时候，用 ii)方法和 iii)方法差了至少一个数量级。

d.素数筛选法

【例题 2】给定 n(n < 10000)个数，范围为[1, 2^32)，判定它是素数还是合数。

首先 1 不是素数，如果 n>1，则枚举[1，sqrt(n)]范围内的素数进行试除，如果至少有一个素数能够整除 n，则表明 n 是合数，否则 n 是素数。

[1，sqrt(n)]范围内的素数可以通过筛选法预先筛出来，用一个数组 notprime[i]标记 i 是素数与否，筛选法有很多，这里介绍一种最常用的筛选法——Eratosthenes 筛选法。

直接给出伪代码：

```
#define MAXP 65536
#define LL __int64
 void Eratosthenes() {
        notprime[1] = true;
        primes[0] = 0;
```

```
        for(int i = 2; i < MAXP; i++) {
            if( !notprime[i] ) {
                primes[ ++primes[0] ] = i;
                //需要注意 i*i 超出整型后变成负数的问题，所以转化成 __int64
                for(LL j = (LL)i*i; j < MAXP; j += i) {
                    notprime[j] = true;
                }
            }
        }
    }
```

notprime[i]为真表明 i 为合数，否则 i 为素数（因为全局变量初始值为 false，筛选法预处理只做一次，所以不需要初始化）。算法的核心就是不断将 notprime[i]标记为 true 的过程，首先从小到大进行枚举，遇到 notprime[i]为假的，表明 i 是素数，将 i 保存到数组 primes 中，然后将 i 的倍数都标记为合数，由于 i*2、i*3、i*(i-1)在[1, i)的筛选过程中必定已经被标记为合数了，所以 i 的倍数只需要从 i*i 开始即可，避免不必要的时间开销。

虽然这个算法有两个嵌套的轮询，但是第二个轮询只有在 i 是素数的时候才会执行，而且随着 i 的增大，它的倍数会越来越少，所以整个算法的时间复杂度并不是 O(n^2)，而且远远小于 O(n^2)，在 notprime 进行赋值的时候加入一个计数器 count，计数器的值就是该程序的总执行次数，对 MAXP 进行不同的值测试发现 int(count / MAXP) 的值随着 MAXP 的增长变化非常小，总是维持在 2 左右，所以这个算法的复杂度可以近似看成是 O(n)，更加确切的可以说是 O(nC)，其中 C 为常数，C 一般取 2。

事实上，实际应用中由于空间的限制（空间复杂度为 O(n)），MAXP 的值并不会取的很大，10^7 基本已经算是极限了，再大的素数测试就需要用到 Rabin-Miller

3、因数分解

a、算术基本定理

算术基本定理可以描述为：对于每个整数 n，都可以唯一分解成素数的乘积。

这里的素数并不要求是不一样的，所以可以将相同的素数进行合并。

证明方法采用数学归纳法

b、素数拆分

给定一个数 n，如何将它拆分成素数的乘积呢？

还是用到上面讲到的试除法，假设 n = pm 并且 m>1，其中 p 为素数，如果 p > sqrt(n)，那么根据算数基本定理，m 中必定存在一个小于等于 sqrt(n)的素数，所以我们不妨设 p <= sqrt(n)。

然后通过枚举[2, sqrt(n)]的素数，如果能够找到一个素数 p，使得 n mod p == 0（mod 表示取余数、也称为模）。于是 m = n/p，这时还需要注意一点，因为 m 中可能也有 p 这个素因子，所以如果 p|m，需要继续试除，令 m' = m/p，直到将所有的素因子 p 除尽，统计除的次数 e，于是我们得到了 n = (p^e) * n'，然后继续枚举素数对 n'做同样的试除。

这时有两种情况：

i) S == 1，则素数分解完毕；

ii) S > 1，根据算术基本定理，S 必定为素数，而且是大于 sqrt(n)的素数，并且最多只有 1 个，这种情况同样适用于 n 本身就是素数的情况，这时 n = S。

这样的分解方式称为因数分解，各个素因子可以用一个二元的结构体来存储。算法时间复杂度为 O(s)，s 为 sqrt(n)内素数的个数。

c、因子个数

朴素的求因子个数的方法为枚举[1, n]的数进行余数判定，复杂度为 O(n)，这里加入一个小优化，如果 m 为 n 的因子，那么必然 n/m 也为 n 的因子，不妨设 m <= n/m，则有 m <= sqrt(n)，所以只要枚

举从[1, sqrt(n)]的因子然后计数即可，复杂度变为 O(sqrt(n))。

【例题 3】给定 X,Y(X, Y < 2^31)，求 X^Y 的因子数 mod 10007。

由于这里的 X^Y 已经是天文数字，利用上述的枚举法已经无法满足要求，所以我们需要换个思路。考虑到任何整数都能表示成素数的乘积，那么 X^Y 也不例外，我们首先将 X 进行因数分解容易发现 X^Y 的因子一定是 p1、p2、...、pk 的组合，并且 p1 可以取的个数为[0, Ye1]，p2 可以取的个数为[0, Ye2]，pk 可以取的个数为[0, Yek]，所以根据乘法原理，总的因子个数就是这些指数+1 的连乘，即(1 + Ye1) * (1 + Ye2) * ... * (1 + Yek)。

通过这个问题，可以得到更加一般的求因子个数的公式，如果用 ei 表示 X 分解素因子之后的指数，那么 X 的因子个数就是(1 + e1) * (1 + e2) * ... * (1 + ek)。

d、因子和

【例题 4】给定 X,Y(X, Y < 2^31)，求 X^Y 的所有因子之和 mod 10007。

同样还是将 X^Y 表示成图一-3-4 的形式，然后就变成了标准素数分解后的数的因子和问题了。考虑数 n，令 n 的因子和为 s(n)，对 n 进行素数分解后的，假设最小素数为 p，素因子 p 的个数为 e，那么 n = (p^e)n'。

容易得知当 n 的因子中 p 的个数为 0 时，因子之和为 s(n')。更加一般地，当 n 的因子中 p 的个数为 k 的时候，因子之和为(p^k)*s(n')，所以 n 的所有因子之和就可以表示成：

s(n) = (1 + p^1 + p^2 + ... p^e) * s(n') = (p^(e+1) - 1) / (p-1) * s(n')

s(n')可以通过相同方法递归计算。最后可以表示成一系列等比数列和的乘积。

令 g(p, e) = (p^(e+1) - 1) / (p-1)，则 s(n) = g(p1, e1) * g(p2, e2) * ... * g(pk, ek)。

4、最大公约数(GCD)和最小公倍数(LCM)

两个数 a 和 b 的最大公约数(Greatest Common Divisor)是指同时整除 a 和 b 的最大因数，记为 gcd(a, b)。特殊的，当 gcd(a, b) = 1，我们称 a 和 b 互质（上文谈到整除的时候略有提及）。

两个数 a 和 b 的最小公倍数(Leatest Common Multiple)是指同时被 a 和 b 整除的最小倍数，记为 lcm(a, b)。特殊的，当 a 和 b 互质时，lcm(a, b) = ab。

gcd 是基础数论中非常重要的概念，求解 gcd 一般采用辗转相除法（这个方法会在第二章开头着重介绍，这里先引出概念），而求 lcm 需要先求 gcd，然后通过 lcm(a, b) = ab / gcd(a, b)求解。

这里无意中**引出了一个恒等式：lcm(a, b) * gcd(a, b) = ab**。可以通过算术基本定理进行证明。

需要说明的是这里的 a 和 b 的分解式中的指数是可以为 0 的，也就是说 p1 是 a 和 b 中某一个数的最小素因子，p2 是次小的素因子。lcm(a, b)和 gcd(a, b)相乘，相当于等式右边的每个素因子的指数相加，即 min{xi, yi} + max{xi, yi} = xi + yi，正好对应了 a 和 b 的第 i 个素数分量的指数之和，得证。

给这样的 gcd 和 lcm 表示法冠个名以便后续使用——指数最值表示法。

【例题 5】三个未知数 x, y, z，它们的 gcd 为 G，lcm 为 L，G 和 L 已知，求(x, y, z)三元组的个数。

三个数的 gcd 可以参照两个数 gcd 的指数最值表示法，只不过每个素因子的指数上是三个数的最值（即 min{x1, y1, z1}），那么这个问题首先要做的就是将 G 和 L 分别进行素因子分解，然后轮询 L 的每个素因子，对于每个素因子单独处理。

假设素因子为 p，L 分解式中 p 的指数为 l，G 分解式中 p 的指数为 g，那么显然 l < g 时不可能存在满足条件的三元组，所以只需要讨论 l >= g 的情况，对于单个 p 因子，问题转化成了求三个数 x1, y1, z1，满足 min{x1, y1, z1} = g 且 max{x1, y1, z1} = l，更加通俗的意思就是三个数中最小的数是 g，最大的数是 l，另一个数在[g, l]范围内，这是一个排列组合问题，三元组{x1, y1, z1}的种类数当 l == g 时只有 1 中，否则答案就是 6(l - g)。

最后根据乘法原理将每个素因子对应的种类数相乘就是最后的答案了。

5、同余

a、模运算

给定一个正整数 p，任意一个整数 n，一定存在等式 n = kp + r； 其中 k、r 是整数，且满足 0 <=

r < p，称 k 为 n 除以 p 的商， r 为 n 除以 p 的余数，表示成 n % p = r (这里采用 C++语法，%表示取模运算)。

对于正整数和整数 a, b, 定义如下运算：

取模运算：a % p (a mod p)，表示 a 除以 p 的余数。

模 p 加法：(a + b) % p = (a%p + b%p) % p

模 p 减法：(a - b) % p = (a%p - b%p) % p

模 p 乘法：(a * b) % p = ((a % p)*(b % p)) % p

幂模 p : (a^b) % p = ((a % p)^b) % p

模运算满足结合律、交换律和分配律。

a≡b (mod n) 表示 a 和 b 模 n 同余，即 a 和 b 除以 n 的余数相等。

【例题 6】一个 n 位十进制数(n <= 1000000)必定包含 1、2、3、4 四个数字，现在将它顺序重排，求给出一种方案，使得重排后的数是 7 的倍数。

取出 1、2、3、4 后，将剩下的数字随便排列得到一个数 a，令剩下的四个数字排列出来的数为 b，那么就是要找到一种方案使得(a*10000 + b) % 7 等于 0。

但是 a 真的可以随便排吗？也就是说如果无论 a 等于多少，都能找到这样的 b 满足等式成立，那么 a 就可以随便排。

我们将等式简化：

(a*10000 + b) % 7 = (a*10000%7 + b%7) % 7

令 k = a*10000%7 = a*4%7，容易发现 k 的取值为[0, 7)，如果 b%7 的取值也是[0, 7)，那这个问题就可以完美解决了，很幸运的是，的确可以构造出 7 个这样的 b。具体参见下图：

图一-5-1

b、快速幂取模

幂取模常常用在 RSA 加密算法的加密和解密过程中，是指给定整数 a，正整数 n，以及非零整数 p，求 a^n % p。利用模 p 乘法，这个问题可以递归求解，即令 f(n) = a^n%p，那么 f(n-1) = a^(n-1)%p，f(n) = a*f(n-1) % p，这样就转化成了递归式。但是递归求解的时间复杂度为 O(n)，往往当 n 很大的时候就很难在规定时间内出解了。

