# 目录



## 一、深度优先搜索

### 1、DFS

1) 算法原理

深度优先搜索即Depth First Search，是图遍历算法的一种。用一句话概括就是：“一直往下走，走不通回头，换条路再走，直到无路可走”。

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（也可以用递归实现，更加方便易懂）。

如图1所示，对以下图以深度优先的方式进行遍历，假设起点是1，访问顺序为1 -> 2 -> 4，由于结点4没有未访问的相邻结点，所以这里需要回溯到2，然后发现2还有未访问的相邻结点5，于是继续访问2 -> 5 -> 6 -> 3 -> 7，这时候7回溯到3，3回溯到6，6回溯到5，5回溯到2，最后2回溯到起点1，1已经没有未访问的结点了，搜索终止，图中圆圈代表路点，红色箭头表示搜索路径，蓝色虚线表示回溯路径。

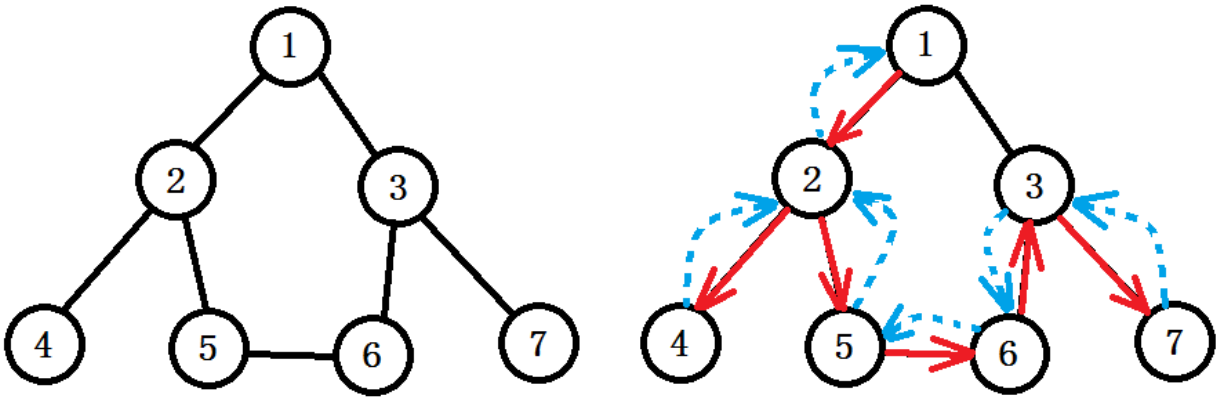



图1

2) 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。

3) 基础应用

a. 求N的阶乘；

令f(N) = N!，那么有f(N) = N * f(N-1)（其中N>0）。由于满足递归的性质，可以认为是一个N个结点的图，结点 i (i >= 1) 到结点 i-1 有一条权值为i的有向边，从N开始深度优先遍历，遍历的终点是结点0，返回1(因为0! = 1)。如图2所示，N!的递归计算看成是一个深度优先遍历的过程，并且每次回溯的时候会将遍历的结果返回给上一个结点（这只是一个思想，并不代表这是求N!的高效算法）。

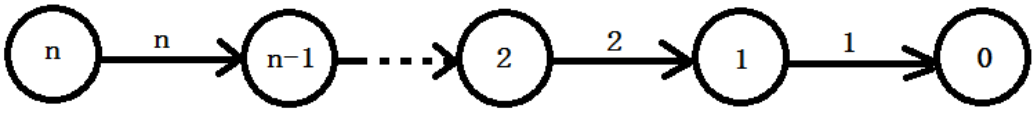



图2

b. 求斐波那契数列的第N项；

令g(N) = g(N-1) + g(N-2)，(N > 2)，其中g(1) = g(2) = 1，同样可以利用图论的思想，从结点N向N-1和N-2分别引一条权值为1的有向边，每次求g(N)就是以N作为起点，对N进行深度优先遍历，然后将N-1和N-2回溯的结果相加作为N结点的值，即g(N)。这里会带来一个问题，g(n)的计算需要用到g(n-1)和g(n-2)，而g(n-1)的计算需要用到g(n-2)和g(n-3)，所以我们发现g(n-2)被用到了两次，而且每个结点都存在这个问题，这样就使得整个算法的复杂度变成指数级了，为了规避这个问题，下面会讲到基于深搜的记忆化搜索。

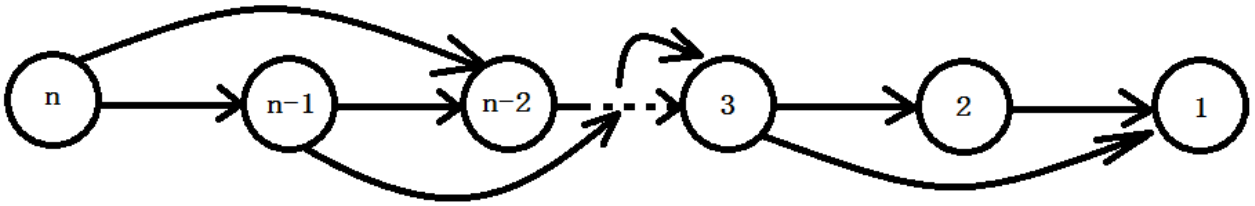



图3

c. 求N个数的全排列；

全排列的种数是N!，要求按照字典序输出。这是最典型的深搜问题。我们可以把N个数两两建立无向边（即任意两个结点之间都有边，也就是一个N个结点的完全图），然后对每个点作为起点，分别做一次深度优先遍历，当所有点都已经标记时输出当前的遍历路径，就是其中一个排列，这里需要注意，回溯的时候需要将原先标记的点的标记取消，否则只能输出一个排列。如果要按照字典序，则需要在遍历的时候保证每次遍历都是按照结点从小到大的方式进行遍历的。

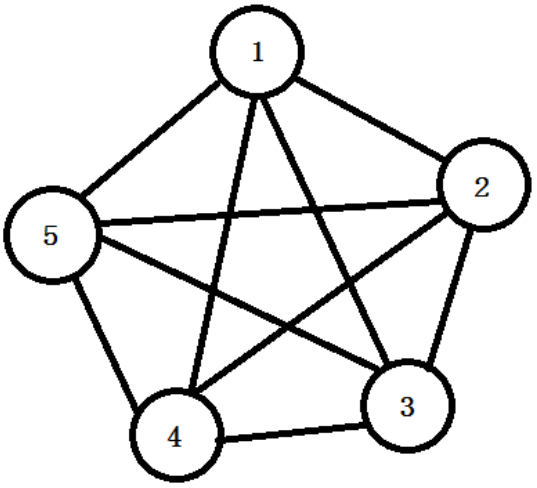



图4

4) 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.博客园

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。

**2、基于DFS的记忆化搜索**

1) 算法原理

上文中已经提到记忆化搜索，其实就是类似动态规划的思想，每次将已经计算出来的状态的值存储到数组中，下次需要的时候直接读数组中的值，避免重复计算。

来看个例子，如图5所示，图中的橙色小方块就是传说中的作者，他可以在一个N*M的棋盘上行走，但是只有两个方向，一个是向右，一个是向下（如绿色箭头所示），棋盘上有很多的金矿，走到格子上就能取走那里的金矿，每个格子的金矿数目不同（用蓝色数字表示金矿的

数量），问作者在这样一个棋盘上最多可以拿到多少金矿。

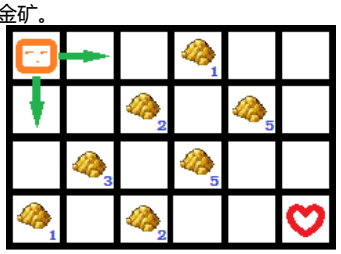


图5

我们用函数DFS(i, j)表示从(1, 1)到(i, j)可以取得金矿的最大值，那么状态转移方程 DFS(i, j) = v[i][j] + max{ DFS(i, j-1), DFS(i-1, j) }（到达(i, j)这个点的金矿最大值的那条路径要么是上面过来的，要么是左边过来的），满足递归性质就可以进行深度优先搜索了，于是遇到了和求斐波那契数列一样的问题，DFS(i, j)可能会被计算两次，每个结点都被计算两次的话复杂度就是指数级了。

所以这里我们可以利用一个二维数组，令D[i][j] = DFS(i, j)，初始化所有的D[i][j] = -1，表示尚未计算，每次搜索到(i, j)这个点时，检查D[i][j]的值，如果为-1，则进行计算，将计算结果赋值给D[i][j]；否则直接返回D[i][j]的值。

记忆化搜索虽然叫搜索，实际上还是一个动态规划问题，能够记忆化搜索的一般都能用动态规划求解，但是记忆化搜索的编码更加直观、易写。

### 3、基于DFS的剪枝

1) 算法原理

搜索的过程可以看作是从树根出发，遍历一棵倒置的树——搜索树的过程。而剪枝，顾名思义，就是通过某种判断，避免一些不必要的遍历过程，形象的说，就是剪去了搜索树中的某些“枝条”，故称剪枝（原话取自1999年OI国家集训队论文《搜索方法中的剪枝优化》（齐鑫））。如图6所示，它是一棵利用深度优先搜索遍历的搜索树，可行解（或最优解）位于黄色的叶子结点，那么根结点的最左边的子树完全没有必要搜索（因为不可能出解）。如果我们在搜索的过程中能够清楚地知道哪些子树不可能出解，就没必要往下搜索了，也就是将连接不可能出解的子树的那根“枝条”剪掉，图中红色的叉对应的“枝条”都是可以剪掉的。

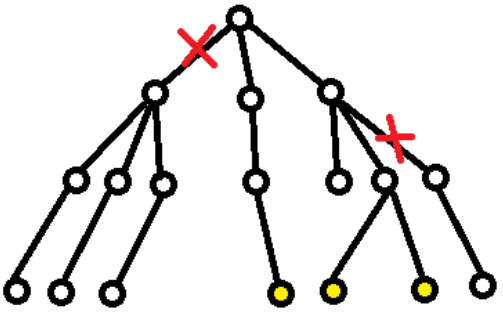


图6

好的剪枝可以大大提升程序的运行效率，那么问题来了，如何进行剪枝？我们先来看剪枝需要满足什么原则：

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。

剪枝大致可以分成两类：可行性剪枝、最优性剪枝（上下界剪枝）。

2) 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。

举个最简单的例子，如图7，问作者能否在正好第11秒的时候避过各种障碍物（图中的东西一看就知道哪些是障碍物了，^_^）最终取得爱心，作者每秒能且只能移动一格，允许走重复的格子。

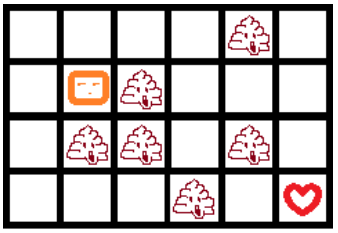


图7

仔细分析可以发现，这是永远不可能的，因为作者无论怎么走，都只能在第偶数秒的时候到达爱心的位置，这是他们的曼哈顿距离（两点的XY坐标差的绝对值之和）的奇偶性决定的，所以这里我们可以在搜索的时候做奇偶性剪枝（可行性剪枝）。

类似的求可行解的问题还有很多，如N (N <= 25) 根长度不一的木棒，问能否选取其中几根，拼出长度为K的木棒，具体就是枚举取木棒的过程，每根木棒都有取或不取两种状态，所以总的状态数为2^25，需要进行剪枝。用到的是剩余和不可达剪枝（随便取的名字，即当前S根木棒取了S1根后，剩下的N-S根木棒的总和 加上 之前取的S1根木棒总和如果小于K，那么必然不满足，没必要继续往下搜索），这个问题其实是个01背包，当N比较大的时候就是动态规划了。

3) 最优性剪枝（上下界剪枝）

最优性剪枝一般是处理最优解的问题。以求两个状态之间的最小步数为例，搜索最小步数的过程：一般情况下，需要保存一个“当前最小步数”，这个最小步数就是当前解的一个下界d。在遍历到搜索树的叶子结点时，得到了一个新解，与保存的下界作比较，如果新解的步数更小，则令它成为新的下界。搜索结束后，所保存的解就是最小步数。而当我们已经搜索了k步，如果能够通过某种方式估算出当前状态到目标状态的理论最少步数s时，就可以计算出起点到目标点的理论最小步数，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。

**4、基于DFS的A*（迭代加深，IDA*）**

1) 算法原理

迭代加深分两步走：

1、枚举深度。

2、根据限定的深度进行DFS，并且利用估价函数进行剪枝。

2) 算法实现

迭代加深写成伪代码如下：

```
def IDA_Star(STATE startState):
    maxDepth = 0
    while true:
        if( DFS(startState, 0, maxDepth) ):
            return
        maxDepth = maxDepth + 1
```

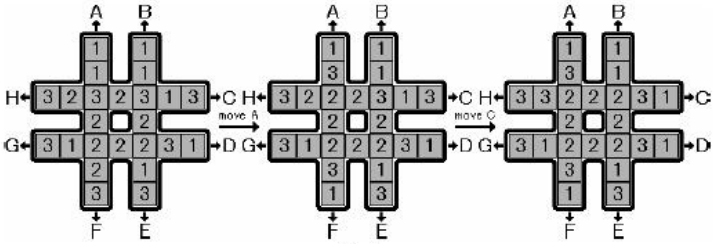



Fig.1

图8

3) 基础应用

如图8所示，一个“井”字形的玩具，上面有三种数字1、2、3，给出8种操作方式，A表示将第一个竖着的列循环上移一格，并且A和F是一个逆操作，B、C、D...的操作方式依此类推，初始状态给定，目标状态是中间8个数字相同。问最少的操作方式，并且要求给出操作的序列，步数一样的时候选择字典序最小的输出。图中的操作序列为AC。

大致分析一下，一共24个格子，每个格子三种情况，所以最坏情况状态总数为3^24，但实际上，我们可以分三种情况讨论，先确定中间的8个数字的值，假设为1的话，2和3就可以看成是一样的，于是状态数变成了2^24。

对三种情况分别进行迭代加深搜索，令当前需要搜索的中间8个数字为k，首先枚举本次搜索的最大深度maxDepth（即需要的步数），从初始状态进行状态扩展，每次扩展8个结点，当搜索到深度为depth的时候，那么剩下可以移动的步数为maxDepth - depth，我们发现每次移动，中间的8个格子最多多一个k，所以如果当前状态下中间8个格子有sum个k，那么需要的剩余步数的理想最小值s = 8 - sum，那么估价函数：

h = depth + (8 - sum)