当 n 为偶数时，我们可以将 a^n%p 拆成两部分，令 b = a^(n/2)%p，则 a^n%p = b*b%p；

当 n 为奇数时，可以拆成三部分，令 b = a^(n/2)%p，则 a^n%p = a*b*b%p；

上述两个等式中的 b 可以通过递归计算，由于每次都是除 2，所以时间复杂度是 O(logn)。

c、循环节

【例题 7】f[1] = a, f[2] = b, f[3] = c, 当 n>3 时 f[n] = (A*f[n-1] + B*f[n-2] + C*f[n-3]) % 53，给定 a, b, c, A, B, C，求 f[n] (n < 2^31)。

由于 n 非常大，循环模拟求解肯定是不现实的，仔细观察可以发现当 n>3 时，f[n]的值域为[0, 53)，并且连续三个数 f[n-1]、f[n-2]、f[n-3]一旦确定，那么 f[n]也就确定了，而 f[n-1]、f[n-2]、f[n-3]这三个数的组合数为 53*53*53 种情况，那么对于一个下标 k<n，假设 f[k]已经求出，并且满足 f[k-1] == f[n-1]且 f[k-2] == f[n-2]且 f[k-3] == f[n-3], 则 f[n]必定等于 f[k]，这里的 f[k...n-1]就被称为这个数列的循环节。

并且在 53*53*53 次计算之内必定能够找到循环节，这个是显而易见的。

数论基础知识

1、欧几里德定理（辗转相除法）

定理：gcd(a, b) = gcd(b, a % b)。

证明：a = kb + r = kb + a%b，则 a % b = a - kb。令 d 为 a 和 b 的公约数，则 d|a 且 d|b 根据整除的组合性原则，有 d|(a-kb)，即 d|(a%b)。

这就说明如果 d 是 a 和 b 的公约数，那么 d 也一定是 b 和 a%b 的公约数，即两者的公约数是一样的，所以最大公约数也必定相等。

这个定理可以直接用递归实现，代码如下：

```
int gcd(int a, int b) {
   return b ? gcd(b, a%b) : a;
}
```

这个函数揭示了一个约定俗成的概念，即任何非零整数和零的最大公约数为它本身。

【例题 8】f[0] = 0, 当 n>1 时，f[n] = (f[n-1]+a) % b，给定 a 和 b，问是否存在一个自然数 k (0 <= k< b)，是 f[n]永远都取不到的。

永远有多远？并不是本题的范畴。

但是可以发现的是这里的 f[...]一定是有循环节的，如果在某个循环节内都无法找到那个自然数 k，那么必定是永远都找不到了。

求出 f[n]的通项公式，为 f[n] = an % b，令 an = kb + r，那么这里的 r = f[n]，如果 t = gcd(a, b)，r = an-kb = t ((a/t)n - (b/t)k)，则有 t|r，要满足所有的 r 使得 t|r,只有当 t = 1 的时候，于是这个问题的解也就出来了，只要求 a 和 b 的 gcd，如果 gcd(a, b) > 1，则存在一个 k 使得 f[n]永远都取不到，**直观的理解是当 gcd(a, b) > 1，那么 f[n]不可能是素数。**

2、扩展欧几里德定理

a、线性同余

线性同余方程 (也可以叫模线性方程) 是最基本的同余方程，即 ax≡b (mod n)，其中 a、b、n 都为常量，x 是未知数，这个方程可以进行一定的转化，得到：ax = kn + b,这里的 k 为任意整数，于是我们可以得到更加一般的形式即：ax + by + c = 0，这个方程就是二维空间中的直线方程，但是 x 和 y 的取值为整数，所以这个方程的解是一些排列成直线的点集。

b、同余方程求解

求解同余方程第一步是转化成一般式：ax + by = c，这个方程的求解步骤如下：

i) 首先求出 a 和 b 的最大公约数 d = gcd(a, b)，那么原方程可以转化成 d(ax/d + by/d) = c，容易知道 (ax/d + by/d)为整数，如若 d 不能整除 b，方程必然无解，算法结束；否则进入 ii)。

ii) 由 i)可以得知，方程有解则一定可以表示成 ax + by = c = gcd(a, b)*c'，那么我们先来看如何求解 d = gcd(a, b) = ax + by，根据欧几里德定理，有：

d = gcd(a, b) = gcd(b, a%b) = bx' + (a%b)y' = bx' + [a-b*(a/b)]y' = ay' + b[x' - (a/b)y']

于是有 x = y', y = x' - (a/b)y'。

由于 gcd(a, b)是一个递归的计算，所以在求解(x, y)时，(x', y')其实已经利用递归计算出来了，递归出口为 b == 0 的时候 (对比辗转相除，也是 b == 0 的时候递归结束)，那么这时方程的解 x0 = 1, y0 = 0。

【例题 9】有两只青蛙，青蛙 A 和青蛙 B，它们在一个首尾相接的数轴上。设青蛙 A 的出发点坐标是 x，青蛙 B 的出发点坐标是 y。青蛙 A 一次能跳 m 米，青蛙 B 一次能跳 n 米，两只青蛙跳一次所花费的时间相同。数轴总长 L 米。要求它们至少跳了几次以后才会碰面。

假设跳了 t 次后相遇，则可以列出方程：(x + mt) % L = (y + nt) % L

将未知数 t 移到等式左边，常数移到等式右边，得到模线性方程：(m-n)t%L = (y-x)%L (即 ax≡b (mod n) 的形式)

利用扩展欧几里德定理可以求得 t 的通解{ t0 + kd | k 为任意整数 }，由于这里需要求 t 的最小正整数，而 t0 不一定是最小的正整数，甚至有可能是负数，我们发现 t 的通解是关于 d 同余的，所以最后的解可以做如下处理：ans = (t0 % d + d) % d。

c、逆元

模逆元的最通俗含义可以效仿乘法，a*x = 1，则称 x 为 a 在乘法域上的逆 (倒数)；同样，如果 ax≡1 (mod n)，则称 b 为 a 模 n 的逆，简称逆元。求 a 模 n 的逆元，就是模线性方程 ax≡b (mod n)中 b 等于 1 的特

殊形式，可以用扩展欧几里德求解。并且在 gcd(a, n) > 1 时逆不存在。

3、中国剩余定理

上文提到了模线性方程的求解，再来介绍一种模线性方程组的求解，模线性方程组如图二-3-1 所示，其中(ai, mi)都是已知量，求最小的 x 满足以下 n 个等式：

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

将模数保存在 mod 数组中，余数保存在 rem 数组中，则上面的问题可以表示成以下几个式子，我们的目的是要求出一个最小的正整数 K 满足所有等式：

K = mod[0] * x[0] + rem[0]　　(0)

K = mod[1] * x[1] + rem[1]　　(1)

K = mod[2] * x[2] + rem[2]　　(2)

K = mod[3] * x[3] + rem[3]　　(3)

… …

这里给出我的算法，大体的思想就是每次合并两个方程，经过 n-1 次合并后剩下一个方程，方程的自变量取 0 时得到最小正整数解。算法描述如下：

i)　迭代器 i = 0

ii)　x[i] = (newMod[i]*k + newRem[i])(k 为任意整数)

iii)　合并(i)和(i+1)，得 mod[i] * x[i] - mod[i+1] * x[i+1] = rem[i+1] - rem[i]

将 x[i]代入上式，有 newMod[i]*mod[i]*k - mod[i+1] * x[i+1] = rem[i+1] - rem[i] - newRem[i]*mod[i]

iv)　那么产生了一个形如 a*k + b*x[i+1] = c 的同余方程，

其中 a = newMod[i]*mod[i], b = - mod[i+1], c = rem[i+1] - rem[i] - newRem[i]*mod[i]

求解同余方程，如果 a 和 b 的 gcd 不能整除 c，则整个同余方程组无解，算法结束；

否则，利用扩展欧几里德求解 x[i+1]的通解，通解可以表示成 x[i+1] = (newMod[i+1]*k + newRem[i+1])(k 为任意整数)

v)　迭代器 i++,如果 i == n 算法结束，最后答案为 newRem[n-1] * mod[n-1] + rem[n-1]；否则跳转到 ii)继续迭代计算。

4、欧拉函数

a、互质

两个数 a 和 b 互质的定义为：gcd(a, b) = 1，那么如何求不大于 n 且与 n 互质的数的个数呢？

朴素算法，枚举 i 从 1 到 n，当 gcd(i, n)=1 时计数器++，算法时间复杂度 O(n)。

这里引入一个新的概念：用 φ(n)表示不大于 n 且与 n 互质的数的个数，该函数以欧拉的名字命名，称为欧拉函数。

如果 n 是一个素数，即 n = p，那么 φ(n) = p-1（所有小于 n 的都互质）；

如果 n 是素数的 k 次幂，即 n = p^k，那么 φ(n) = p^k - p^(k-1) （除了 p 的倍数其它都互质）；

如果 m 和 n 互质，那么 φ(mn) = φ(m)φ(n)（可以利用上面两个性质进行推导）。

前面已经讲到 n 的因子分解复杂度为 O(k)，所以欧拉函数的求解就是 O(k)。

b、筛选法求解欧拉函数

由于欧拉函数的表示法和整数的素数拆分表示法很类似，都可以表示成一些素数的函数的乘积，所以同样可以利用筛选法进行求解。

c、欧拉定理和费马小定理

欧拉定理：若 n,a 为正整数，且 n,a 互质，则: $a^{\varphi(n)} \equiv 1(\mathrm{mod}\, n)$ 。

费马小定理：若 p 为素数，a 为正整数且和 p 互质，则： $a^{p-1} \equiv 1(mod\, p)$ 。

由于当 n 为素数时 φ(n) = p-1，可见费马小定理是欧拉定理的特殊形式。

【例题 10】整数 a 和 n 互质，求 a 的 k 次幂模 n，其中 k = X^Y, 正整数 a,n,X,Y（X,Y<=10^9）为给定值。

问题要求的是 a^(X^Y) % n，指数上还是存在指数，需要将指数化简，注意到 a 和 n 互质，所以可以利用欧拉定理，令 X^Y = kφ(n) + r，那么 kφ(n)部分并不需要考虑，问题转化成求 r = X^Y % φ(n)，可以采用快速幂取模，二分求解，得到 r 后再采用快速幂取模求解 a^r % n。

5、容斥原理

容斥原理是应用在集合上的，来看图二-5-1，要求图中两个圆的并面积，我们的做法是先将两个圆的面积相加，然后发现相交的部分多加了一次，予以减去；对于图二-5-2 的三个圆的并面积，则是先将三个圆的面积相加，然后减去两两相交的部分，而三个圆相交的部分被多减了一次，予以加回。

【例题 11】求小于等于 m(m < 2^31)并且与 n(n < 2^31)互质的数的个数。

当 m 等于 n，就是一个简单的欧拉函数求解。

但是一般情况 m 都是不等于 n 的，所以可以直接摈弃欧拉函数的思路了。

考虑将 n 分解成素数幂的乘积，来看一种最简单的情况，当 n 为素数的幂即 n = p^k 时，显然答案等于 m - m/p（m/p 表示的是 p 的倍数，去掉 p 的倍数，则都是和 n 互质的数了）；然后再来讨论 n 是两个素数的幂的乘积的情况，即 n = p1^k1 * p2^k2，那么我们需要做的就是找到 p1 的倍数和 p2 的倍数，并且要减去 p1 和 p2 的公公倍数，这个思想其实已经是容斥了，所以这种情况下答案为：m -（m/p1 + m/p2 - m/(p1*p2)）。

类比两个素因子，如果 n 分解成 s 个素因子，也同样可以用容斥原理求解。

容斥原理其实是枚举子集的过程，常见的枚举方法为 dfs，也可以采用二进制法（0 表示取，1 表示不取）。

数论常用算法

1、Rabin-Miller 大素数判定

对于一个很大的数 n（例如十进制表示有 100 位），如果还是采用试除法进行判定，时间复杂度必定难以承受，目前比较稳定的大素数判定法是拉宾-米勒（Rabin-Miller）素数判定。

拉宾-米勒判定是基于费马小定理的，即如果一个数 p 为素数的条件是对于所有和 p 互质的正整数 a 满足以下等式： $a^{p-1} \equiv 1(mod\, p)$

然而我们不可能试遍所有和 p 互质的正整数，这样的话和试除比算法的复杂度反而更高，事实上我们只需要取比 p 小的几个素数进行测试就行了。

具体判断 n 是否为素数的算法如下：

i) 如果 n==2，返回 true；如果 n<2|| !(n&1), 返回 false；否则跳到 ii)。

ii) 令 n = m*（2^k） + 1，其中 m 为奇数，则 n-1 = m*（2^k）。

iii) 枚举小于 n 的素数 p(至多枚举 10 个)，对每个素数执行费马测试，费马测试如下：计算 pre = p^m % n，如果 pre 等于 1，则该测试失效，继续回到 iii)测试下一个素数；否则进行 k 次计算 next = pre^2 % n，如果 next == 1 && pre ！= 1 && pre != n-1 则 n 必定是合数，直接返回；k 次计算结束判断 pre 的值，如果不等于 1，必定是合数。

iv) 10 次判定完毕，如果 n 都没有检测出是合数，那么 n 为素数。

伪代码如下：

```
bool Rabin_Miller(LL n) {
```

```
        LL k = 0, m = n-1;
        if(n == 2) return true;
        if(n < 2 || !(n & 1)) return false;
        // 将 n-1 表示成 m*2^k
        while( !(m & 1) ) k++, m >>= 1;
        for(int i = 0; i < 10; i++) {
            if(p[i] == n)
                return true;
            if( isRealComposite(p[i], n, m, k) ) {
                return false;
            }
        }
        return true;
}
```

这里的函数 isRealComposite(p, n, m, k)就是费马测试，p^(m*2^k) % n 不等于 1 则 n 必定为合数，这是根据费马小定理得出的（注意）。n-1 = m*（2^k）

isRealComposite 实现如下：

```
    bool isRealComposite(LL p, LL n, LL m, LL k) {
        LL pre = Power_Mod(p, m, n);
        if(pre == 1) {
            return false;
        }
        while(k--) {
            LL next = Product_Mod(pre, pre, n);
            if(next == 1 && pre != 1 && pre != n-1)
                return true;
            pre = next;
        }
        return ( pre != 1 );
    }
```

这里 Power_Mod(a, b, n)即 a^b%n，Product_Mod(a, b, n)即 a*b%n，而 k 次测试的基于费马小定理的一个推论：x^2 % n = 1 当 n 为素数时 x 的解只有两个，即 1 和 n-1。

2、Pollard-rho 大数因式分解

有了大数判素，就会伴随着大数的因式分解，Pollard-rho 是一个大数分解的随机算法，能够在 O(n^(1/4))的时间内找到 n 的一个素因子 p，然后再递归计算 n' = n/p，直到 n 为素数为止，通过这样的方法将 n 进行素因子分解。

Pollard-rho 的策略为：从[2, n)中随机选取 k 个数 x1、x2、x3、...、xk，求任意两个数 xi、xj 的差和 n 的最大公约数，即 d = gcd(xi - xj, n)，如果 1 < d < n，则 d 为 n 的一个因子，直接返回 d 即可。

然后来看如何选取这 k 个数，我们采用生成函数法，令 x1 = rand()%(n-1) + 1，xi = （xi-1 ^ 2 + 1）mod n，很明显，这个序列是有循环节的。

我们需要做的就是在它进入循环的时候及时跳出循环，因为 x1 是随机选的，x1 选的不好可能使得这个算法永远都找不到 n 的一个范围在(1，n)的因子，这里采用步进法，保证在进入环的时候直接跳出循环，具体算法伪代码如下：

```
  LL Pollard_rho(LL n) {
```

```
        LL x = rand() % (n - 1) + 1;
        LL y = x;
        LL i = 1, k = 2;
        do {
            i++;
            LL p = gcd(n + y - x, n);    // 这里传入的 gcd 需要是正数
            if(1 < p && p < n) {
                return p;
            }
            if(i == k) {
                k <<= 1;
                y = x;
            }
            x = Func(x, n);
        }while(x != y);
        return n;
    }
```

3、RSA 原理

RSA 算法有三个参数，n、pub、pri，其中 n 等于两个大素数 p 和 q 的乘积(n = p*q)，pub 可以任意取，但是要求与(p-1)*(q-1)互质，pub*pri % () = 1 (可以理解为 pri 是 pub 的逆元)，那么这里的(n, pub)称为公钥，(n, pri)称为私钥。(p-1)*(q-1)

RSA 算法的加密和解密是一致的，令 x 为明文，y 为密文，则：

加密：y = x ^ pub % n　　(利用公钥加密，y = encode(x))

解密：x = y ^ pri % n(利用私钥解密，x = decode(y))

那么我们来看看这个算法是如何运作的。

假设你得到了一个密文 y，并且手上只有公钥，如何得到明文 x，从 decode 的情况来看，只要知道私钥貌似就可以了，而私钥的获取方式只有一个，就是求公钥对(p-1)*(q-1)的逆元，如果(p-1)*(q-1)已知，那么可以利用扩展欧几里德定理进行求解，问题是(p-1)*(q-1)是未知的，但是我们有 n = p*q，于是问题归根结底其实是难在了对 n 进行素因子分解上了，Pollard-rho 的分解算法时间 复杂度只能达到 O(n ^(1/4))，对 int64 范围内的整数可以在几十毫秒内出解，而当 n 是几百位的大数的时候计算时间就只能用天来衡量了。

最近公共祖先

一、引例

1、树-结点间最短距离【例题 1】给定一棵 n(n <= 100000)个结点的树，以及 m(m <= 100000)条询问(u, v),对于每条询问，求 u 和 v 的最短距离？

我们知道，一个普通的无向图求两点间最短距离可以采用单源最短路，将时间复杂度大致控制在 O(nlogn)，但是当询问足够多的时候，这并不是一个高效的算法。从树的性质可知，树上任意两点间有且仅有一条简单路径（路径上没有重点和重边），所以树上任意两点间的最短距离其实就是这条简单路径的长度。如图一-1-1 所示，要求 u 到 v 的距离，我们需要知道红色路径 A(u->r），蓝色路径 B(v->r)，红蓝公共路径 C(a->r)，那么 u->v 的路径显然就可以通过 A、B、C 三者的长度计算出来，令 dist[x]表示从 x 到树根 r 的简单路径的长度，则 u 到 v 的距离可以表示成如下：dist[u] + dist[v] - 2*dist[a]。

那么问题来了，a 是什么，我们来看 a->r 路径上的所有结点既是 u 的祖先，也是 v 的祖先，所以我们给它们一个定义称为公共祖先（Common Ancestors），而 a 作为深度最深的祖先，或者说离 u 和 v 最近的，我们称它为最近公共祖先（Lowest Common Ancestors），简称 LCA。

二、LCA（最近公共祖先）

1、朴素算法

于是求树上两点间的距离转化成了求两个结点的最近公共祖先问题上来了，最容易想到的办法是将 u->r 和 v->r 的两条路径通过递归生成出来，并且逆序保存，然后比较两条路径的公共前缀路径，显然公共前缀路径的尾结点就是 u 和 v 的最近公共祖先 ,但是**这个算法在树退化成链的时候达到最坏复杂度 O(n)，并不可行。**

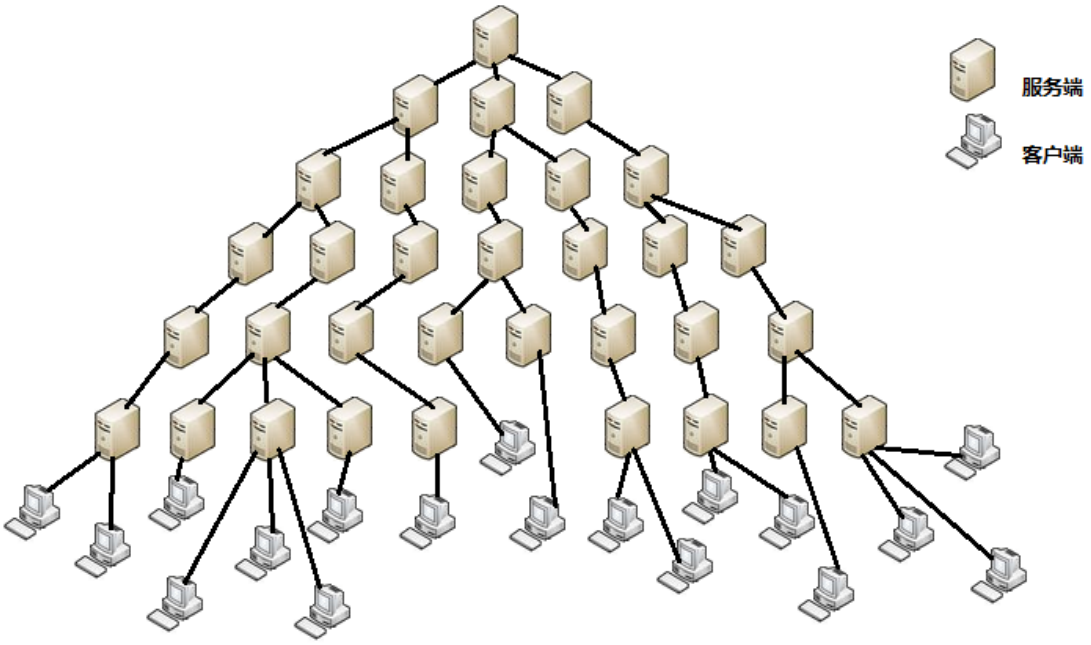
2、步进法

对于两个结点 u 和 v，假设它们的最近公共祖先为 lca(u, v)，用 depth[x]表示 x 这个结点的深度，pre[x]表示 x 的父结点。那么很显然，有 depth[lca(u, v)] <= min{depth[u], depth[v]}，通过这样一个性质，我们可以很容易得出一个算法：