当h > maxDepth时，表明在当前这种状态下，不可能在maxDepth步以内达成目标，直接回溯。

当某个深度maxDepth至少有一个可行解时，整个算法也就结束了，可以设定一个标记，直接回溯到最上层，或者在DFS的返回值给定，对于某个搜索树，只要该子树下有解就返回1，否则返回0。

迭代加深适合深度不是很深，但是每次扩展的结点数很多的搜索问题。

**一、广度优先搜索**

**1、BFS**

1) 算法原理

广度优先搜索即Breadth First Search，也是图遍历算法的一种。用一句话概括就是：“我会分身我怕谁？！”。

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。

如图9所示，对以下图进行广度优先搜索，假设起点为1，将它放入队列后。那么第一次从队列中弹出的一定是1，将和1相邻未被访问的结点继续按顺序放入队列中，分别是2、3、4、5、7，并且记录下它们距离起点的距离dis[x] = dis[1] + 1 (x 属于集合 {2, 3, 4, 5, 7})；然后弹出的元素是2，和2相邻未被访问的结点是10，将它也放入队列中，记录dis[10] = dis[2] + 1；然后弹出5，放入6（4由于已经被访问过，所以不需要再放入队列中）；弹出7，放入8、9。队列为空后结束搜索，搜索完毕后，dis数组就记录了起点1到各个点的最短距离；

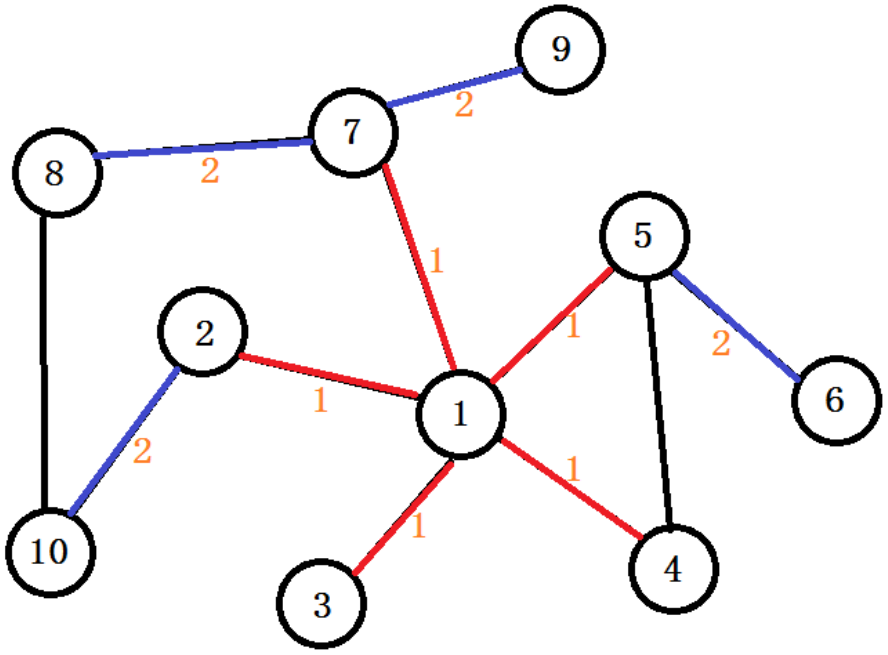



图9

2) 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)
```

3) 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。

绝大部分四向、八向迷宫的最短路问题。

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；

4) 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，试问没有稳定的婚姻，如何有心思学习算法，所以一定要学好BFS啊；

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。

详细算法参见：http://www.cppblog.com/menjitianya/archive/2014/07/10/207604.html

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，HDU1813四HDU3278以及HDU3900都是很好的例子；

f. 其它：

还有好多，一时间想不起来了，占坑；

**2、基于BFS的A***

1) 算法原理

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。

2) 基础应用

a.八数码问题

如图10所示，一个3*3的棋盘，放置8个棋子，编号1-8，给定任意一个初始状态，每次可以交换相邻两个棋子的位置，问最少经过多少次交换使棋盘有序。

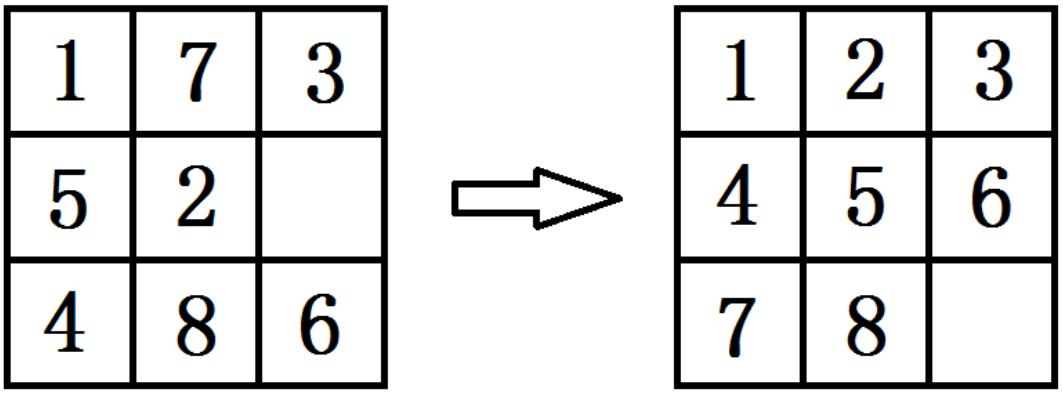



图10

遇到搜索问题一般都是先分析状态，这题的状态数可以这么考虑：将数字1放在九个格子中的任意一个，那么数字2有八种摆放方式，3有七种，依此类推；所以状态总数为9的排列数，即9!（9的阶乘） = 362880。每个状态可以映射到0到362880-1的一个整数，

对于广搜来说这个状态量不算大，但是也不小，如果遇到无解的情况，就会把所有状态搜遍，所以这里必须先将无解的情况进行特判，采用的是曼哈顿距离和逆序数进行剪枝，具体参见 SGU 139的解法：

http://www.cppblog.com/menjitianya/archive/2014/06/23/207386.html

网上对A*的描述写的都很复杂，我尝试用我的理解简单描述一下，首先还是从公式入手：

f(state) = g(state) + h(state)

g(state)　表示从**初始状态** 到 **state** 的实际行走步数，这个是通过BFS进行实时记录的，是一个已知量；

h(state)　表示从 **state** 到 **目标状态** 的期望步数，这个是一个估计值，不能准确得到，只能通过一些方法估计出一个值，并不准确；

f(state)　表示从 **初始状态** 到 **目标状态** 的期望步数，这个没什么好说的，就是前两个数相加得到，也肯定是个估计值；

对于广搜的状态，我们是用队列来维护的，所以state都是存在于队列中的，我们希望队列中状态的f(state)值是单调不降的（这样才能尽量早得搜到一个解），g(state)可以在状态扩展的时候由当前状态的父状态pstate的g(pstate)+1得到；那么问题就在于h(state)，用什么来作为state的期望步数，这个对于每个问题都是不一样的，在八数码问题中，我们可以这样想：

这个棋盘上每个**有数字的格子**都住了一位**老爷爷**（-_-|||），每位**老爷爷**都想回家，**老爷爷**的家就对应了目标状态每个数字所在的位置，对于 i 号老爷爷，他要回家的话至少要走的路程为当前状态state它在的格子pos[i] 和 目标状态他的家target[i] 的曼哈顿距离。每位老爷爷都要回家，所以最少的回家距离就是所有的这些曼哈顿距离之和，这就是我们在state状态要到达目标状态的期望步数h(state)，不理解请回到两行前再读一遍或者看下面的公式。

h(state) = sum( abs(pos[i].x - (i-1)/3) + abs(pos[i].y - (i-1)%3) )（其中 1 <= i <= 8,  0 <= pos[i].x, pos[i].y < 3 ）

最后附上原题链接：http://acm.hdu.edu.cn/showproblem.php?pid=1043

b.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。

还是一个h(state)函数，这里可以采用state到目标结点的最短距离为期望距离；

附上原题链接：http://poj.org/problem?id=2449

**3、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；

如图11，S表示初始状态，T表示目标状态，红色路径连接的点为S扩展出来的，蓝色路径连接的点为T扩展出来的，当S扩展到第三层的时候发现有一个结点已经在T扩展出来的集合中，于是搜索结束，最短距离等于3 + 2 = 5；

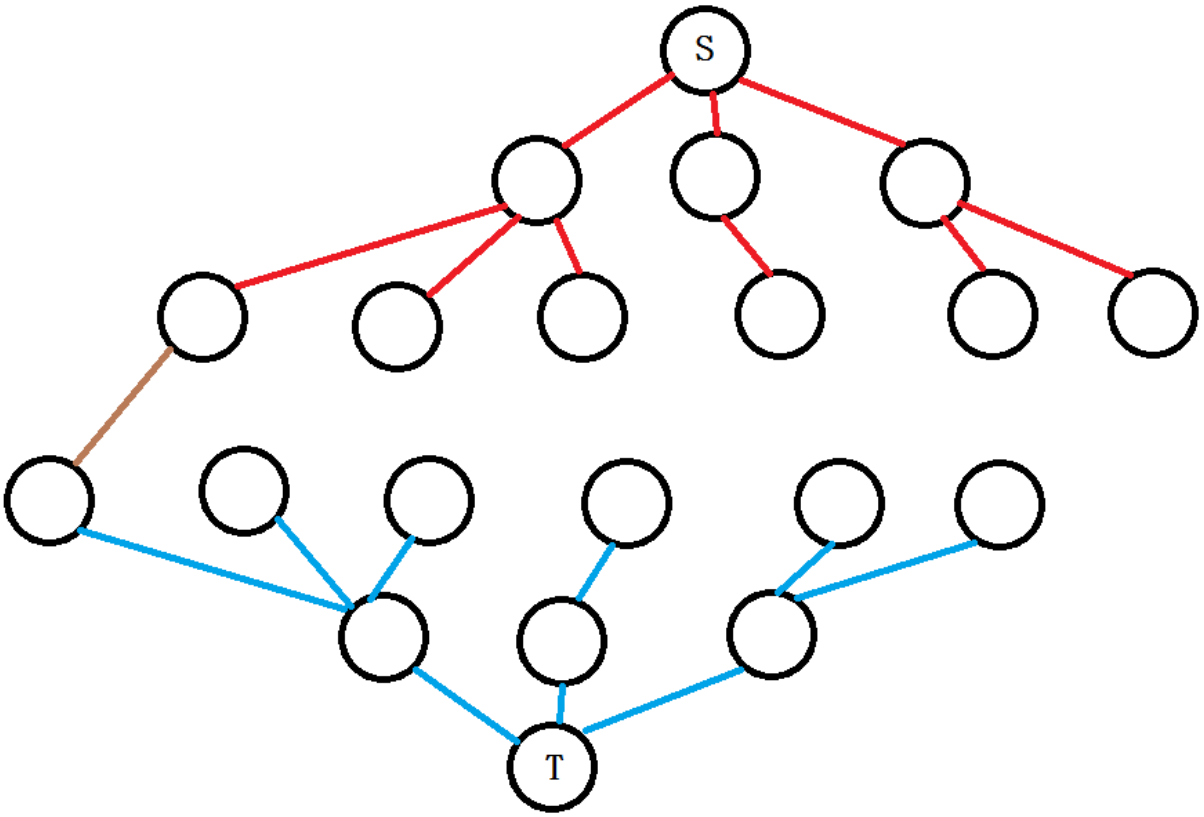



图11

双广的思想很简单，自己写上一两个基本上就能总结出固定套路了，和BFS一样属于盲搜。

## 三、搜索题集整理

1、DFS

| | | |
|---|---|---|
| Red and Black | ★☆☆☆☆ | FloodFill |
| The Game | ★☆☆☆☆ | FloodFill |
| Frogger | ★☆☆☆☆ | 二分枚举答案 + FloodFill |
| Nearest Common Ancestors | ★☆☆☆☆ | 最近公共祖先 |
| Robot Motion | ★☆☆☆☆ | 递归模拟 |
| Dessert | ★☆☆☆☆ | 枚举 |
| Matrix | ★☆☆☆☆ | 枚举 |
| Frame Stacking | ★☆☆☆☆ | 枚举 |
| Transportation | ★☆☆☆☆ | 枚举 |
| Pairs of Integers | ★★☆☆☆ | 枚举 |
| 方程的解数 | ★★☆☆☆ | 枚举 + 散列HASH |
| Maze | ★★☆☆☆ | 建完图后FloodFill |
| Trees Made to Order | ★★☆☆☆ | 递归构造解 |
| Cycles of Lanes | ★★☆☆☆ | 简单图最长环 |
| The Settlers of Catan | ★★☆☆☆ | 简单图最长链 |
| Parity game | ★★☆☆☆ | 简单图判奇环(交错染色) |
| Increasing Sequences | ★★☆☆☆ | 枚举 |
| Necklace Decomposition | ★★☆☆☆ | 枚举 |
| Rikka with Tree | ★★☆☆☆ | 统计 |
| Mahjong tree | ★★★☆☆ | 统计 |
| Machine Schedule | ★★★☆☆ | 二分图最大匹配 |
| Chessboard | ★★★☆☆ | 棋盘覆盖问题 |
| Air Raid | ★★★☆☆ | DAG图 最小路径覆盖 |
| Entropy | ★★★☆☆ | 枚举 + 适当剪枝 |
| Dropping the stones | ★★★☆☆ | 枚举 + 适当剪枝 |
| Dreisam Equations | ★★★☆☆ | 表达式求值 |
| Firefighters | ★★★☆☆ | 表达式求值 |
| Cartesian Tree | ★★★☆☆ | 笛卡尔树的构造 |
| Binary Stirling Numbers | ★★★☆☆ | 分形 |
| Obfuscation | ★★★☆☆ | 字符串匹配 |
| Graph Coloring | ★★★☆☆ | 最大团 |
| Pusher | ★★★☆☆ | 暴力搜索 |
| Self-Replicating Numbers | ★★★★☆ | 枚举 |
| Last Digits | ★★★★☆ | DFS + 欧拉函数 |
| Secret Code | ★★★★☆ | 实复数进制转化 |
| Anniversary Cake | ★★★★☆ | 矩形填充 |
| A Puzzling Problem | ★★★★☆ | 枚举摆放 |

| | | |
|---|---|---|
| Vase collection | ★★★★☆ | 图的完美匹配 |
| Packing Rectangles | ★★★★☆ | 枚举摆放 |
| Missing Piece 2001 | ★★★★☆ | N*N-1 数码问题，强剪枝 |

2、IDA* (确定是迭代加深后就一个套路，枚举深度，然后 暴力搜索+强剪枝)

| | | |
|---|---|---|
| Addition Chains | ★★☆☆☆ | |
| DNA sequence | ★★☆☆☆ | |
| Booksort | ★★★☆☆ | |
| The Rotation Game | ★★★☆☆ | 迭代加深的公认经典题，理解“最少剩余步数” |
| Paint on a Wall | ★★★☆☆ | The Rotation Game 的简单变形 |
| Escape from Tetris | ★★★★☆ | |
| Maze | ★★★★☆ | |
| Rubik 2×2×2 | ★★★★★ | 编码麻烦的魔方题 |

3、BFS

| | | |
|---|---|---|
| Pushing Boxes | ★☆☆☆☆ | 经典广搜 - 推箱子 |
| Jugs | ★☆☆☆☆ | 经典广搜 - 倒水问题 |
| Space Station Shielding | ★☆☆☆☆ | FloodFill |
| Knight Moves | ★☆☆☆☆ | 棋盘搜索 |
| Knight Moves | ★☆☆☆☆ | 棋盘搜索 |
| Eight | ★★☆☆☆ | 经典八数码 |
| Currency Exchange | ★★☆☆☆ | SPFA |
| The Postal Worker Rings | ★★☆☆☆ | SPFA |
| ROADS | ★★☆☆☆ | 优先队列应用 |
| Ones | ★★☆☆☆ | 同余搜索 |
| Dogs | ★★☆☆☆ | |
| Lode Runner | ★★☆☆☆ | |
| Hike on a Graph | ★★☆☆☆ | |
| Find The Multiple | ★★☆☆☆ | 同余搜索 |
| Different Digits | ★★★☆☆ | 同余搜索 |
| Magic Multiplying Machine | ★★★☆☆ | 同余搜索 |
| Remainder | ★★★☆☆ | 同余搜索 |
| Escape from Enemy Territory | ★★★☆☆ | 二分答案 + BFS |
| Will Indiana Jones Get | ★★★☆☆ | 二分答案 + BFS |
| Fast Food | ★★★☆☆ | SPFA |
| Invitation Cards | ★★★☆☆ | SPFA |
| Galactic Import | ★★★☆☆ | SPFA |
| XYZZY | ★★★☆☆ | 最长路判环 |
| Intervals | ★★★☆☆ | 差分约束 |
| King | ★★★☆☆ | 差分约束 |
| Integer Intervals | ★★★☆☆ | 差分约束 |
| Sightseeing trip | ★★★☆☆ | 无向图最小环 |
| N-Credible Mazes | ★★★☆☆ | 多维空间搜索，散列HASH |
| Spreadsheet | ★★★☆☆ | 建立拓扑图后广搜 |
| Frogger | ★★★☆☆ | 同余搜索 |
| Ministry | ★★★☆☆ | 需要存路径 |
| Gap | ★★★☆☆ | A* |
| Maze | ★★★☆☆ | 二进制压缩钥匙的状态 |
| Hike on a Graph | ★★★☆☆ | |
| All Discs Considered | ★★★☆☆ | |
| Roads Scholar | ★★★☆☆ | SPFA |
| Holedox Moving | ★★★☆☆ | |
| 昂贵的聘礼 | ★★★☆☆ | |
| Maze Stretching | ★★★☆☆ | |
| Treasure of the Chimp | ★★★☆☆ | |
| Is the Information Reliable | ★★★☆☆ | 最长路判环 |
| It's not a Bug, It's a | ★★★☆☆ | |
| Warcraft | ★★★☆☆ | |
| Escape | ★★★☆☆ | |
| Bloxorz I | ★★★☆☆ | 当年比较流行这个游戏 |
| Up and Down | ★★★★☆ | 离散化 + BFS |
| Sightseeing | ★★★★☆ | SPFA |
| Cliff Climbing | ★★★★☆ | 日本人的题就是这么长 |
| Cheesy Chess | ★★★★☆ | 仔细看题 |
| The Treasure | ★★★★☆ | |
| Chessman | ★★★★★ | 弄清状态同余的概念 |
| Puzzle | ★★★★★ | 几乎尝试了所有的搜索 -_-\|\|\| 让人欲仙欲死的题 |
| Unblock Me | ★★★★★ | 8进制压缩状态，散列HASH，位运算加速 |

4、双向BFS(适用于起始状态都给定的问题，一般一眼就能看出来，固定套路，很难有好的剪枝)

| | |
|---|---|
| Solitaire | ★★★☆☆ |
| A Game on the Chessboard | ★★★☆☆ |
| 魔板 | ★★★★☆ |
| Tobo or not Tobo | ★★★★☆ |
| Eight II | ★★★★★ |

# 目录



## 一、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互素(互素的概念下文会讲到)，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)。

拿一个我还未出生时的初二数学竞赛题就能概括整除的性质了。

【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互素，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（C++中的'%'运算符），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，用极限表示如图一-2-1所示：

$$\lim_{x\to\infty}\frac{\pi(x)}{x/\ln(x)}=1.$$

图一-2-1

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
（第三章中会介绍该算法的具体实现）大数判素了。

**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，如图一-3-1所示：

$$n = p_1 p_2 p_3 \dots p_k$$
$$(p_1 \le p_2 \le p_3 \dots \le p_k)$$

图一-3-1

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，采用素数幂的乘积进行表示，如图一-3-2所示：

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \dots p_k^{e_k}$$
$$(p_1 < p_2 < p_3 \dots < p_k \text{且} e_i > 0)$$

图一-3-2

证明方法采用数学归纳法，此处略去。

**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。

枚举完[2, sqrt(n)]的素数后，得到表达式如图一-3-3所示：

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \dots p_k^{e_k} S$$

图一-3-3

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，那么X^Y可以表示成图一-3-4所示的形式：

$$X^Y = (p_1^{e_1} p_2^{e_2} p_3^{e_3} \dots p_k^{e_k})^Y$$

$$= p_1^{Ye_1} p_2^{Ye_2} p_3^{Ye_3} \dots p_k^{Ye_k}$$

图一-3-4

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互素（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互素时，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。这个等式可以通过算术基本定理进行证明，证明过程可以通过图一-4-1秒懂。

$$a = p_1^{x_1} p_2^{x_2} p_3^{x_3} \dots p_k^{x_k}$$

$$b = p_1^{y_1} p_2^{y_2} p_3^{y_3} \dots p_k^{y_k}$$

$$\gcd(a,b) = p_1^{\min\{x_1,y_1\}} p_2^{\min\{x_2,y_2\}} p_3^{\min\{x_3,y_3\}} \dots p_k^{\min\{x_k,y_k\}}$$

$$\mathrm{lcm}(a,b) = p_1^{\max\{x_1,y_1\}} p_2^{\max\{x_2,y_2\}} p_3^{\max\{x_3,y_3\}} \dots p_k^{\max\{x_k,y_k\}}$$

图一-4-1

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：

| b%7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| b | 4312 | 2143 | 1234 | 4231 | 1243 | 2413 | 4213 |

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。

## 二、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。代码如下：

```
#define LL __int64
LL Extend_Euclid(LL a, LL b, LL &X, LL &Y) {
    LL q, temp;
    if( !b ) {
        X = 1; Y = 0;
        return a;
    }else {
        q = Extend_Euclid(b, a % b, X, Y);
        temp = X;
        X = Y;
        Y = temp - (a / b) * Y;
        return q;
    }
}
```

扩展欧几里德算法和欧几里德算法的返回值一致，都是gcd(a, b)，传参多了两个未知数X, Y，采用引用的形式进行传递，对应上文提到的x, y，递归出口为b == 0，这时返回值为当前的a，因为gcd(a, 0) = a，(X, Y)初值为(1, 0)，然后经过回溯不断计算新的(X, Y)，这个计算是利用了之前的(X, Y)进行迭代计算的，直到回溯到最上层算法终止。最后得到的(X, Y)就是方程gcd(a, b) = ax + by的解。

通过扩展欧几里德求的是ax + by = gcd(a, b)的解，令解为(x0, y0)，代入原方程，得：ax0 + by0 = gcd(a, b)，如果要求ax + by = c = gcd(a, b)*c'，可以将上式代入，得：ax + by = c = (ax0 + by0)c'，则x = x0c', y = y0c'，这里的(x, y)只是这个方程的其中一组解，x的通解为 { x0c' + kb/gcd(a, b) | k为任意整数 }，y的通解可以通过x通解的代入得出。

**【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：

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

图二-3-1

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**、互素

两个数a和b互素的定义为：gcd(a, b) = 1，那么如何求不大于n且与n互素的数的个数呢？
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互素的数的个数，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的都互素）；
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的倍数其它都互素）；
如果m和n互素，那么φ(mn) = φ(m)φ(n)（可以利用上面两个性质进行推导）。
将n分解成如图二-4-1的素因子形式，那么利用上面的定理可得φ(n)如图二-4-2所示：

$$n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \dots p_k^{e_k}$$
$$(p_1 < p_2 < p_3 \dots < p_k 且 e_i > 0)$$

图二-4-1

$$\varphi(n) = p_1^{e_1-1}(p_1 - 1) * p_2^{e_2-1}(p_2 - 1) * \dots * p_k^{e_k-1}(p_k - 1)$$

图二-4-2

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。伪代码如下：

```
#define MAXP 2000010
#define LL __int64
void Eratosthenes_Phi() {
    notprime[1] = true;
    for(int i = 1; i < MAXP; i++) phi[i] = 1;
    for(int i = 2; i < MAXP; i++) {
        if( !notprime[i] ) {
            phi[i] *= i - 1;
            // 和传统素数筛法的区别在于这个i+i
            for(int j = i+i; j < MAXP; j += i) {
                notprime[j] = true;
                int n = j / i;
                phi[j] *= (i - 1);
                while(n % i == 0) n /= i, phi[j] *= i;
            }
        }
    }
}
```

这里的phi[i]保存了i这个数的欧拉函数，还是利用素数筛选将所有素数筛选出来，然后针对每个素因子计算它的倍数含有该素因子的个数，利用欧拉公式计算该素因子带来的欧拉函数分量，整个筛选过程可以参考素数筛选。

**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互素，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互素，则：$a^{p-1} \equiv 1 (mod\, p)$。

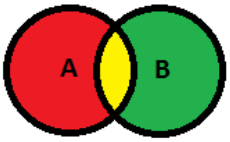
由于当n为素数时φ(n) = p-1，可见费马小定理是欧拉定理的特殊形式。
证明随处可见，这里讲一下应用。

【例题**10**】整数**a**和**n**互素，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。

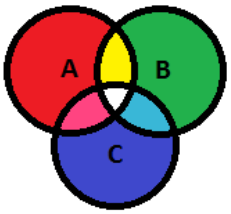
问题要求的是a^(X^Y) % n，指数上还是存在指数，需要将指数化简，注意到a和n互素，所以可以利用欧拉定理，令X^Y = kφ(n) + r，那么kφ(n)部分并不需要考虑，问题转化成求r = X^Y % φ(n)，可以采用快速幂取模，二分求解，得到r后再采用快速幂取模求解a^r % n。

**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。




图二-5-1




图二-5-2

这里的“加”就是“容”，“减”就是“斥”，并且“容”和“斥”总是交替进行的（一个的加上，两个的减去，三个的加上，四个的减去），而且可以推广到n个元素的情况。

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

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互素的数了）；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）。这里给出一版dfs版本的容斥原理的伪代码，用于求解小于等于m且与n互素的数的个数。

```
#define LL __int64
void IncludeExclude(int depth, LL m, LL mul, int op, int* p, LL &ans) {
    if(m < mul) return ;
    if(depth == p[0]) {
        ans += (op ? -1 : 1) * (m / mul);
        return ;
    }
    for(int i = 0; i < 2; i++) {
        // 0 表示不取, 1表示取
        IncludeExclude( depth+1, m, mul * (i?p[depth+1]:1), op^i, p, ans );
    }
}
```

p[ 1 : p[0] ]存储的是n的所有素因子，p[0]表示数组长度，mul表示该次的素因子子集的乘积，op表示子集的奇偶性，ans存储最后的答案。

例如求[1, 9]中和6互素的数的个数，这时p = [2, 2, 3] （注意p[0]是存素数的个数的，6分解的素因子为2和3）。

ans = 9/1 - （9/2 + 9/3） + 9/6 = 3，ans分为三部分，0个数的组合，1个数的组合，2个数的组合。

## 三、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互素的正整数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互素的正整数，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 );
    }
```

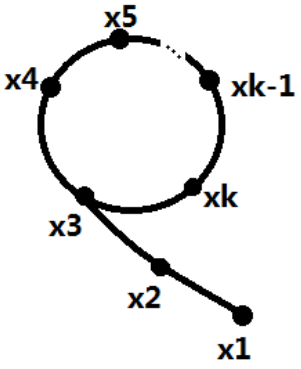
这里Power_Mod(a, b, n)即a^b%n，Product_Mod(a, b, n)即a*b%n，而k次测试的基于费马小定理的一个推论：x^2 % n = 1当n为素数时x的解只有两个，即1和n-1。

**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，就像图三-2-1那样。




图三-2-1

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)互素，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。

## 四、数论题集整理

### 1、素数和因数分解

| | | |
|---|---|---|
| Largest prime factor | ★☆☆☆☆ | 素数筛选 |
| The number of divisors | ★☆☆☆☆ | 因子数 |
| 七夕节 | ★☆☆☆☆ | 因子和 |
| Happy 2004 | ★☆☆☆☆ | X^Y的因子和 |
| Number Sequence | ★☆☆☆☆ | 循环节的经典问题 |
| Beijing 2008 | ★★☆☆☆ | X^Y的因子和 |
| f(n) | ★★☆☆☆ | 找规律+素数筛选 |
| 本原串 | ★★☆☆☆ | 整除性质 + 因子枚举 |
| Special Prime | ★★☆☆☆ | 3n^2+3n+1 的素数判定问题 |
| Factorial Simplificat | ★★★☆☆ | 因式分解+树状数组+DFS |
| Gcd & Lcm game | ★★★★☆ | 因式分解+线段树 |

### 2、GCD && LCM

| | | |
|---|---|---|
| hide handkerchief | ★☆☆☆☆ | 互素判定 |
| GCD and LCM | ★★☆☆☆ | GCD和LCM性质 + 排列组合 |
| Revenge of GCD | ★★☆☆☆ | 辗转相除+因子枚举 |
| Least common multiple | ★★★☆☆ | GCD性质 + 完全背包 |

### 3、同余性质 和 循环节

| | | |
|---|---|---|
| N!Again | ★☆☆☆☆ | 同余的乘法性质 |
| Ice Rain | ★★☆☆☆ | 余数性质 |
| Love you TenThous years | ★★☆☆☆ | 规律 |
| TCE-frep number system | ★★☆☆☆ | 完全数 |
| Perfect Squares | ★★☆☆☆ | 同余性 + 循环节 |
| X mod f(x) | ★★★☆☆ | 利用同余原理进行动态规划 |
| Interesting Fibonacci | ★★★★☆ | 复杂的循环节 |