1) 当 depth[u] < depth[v]时，lca(u, v) = lca(u, pre[v]);

2) 当 depth[u] > depth[v]时，lca(u, v) = lca(pre[u], v);

利用以上两个条件，可以将 u 和 v 不断向根进行步进，递归求解，直到 u == v 时，这里的 u 或 v 就是原先要求的(u, v)的最近公共祖先。



3、记忆化步进法

【例题 2】如图二-3-1 的网络拓扑树，给定两个客户端(x, y)，需要找到一个离他们最近的服务器来处理两个客户端的交互，客户端和服务端数量小于等于 1000，但是询问数 Q <= 10^8。

这是一个典型的 LCA 问题，并且询问很多，还有一个特点是总的树结点树并不是很多，所以在步进法计算 LCA 的时候，可能会遇到很多重复的计算，具体是指计算 lca(u, v)的时候可能在某次查询的时候已经被计算过了，那么我们可以把这个二元组预先存在数组中，即 lca[u][v]，这样做可以避免多次查询中遇到的冗余计算。但同时也带来一个问题，就是空间复杂度是 O(n^2)，对于 n = 100000 的情况下内存已经吃不消了，记忆化步进法只适用于 n 在千量级的情况。

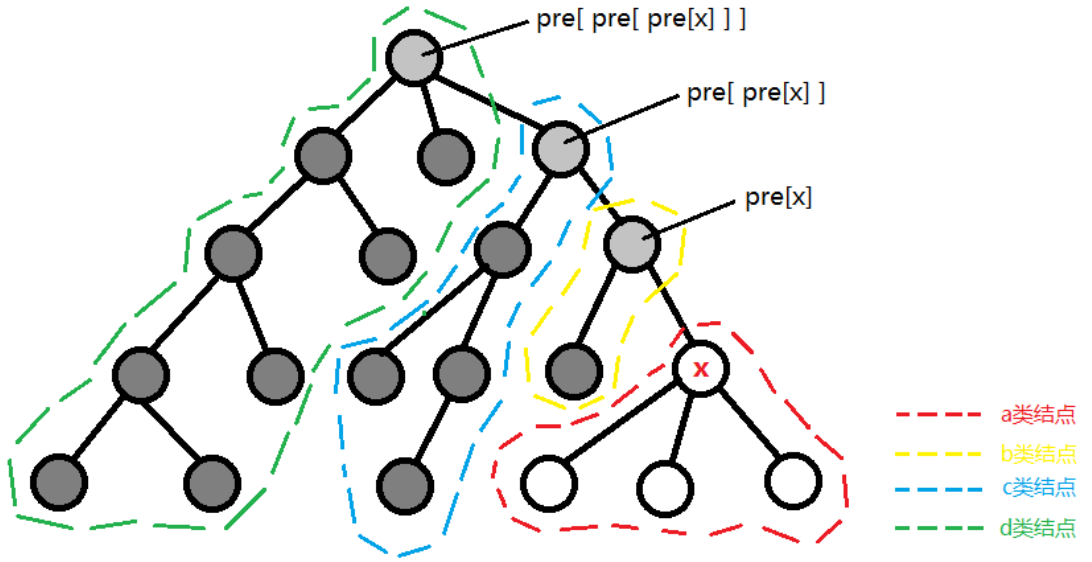
4、tarjan 算法

tarjan 算法采用深度优先搜索递归计算结点(u, v)的 LCA。具体的思想是在搜索过程中将一棵树进行分类，分类如下：

a.以结点 x 为根的子树作为 a 类结点；

b.以 pre[x]为根的子树去掉 a 类结点，为 b 类结点；

c.以 pre[pre[x]]为根的子树并且去掉 a、b 两类，为 c 类结点；依此类推...



如图二-4-1 所示，对这棵树进行深度优先搜索，深灰色结点（之所以不用黑色是因为线条也是黑的）表示以该结点为根的子树已经尽数访问完毕；浅灰色结点表示该结点为根的子树正在进行访问，且尚未访问完毕；白色结点表示以该结点为根的子树还没开始访问。图中红色圈出的部分为上文提到的 a 类结点；黄色圈出的部分为 b 类结点；蓝色为 c 类结点；绿色为 d 类结点，以此类推，不同的颜色属于不同的集合。所谓一图在手，胜过万语千言，从图中很容易就能看出，x 和所有绿色结点的 LCA 都为 pre[pre[pre[x]]]，即 x 的曾祖父结点；和所有蓝色结点的 LCA 都为 pre[pre[x]]，即 x 的祖父结点；和所有黄色结点的 LCA 都为 pre[x]，即 x 的父结点。

可能有人对这里集合的概念表示不理解，举个例子就能明白了，我们还是沿用上图，将图上的结点进行编号，如图二-4-2 所示，可以得到其中三个集合为：

B = {8,13}　C = {4,7,11,12,16}D = {1,2,3,5,6,9,10,14,15}

每个集合对应一棵子树，按照 tarjan 算法的思路，当给定任意一个结点，我们需要得到这个结点所在集合对应的子树的根结点，这里分两步走：

1）找到结点对应的集合；

2）找到集合的根；

第 2)步可以预先将值保存在数组中，但是集合不像数字，不能作为数组的下标。而我们注意到这里的集合都是互不相交的，这一点是非常关键的，这就意味着我们可以用一个集合中的元素来作为这个集合的“代表元”。假设 B 的代表元为 13，C 的代表元为 7，D 的代表元为 5，用 ancestor 数组来存储集合的根结点，则有 ancestor[13] = 8, ancestor[7] = 4, ancestor[5] = 1，于是第 2)步就能在 O(1)的时间内完成了。
第 1)步其实可以描述成给定一个结点，求该结点所在集合的代表元。这里先不讨论实现，因为这个操作会在第三节并查集中花很长的篇幅来讲。
对集合有一定了解后，让我们最后总结下这个算法究竟是如何实现的。
1) 初始化所有结点颜色 colors[i] = 0，对树 T 进行深度优先搜索；
2) 访问到结点 u 时，将 u 单独作为一个集合，并且令 ancestor[u] = u，即这个集合的根结点为 u，然后跳转到 3)；
3) 访问 u 的所有子结点 v，递归执行 2)，当 v 所在子树结点全部访问完毕后，将 v 所在集合和 u 所在集合合并（执行 merge(u, v)），并且令新的集合的祖先为 u（执行 ancestor[find(u)] = u）；
4) 当 u 所在子树结点全部访问完毕后，置 colors[u] = 1，枚举所有和 u 有关的询问：(u, v)，如果 colors[v] 等于 1，那么记录 u 和 v 的最近公共祖先为 ancestor[find(v)]；
这里有两个神奇的函数，merge(u, v)和 find(u)，其中 merge(u, v)表示合并 u 和 v 所在的两个集合，find(u)则表示找到 u 所在集合的代表元。如果对这个算法已经有一定的认知，那么趁热打铁，来看下伪代码是如何实现的。

```
void LCA_Tarjan(int u) {
   make_set(u); // 注释 1
   ancestor[ find(u) ] = u; // 注释 2
   for(int i = 0; i < edge[u].size(); i++) {   // 注释 3
int v = edge[u][i].v;
LCA_Tarjan(v); // 注释 4
merge(u, v);     // 注释 5
ancestor[ find(u) ] = u;     // 注释 6
   }
   colors[u] = 1; // 注释 7
   for(int i = 0; i < lcaInfo[u].size(); i++) {
int v = lcaInfo[u][i].v;
if(colors[v]) {// 注释 8
lcaDataAns[ lcaInfo[u][i].idx ].lca = ancestor[ find(v) ];
}
   }
}
```

注释 1:创建一个集合，集合中只有一个元素 u，即{ u }
注释 2:因为 u 所在集合只有一个元素，所以也可以写成 ancestor[u] = u
注释 3:edge[u][0...size-1]存储的是 u 的直接子结点
注释 4:递归计算 u 的所有直接子结点 v
注释 5:回溯的时候进行集合合并，将以 v 为根的子树和 u 所在集合进行合并
注释 6:对合并完的集合设置集合对应子树的根结点，find(u)为该集合的代表元
注释 7:u 为根的子树访问完毕，设置结点颜色
注释 8:枚举所有和 u 相关的询问(u, v)，如果以 v 为根的子树已经访问过，那么 ancestor[find(v)]肯定已经计算出来，并且必定是 u 和 v 的 LCA
tarjan 算法的时间复杂度为 O(n+q)，其中 n 为结点个数，q 为询问对(u, v)的个数。但是在进行深搜之前

首先需要知道所有的询问对，所以是一个离线算法，不能实时进行计算，局限性较大。
【例题 3】在一个可视化的界面上的一棵树，选中某些结点，然后要求在文件中保存一棵最小的子树，使得这棵子树包含所有这些选中的结点。
doubly 算这个是实际文件保存中比较经典的问题，我们可以选择两个结点求出 LCA，然后用这个 LCA 再和两一个点求 LCA，以此类推...n 个结点经过 n-1 次迭代就能求出 n 个结点的 LCA，这个 LCA 就是要保存的子树的根结点了。

5、doubly 算法

doubly 算法（倍增法）是在线算法，可以实时计算任意两点间的 LCA，并且每次计算时间复杂度是 O(1)的。
该算法同样也是基于深度优先搜索的，遍历的时候从根结点开始遍历，将遍历的边保存下来，对于一棵 n 个结点的树来说总共有 n-1 条边，那么遍历的时候需要保存 2*(n-1)条边（自顶向下的边，以及回溯时的边，所以总共两倍的边），这些边顺序存储后相邻两条边的首尾结点必定是一致的，所以可以压缩到一个一维数组 E 中，数组的长度为 2*(n-1)+1，然后建立一个辅助数组 D，长度和 E 一致，记录的是 E 数组中对应结点的深度，这个在遍历的时候可以一并保存，然后再用一个辅助数组 I[i]来保存 i 这个结点在 E 数组中第一次出现的下标。至此，初始化工作就算完毕了。
那么当询问两个结点 u 和 v 的 LCA 时，我们首先通过辅助数组 I 获取两个结点在 D 数组中的位置，即 I[u]和 I[v]，不妨设 I[u] <= I[v],那么在 D 数组中的某个深度序列 D[I[u], I[u]+1, ... I[v]-1, I[v]]，其实表示的是从 u 到 v 的路径上每个结点的深度，而之前我们已经知道树上任意两个结点的简单路径有且仅有一条，并且路径上深度最小的点就是 u 和 v 的 LCA，所以问题转化成了求 D[I[u], I[u]+1, ... I[v]-1, I[v]]的最小值，找到最小值所在下标后再到 E 数组索引得到结点的值就是 u 和 v 的 LCA 了。