### 4、模线性方程和逆元

| | | |
|---|---|---|
| 青蛙的约会 | ★☆☆☆☆ | 线性同余 |
| Romantic | ★☆☆☆☆ | 线性同余 |
| Robot | ★★☆☆☆ | 逆元 |
| An easy problem | ★★☆☆☆ | 逆元 |
| A/B | ★★☆☆☆ | 逆元入门题 |
| A New Change Problem | ★★☆☆☆ | 同余推导 |
| Central Meridian Number | ★★☆☆☆ | 线性同余+枚举 |
| number theory | ★★☆☆☆ | 快速幂取模 + 欧几里德定理 |
| Multiply game | ★★★★☆ | 树状数组 + 逆元 |

### 5、模线性方程组

| | | |
|---|---|---|
| Chinese remainder theorem again | ★★☆☆☆ | 中国剩余定理 简化版 |
| Strange Way to Express Integers | ★★★☆☆ | 中国剩余定理 模板题 |
| Hello Kiki | ★★★☆☆ | 中国剩余定理 模板题 |
| X问题 | ★★★☆☆ | 中国剩余定理 模板题 |
| And Now, a Remainder from | ★★★☆☆ | 中国剩余定理 模板题 |

### 6、欧拉函数、欧拉定理、费马小定理

| | | |
|---|---|---|
| 2^x mod n = 1 | ★☆☆☆☆ | 费马小定理 简化版的 |
| HeHe | ★☆☆☆☆ | 欧拉函数 |
| GCD | ★★☆☆☆ | 欧拉函数 |
| Become A Hero | ★★☆☆☆ | 筛选法求欧拉函数 |
| The Euler function | ★★☆☆☆ | 筛选法求欧拉函数 |
| The Luckiest number | ★★★☆☆ | 费马小定理 |
| Calculation | ★★★☆☆ | 费马小定理 |
| Description has only two Sentences | ★★★☆☆ | 费马小定理，我的题 |

### 6、容斥原理

| | | |
|---|---|---|
| How many integers | ★★☆☆☆ | 容斥原理 |
| Visible Trees | ★★☆☆☆ | 容斥原理 |
| GCD Again | ★★☆☆☆ | 容斥原理 |
| GCD | ★★☆☆☆ | 容斥原理 |
| GCD | ★★☆☆☆ | 容斥原理 |
| Coprime | ★★★☆☆ | 二分枚举+容斥原理 |
| Sky Code | ★★★☆☆ | 容斥原理 |

### 7、大素数判定

| | | |
|---|---|---|
| GCD & LCM Inverse | ★★★☆☆ | 拉宾米勒大数判素+dfs |
| Pseudoprime numbers | ★★★☆☆ | 拉宾米勒 |
| Problem about GCD | ★★★★☆ | 拉宾米勒 |
| Prime Test | ★★★★☆ | 拉宾米勒+Pollard-rho |
| RSA | ★★★★☆ | 拉宾米勒 + 线性同余 |

8、离散对数-Baby Step Gaint Step算法

| | | |
|---|---|---|
| Discrete Logging | ★★★☆☆ | 基础 |
| Mod Tree | ★★★★☆ | 扩展Baby Step Giant Step |
| Matrix Puzzle | ★★★★★ | Baby Step Giant Step + 高斯消元 |

9、其它

Counting Problem
The Two Note Rag
Disgruntled Judge
Can you find it
YAPTCHA
Jacobi symbol
GCD of Sequence
Sum Of Gcd
GCD Array
GCD pair
GCD
Gcd and Lcm
The sum of gcd
GCD?LCM!
GCD Tree

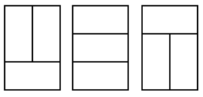
## 五、动态规划题集整理

### 一、动态规划初探

#### 1、递推

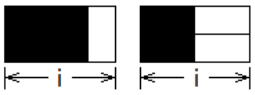
暂且先不说动态规划是怎么样一个算法，由最简单的递推问题说起应该是最恰当不过得了。因为一来，递推的思想非常浅显，从初中开始就已经有涉及，等差数列 f[i] = f[i-1] + d（ i > 0， d为公差，f[0]为初项）就是最简单的递推公式之一；二来，递推作为动态规划的基本方法，对理解动态规划起着至关重要的作用。理论的开始总是枯燥的，所以让读者提前进入思考是最能引起读者兴趣的利器，于是【例题1】应运而生。

**【例题1】在一个3 X N的长方形方格中，铺满1X2的骨牌（骨牌个数不限制），给定N，求方案数（图一 -1-1为N=2的所有方案），所以N=2时方案数为3。**



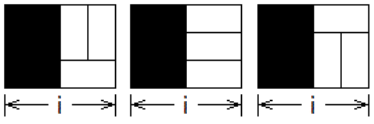
图一 -1-1

这是一个经典的递推问题，如果觉得无从下手，我们可以来看一个更加简单的问题，把问题中的“3”变成“2”（即在一个2XN的长方形方格中铺满1X2的骨牌的方案）。这样问题就简单很多了，我们用f[i]表示2Xi的方格铺满骨牌的方案数，那么考虑第i列，要么竖着放置一个骨牌；要么连同i-1列，横着放置两个骨牌，如图2所示。由于骨牌的长度为1X2，所以在第i列放置的骨牌无法影响到第i-2列。很显然，图一 -1-2中两块黑色的部分分别表示f[i-1]和f[i-2]，所以可以得到递推式f[i] = f[i-1] + f[i-2] (i >= 2)，并且边界条件f[0] = f[1] = 1。



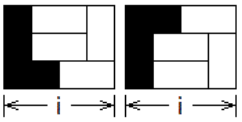

图一 -1-2

再回头来看3 X N的情况，首先可以明确当N等于奇数的时候，方案数一定为0。所以如果用f[i] (i 为偶数) 表示3Xi的方格铺满骨牌的方案数，f[i]的方案数不可能由f[i-1]递推而来。那么我们猜想f[i]和f[i-2]一定是有关系的，如图一 -1-3所示，我们把第i列和第i-1列用1X2的骨牌填满后，轻易转化成了f[i-2]的问题，那是不是代表f[i] = 3*f[i-2]呢？



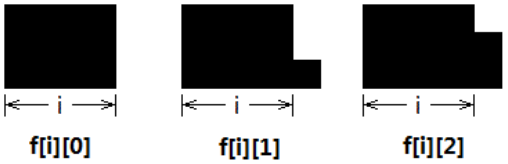

图一 -1-3

仔细想想才发现不对，原因是我们少考虑了图一 -1-4的情况，这些情况用图一 -1-3的情况无法表示，再填充完黑色区域后，发现和f[i-4]也有关系，但是还是漏掉了一些情况。




图一 -1-4

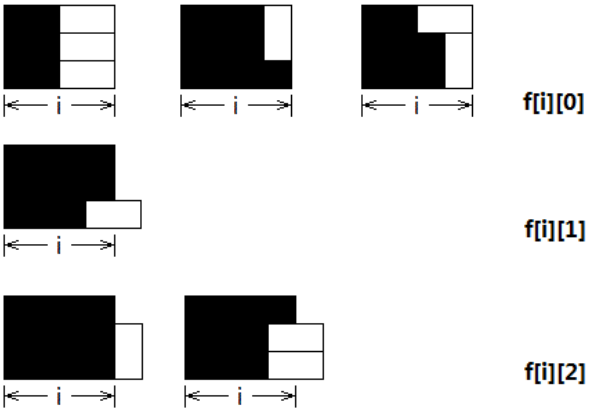
上面的问题说明我们在设计状态（状态在动态规划中是个很重要的概念，在本章的第4小节会进行介绍总结）的时候的思维定式，当一维的状态已经无法满足我们的需求时，我们可以试着增加一维，用二维来表示状态，用f[i][j]表示(3 X i) + j个多余块的摆放方案数，如图一 -1-5所示：




图一 -1-5

转化成二维后，我们可以轻易写出三种情况的递推式，具体推导方法见图一 -1-6。

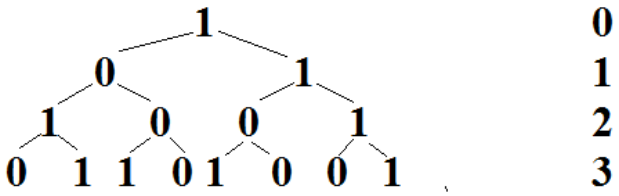
f[i][0] = f[i-2][0] + f[i-1][1] + f[i-2][2]
f[i][1] = f[i-1][2]
f[i][2] = f[i][0] + f[i-1][1]
边界条件　f[0][0] = f[1][1] = f[0][2] = 1




图一 -1-6

如果N不是很大的情况，到这一步，我们的问题已经完美解决了，其实并不需要求它的通项公式，因为我们是程序猿，一个for循环就能搞定了<*_*>，接下来的求解就全仰仗于计算机来完成了。

**【例题2】对一个“01”串进行一次μ变换被定义为：将其中的"0"变成"10"，"1"变成"01"，初始串为"1"，求经过N(N <= 1000)次μ变换后的串中有多少对"00"（有没有人会纠结会不会出现"000"的情况？这个请放心，由于问题的特殊性，不会出现"000"的情况）。图一 -1-7表示经过小于4次变换时串的情况。**

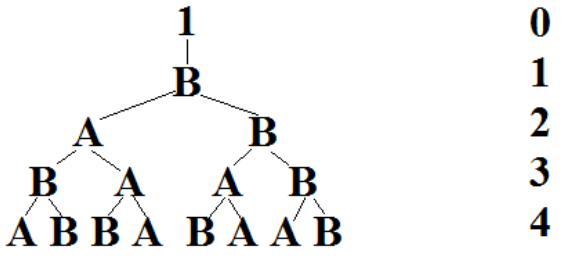



图一 -1-7

如果纯模拟的话，每次μ变换串的长度都会加倍，所以时间和空间复杂度都是O(2^n)，对于n为1000的情况，完全不可能计算出来。仔细观察这个树形结构，可以发现要出现"00"，一定是"10"和"01"相邻产生的。为了将问题简化，我们不妨设A = "10", B = "01"，构造出的树形递推图如图一 -1-8所示，如果要出现"00"，一定是AB（"1001"）。

令FA[i]为A经过i次μ变换后"00"的数量，FA[0] = 0;FB[i]为B经过i次μ变换后"00"的数量，FB[0] = 0。

从图中观察得出，以A为根的树，它的左子树的最右端点一定是B，也就是说无论经过多少次变换，两棵子树的交界处都不可能产生AB，所以FA[i] = FB[i-1] + FA[i-1]（直接累加两棵子树的"00"的数量）；而以B为根的树，它的左子树的右端点一定是A，而右子树的左端点呈BABABA...交替排布，所以隔代产生一次AB，于是FB[i] = FA[i-1] + FB[i-1] + (i mod 2)。最后要求的答案就是FB[N-1]，递推求解。




图一 -1-8

## 2、记忆化搜索

递推说白了就是在知道前i-1项的值的前提下，计算第i项的值，而记忆化搜索则是另外一种思路。它是直接计算第i项，需要用到第 j 项的值( j < i)时去查表，如果表里已经有第 j 项的话，则直接取出来用，否则递归计算第 j 项，并且在计算完毕后把值记录在表中。记忆化搜索在求解多维的情况下比递推更加方便，【例题3】是我遇到的第一个记忆化搜索的问题，记忆犹新。

**【例题3】这个问题直接给出了一段求函数w(a, b, c)的伪代码：**

```
function w(a, b, c):
    if a <=0 or b <=0 or c <=0, then returns:1
    if a >20or b >20or c >20, then returns: w(20,20,20)
    if a < b and b < c, then returns: w(a, b, c-1)+ w(a, b-1, c-1)- w(a, b-1, c)
    otherwise it returns: w(a-1, b, c)+ w(a-1, b-1, c)+ w(a-1, b, c-1)
```

**要求给定a, b, c，求w(a, b, c)的值。**

乍看下只要将伪代码翻译成实际代码，然后直接对于给定的a, b, c，调用函数w(a, b, c)就能得到值了。但是只要稍加分析就能看出这个函数的时间复杂度是指数级的（尽管这个三元组的最大元素只有20，这是个陷阱）。对于任意一个三元组(a, b, c)，w(a, b, c)可能被计算多次，而对于固定的(a, b, c)，w(a, b, c)其实是个固定的值，没必要多次计算，所以只要将计算过的值保存在f[a][b][c]中，整个计算就只有一次了，总的时间复杂度就是O(n^3)，这个问题的n只有20。

## 3、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，例如【例题1】中的f[i][j]，【例题2】中的FA[i]、FB[i]，【例题3】中的f[a][b][c]，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）。

**4、最优化原理和最优子结构**

在介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。这里我尽力减少理论化的概念，而改用一个简单的例题来加深对这句话的理解。

**【例题4】给定一个长度为n(1 <= n <= 1000)的整数序列a[i]，求它的一个子序列(子序列即在原序列任意位置删除0或多个元素后的序列)，满足如下条件：**

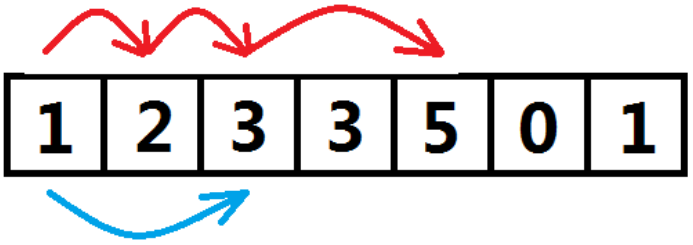
**1、该序列单调递增；**

**2、在所有满足条件1的序列中长度是最长的；**

**这个问题是经典的动态规划问题，被称为最长单调子序列。**

我们假设现在没有任何动态规划的基础，那么看到这个问题首先想到的是什么？

我想到的是万金油算法---枚举（DFS），即枚举a[i]这个元素取或不取，所有取的元素组成一个合法的子序列，枚举的时候需要满足单调递增这个限制，那么对于一个n个元素的序列，最坏时间复杂度自然就是$O(2^n)$，n等于30就已经很变态了更别说是1000。但是方向是对的，动态规划求解之前先试想一下搜索的正确性，这里搜索的正确性是很显然的，因为已经枚举了所有情况，总有一种情况是我们要求的解。我们尝试将搜索的算法进行一些改进，假设第i个数取的情况下已经搜索出的最大长度记录在数组d中，即用d[i]表示当前搜索到的以a[i]结尾的最长单调子序列的长度，那么如果下次搜索得到的序列长度小于等于d[i]，就不必往下搜索了（因为即便继续往后枚举，能够得到的解必定不会比之前更长）；反之，则需要更新d[i]的值。如图一-4-1，红色路径表示第一次搜索得到的一个最长子序列1、2、3、5，蓝色路径表示第二次搜索，当枚举第3个元素取的情况时，发现以第3个数结尾的最长长度d[3] = 3，比本次枚举的长度要大（本次枚举的长度为2），所以放弃往下枚举，大大减少了搜索的状态空间。