如图二-5-1 为 n = 7 个结点的一棵树，其中 0 为根结点，6 条边分别为(0,5)(5,2)(2,4)(0,3)(3,1)(3,6)。注意这里的边是有向边，即一定是父结点指向子结点，如果给定的是无向边，需要预先进行处理。如右图，从根结点进行遍历，其中红色的边为自顶向下的边，绿色的边为回溯边，回溯边一定是子结点指向父结点的，蓝色的小数字代表边的遍历顺序，即第几条边，将所有的边串起来就变成这样了：
(0,5) -> (5,2) -> (2,4) -> (4,2) -> (2,5) -> (5,0) -> (0,3) -> (3,1) -> (1,3) -> (3,6) -> (6,3) -> (3,0)
然后我们将这些边压缩到数组 E 中，得到：
E[1 ... 2n-1] = 0 5 2 4 2 5 0 3 1 3 6 3 0
对数组 E 中对应的结点在树上的深度记录在数组 D 中，得到：
D[1 ... 2n-1] = 0 1 2 3 2 1 0 1 2 1 2 1 0
再将每个结点在数组 E 中第一次出现的位置记录在数组 I 中，得到：
I[0 ... n-1] = 1 9 3 8 4 2 11
注意：这里的数组 E 和 D 都是 1-based，而 I 数组是 0-based，这个是个人习惯，可自行约定，不必纠结。
根据 LCA 的性质，有 LCA(x, y) = E[Min_Index(D, I[x], I[y])]，其中 Min_Index(Array, i, j)表示 Array 数组中[i, j]区间内的最小值对应的下标，那么问题的难点其实就是在于求 Min_Index(Array, i, j)了，这个问题就是经典的区间最值问题，最常见的可以采用线段树求解，建好树后单次查询的时间复杂度为 O(logn)，当然还有一种更加高效的算法，查询复杂度可以达到严格的 O(1)，这就是第四节要讨论的 RMQ 问题。
由于 tarjan 算法中还有一个有关集合操作的遗留问题尚未介绍，这里先来看史上最轻量级的数据结构——并查集。

三、并查集

1、"并"和"查"

并查集是一种处理不相交集合的数据结构，它支持两种操作：
1）合并操作 merge(x, y)。即合并两个原本不相交的集合，此所谓“并”。
2）查找操作 find(x)。即检索某一个元素属于哪个集合，此所谓“查”。

2、朴素算法

接下来来讨论下并查集的朴素实现，既然是朴素实现，当然是越朴素越好。朴素的只需要一个数组就能表示集合，我们用 set[i]表示 i 所在的集合，这样查找操作的时间复杂度就能通过下标索引达到 O(1)（可以把 set 数组理解成哈希表）；合并 x 和 y 的操作需要判断 set[x]和 set[y]是否相等，如果不相等，需要将所有满足 set[i]等于 set[x]的 set[i]变成 set[y]，由于这里需要遍历 set[i]。

初始化 set[i] = i,每次得到一组数据(x, y),就执行 merge(x, y)，统计完所有数据后，对 set 数组进行一次线扫，就能统计出总共有多少个门派。

3、森林实现

由于朴素实现合并操作的时间复杂度太高，在人数很多的情况下，效率上非常吃亏，如果有 n 次合并操作，那么总的时间复杂度就是 O(n^2)。所以我们将集合的表示进行一定的优化，将一个个集合用树的形式来组织，多个集合就组成了一个森林。用 pre[i]表示 i 在集合树上的父结点，当 pre[i]等于 i 时，则表示 i 为这棵集合树的根结点。那么显而易见的，对于合并操作(x, y)，只需要查找 x 和 y 在各自集合树的根结点 rx 和 ry，如果 rx 和 ry 不相等，则将 rx 设为 ry 的父结点，即令 pre[ry] = rx。查找操作更加简单，只要顺着父结点一直找到根结点，就找到了该结点所在的集合。初始化 pre[i] = i，即表示森林中有 n 棵一个结点的树。

```
int find (int x) {
   return x == pre[x] ? x : find(pre[x]);
}
void merge(int x, int y) {
   int rx = find(x), ry = find(y);
   pre[ry] = rx;
}
```

代码量较朴素算法减少了不少，那时间复杂度呢？仔细观察，不难发现两个操作的时间复杂度其实是一样的，瓶颈都在查找操作上，来看一个简单的例子。

【例题 3】因为天下武功出少林，所以很多人都想加入少林习武，令少林寺的编号为 1。然后给定 m(m <= 100000)组数据(x, y)，表示 x 和 y 结成朋友，当 x 或 y 等于 1 时，表示另一个不等于 1 的人带领他的朋友一起加入少林，已知总人数 n，求最后少林寺来了多少人。

一个最基础的并查集操作，对于每条数据执行 merge(x, y)即可，最后一次线性扫描统计 1 所在集合的人数的个数，但是对于极限情况，还是会退化成 O(n)的查找，如图三-3-2 所示，每次合并都是一条链合并到一个结点上，使得原本的树退化成了链，合并本身是 O(1)的，但是在合并前的查找根结点的过程中已经是 O(n)的了，**为了避免集合成链的情况，需要进行启发式合并。**

4、启发式合并

启发式合并是为了解决合并过程中树退化成链的情况，用 depth[i]表示根为 i 的树的最大深度，合并 ra 和 rb 时，采用最大深度小的向最大深度大的进行合并，如果两棵树的最大深度一样，则随便选择一个作为根，并且将根的最大深度 depth 自增 1，这样做的好处是在 n 次操作后，任何一棵集合树的最大深度都不会超过 log(n)，所以使得查找的复杂度降为 O(log(n))。

给出启发式合并的伪代码如下：

```
int find (int x) {
   return x == pre[x] ? x : find(pre[x]);
}
int merge(int x, int y) {
   int rx = find(x), ry = find(y);
   if(rx != ry) {
if( depth[rx] == depth[ry] ) {
```

```
pre[ry] = rx;
depth[rx]++;
}else if( depth[rx] < depth[ry] ) {
pre[rx] = ry;
}else {
pre[ry] = rx;
}
  }
}
```

启发式合并的查找操作不变，合并操作引入了 depth 数组，并且在合并过程中即时更新。

5、路径压缩

小的集合归并到大的集合

【例题 4】n(n <= 100000)个门派编号为 1-n，经过 m(m <= 100000)次江湖上的血雨腥风，不断产生门派吞并，每次吞并可以表示成(x, y)，即 x 吞并了 y，最后问从前往后数还存在的编号第 k 大的那个门派的编号。

启发式合并通过改善合并操作提高了效率，但是这个问题的合并是有向的，即一定是 y 向 x 合并，所以无法采用按深度的启发式合并，那么是否有办法优化查找操作来改善效率呢？答案是一定的，我们可以在结点 x 找到树根 rx 的时候 ,将 x 到 rx 路径上的点的父结点都设置为 rx ,这样做并不会改变原有的集合关系，如图三-5-1 所示。

由于每次查找过程中都对路径进行了压缩，使得任何时候树的深度都是小于 4 的，从而查找操作可以认为是常数时间。

当然，如果合并是无向的同样可以采用路径压缩，这样做会导致树的深度可能时刻在变化，所以启发式合并的启发值不能采用树的深度，可以通过一个 rank 值来进行合并，rank 值初始为 0，当两棵树进行合并时，如果 rank 值相同，随便选一棵树的根作为新根进行合并，rank 值+1；否则 rank 小的向大的合并，rank 值保持不变。

给出路径压缩的伪代码如下：

```
    int find(int x) {
        return x == pre[x] ? x : pre[x] = find(pre[x]);
    }
    int merge(int x, int y) {
        int rx = find(x), ry = find(y);
        if(rx != ry) {
            if( rank[rx] == rank[ry] ) {
                pre[ry] = rx;
                rank[rx]++;
            }else if( rank[rx] < rank[ry] ) {
                pre[rx] = ry;
            }else {
                pre[ry] = rx;
            }
        }
}
```

仔细观察不难发现，路径压缩版本的 find 和 merge 操作与启发式合并的版本除了红色标注部分，其它完全相同（只是 merge 函数中 depth 变量名换成了 rank），find 函数中的赋值操作（红色部分代码）很好

地诠释了深度优先搜索在回溯时的完美表现，find 函数的返回值一定是这棵集合树的根结点 root，回溯的时候会经过从 x 到 root 的路径，通过这一步赋值可以很轻松的将该路径上所有结点的父结点都设为根结点 root。

6、元素删除

【例题 5】话说有人加入少林，当然也有人还俗，虚竹就是个很好的例子，还俗后加入了逍遥派，这就是所谓的集合元素的删除。

并查集的删除操作可以描述成：在某个集合中，将某个元素孤立出来成为一个只有它本身的新的集合。这步操作不能破坏原有的树结构，单纯“拆”树是不现实的，如图三-6-1 所示,是我们的美好预期，但是事实上并做不到，因为在并查集的数据结构中只记录了 x 的父亲，而未记录 x 的子结点，没办法改变 x 子结点的父结点。

而且就算是记录了子结点，每次删除操作最坏情况会更新 n 个子结点的父结点，使得删除操作的复杂度退化成 O(n)，还有一个问题就是 x 本身如果就是根结点，那么一旦 x 删除，它的子结点就会妻离子散，家破人亡，需要重新建立关系，越想越复杂。

所以，及时制止记录子结点的想法，采用一种新的思维——二次哈希法（ReHash），对于每个结点都有一个哈希值，在进行查找之前需要将 x 转化成它的哈希值 HASH[x]，那么在进行删除的时候，只要将 x 的哈希值进行改变，变成一个从来没有出现过的值（可以采用一个计数器来实现这一步），然后对新的值建立集合，因为只有它一个元素，所以必定是一个新的集合。这样做可以保证每次删除操作的时间复杂度都是 O(1)的，而且不会破坏原有树结构，唯一的一个缺点就是每删除一个结点其实是多申请了一块内存，如果删除操作无限制，那么内存会无限增长。

四、RMQ

1、朴素算法

RMQ (Range Minimum/Maximum Query)问题是指：对于长度为 n 的数列 Array，回答若干询问 RMQ(Array,i,j)(i,j<=n)，返回数列 Array 中下标在 i,j 里的最小(大)值（或者最小（大）值所在的下标），也就是说，RMQ 问题是指求区间最值的问题。

朴素算法就是枚举区间的值进行大小判定，如果长度为 n 的数组，询问个数为 q，那么算法的时间复杂度为 O(qn)。

2、线段树（简介）

线段树是一种二叉搜索树，它的每个结点都保存一个区间[l, r]，如果当 l 等 r 时，表示该结点是一个叶子结点；否则它必定有两个子结点，两个子结点保存的区间分别为[l, mid]和[mid+1, r],其中 mid = (l+r)/2 的下整，利用二分（分治）的思想将一个长度为 n 的区间分割成一系列单位区间（叶子区间）。

线段树的每个结点都可以保存一些信息，最经典的应用就是区间最值，可以在结点上保存该结点对应区间[l, r]上的最值，每次询问[A, B]区间最值时将区间[A, B]进行划分，划分成一些区间的并集，并且集合中的区间必定都能在线段树结点上一一对应，可以证明一定存在这样一个划分，并且划分的集合个数不会超过 logn，从而使得每次查询的时间复杂度能够保证在 O(logn)。

3、ST 算法

ST（Sparse Table）算法是基于动态规划的，用 f[i][j]表示区间起点为 j 长度为 2^i 的区间内的最小值所在下标，通俗的说，就是区间[j, j + 2^i)的区间内的最小值的下标。

从定义可知，这种表示法的区间长度一定是 2 的幂，所以除了单位区间(长度为 1 的区间)以外，任意一个区间都能够分成两份，并且同样可以用这种表示法进行表示，[j, j + 2^i)的区间可以分成[j, j+2^(i-1))和[j + 2^i)，于是可以列出状态转移方程为： f[i][j] = RMQ(f[i-1][j], f[i-1][j+2^(i-1)])。f[m][n]的状态数目为 n*m = nlogn，每次状态转移耗时 O(1)，所以预处理总时间为 O(nlogn)。

原数组长度为 n，当[j, j + 2^i)区间右端点 j + 2^i - 1 > n 时如何处理？在状态转移方程中只有一个地方会下标越界，所以当越界的时候状态转移只有一个方向，即当 j + 2^(i-1) > n 时，f[i][j] = f[i-1][j]。

求解 f[i][j]的代码就不给出了，只需要两层循环的状态转移就搞定了。

f[i][j]的计算只是做了一步预处理，但是我们在询问的时候，不能保证每个询问区间长度都是 2 的幂，如何利用预处理出来的值计算任何长度区间的值就是我们接下来要解决的问题。

首先只考虑区间长度大于 1 的情况（区间长度为 1 的情况最小值就等于它本身），给定任意区间[a, b] (1 <= a < b <= n)，必定可以找到两个区间 X 和 Y，它们的并是[a, b]，并且区间 X 的左端点是 a，区间 Y 的右端点是 b，而且两个区间长度相当，且都是 2 的幂，如图所示：

设区间长度为 2^k，则 X 表示的区间为[a, a + 2^k)，Y 表示的区间为(b - 2^k, b]，则需要满足一个条件就是 X 的右端点必须大于等于 Y 的左端点减一，即 a+2^k-1 >= b-2^k，则 2^(k+1) >= (b-a+1), 两边取对数（以 2 为底），得 k+1 >= lg(b-a+1)，则 k >= lg(b-a+1) - 1，k 只要需要取最小的满足条件的整数即可(lg(x)代表以 2 为底 x 的对数)。

仔细观察发现 b-a+1 正好为区间[a, b]的长度 len，所以只要区间长度一定，k 就能在常数时间内求出来。而区间长度只有 n 种情况，所以 k 可以通过预处理进行预存。

当 lg(len)为整数时，k 取 lg(len)-1，否则 k 为 lg(len)-1 的上整（并且只有当 len 为 2 的幂时，lg(len)才为整数）。

代码如下：

```
for(i = 1, k = 0; i <= n; i++) {
   lg2K[i] = k - 1;
   if((1<<k) == i) k++;
}
int RMQ_Query(ValueType val[], int (*f)[MAXN], int a, int b) {
   if(a == b) return a;
   int k = lg2K[ abs(b-a) + 1 ];
   int x = f[k][a], y = f[k][b-(1<<k)+1];
   return val[x] < val[y] ? x : y;
}
```

val 数组代表原数组，f 是个二维数组，即上文提到的动态规划数组，x 和 y 分别对应了图四-3-1 中 X 区间和 Y 区间的最小值所在的下标。将这两部分在原数组中的数值进行比较就能得到整个[a, b]区间内的最小值所在下标了。

ST 算法可以扩展到二维，用四维的数组来保存状态，每个状态表示的是一个矩形区域中的最值，可以用来求解矩形区域内的最值问题。

线段树

一、引例

1、区间最值

【例题 1】给定一个 n（n <= 100000）个元素的数组 A，有 m(m <= 100000)个操作，共两种操作：

1、Q a b　　询问：表示询问区间[a, b]的最大值；

2、C a c　　更新：表示将第 a 个元素变成 c；

静态的区间最值可以利用 RMQ 来解决，但是 RMQ 的 ST 算法是在元素值给定的情况下进行的预处理，然后在 O(1)时间内进行询问，这里第二种操作需要实时修改某个元素的值，所以无法进行预处理。

由于每次操作都是独立事件，所以 m 次操作都无法互相影响，于是时间复杂度的改善只能在单次操作上进行优化了，我们可以试想能否将任何的区间[a, b]（a < b）都拆成 log(b-a+1)个小区间，然后只对这些拆散的区间进行询问，这样每次操作的最坏时间复杂度就变成 log(n)了。

2、区间求和

【例题 2】给定一个 n(n <= 100000)个元素的数组 A，有 m(m <= 100000)个操作，共两种操作：

1、Q a b　　询问：表示询问区间[a, b]的元素和；

2、A a b c 更新：表示将区间[a, b]的每个元素加上一个值 c；

先来看朴素算法，两个操作都用遍历来完成，单次时间复杂度在最坏情况下都是 O(n)的，所以 m 次操作下来总的时间复杂度就是 O(nm)了，复杂度太高。

再来看看树状数组，对于第一类操作，树状数组可以在 log(n)的时间内出解；然而第二类操作，还是需要遍历每个元素执行 add 操作，复杂度为 nlog(n)，所以也不可行。这个问题同样也需要利用区间拆分的思想。

线段树就是利用了区间拆分的思想，完美解决了上述问题。

二、线段树的基本概念

1、二叉搜索树

线段树是一种二叉搜索树，即每个结点最多有两棵子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。线段树的每个结点存储了一个区间（线段），故而得名。基于线段树的二分性质，所以它是一棵平衡树，树的高度为 log(n)。

2、数据域

首先，既然线段树的每个结点表示的是一个区间，那么必须知道这个结点管辖的是哪个区间，所以其中最重要的数据域就是区间左右端点[l, r]。然而有时候为了节省全局空间，往往不会将区间端点存储在结点中，而是通过递归的传参进行传递，实时获取。再者，以区间最大值为例，每个结点除了需要知道所管辖的区间范围[l, r]以外，还需要存储一个当前区间内的最大值 max。

以数组 A[1:6] = [1 7 2 5 6 3]为例，建立如图二-2-1 的线段树，叶子结点的 max 域为数组对应下标的元素值，非叶子结点的 max 域则通过自底向上的计算由两个儿子结点的 max 域比较得出。这是一棵初始的线段树，接下来讨论下线段树的询问和更新操作。

在询问某个区间的最大值时，我们一定可以将这个区间拆分成 log(n)个子区间，并且这些子区间一定都能在线段树的结点上找到（这一点下文会着重讲解），然后只要比较这些结点的 max 域，就能得出原区间的最大值了，因为子区间数量为 log(n)，所以时间复杂度是 O(log(n))。

更新数组某个元素的值时我们首先修改对应的叶子结点的 max 域，然后修改它的父结点的 max 域，以及祖先结点的 max 域，换言之，修改的只是线段树的叶子结点到根结点的某一条路径上的 max 域，又因为树高是 log(n)，所以这一步操作的时间复杂度也是 log(n)的。

3、指针表示

接下来讨论一下结点的表示法，每个结点可以看成是一个结构体指针，由数据域和指针域组成，其中指针域有两个，分别为左儿子指针和右儿子指针，分别指向左右子树；数据域存储对应数据，根据情况而定(如

果是求区间最值，就存最值 max；求区间和就存和 sum)，这样就可以利用指针从根结点进行深度优先遍历了。
以下是简单的线段树结点的 C++结构体：

```
struct treeNode {
   Data data; // 数据域
   treeNode *lson, *rson; // 指针域
}*root;
```

4、数组表示

实际计算过程中，还有一种更加方便的表示方法，就是基于数组的静态表示法，需要一个全局的结构体数组，每个结点对应数组中的一个元素，利用下标索引。
例如，假设某个结点在数组中下标为 p，那么它的左儿子结点的下标就是 2*p，右儿子结点的下标就是 2*p+1(类似于一般数据结构书上说的堆在数组中的编号方式)，这样可以将所有的线段树结点存储在相对连续的空间内。之所以说是相对连续的空间，是因为有些下标可能永远用不到。
还是以长度为 6 的数组为例，如图二-4-1 所示，红色数字表示结点对应的数组下标，由于树的结构和编号方式，导致数组的第 10、11 位置空缺。

图二-4-1
这种存储方式可以不用存子结点指针，取而代之的是当前结点的数组下标索引，以下是数组存储方式的线段树结点的 C++结构体：

```
struct treeNode {
   Data data; // 数据域
   int pid;     // 数组下标索引
   int lson() { return pid << 1; }
   int rson() { return pid<<1|1; }// 利用位运算加速获取子结点编号
}nodes[ MAXNODES ];
```

接下来我们关心的就是 MAXNODES 的取值了，由于线段树是一种二叉树，所以当区间长度为 2 的幂时，它正好是一棵满二叉树，数组存储的利用率达到最高（即 100%），根据等比数列求和可以得出，满二叉树的结点个数为 2*n-1，其中 n 为区间长度（由于 C++中数组长度从 0 计数，编号从 1 开始，所以 MAXNODES 要取 2*n）。那么是否对于所有的区间长度 n 都满足这个公式呢？答案是否定的，当区间长度为 6 时，最大的结点编号为 13，而公式算出来的是 12（2*6）。
那么 MAXNODES 取多少合适呢？
为了保险起见，我们可以先找到比 n 大的最小的二次幂，然后再套用等比数列求和公式，这样就万无一失了。举个例子，当区间长度为 6 时，MAXNODES = 2 * 8；当区间长度为 1000，则 MAXNODES = 2 * 1024；当区间长度为 10000，MAXNODES = 2 * 16384。至于为什么可以这样，明眼人一看便知。

三、线段树的基本操作

线段树的基本操作包括构造、更新、询问，都是深度优先搜索的过程。

1、构造

线段树的构造是一个二分递归的过程，封装好了之后代码非常简洁，总体思路就是从区间[1，n]开始拆分，拆分方式为二分的形式，将左半区间分配给左子树，右半区间分配给右子树，继续递归构造左右子树。
当区间拆分到单位区间时（即遍历到了线段树的叶子结点），则执行回溯。回溯时对于任何一个非叶子结点需要根据两棵子树的情况进行统计，计算当前结点的数据域，详见注释 4。

```
void segtree_build(int p, int l, int r) {
   nodes[p].reset(p, l, r);// 注释 1
```

```
  if (l < r) {
int mid = (l + r) >> 1;
segtree_build(p<<1, l, mid);// 注释 2
segtree_build(p<<1|1, mid+1, r);// 注释 3
nodes[p].updateFromSon(); // 注释 4
  }
}
```

注释 1：初始化第 p 个结点的数据域，根据实际情况实现 reset 函数

注释 2：递归构造左子树

注释 3：递归构造右子树

注释 4：回溯，利用左右子树的信息来更新当前结点，updateFromSon 这个函数的实现需要根据实际情况进行求解，在第四节会详细讨论

构造线段树的调用如下：segtree_build(1, 1, n);

2、更新

线段树的更新是指更新数组在[x, y]区间的值，具体更新这件事情是做了什么要根据具体情况而定，可以是将[x, y]区间的值都变成 val（覆盖），也可以是将[x, y]区间的值都加上 val（累加）。