图一-4-1

这时候，我们其实已经不经意间设计好了状态，就是上文中提到的那个d[i]数组，它表示的是以a[i]结尾的最长单调子序列的长度，那么对于任意的i，d[i] 一定等于 d[j] + 1（j < i），而且还得满足 a[j] < a[i]。因为这里的d[i]表示的是最长长度，所以d[i]的表达式可以更加明确，即：

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

这个表达式很好的阐释了最优化原理，其中d[j]作为d[i]的子问题，d[i]最长（优）当且仅当d[j]最长（优）。当然，这个方程就是这个问题的状态转移方程。状态总数量O(n)，每次转移需要用到前i项的结果，平摊下来也是O(n)的，所以该问题的时间复杂度是O(n^2)，然而它并不是求解这类问题的最优解，下文会提到最长单调子序列的O(nlogn)的优化算法。

**5、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）。

**【例题5】老王想在未来的n年内每年都持有电脑，m(y, z)表示第y年到第z年的电脑维护费用，其中y的范围为[1, n]，z的范围为[y, n]，c表示买一台新的电脑的固定费用。 给定矩阵m，固定费用c，求在未来n年都有电脑的最少花费。**

考虑第 i 年是否要换电脑，换和不换是不一样的决策，那么我们定义一个二元组(a, b)，其中 a < b，它表示了第a年和第b年都要换电脑（第a年和第b年之间不再换电脑），如果假设我们到第a年为止换电脑的最优方案已经确定，那么第a年以前如何换电脑的一些列步骤变得不再重要，因为它并不会影响第b年的情况，这就是无后效性。

更加具体得，令d[i]表示在第i年买了一台电脑的最小花费(由于这台电脑能用多久不确定，所以第i年的维护费用暂时不计在这里面)，如果上一次更换电脑的时间在第j年，那么第j年更换电脑到第i年之前的总开销就是c + m(j, i-1)，于是有状态转移方程：

d[i] = min{ d[j] + m(j, i-1) | 1 <= j < i } + c

这里的d[i]并不是最后问题的解，因为它漏算了第i年到第n年的维护费用，所以最后问题的答案：

ans = min{ d[i] + m(i, n) | 1 <= i < n }

我们发现两个方程看起来很类似，其实是可以合并的，我们可以假设第n+1年必须换电脑，并且第n+1年换电脑的费用为0，那么整个阶段的状态转移方程就是：

d[i] = min{ d[j] + m(j, i-1) | 1 <= j < i } + w(i)　其中w(i) = (i==n+1)?0:c;

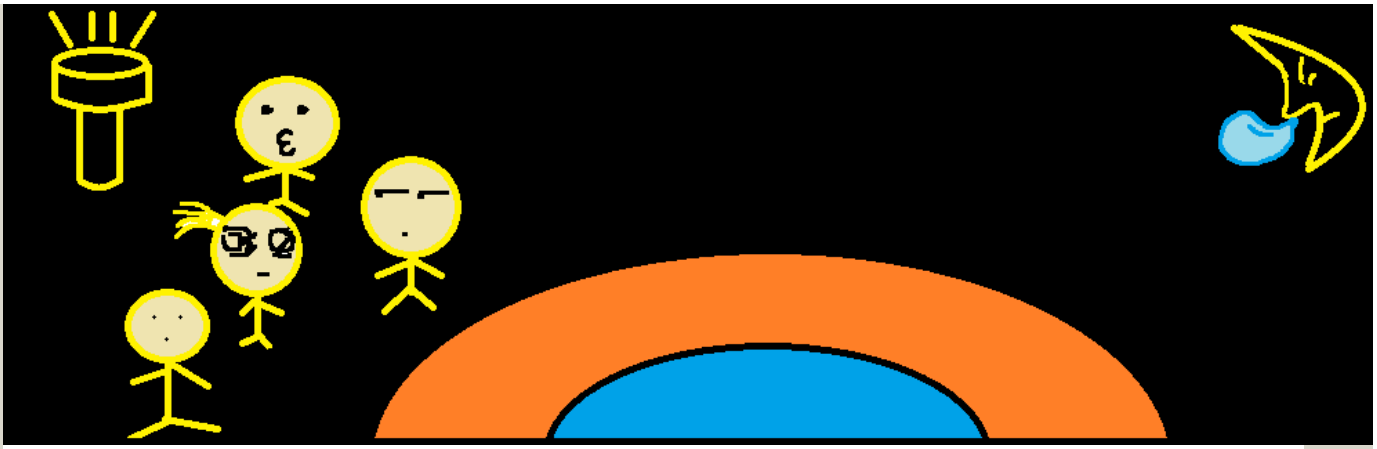
d[n+1]就是我们需要求的最小费用了。

## 二、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。【例题6】是一个经典的面试题，我们将它作为线性模型的敲门砖。

**【例题6】在一个夜黑风高的晚上，有n（n <= 50）个小朋友在桥的这边，现在他们需要过桥，但是由于桥很窄，每次只允许不大于两人通过，他们只有一个手电筒，所以每次过桥的两个人需要把手电筒带回来，i号小朋友过桥的时间为T[i]，两个人过桥的总时间为二者中时间长者。问所有小朋友过桥的总时间最短是多少。**



图二-1-1

每次过桥的时候最多两个人，如果桥这边还有人，那么还得回来一个人（送手电筒），也就是说N个人过桥的次数为2*N-3（倒推，当桥这边只剩两个人时只需要一次，三个人的情况为来回一次后加上两个人的情况...）。有一个人需要来回跑，将手电筒送回来（也许不是同一个人，realy？！）这个回来的时间是没办法省去的，并且回来的次数也是确定的，为N-2，如果是我，我会选择让跑的最快的人来干这件事情，但是我错了...如果总是跑得最快的人跑回来的话，那么他在每次别人过桥的时候一定得跟过去，于是就变成就是很简单的问题了，花费的总时间：

T = minPTime * (N-2) + (totalSum-minPTime)

来看一组数据 四个人过桥花费的时间分别为 1 2 5 10，按照上面的公式答案是19，但是实际答案应该是17。

具体步骤是这样的：

第一步：1和2过去，花费时间2，然后1回来（花费时间1）；

第二步：3和4过去，花费时间10，然后2回来（花费时间2）；

第三部：1和2过去，花费时间2，总耗时17。

所以之前的贪心想法是不对的。

我们先将所有人按花费时间递增进行排序，假设前i个人过河花费的最少时间为opt[i]，那么考虑前i-1个人过河的情况，即河这边还有1个人，河那边有i-1个人，并且这时候手电筒肯定在对岸，所以

opt[i] = opt[i-1] + a[1] + a[i]　　(让花费时间最少的人把手电筒送过来，然后和第i个人一起过河)

如果河这边还有两个人，一个是第i号，另外一个无所谓，河那边有i-2个人，并且手电筒肯定在对岸，所以

opt[i] = opt[i-2] + a[1] + a[i] + 2*a[2]　(让花费时间最少的人把电筒送过来，然后第i个人和另外一个人一起过河，由于花费时间最少的人在这边，所以下一次送手电筒过来的一定是花费次少的，送过来后花费最少的和花费次少的一起过河，解决问题)

所以 opt[i] = min{opt[i-1] + a[1] + a[i] , opt[i-2] + a[1] + a[i] + 2*a[2] }

### 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。

**【例题7】给定一个长度为n（n <= 1000）的字符串A，求插入最少多少个字符使得它变成一个回文串。**

典型的区间模型，回文串拥有很明显的子结构特征，即当字符串X是一个回文串时，在X两边各添加一个字符'a'后，aXa仍然是一个回文串，我们用d[i][j]来表示A[i...j]这个子串变成回文串所需要添加的最少的字符数，那么对于A[i] == A[j]的情况，很明显有 d[i][j] = d[i+1][j-1]（这里需要明确一点，当i+1 > j-1时也是有意义的，它代表的是空串，空串也是一个回文串，所以这种情况下d[i+1][j-1] = 0）；当A[i] != A[j]时，我们将它变成更小的子问题求解，我们有两种决策：

1、在A[j]后面添加一个字符A[i]；

2、在A[i]前面添加一个字符A[j]；

根据两种决策列出状态转移方程为：

d[i][j] = min{ d[i+1][j], d[i][j-1] } + 1;　　(每次状态转移，区间长度增加1)

空间复杂度O(n^2)，时间复杂度O(n^2)， 下文会提到将空间复杂度降为O(n)的优化算法。

### 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。由于网上有非常详尽的背包讲解，这里只将常用部分抽出来，具体推导过程详见《背包九讲》。

a.0/1背包

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）。

b.完全背包

有N种物品（每种物品无限件）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。

f[i][v] = max{ f[i-1][v - kCi] + kWi | 0 <= k <= v/Ci }　　(当k的取值为0,1时，这就是01背包的状态转移方程)

时间复杂度O( VNsum{V/Ci} )，空间复杂度在用滚动数组优化后可以达到O( V )。

进行优化后（此处省略500字），状态转移方程变成：

f[i][v] = max{ f[i-1][v],  f[i][v - Ci] +Wi }

时间复杂度降为O(VN)。

c.多重背包

有N种物品（每种物品Mi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。

f[i][v] = max{ f[i-1][v - kCi] + kWi | 0 <= k <= Mi }

时间复杂度O( Vsum(Mi) )，空间复杂度仍然可以用滚动数组优化后可以达到O( V )。

优化：采用二进制拆分物品，将Mi个物品拆分成容量为1、2、4、8、... 2^k、Mi-( 2^(k+1) - 1 ) 个对应价值为Wi、2Wi、4Wi、8Wi、...、2^kWi、( Mi-( 2^(k+1) - 1 ) ) Wi的物品，然后采用01背包求解。

这样做的时间复杂度降为O(Vsum(logMi) )。

**【例题8】一群强盗想要抢劫银行，总共N(N <= 100)个银行，第i个银行的资金为Bi亿，抢劫该银行被抓概率Pi，问在被抓概率小于p的情况下能够抢劫的最大资金是多少？**

p表示的是强盗在抢银行时至少有一次被抓概率的上限，那么选择一些银行，并且计算抢劫这些银行都不被抓的的概率pc，则需要满足1 - pc < p。这里的pc是所有选出来的银行的抢劫时不被抓概率（即1 - Pi）的乘积，于是我们用资金作为背包物品的容量，概率作为背包物品的价值，求01背包。状态转移方程为：

f[j] = max{ f[j], f[j - pack[i].B] * (1-pack[i].p) }

最后得到的f[i]表示的是抢劫到 i 亿资金的最大不被抓概率。令所有银行资金总和为V，那么从V-0进行枚举，第一个满足1 - f[i] < p的i就是我们所要求的被抓概率小于p的最大资金。

### 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。

**【例题9】**对于一条n(n <= 11)个点的哈密尔顿路径C1C2...CN（经过每个点一次的路径）的值由三部分组成：

**1、每个顶点的权值Vi的和**

**2、对于路径上相邻的任意两个顶点CiCi+1，累加权值乘积 Vi*Vi+1**

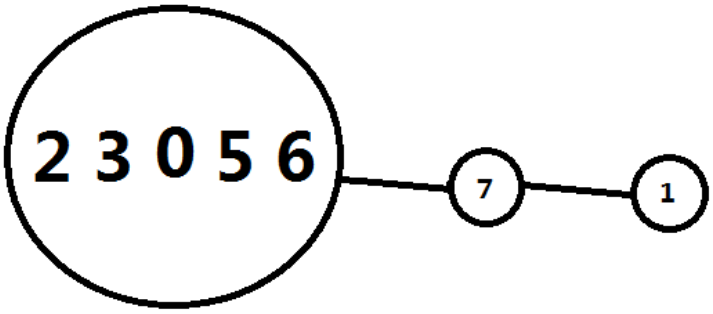
**3、对于相邻的三个顶点CiCi+1Ci+2，如果Ci和Ci+2之间有边，那么累加权值三乘积 Vi*Vi+1*Vi+2**

**求值最大的哈密尔顿路径的权值 和 这样的路径的个数。**

枚举所有路径，判断找出值最大的，复杂度为O(n！)，取缔！

由于点数较少，采用二进制表示状态，用d[i][j][k]表示某条哈密尔顿路径的最大权值，其中i是一个二进制整数，它的第t位为1表示t这个顶点在这条哈密尔顿路径上，为0表示不在路径上。j和k分别为路径的最后两个顶点。那么图二-4-1表示的状态就是：

$d[(11101111)_2][7][1]$




图二-4-1

明确了状态表示，那么我们假设02356这5个点中和7直接相连的是i，于是就转化成了子问题...->j -> i -> 7，我们可以枚举i = 0， 2， 3， 5， 6。

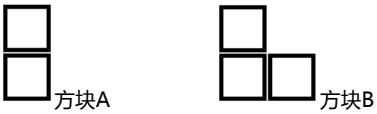
直接给出状态转移方程：

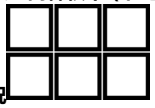
d[i][j][k] = max{ d[i ^ (1<<k)][t][j] + w(t, j, k) | (i & (1<<t)) != 0 }

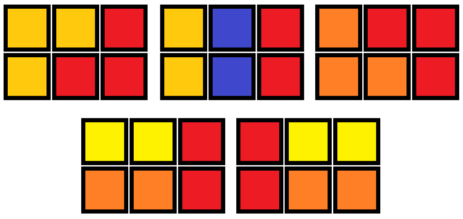
这里用到了几个位运算:i ^ (1<<k)表示将i的二进制的第k位从1变成0，i & (1<<t)则为判断i的二进制表示的第t位是否为1，即该路径中是否存在t这个点。这个状态转移的实质就是将原本的 ...->j -> k 转化成更加小规模的去掉k点后的子问题 ... -> t -> j 求解。而w(t, j, k)则表示 t->j->k这条子路径上产生的权值和，这个可以由定义在O(1)的时间计算出来。

d[ (1<<j) | (1<<k) ][j][k] 为所有的两个点的路径的最大值，即最小的子问题。这个问题的状态并非线性的，所以用记忆化搜索来求解状态的值会事半功倍。