更新过程采用二分，将[1, n]区间不断拆分成一个个子区间[l, r]，当更新区间[x, y]完全覆盖被拆分的区间[l, r]时，则更新管辖[l, r]区间的结点的数据域，详见注释 2 和注释 3。

```
void segtree_insert(int p, int l, int r, int x, int y, ValueType val) {
      if( !is_intersect(l, r, x, y) ) {   // 注释 1
      return ;
}
if( is_contain(l, r, x, y) ) { // 注释 2
      nodes[p].updateByValue(val);    // 注释 3
      return ;
}
  nodes[p].giveLazyToSon();// 注释 4
  int mid = (l + r) >> 1;
  segtree_insert(p<<1, l, mid, x, y, val); // 注释 5
  segtree_insert(p<<1|1, mid+1, r, x, y, val);    // 注释 6
  nodes[p].updateFromSon();// 注释 7
}
```

注释 1：区间[l, r]和区间[x, y]无交集，直接返回

注释 2：区间[x, y]完全覆盖[l, r]

注释 3：更新第 p 个结点的数据域，updateByValue 这个函数的实现需要根据具体情况而定，会在第四节进行详细讨论

注释 4：这里先卖个关子，参见第五节的 lazy-tag

注释 5：递归更新左子树

注释 6：递归更新右子树

注释 7：回溯，利用左右子树的信息来更新当前结点

更新区间[x, y]的值为 val 的调用如下：segtree_insert(1, 1, n, x, y, val);

3、询问

线段树的询问和更新类似，大部分代码都是一样的，只有红色部分是不同的，同样是将大区间[1, n]拆分成一个个小区间[l, r]，这里需要存储一个询问得到的结果 ans，当询问区间[x, y]完全覆盖被拆分的区间[l, r]

时，则用管辖[l, r]区间的结点的数据域来更新 ans，详见注释 1 的 mergeQuery 接口。

```
void segtree_query (int p, int l, int r, int x, int y, treeNode& ans) {
  if( !is_intersect(l, r, x, y) ) {
return ;
  }
  if( is_contain(l, r, x, y) ) {
ans.mergeQuery(p);  // 注释 1
return;
  }
  nodes[p].giveLazyToSon();
  int mid = (l + r) >> 1;
  segtree_query(p<<1, l, mid, x, y, ans);
  segtree_query(p<<1|1, mid+1, r, x, y, ans);
  nodes[p].updateFromSon(); // 注释 2
}
```

注释 1：更新当前解 ans，会在第四节进行详细讨论

注释 2：和更新一样的代码，不再累述

四、线段树的经典案例

线段树的用法千奇百怪，接下来介绍几个线段树的经典案例，加深对线段树的理解。

1、区间最值

区间最值是最常见的线段树问题，引例中已经提到。接下来从几个方面来讨论下区间最值是如何运作的。

数据域：

```
  int pid;    // 数组索引
  int l, r;  // 结点区间(一般不需要存储)
  ValyeType max;    // 区间最大值
```

初始化：

```
  void treeNode::reset(int p, int l, int r) {
pid = p;
max = srcArray[l]; // 初始化只对叶子结点有效
  }
```

单点更新：

```
  void treeNode::updateByValue(ValyeType val) {
max = val;
  }
```

合并结点：

```
  void treeNode::mergeQuery(int p) {
max = getmax( max, nodes[p].max );
  }
```

回溯统计：

```
  void treeNode::updateFromSon() {
max = nodes[ lson() ].max;
mergeQuery( rson() );
  }
```

结合上一节线段树的基本操作，在构造线段树的时候，对每个结点执行了一次初始化，初始化同时也是单

点更新的过程，然后在回溯的时候统计，统计实质上是合并左右结点的过程，合并结点做的事情就是更新最大值；询问就是将给定区间拆成一个个能够在线段树结点上找到的区间，然后合并这些结点的过程，合并的结果 ans 一般通过引用进行传参，或者作为全局变量，不过尽量避免使用全局变量。

2、区间求和

函数的实现。

数据域：

```
  int pid;    // 数组索引
  int len;    // 结点区间长度
  ValyeType sum;    // 区间元素和
  ValyeType lazy;   // lazy tag
```

初始化：

```
  void treeNode::reset(int p, int l, int r) {
pid = p;
len = r - l + 1;
sum = lazy = 0;
  }
```

单点更新：

```
  void treeNode::updateByValue(ValyeType val) {
lazy += val;
sum += val * len;
  }
```

lazy 标记继承：

```
  void treeNode::giveLazyToSon() {
if( lazy ) {
    nodes[ lson() ].updateByValue(lazy);
    nodes[ rson() ].updateByValue(lazy);
    lazy = 0;
}
  }
```

合并结点：

```
  void treeNode::mergeQuery(int p) {
sum += nodes[p].sum;
  }
```

回溯统计：

```
  void treeNode::updateFromSon() {
sum = nodes[ lson() ].sum;
mergeQuery( rson() );
  }
```

对比区间最值，区间求和的几个函数的实现主旨是一致的，因为引入了 lazy-tag，所以需要多实现一个函数用于 lazy 标记的继承，在进行区间求和的时候还需要记录一个区间的长度 len，用于更新的时候计算累加的 sum 值。

3、区间染色

【例题 3】给定一个长度为 n(n <= 100000)的木板，支持两种操作：

1、P a b c 将[a, b]区间段染色成 c；

2、Q a b　　询问[a, b]区间内有多少种颜色；

保证染色的颜色数少于 30 种。

对比区间求和，不同点在于区间求和的更新是对区间和进行累加；而这类染色问题则是对区间的值进行替换（或者叫覆盖），有一个比较特殊的条件是颜色数目小于 30。

我们是不是要将 30 种颜色的有无与否都存在线段树的结点上呢？答案是肯定的，但是这样一来每个结点都要存储 30 个 bool 值，空间太浪费，而且在计算合并操作的时候有一步 30 个元素的遍历，大大降低效率。然而 30 个 bool 值正好可以压缩在一个 int32 中，利用二进制压缩可以用一个 32 位的整型完美的存储 30 种颜色的有无情况。

因为任何一个整数都可以分解成二进制整数，二进制整数的每一位要么是 0，要么是 1。二进制整数的第 i 位是 1 表示存在第 i 种颜色；反之不存在。

数据域需要存一个颜色种类的位或和 colorBit，一个颜色的 lazy 标记表示这个结点被完全染成了 lazy，基本操作的几个函数和区间求和非常像，这里就不出示代码了。

和区间求和不同的是回溯统计的时候，对于两个子结点的数据域不再是加和，而是位或和。

4、矩形面积并

【例题 4】给定 n(n <= 100000)个平行于 XY 轴的矩形，求它们的面积并。这类二维的问题同样也可以用线段树求解，核心思想是降维，将某一维套用线段树，另外一维则用来枚举。具体过程如下：

第一步：将所有矩形拆成两条垂直于 x 轴的线段，平行 x 轴的边可以舍去，

第二步：定义矩形的两条垂直于 x 轴的边中 x 坐标较小的为入边，x 坐标较大的为出边，入边权值为+1，出边权值为-1，并将所有的线段按照 x 坐标递增排序，第 i 条线段的 x 坐标记为 X[i]。

第三步：将所有矩形端点的 y 坐标进行重映射(也可以叫离散化)，原因是坐标有可能很大而且不一定是整数，将原坐标映射成小范围的整数可以作为数组下标，更方便计算，映射可以将所有 y 坐标进行排序去重，然后二分查找确定映射后的值，离散化的具体步骤下文会详细讲解。

第四步：以 x 坐标递增的方式枚举每条垂直线段，y 方向用一个长度为 m-1 的数组来维护“单位线段”的权值，展示了每条线段按 x 递增方式插入之后每个“单位线段”的权值。

当枚举到第 i 条线段时，检查所有“单位线段”的权值，所有权值大于零的“单位线段”的实际长度之和(离散化前的长度)被称为“合法长度”，记为 L，那么(X[i] - X[i-1]) * L，就是第 i 条线段和第 i-1 条线段之间的矩形面积和，计算完第 i 条垂直线段后将它插入，所谓"插入"就是利用该线段的权值更新该线段对应的“单位线段”的权值和（这里的更新就是累加）。红色、黄色、蓝色三个矩形分别是 3 对相邻线段间的矩形面积和，其中红色部分的 y 方向由<1-2>、<2-3>两个“单位线段”组成，黄色部分的 y 方向由<1-2>、<2-3>、<3-4>三个“单位线段”组成，蓝色部分的 y 方向由<2-3>、<3-4>两个“单位线段”组成。特殊的，在计算蓝色部分的时候，<1-2>部分的权值由于第 3 条线段的插入(第 3 条线段权值为-1)而变为零，所以不能计入“合法长度”。以上所有相邻线段之间的面积和就是最后要求的矩形面积并。

那么这里带来几个问题：

1、是否任意相邻两条垂直 x 轴的线段之间组成的封闭图形都是矩形呢？答案是否定的，如图四-4-7 所示，其中绿色部分为四个矩形的面积并中的某块有效部分，它们同处于两条相邻线段之间，但是中间有空隙，所以它并不是一个完整的矩形。

2、每次枚举一条垂直线段的时候，需要检查所有“单位线段”的权值，如果用数组维护权值，那么这一步检查操作是 O(m)的，所以总的时间复杂度为 O(nm)，其中 n 表示垂直线段的个数，复杂度太大需要优化。

优化自然就是用线段树了，之前提到了降维的思想，x 方向我们继续采用枚举，而 y 方向的“单位线段”则可以采用线段树来维护，和一般问题一样，首先讨论数据域。

数据域：

```
int pid;   // 数组索引
int l, r; // 结点代表的“单位线段”区间[l, r] (注意，l 和 r 均为离散后的下标)
int cover;// [l, r]区间被完全覆盖的次数
```

int len; // 该结点表示的区间内的合法长度

注意，这次的线段树和之前的线段树稍微有点区别，就是叶子结点的区间端点不再相等，而是相差 1，即 l+1 == r。因为一个点对于计算面积来说是没有意义的。

算法采用深度优先搜索的后序遍历，记插入线段为[a, b, v]，其中[a, b]为线段的两个端点，是离散化后的坐标；v 是+1 或-1，代表是入边还是出边，每次插入操作二分枚举区间，当线段树的结点代表的区间被插入区间完全覆盖时，将权值 v 累加到结点的 cover 域上。由于是后续遍历，在子树全部遍历完毕后需要进行统计。插入过程修改 cover，同时更新 len。

回溯统计过程对 cover 域分情况讨论：

当 cover > 0 时，表示该结点代表的区间至少有一条入边没有被出边抵消，换言之，这块区间都应该在“合法长度”之内，则 len = Y[r] - Y[l]（Y[i]代表离散前第 i 大的点的 y 坐标）；更加通俗的理解是至少存在一个矩形的入边被扫描到了，而出边还未被扫描到，所以这块面积需要被计算进来。

当 cover 等于 0 时，如果该区间是一个单位区间（即上文所说的“单位线段”，l+1 == r，也是线段树的叶子结点），则 len = 0；否则，len 需要由左子树和右子树的计算结果得出，又因为是后序遍历，所以左右子树的 len 都已经计算完毕，从而不需要再进行递归求解，直接将左右儿子的 len 加和就是答案，即 len = lson.len + rson.len。