**【例题10】**




**利用以上两种积木（任意数量，可进行旋转和翻转），拼出一个m*n( 1<= m <= 9, 1 <= n <= 9 )的矩形，问这样的方式有多少种。如m = 2, n = 3的情况**  **，有以下5种拼接方式：**

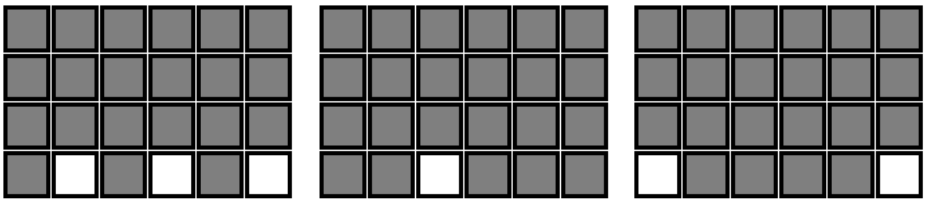


图二-4-2

经典问题，2进制状态压缩。有固定套路，就不纠结是怎么想出来的了， 反正第一次看到这种问题我是想不出来，你呢？但是照例还是得引导一下。

如果问题不是求放满的方案数，而是求前M-1行放满，并且第M行的奇数格放上骨牌而偶数格不放 或者 第M行有一个格子留空 或者 第M行的首尾两个格子留空，求方案数（这是三个问题，分别对应图二-4-3的三个图）。这样的问题可以出一箩筐了，因为第M行的情况总共有2^n，按照这个思路下去，我们发现第i (1 <= i <= m)行的状态顶多也就2^n种，这里的状态可以用一个二进制整数来表示，**对于第i行，如果这一行的第j个格子被骨牌填充则这个二进制整数的第j位为1，否则为0。**

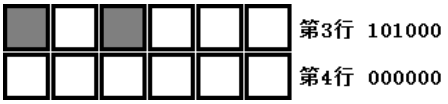
图二-4-3中的三个图的第M行状态可以分别表示为$(101010)_2$、$(110111)_2$、$(011110)_2$，那么如果我们已知第i行的状态k对应的方案数，并且状态k放置几个骨牌后能够将i+1行的状态变成k'，那么第i+1行的k'这个状态的方案数必然包含了第i行的状态k的方案数，这个放置骨牌的过程就是状态转移。



图二-4-3

用一个二维数组DP[i][j] 表示第i行的状态为j的骨牌放置方案数(其中 $1<=i<=m$, $0 <= j < 2^n$)，为了将问题简化，我们虚拟出一个第0行，则有DP[0][j] = (j == $2^n$-1) ? 1 : 0；这个就是我们的**初始状态**，它的含义是这样的，因为第0行是我们虚拟出来的，所以第0行只有完全放满的时候才有意义，也就是第0行全部放满（状态的二进制表示全为1，即十进制表示的$2^n$-1 ）的方案数为1，其它情况均为0。

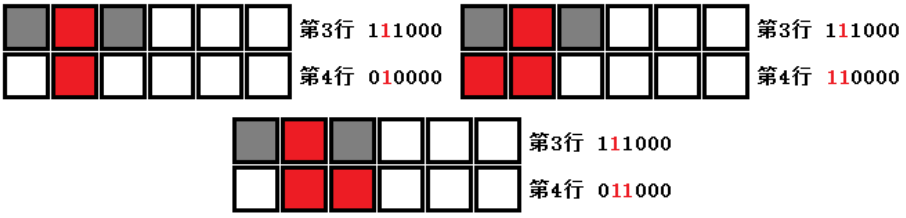
那么如何进行状态转移呢？假设第3行的某个状态$(101000)_2$的方案数DP[3][$(101000)_2$] = 5，如图二-4-4所示：




图二-4-4

我们需要做的就是通过各种方法将第3行填满，从而得到一系列第4行可能的状态集合S，并且对于每一个在S中的状态s，执行DP[4][s] += DP[3][$(101000)_2$]（两个状态可达，所以方案数是可传递的，又因为多个不同的状态可能到达同一个状态，所以采用累加的方式）。

根据给定的骨牌，我们可以枚举它的摆放方式，图二-4-5展示了三种骨牌的摆放方式以及能够转移到的状态，但是这里的状态转移还没结束，因为第3行尚未放满，问题求的是将整个棋盘铺满的方案数，所以只有当第i行全部放满后，才能将状态转移给i+1行。

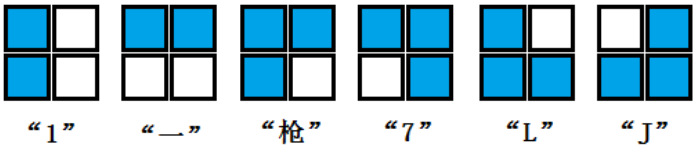



图二-4-5

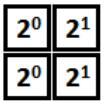
枚举摆放的这一步可以采用dfs递归枚举列，递归出口为列数col == N时。dfs函数的原型可以写成如下的形式：

```
void dfs( int col, int nextrow, int nowstate, int nextstate, LL cnt);
// col       表示当前枚举到的列编号
// nextrow   表示下一行的行编号
// nowstate  表示当前枚举骨牌摆放时第i 行的状态（随着放置骨后会更新）
// nextstate 表示当前枚举骨牌摆放时第i+1行的状态（随着放置骨后会更新）
// cnt       状态转移前的方案数，即第i行在放置骨牌前的方案数
```

然后再来看如何将骨牌摆上去，这里对骨牌进行归类，旋转之后得到如下六种情况：




图二-4-6




图二-4-7

为了方便叙述，分别给每个类型的骨牌强加了一个奇怪的名字，都是按照它自身的形状来命名的，o(ﾟ□ﾟ)o。然后我们发现它们都被圈定在一个2X2的格子里，所以每个骨牌都可以用2个2位的2进制整数来表示，编码方式类似上面的状态表示法（参照图6，如果骨牌对应格子为蓝色则累加格子上的权值），定义如下：

```
int blockMask[6][2] = {
    {1, 1},    // 竖向2X1
    {3, 0},    // 横向1X2
    {3, 1},    // 枪
    {3, 2},    // 7
    {1, 3},    // L
    {2, 3},    // J
};
```

blockMask[k][0]表示骨牌第一行的状态，blockMask[k][1]表示骨牌第二行的状态。这样一来就可以通过简单的位运算来判断第k块骨牌是否可以放在(i, col)这个格子上，这里的i表示行编号，col则表示列编号。接下来需要用到位运算进行状态转移，所以先介绍几种常用的位运算：

a. x & (1<<i)　值如果非0，表示x这个数字的二进制表示的第i(i >= 0)位为1，否则为0；

b. x & (y<<i)　值如果非0，表示存在一个p(i <= p < i+k)，使得x这个数字的二进制表示的第p位和y的p-i位均为1（k为y的二进制表示的位数）；

c. x | (1<<i)　将x的第i位变成1（当然，有可能原本就为1，这个不影响）；

d. x | (y<<i)　将x的第i~i+k-1位和y进行位或运算（k为y的二进制表示的位数），这一步就是模拟了**骨牌摆放**；

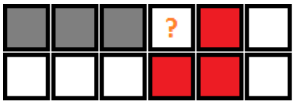
那么这个格子可以放置第k个骨牌的条件有如下几个：

1、当前骨牌横向长度记为w，那么w必须满足 col + w <= N，否则就超出右边界了。

2、 nowstate & (blockMask[k][0]<<col) == 0，即第i行，骨牌放入前**对应的格子**为空（**对应的格子**表示骨牌占据的格子，下同）

3、nextstate & (blockMask[k][1]<<col) == 0，即第i+1行，骨牌放入前对应的格子为空

4、最容易忽略的一点是，“J”骨牌放置时，它的缺口部分之前必须要被骨牌铺满，否则就无法满足第i行全铺满这个条件了，如图二-4-8所示的情况。




图二-4-8

当四个条件都满足就可以递归进入下一层了，递归的时候也是采用位运算，实现如下：

```
dfs( col+1, nextrow, nowstate|(blockMask[k][0]<<col), nextstate|(blockMask[k][1]<<col), cnt );
```

这里的位或运算（|）就是模拟将一个骨牌摆放到指定位置的操作（参见位运算d）。

当然，在枚举到第col列的时候，有可能(i, col)这个格子已经被上一行的骨牌给“占据”了（是否被占据可以通过 (1<<col) & nowstate 得到），这时候我们只需要继续递归下一层，只递增col，其它量都不变即可，这表示了这个格子什么都不放的情况。

### 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。

**【例题11】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。

## 三、动态规划的常用状态转移方程

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$O(n^t)$，每个子问题需要用到$O(n^e)$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)

【例题4】和【例题5】都是这类方程。

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)

【例题7】是这类方程的变形，最典型的见最长公共子序列问题。

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$O(n^{t+e})$，空间复杂度是$O(n^t)$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。

## 四、动态规划和数据结构结合的常用优化

### 1、滚动数组

【例题12】例题7（回文串那题）的N变成5000，其余不变。

回忆一下那个问题的状态转移方程如下：

```
d[i][j] =  {
                d[i+1][j-1]                    | A[i] == A[j]
                min{ d[i+1][j], d[i][j-1] } + 1  |  A[i] != A[j]
           }
```

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$O(n^2)$，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：

**【例题13】例题4（最长递增子序列那题）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$O(n^2)$，利用单调性优化后可以将复杂度将至O(nlogn)。

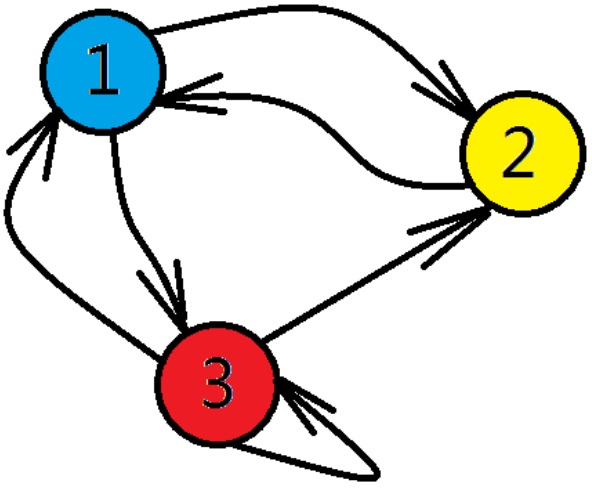
【例题14】**给定n个元素(n <= 100000)的序列，将序列的所有数分成x堆，每堆都是单调不增的，求x的最小值。**

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

**【例题15】三个小岛，编号1、2、3，老王第0天在1号岛上。这些岛有一些奇怪的规则，每过1天，1号岛上的人必须进入2、3号岛；2号岛上的人必须进入1号岛；3号岛上的人可以前往1、2或留在3号岛。问第n(n <=$10^9$)天老王在到达1号岛的行走方案，由于数据比较大，只需要输出 ans MOD 100000007的值即可。**




图四-3-1

临时想的一个问题，首先看问题有几个维度，岛和天数，而且状态转移是常数级的，所以这是个2D/0D问题，我们用f[i][j]表示第i天在j号岛上的方案数，那么初始状态f[0][1] = 1, f[0][2] = f[0][3] = 0。

状态转移可以参考图四-3-1，有：

f[i][1] = f[i-1][2] + f[i-1][3]

f[i][2] = f[i-1][1] + f[i-1][3]

f[i][3] = f[i-1][1] + f[i-1][3]

我们要求的结果就是f[n][1]，再来看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，如图四-3-2

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 0 |
| 3 | 1 | 1 | 1 |

图四-3-2

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，$A^2$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，$A^n$ 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$A^n$ MOD 100000007的值了。　　$A^n$当n为偶数的时候等于$(A^{n/2})*(A^{n/2})$；当n为奇数的时候，等于$(A^{n/2})*(A^{n/2})*A$，这样求解矩阵$A^n$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 = $A^n$ MOD 100000007，那么T[1][1]就是我们要求的解了。

**4、斜率优化**

**【例题16】n（n <= 500000）个单词，每个单词输出的花费为Ci(1 <= i <= n)，将k个连续的单词输出在一行的花费为：**

$$(\sum_{i=1}^{k} C_i)^2 + M$$

**其中M为常数，求一个最佳方案，使得输出所有单词总的花费最小。**

令d[i]为前i个单词组织出的最小花费，s[i] = sum{Ck | 0 <= k <= i }，其中s[0] = 0

状态转移方程 d[i] = min{ d[j] + (s[i] - s[j])$^2$ + M | 0 <= j < i }　(1 <= i <= n)

这是个1D/1D问题，空间复杂度O(n), 时间复杂度为O(n$^2$)，对于500000的数据来说是不可能在给定时间出解的。

令g[j] = d[j] + (s[i] - s[j])$^2$+ M，表示由 j 到 i 的决策。

对于两个决策点 j < k，如果k的决策值小于j （即g[k] < g[j] ），则有：

d[k] + (s[i] - s[k]) $^2$ + M < d[j] + (s[i] - s[j])$^2$+ M，然后将左边转化成关于j和k的表达式，右边转化成只有i的表达式。

(中间省略t行推导过程...，t >= 5)

(d[j] - d[k] + s[j]$^2$- s[k]$^2$) / (s[j] - s[k]) < 2*s[i]

令xj = d[j] + s[j]$^2$,　yj = s[j]，则原式转化成：(xj - xk) / (yj - yk) < 2*s[i]

不等式左边是个斜率的形式，我们用斜率函数来表示 slope(j, k) = (xj - xk) / (yj - yk)

那么这里我们可以得出两个结论：

1、当两个决策j、k (j < k)满足 slope(j, k) < 2*s[i]时，j决策是个无用决策，并且因为s[i]是个单调不降的，所以对于i < i'，则有slope(j, k) < 2*s[i] < 2*s[i']，即j决策在随着i增大的过程中也是一直都用不到的。