这样就可以利用二分，在 O(n)的时间内递归构造初始的线段树。

5、区间 K 大数

【例题 5】给定 n(n <= 100000)个数的数组，然后 m(m <= 100000)条询问，询问格式如下：

1、l r k 询问[l, r]的第 K 大的数的值

这是一个经典的面试题，利用了线段树划分区间的思想，线段树的每个结点存的不只是区间端点，而是这个区间内所有的数，并且是按照递增顺序有序排列的，建树过程是一个归并排序的过程，从叶子结点自底向上进行归并，对于一个长度为 6 的数组[4, 3, 2, 1, 5, 6]，建立线段树如图四-5-1 所示。

从图中可以看出，线段树的任何一个结点存储了对应区间的数，并且进行有序排列，所以根结点存储的一定是一个长度为数组总长的有序数组，叶子结点存储的递增序列为原数组元素。

每次询问，我们将给定区间拆分成一个个线段树上的子区间，然后二分枚举答案 T，再利用二分查找统计这些子区间中大于等于 T 的数的个数，从而确定 T 是否是第 K 大的。

五、线段树的常用技巧

1、离散化

在讲解矩形面积并的时候曾经提了一下离散化，现在再详细的说明一下，所谓离散化就是将无限的个体映射到有限的个体中，从而提高算法效率。

举个简单的例子，一个实数数组，我想很快的得到某个数在整个数组里是第几大的，并且询问数很多，不允许每次都遍历数组进行比较。

那么，最直观的想法就是对原数组先进行一个排序，询问的时候只需要通过二分查找就能在 O(log(n))的时间内得出这个数是第几大的了，离散化就是做了这一步映射。

对于一个数组[1.6, 7.8, 5.5, 11.1111, 99999, 5.5]，离散化就是将原来的实数映射成整数(下标)。

这样就可以将原来的实数保存在一个有序数组中，询问第 K 大的是什么称为正查，可以利用下标索引在 O(1)的时间内得到答案；询问某个数是第几大的称为反查，可以利用二分查找或者 Hash 得到答案，复杂度取决于具体算法，一般为 O(log(n))。

2、lazy-tag

这个标记一般用于处理线段树的区间更新。

线段树在进行区间更新的时候，为了提高更新的效率，所以每次更新只更新到更新区间完全覆盖线段树结点区间为止，这样就会导致被更新结点的子孙结点的区间得不到需要更新的信息，所以在被更新结点上打上一个标记，称为 lazy-tag，等到下次访问这个结点的子结点时再将这个标记传递给子结点，所以也可以叫延迟标记。

3、子树收缩

子树收缩是子树继承的逆过程，子树继承是为了两棵子树获得父结点的信息；而子树收缩则是在回溯的时候，如果两棵子树拥有相同数据的时候在将数据传递给父结点，子树的数据清空，这样下次在访问的时候就可以减少访问的结点数。

六、线段树的多维推广

1、二维线段树 - 矩形树

线段树是处理区间问题的，二维线段树就是处理平面问题的了，曾经写过一篇二维线段树的文章，就不贴过来了，直接给出传送门：二维线段树。

2、三维线段树 - 空间树

线段树-二叉树，二维线段树-四叉树，三维线段树自然就是八叉树了，分割的是空间，一般用于三维计算几何，当然也不一定用在实质的空间内的问题。

比如需要找出身高、体重、年龄在一定范围内并且颜值最高的女孩子，就可以用三维线段树（三维空间最值问题），嘿嘿嘿！！！

计算几何技巧

Pick 定理和叉积求多边形的面积。
Pick 定理：一个计算点阵中顶点在格点上的多边形面积公式：S=a+b/2-1，其中 a 表示多边形内部的点数，b 表示多边形边界上的点数，s 表示多边形的面积。

多边形面积：
1) △ABC 的面积为向量 AB 与向量 AC 的叉乘的一半。
2)对于一个多边形，选定一个顶点 P1，与其他顶点连线，可将多边形分为若干个三角形。
3)多边形面积为 abs(Sum{CrossMul(A,B,P1)|A,B 为相邻的两个顶点}) (先求和再取 abs，否则对于凹多边形会出错）

求在边上的顶点数：
对于 Pa(x1,y1),Pb(x2,y2)所连成的选段，经过的格点的个数为 Gcd(abs(x1-x2),abs(y1-y2))+1.

计算机专业常见术语

A
access，获取，存取
Active Directory，活动目录
affinity，绑定
agent，代理
agent-based interface，代理人界面
AI，Artificial Intelligence，人工智能
air waves，无线电波
algorithm，算法
analog，模拟的
application，应用，应用程序，应用软件
application pool，应用程序池
atomic opreation，原子操作
atomic transaction，原子事务
atomicity，原子性
augmented reality，增强实现
authorization，授权
automation，自动化
autonomous，独立性
availability (set)，可用性(集)

B
bandwidth，带宽
bar code，条形码
baud，波特
behavior，行为
big data，大数据
binary，二进制
bit，比特
bitnik，比特族
blob，BLOB
block，阻断
block blob，块 BLOB
bottleneck，瓶颈
bps，bits per second，比特/秒
broadcast，广播
browser-server，浏览器-服务器
bug，缺陷
built-in，内置的，内建的；嵌入的；内置
business layer，业务层
busy (status)，忙（状态）；繁忙（状态）
byte，字节

C
Cache/Caching，缓存
call stack，调用堆栈
carbon copy，复写本，副本；抄送（CC）
carriage return，回车
cell，单元
cellular telephone，移动电话
Central Processing Unit，中央处理器（CPU）
certificate，（数字）证书
Certificate Authority，证书认证机构
channel，信道，频道
character，字符
check in，签入
check out，签出
chip，芯片
cipher，密码
claim，声明
client-server，客户端-服务器
clone，克隆，复制
cloud computing，云计算
cloud service，云服务
cluster，集群
command，命令,按钮
command prompt，命令行提示
commingled bits，混合的比特
communication，通信
community，社区
committed，已提交（的）
common name，通用名称
compatibility，兼容性
comcurrency，并发
concurrency mode，并发模式
conditional compilation，条件编译
configuration，配置，设置
connection string，连接字符串
consistenct，一致性
constructor，构造函数
container，容器
content，内容

context，上下文
contribute，贡献
convert，转换
cookie，Cookie
core，内核
CPU，中央处理器（Central Processing Unit）
crash，（程序）崩溃
crash dump，故障转储
cursor，光标

D
data integrity，数据完整性
data mining，数据挖掘
dependenct injection，依赖注入（DI）
deployment，部署
dequeue，出列
derives from 继承
device，设备
DI，依赖注入（dependenct injection）
diagnostics，诊断
directive，指令
discussion forum，论坛
disk，磁盘
distributed system，分布式系统
dummy function，虚构函数
durability，持久性

E
EAP ,早期评估版本(Early Assessment Program）
Early Assessment Program ,早期评估版本(EAP）
Egress，流出
elasticity，弹性
Element (XML)，元素
endpoint，端点
enqueue，入列；加入队列
entity，实体
exception handling，异常处理
Exclusive OR，异或（XOR）
explanatory figures，图示

F
failover，容错转移
fat client，胖客户端
FDD，软盘（Floopy Disk Drive）
Floopy Disk Drive，软盘（FDD）
full-duplex，全双工
Full Packaged Product，零售版（FPP）

G
Geo-Replication，地域复制
Geo Redundant，地域冗余

H
handle，句柄
Hard Disk Drive，硬盘（HDD）
DHH，硬盘（Hard Disk Drive）
header，头；标头；表头
High Avaliability，高可用性
Homogeneous，同质化
Horizontal Scale，水平缩放
Hosting，宿主
Hybrid Cloud，混合云

I
Iaas，设施即服务（Infrastructure as a Service）
image，映像
index，索引
Infrastructure as a Service，设施即服务（Iaas）
ingesting，摄取
ingress，流入
input endpoint，输入端点
instance，实例
Isolation，隔离性

J
Job，作业

K
Key，密钥
Key-Value Pair，键-值对

L
Large，大型
license，许可证
lifetime，生命周期
link，链接
load-balancing，负载平衡
load balancer，负载平衡器

log，日志

M
Mainframe，主机
Maintainability，可维护性
Management Key，管理密钥
Media Service，媒体服务
Medium，中型
Merge，合并
Metadata，元数据
Middleware，中间件
Mobile Service，移动服务
Multitenancy，多租户

N
Namespace，命名空间，名称空间
node，节点
normalize，规格化
notification，通知
notification hub，通知中心
N-Tier，N 层（结构）

O
Overview，概览
P
Paas，平台即服务（Platform as a Service）
Page Blob，页 BLOB
partition，分区
passive，被动（的）
Pay as You Go，即用即付
PC，个人计算机（Personal Computer）
peek，查看
performance，性能
performance counter，性能计数器
Personal Computer，个人计算机（PC）
Platform as a Service，平台即服务（Paas）
polling，轮询
presentation layer，表现层
private cloud，私有云
priority queue，优先级队列
process，进程
production，生产（环境）
protocol，协议
proxy，代理
public cloud，公有云
push，推送

Q
Queue，队列
Quota，配额

R
Rack，机架
Ready (status)，就绪（状态）
real-time，即时、实时
real-time discussions，即时讨论、实时讨论
Redundancy，冗余
Redundant，冗余（的）
Refactor，重构
region，地域
relay，中继
Reliability，可靠性
reporting，报表
Repository，存储库；仓储；仓库
reserved，专属
reverse proxy module，反向代理模块
retail，零售版
role，角色
Rolling Upgrade，滚动升级
round-robin，轮流（分配）；轮叫
router，路由器
row，行

S
Saas，软件即服务（Software as a Service）
Scale，缩放
Scale Out，向外缩放
Scale Up，向上缩放
Schema (database)，架构（数据）
Schema (xml)，架构（xml）
Security，安全（性）
Setting，设置
Shared，共享；分享
Signature，签名
SLA，服务水平协议（Service Level Agreement）
Small，小型
snapshot，快照
Software as a Service，软件即服务（Saas）

SQL Database，SQL 数据库
Sign in，登录
Sign out，注销
Site，站点
Site-to-Site，站点到站点
Storage，存储
Storage Account，存储账户
Subscription，订阅

T
Table，表
Tenant，租户
Terminus，端点
Thin Client，瘦客户端
Thread，线程
Thread Pool，线程池
Topology，拓扑结构
Token，令牌
(Code) Tracing，(代码）追踪
Transaction，事务

U
Unit test，单元测试
Uncommitted，未提交（的）

V
VIP，虚拟 IP（或不译）,会员
VIP Swap，VIP 交换
Virtual Machine，虚拟机

W
Wearable Device，可穿戴设备
Web Role，网站角色
Web Service，网络服务
Web Sites，网站
WINS Proxy，WINS 代理
WINS Resource，WINS 资源
wireless communication，无线通讯
WMI ， Windows 管 理 规 范 （ Windows Management Instrumentation）
Worker Role，辅助角色
Workflow，工作流
workgroup，工作组

X
X.509v3 certificate，X.509 证书
XOR，异或（Exclusive OR）

Y

Z
Zero-downtime Upgrade，零停机升级
zip disk，压缩磁盘
zone，区域
zone list，区域列表
zone transfer，区域传送