2、对于当前需要计算的值f[i]，存在三个决策j、k、l，并且有 j < k < l，如果slope(j, k) > slope(k, l)，则k这个决策是个无用决策，证明需要分情况讨论：

i. slope(k, l) < 2*s[i]，则l的决策比k更优；

ii. slope(k, l) >= 2*s[i]，则 slope(j, k) > slope(k, l) >= 2*s[i]，j的决策比k更优；

综上所述，当slope(j, k) > slope(k, l)时，k是无用决策；

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。

**【例题17】给定一个n(n <= 100000)个元素的序列a[i] (1 <= a[i] <= INF)，定义"和谐序列"为序列的任何两个相邻元素相差不超过K，求a[i]的子序列中，"和谐序列"的个数 MOD 10000007。**

用d[i]表示以第i个元素结尾的"和谐序列"的个数， 令d[0] = 1， 那么sum {d[i] | 1 <= i <= n}就是我们要求的解。列出状态转移方程：

d[i] = sum{ d[j] | j==0 || j < i && abs(a[i] - a[j]) <= K }

这是一个1D/1D问题，如果不进行任何优化，时间复杂度是O(n^2)。

我们首先假设K是个无限大的数，也就是不考虑abs(a[i] - a[j]) <= K这个限制，那这个问题要怎么做？很显然d[1] = 1，d[2] = 1 + d[1]，d[3] = d[1] + d[2] + 1（这里的1其实是状态转移方程中j == 0的情况，也就是只有一个数a[i]的情况），更加一般地：

d[i] = sum{d[j] | j < i} + 1
对于这一步状态转移还是O(n)的，但是我们可以将它稍加变形，如下：
d[i] = sum(1, INF) + 1;　　(这里的sum是树状数组的区间求和函数)
得到d[i]的值后，再将d[i]插入到树状数组中，利用树状数组第1条操作 add(a[i], d[i])；
基于这样的一种思路，我们发现即使有了限制K，同样也是可以求解的，只是在求和的时候进行如下操作：
d[i] = sum(a[i] - K, a[i] + K) + 1;
这样就把原本O(n)的状态转移变成了 O（logn），整个算法时间复杂度O（logn）。

**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。

这里给出线段树的求区间最值需要用到的函数，即：
1、insert(x, v)　　在下标为x的位置插入一个值v；　耗时 O( log(n) )
2、query(l, r)　　求下标在[l, r]上的最值；　　耗时 O( log(n) )

**【例题18】详见 例题13**

还是以最长单调子序列为例，状态转移方程为：
d[i] = max{ d[j] | j < i && a[j] < a[i] } + 1
这里我们首先要保证 1 <= a[i] <= n，如果a[i]是实数域的话，首先要对a[i]进行离散化，排序后从小到大重新编号，最小的a[i]编号为1，最大的a[i]编号为n。
对于状态转移执行线段树的询问操作： d[i] = query( 1, a[i] - 1 ) + 1
然后再执行插入操作： insert( a[i], d[i] )
状态转移同样耗时O(logn)。
这个思想和树状数组的思想很像，大体思路都是将数本身映射到某个数据结构的下标。

**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

## 五、动态规划题集整理

### 1、递推

Recursion Practice　★☆☆☆☆
几个初级递推
Put Apple　★☆☆☆☆
Tri Tiling　★☆☆☆☆
【例题1】
Computer Transformation　★☆☆☆☆　【例题2】
Train Problem II　★☆☆☆☆
How Many Trees?　★☆☆☆☆
Buy the Ticket　★☆☆☆☆
Game of Connections　★☆☆☆☆
Count the Trees　★☆☆☆☆
Circle　★☆☆☆☆
Combinations, Once Again　★★☆☆☆
Closing Ceremony of Sunny Cup　★★☆☆☆
Rooted Trees Problem　★★☆☆☆
Water Treatment Plants　★★☆☆☆
One Person　★★☆☆☆
Relax! It’s just a game　★★☆☆☆
N Knight　★★★☆☆
Connected Graph　★★★★★
楼天城“男人八题”之一

### 2、记忆化搜索

Function Run Fun　★☆☆☆☆
【例题3】
FatMouse and Cheese　★☆☆☆☆
经典迷宫问题
Cheapest Palindrome　★★☆☆☆
A Mini Locomotive　★★☆☆☆

Millenium Leapcow ★★☆☆☆

Brackets Sequence ★★★☆☆
经典记忆化

Chessboard Cutting ★★★☆☆
《算法艺术和信息学竞赛》例题

Number Cutting Game ★★★☆☆

### 3、最长单调子序列

Constructing Roads In JG Kingdom ★★☆☆☆

Stock Exchange ★★☆☆☆

Wooden Sticks ★★☆☆☆

Bridging signals ★★☆☆☆

BUY LOW, BUY LOWER ★★☆☆☆
要求需要输出方案数

Longest Ordered Subsequence ★★☆☆☆

Crossed Matchings ★★☆☆☆

Jack’s struggle ★★★☆☆
稍微做点转化

### 4、最大M子段和

Max Sum
★☆☆☆☆ 最大子段和

Max Sum Plus Plus
★★☆☆☆ 最大M子段和

To The Max
★★☆☆☆ 最大子矩阵

Max Sequence
★★☆☆☆ 最大2子段和

Maximum sum
★★☆☆☆ 最大2子段和

最大连续子序列
★★☆☆☆ 最大子段和

Largest Rectangle in a Histogram ★★☆☆☆ 最大子矩阵变形

City Game
★★☆☆☆ 最大子矩阵扩展

Matrix Swapping II
★★★☆☆ 最大子矩阵变形后扩展

### 5、线性模型

Skiing ★☆☆☆☆

Super Jumping! Jumping! Jumping! ★☆☆☆☆

Milking Time ★★☆☆☆
区间问题的线性模型

Computers ★★☆☆☆
【例题5】

Bridge over a rough river ★★★☆☆ 【例题6】

Crossing River ★★★☆☆
【例题6】大数据版

Blocks ★★★☆☆

Parallel Expectations ★★★★☆ 线性模型黑书案例

## 6、区间模型

Palindrome ★☆☆☆☆

【例题7】

See Palindrome Again ★★★☆☆

## 7、背包模型

饭卡 ★☆☆☆☆

01背包

I NEED A OFFER! ★☆☆☆☆

概率转化

Bone Collector ★☆☆☆☆

01背包

最大报销额 ★☆☆☆☆

01背包

Duty Free Shop ★★☆☆☆

01背包

Robberies ★★☆☆☆

【例题8】

Piggy-Bank ★☆☆☆☆

完全背包

Cash Machine ★☆☆☆☆

多重背包

Coins ★★☆☆☆

多重背包，楼天城“男人八题”之一

I love sneakers! ★★★☆☆

背包变形

## 8、状态压缩模型

ChessboardProblem ★☆☆☆☆

比较基础的状态压缩

Number of Locks ★☆☆☆☆

简单状态压缩问题

Islands and Bridges ★★☆☆☆

【例题9】

Tiling a Grid With Dominoes ★★☆☆☆

骨牌铺方格 4XN的情况

Mondriaan's Dream ★★☆☆☆

【例题10】的简易版

Renovation Problem ★★☆☆☆

简单摆放问题

The Number of set ★★☆☆☆

Hardwood floor ★★★☆☆

【例题10】二进制状态压缩鼻祖

Tetris Comes Back ★★★☆☆

纸老虎题

Bugs Integrated, Inc. ★★★☆☆

三进制状态压缩鼻祖

Another Chocolate Maniac ★★★☆☆

三进制

Emplacement ★★★☆☆

类似Bugs那题，三进制

Toy bricks ★★★☆☆

四进制， 左移运算高于&

Quad Tiling ★★★☆☆
骨牌铺方格 4XN的情况 利用矩阵优化

Eat the Trees ★★★☆☆
插头DP入门题

Formula 1 ★★★☆☆
插头DP入门题

The Hive II ★★★☆☆
插头DP

Plan ★★★☆☆
插头DP

Manhattan Wiring ★★★☆☆
插头DP

Pandora adventure ★★★★☆
插头DP

Tony's Tour ★★★★☆
插头DP，楼天城“男人八题”之一

Pipes ★★★★☆
插头DP

circuits ★★★★☆
插头DP

Beautiful Meadow ★★★★☆
插头DP

I-country ★★★★☆
高维状态表示

Permutaion ★★★★☆
牛逼的状态表示

01-K Code ★★★★☆

Tour in the Castle ★★★★★
插头DP（难）

The Floor Bricks ★★★★★
四进制（需要优化）

## 9、树状模型

Anniversary party ★☆☆☆☆
树形DP入门

Strategic game ★☆☆☆☆
树形DP入门

Computer ★★☆☆☆

Long Live the Queen ★★☆☆☆

最优连通子集 ★★☆☆☆

Computer Network ★★☆☆☆

Rebuilding Roads ★★★☆☆
树形DP+背包

New Year Bonus Grant ★★★☆☆

How Many Paths Are There ★★★☆☆

Intermediate Rounds for Multicast ★★★★☆

Fire ★★★★☆

Walking Race ★★★★☆

Tree ★★★★★
树形DP，楼天城“男人八题”之一

## 10、滚动数组优化常见问题

Palindrome ★☆☆☆☆

Telephone Wire ★☆☆☆☆

Gangsters ★☆☆☆☆

Dominoes ★☆☆☆☆

Cow Exhibition ★☆☆☆☆

Supermarket ★★☆☆☆

## 11、决策单调性

Print Article ★★★☆☆

Lawrence ★★★☆☆

Batch Scheduling ★★★☆☆

K-Anonymous Sequence ★★★☆☆

Cut the Sequence ★★★☆☆

## 12、常用优化

Divisibility ★★☆☆☆
利用同余性质

Magic Multiplying Machine ★★☆☆☆ 利用同余性质

Moving Computer ★★☆☆☆
散列HASH表示状态

Post Office ★★★☆☆
四边形不等式

Minimizing maximizer ★★★☆☆ 线段树优化

Man Down ★★★☆☆
线段树优化

So you want to be a 2n-aire? ★★★☆☆ 期望问题

Expected Allowance ★★★☆☆ 期望问题

Greatest Common Increase Subseq ★★★☆☆ 二维线段树优化

Traversal ★★★☆☆
树状数组优化

Find the nondecreasing subsequences ★★★☆☆ 树状数组优化

Not Too Convex Hull ★★★★☆ 利用凸包进行状态转移

In Action ★★★☆☆
最短路+背包

Search of Concatenated Strings ★★★☆☆ STL bitset 应用

## 13、其他类型的动态规划

Common Subsequence 2D/0D

Advanced Fruits 2D/0D

Travel 2D/1D

RIPOFF 2D/1D

Balls 2D/1D

Projects 2D/1D

Cow Roller Coaster 2D/1D

| | |
|---|---|
| LITTLE SHOP OF FLOWERS | 2D/1D |
| Pearls | 2D/1D |
| Spiderman | 2D/0D |
| The Triangle | 2D/0D |
| Triangles | 2D/0D |
| Magazine Delivery | 3D/0D |
| Tourist | 3D/0D |
| Rectangle | 2D/1D |
| Message | 2D/1D |
| Bigger is Better | 2D/1D |
| Girl Friend II | 2D/1D |
| Phalanx | 2D/1D |
| Spiderman | 最坏复杂度O(NK)，K最大为1000000，呵呵 |
| Find a path | 3D/1D 公式简化，N |

维不能解决的问题试着用N+1维来求解

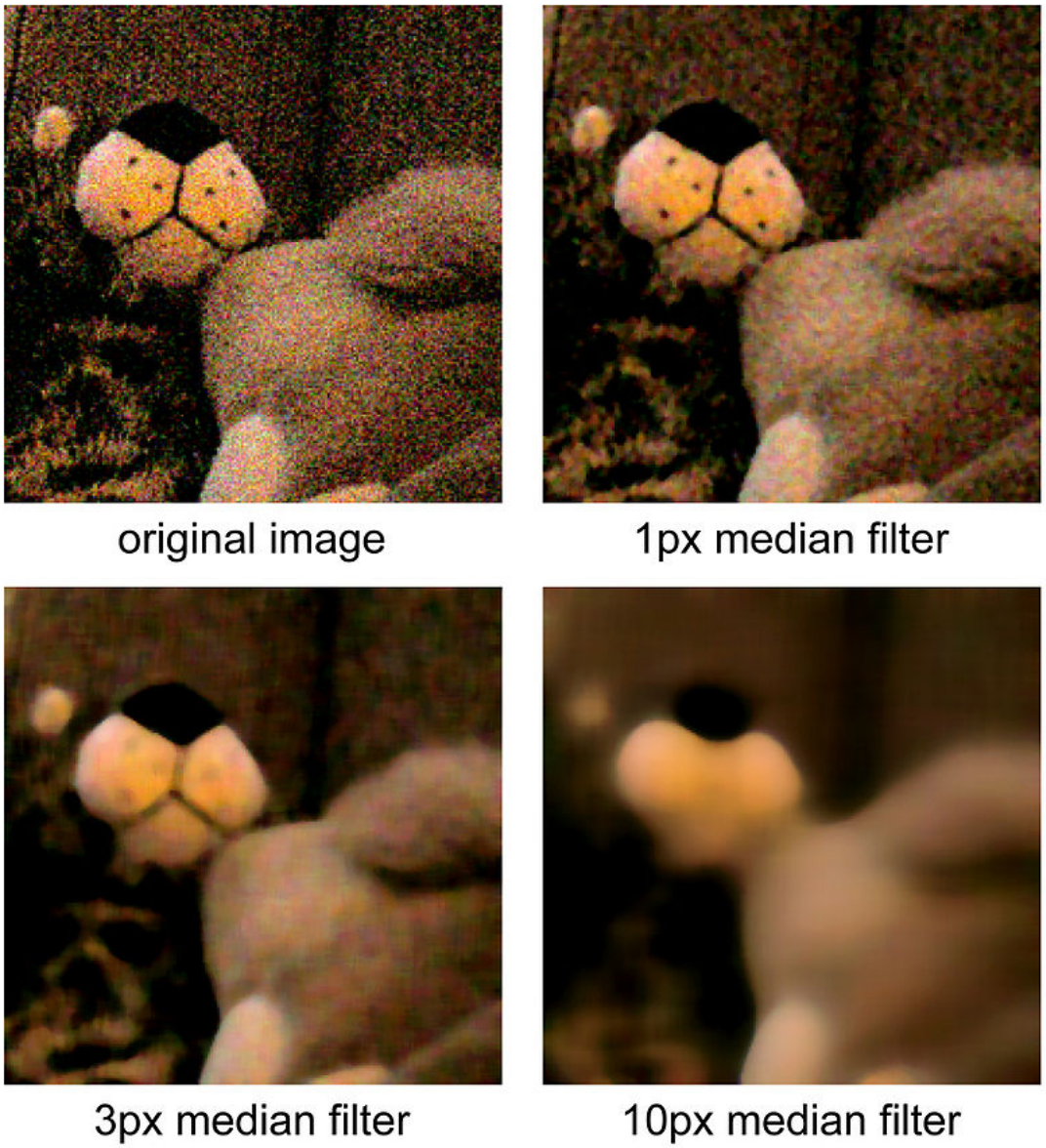
# 目录



## 一、从图形学算法说起

### 1、Median Filter概述

Median Filter 在现代的图形处理中是非常基础并且广泛应用的算法 (翻译叫 **中值滤波器**，为了不让人一看到就觉得是高深的东西，还是选择了使用它的英文名，更加能让人明白是个什么东西) ，如图一-1-1，基本就能猜到这是一个什么样的算法了，可以简单的认为是PS中的"模糊"那个操作。




图一-1-1

**2、r pixel-Median Filter算法**

首先对于一张宽为W，高为H的图片，每个像素点存了一个颜色值，这里为了把问题简化，先讨论黑白图片，黑白图片的每个像素值可以认为是一个[0, 255]的整数（如图一-1-2，为了图片看起来不是那么的密密麻麻，像素值的范围取了[0, 10]）。

**r pixel-Median Filter** 算法描述如下：

对于每个第i行第j列的像素点p(i, j)，像四周扩展一个宽和高均为(2r + 1)的矩形区域，将矩形区域内的像素值按非降序排列后，用排在最中间的那个数字取代原来的像素点p(i, i)的值( 边界的那圈不作考虑 )，下文会将排在最中间的那个数字称为这个序列的**中位数。**

| | | | | | | |
|---|---|---|---|---|---|---|
| 7 | 10 | 5 | 7 | 2 | 5 | 4 |
| 8 | 4 | 9 | 4 | 7 | 1 | 2 |
| 3 | 0 | 2 | 6 | 1 | 7 | 4 |
| 4 | 9 | 1 | 3 | 0 | 1 | 1 |
| 6 | 1 | 1 | 2 | 2 | 9 | 4 |
| 10 | 8 | 1 | 1 | 9 | 2 | 8 |

图一-1-2

如图一-1-2，r = 1，红框代表(2, 3) (下标从0计数，行优先)这个像素点所选取的2r + 1的矩形区域，将内中数字进行非降序排列后，得到**[0 1 1 2 3 4 6 7 9]**，所以(2, 3)这个像素点的值将从 **6** 变成 **3。**

这样就可以粗略得到一个时间复杂度为O(n^4logn )的算法（枚举每个像素点，对于每个像素点取矩形窗口元素排序后取中值）。n代表了图片的尺寸，也就是当图片越大，这个算法的执行效率就越低，而且增长迅速。那么如何将这个算法进行优化呢？如果对于二维的情况不是很容易下手的话，不妨先从一维的情况进行考虑。

**3、一维模型**

将问题转化成一维，可以描述成：给定**n**(n <= 100000)个范围在[0, 255]的数字序列a[i] (1 <= i < = n)和一个值**r** (2r+1 <= n)，对于所有的a[k] (r+1 <= k <= n-r)，将它变成 a[k-r ... k+r] 中的中位数。

```
a[1...7] = [1 7 6 4 3 2 1]   r = 2
d[3] = median( [1 7 6 4 3] ) = 4
d[4] = median( [7 6 4 3 2] ) = 4
d[5] = median( [6 4 3 2 1] ) = 3
```

所以原数组就会变成a[1..7] = [1 7 4 4 3 2 1]

那么总共需要计算的元素为n-2r，取这些元素的左右r个元素的值需要2r+1次操作，(n-2r)*(2r+1) 当r = (n-1)/4 时取得最大值，为 (n+1)^2 / 4，再加上排序的时间复杂度，所以最坏情况的时间复杂度为O( n^2 logn )。n的范围不允许这么高复杂度的算法，尝试进行优化。

考虑第i个元素的2r+1区域**a[ i-r ... i+r ]**和第i+1个元素的2r+1区域**a[ i+1-r ... i+1+r ]**，后者比前者少了一个元素a[i-r]，多了一个元素a[i+1+r]，其它元素都是一样的，那么在计算第i+1个元素的情况时如何利用第i个元素的情况就成了问题的关键。

**4、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)。

具体得，用一个辅助数组d[256]，插入a[i]执行的是d[ a[i] ] ++，删除a[i]执行的是 d[ a[i] ] --；询问操作是对d数组进行顺序统计，顺序枚举i，找到第一个满足**sum{d[j] | 1 <= j <= i}** >= r+1的 i 就是所求中位数，这样就得到了一个时间复杂度为O(Kn)的算法，其中K是数字的值域（这里讨论的问题值域是256）。

相比之前的算法，这种方法已经前进了一大步，至少n的指数下降了大于一个数量级，但是也带来了一个问题，如果数字的值域很大，复杂度还是会很大，所以需要更好的算法支持。

**5、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)

试想一下，如果用HASH来实现这两个函数，那么1的复杂度是O(1)，而2的复杂度就是O(n)了，而树状数组实现的这两个函数可以让两者的复杂度都达到O(logn)，具体的实现先卖个关子，留到第二节着重介绍。

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。

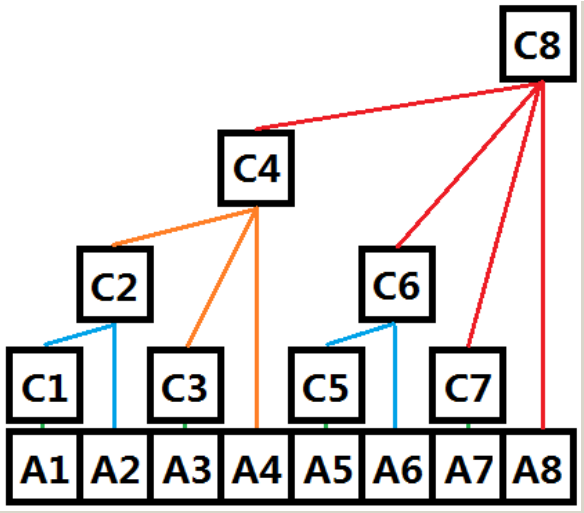
这样一来，一维的Median Filter模型的整体时间复杂度就降到了O(n * logn * logn)，已经是比较高效的算法了。

接下来就是要来说说树状数组的具体实现了。

## 二、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。




图二-1-1

如图二-1-1，其中A为普通数组，C为树状数组（C在物理空间上和A一样都是连续存储的）。树状数组的第4个元素C4的父结点为C8 (4的二进制表示为"100"，所以k=2，那么4 + 2^2 = 8)，C6和C7同理。C2和C3的父结点为C4，同样也是可以用上面的关系得出的，那么从定义出发，奇数下标一定是叶子结点。

**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）。从图中可以得出：

C1 = A1
C2 = C1 + A2 = A1 + A2
C3 = A3
C4 = C2 + C3 + A4 = A1 + A2 + A3 + A4
C5 = A5
C6 = C5 + A6 = A5 + A6
C7 = A7
**C8 = C4 + C6 + C7 + A8 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8**

建议直接看C8，因为它最具代表性。

我们从中可以发现，其实Ci还有一种更加普适的定义，它表示的其实是一段原数组A的连续区间和。根据定义，右区间是很明显的，一定是i，即Ci表示的区间的最后一个元素一定是Ai，那么接下来就是要求Ci表示的第一个元素是什么。从图上可以很容易的清楚，其实就是顺着Ci的最左儿子一直找直到找到叶子结点，那个叶子结点就是Ci表示区间的第一个元素。

更加具体的，如果i的二进制表示为 ABCDE1000，那么它最左边的儿子就是 ABCDE0100，这一步是通过结点父子关系的定义进行逆推得到，并且这条路径可以表示如下：

**ABCDE1000 => ABCDE0100 => ABCDE0010 => ABCDE0001**

这时候，ABCDE0001已经是叶子结点了，所以它就是Ci能够表示的第一个元素的下标，那么我们发现，如果用k来表示i的二进制末尾0的个数，Ci能够表示的A数组的区间的元素个数为2^k，又因为区间和的最后一个数一定是Ai，所以有如下公式：

Ci = sum{ A[j] | i - 2^k + 1 <= j <= i }  （帮助理解：将j的两个端点相减+1 等于2^k）

**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。

那么其实就是求在Ai改变的时候会影响哪些Ci，看图二-1-1的树形结构就一目了然了，Ai的改变只会影响Ci及其祖先结点，即A5的改变影响的是C5、C6、C8；而A1的改变影响的是C1、C2、C4、C8。

也就是每次add(i, v)，我们只要更新Ci以及它的祖先结点，之前已经讨论过两个结点父子关系是如何建立的，所以给定一个x，一定能够在最多log(n) (这里的n是之前提到的值域) 次内更新完所有x的祖先结点，add(i, v)的主体代码（去除边界判断）也只有一行代码：

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

和求和操作类似，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$2^k$，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$2^k$的方法。

来看一段补码小知识：

清楚补码的表示的可以跳过这一段，计算机中的符号数有三种表示方法，即原码、反码和补码。三种表示方法均有符号位和数值位两部分，符号位都是用0表示“正”，用1表示“负”，而数值位，三种表示方法各不相同。这里只讨论整数补码的情况，在计算机系统中，数值一律用补码来表示和存储。原因在于，使用补码，可以将符号位和数值域统一处理；同时，加法和减法也可以统一处理。整数补码的表示分两种：

正数：正数的补码即其二进制表示；

例如一个8位二进制的整数+5，它的补码就是 00000101（标红的是符号位，0表示"正"，1表示“负”）

负数：负数的补码即将其整数对应的二进制表示所有位取反(包括符号位)后+1；

例如一个8位二进制的整数-5，它的二进制表示是00000101，取反后为11111010，再+1就是11111011，这就是它的补码了。

下面的等式可以帮助理解补码在计算机中是如何工作的

+5 + (-5) = 00000101 + 11111011 = 1 00000000 (溢出了!!!) = 0

这里的加法没有将数值位和符号位分开，而是统一作为二进制位进行计算，由于表示的是8进制的整数，所以多出的那个最高位的1会直接舍去，使得结果变成了0，而实际的十进制计算结果也是0，正确。

补码复习完毕，那么来看下下面这个表达式的含义：　　x & (-x)　　(其中 x >= 0)

首先进行&运算，我们需要将x和-x都转化成补码，然后再看&之后会发生什么，我们假设 x 的二进制表示的末尾是连续的 k 个 0，令x的二进制表示为 $X_0X_1X_2...X_{n-2}X_{n-1}$，则 $\{X_i = 0 \mid n-k <= i < n\}$，这里的$X_0$表示符号位。

x 的补码就是由三部分组成：　　(0)($X_1X_2...X_{n-k-1}$)(k个0)　　其中$X_{n-k-1}$为1，因为末尾是k个0，如果它为0，那就变成k+1个0了。

-x的补码也是由三部分组成：　　(1)($Y_1Y_2...Y_{n-k-1}$)(k个0)　　其中$Y_{n-k-1}$为1，其它的$X_i$和$Y_i$相加为1，想想补码是怎么计算的就明白了。

那么 x & (-x) 也就显而易见了，由两部分组成 (1)(k个0)，表示成十进制为 $2^k$ 啦。

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。

## 三、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($n^2$)。不推荐！

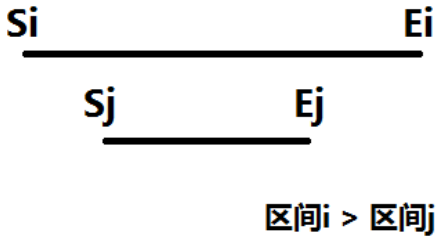
来看一个给定**n**个元素的排列 $X_0\ X_1\ X_2\ \dots\ X_{n-2}\ X_{n-1}$,对于某个 $X_i$ 元素，如果想知道以它为"首"的逆序对的对数( 形如($X_i$,$X_j$) 的逆序对)，就是需要知道 $X_{i+1}\ \dots\ X_{n-2}\ X_{n-1}$ 这个子序列中小于 $X_i$ 的元素的个数。

那么我们只需要对这个排列从后往前枚举，每次枚举到 $X_i$ 元素时，执行cnt += sum($X_i$-1)，然后再执行add($X_i$, 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。

【例题4】给定**N(N <= 100000)**个区间，定义两个区间**(Si, Ei)**和**(Sj, Ej)**的**'>'**如下：如果**Si <= Sj and Ej <= Ei and Ei - Si > Ej - Sj,** 则 **(Si, Ei) > (Sj, Ej)**，现在要求每个区间有多少区间**'>'**它。

将上述三个关系式化简，可以得到 区间i > 区间i 的条件是：区间i 完全覆盖 区间i，并且两者不相等。




图三-3-1

对区间进行排序，排序规则为：左端点递增，如果左端点相同，则右端点递减。

然后枚举区间，不断插入区间右端点，因为区间左端点是保持递增的，所以对于某个区间(Si, Ei)，只需要查询树状数组中[Ei, MAX]这一段有多少已经插入的数据，就能知道有多少个区间是比它大的，这里需要注意的是多个区间相等的情况，因为有排序，所以它们在排序后的数组中一定是相邻的，所以在遇到有相等区间的情况，需要"延迟"插入。等下一个不相等区间出现时才把之前保存下来的区间右端点进行插入。插入完毕再进行统计。

这里的插入即add(Ej, 1)，统计则是sum(MAX) - sum(Ei - 1)（其中j < i)。

**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。**

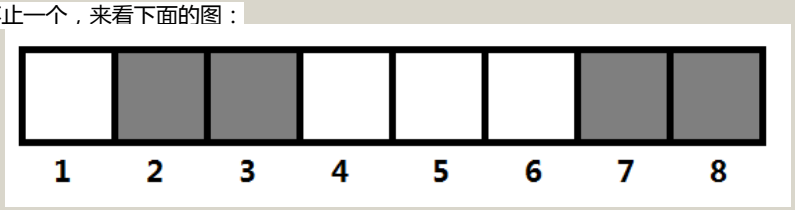
**现在要你来模拟这个神奇的容器的功能。可以抽象成两种操作：**

**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，来看下面的图：

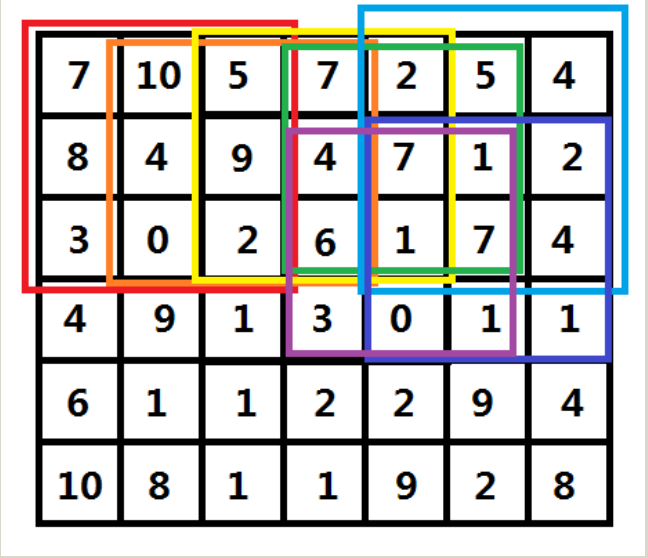



图三-4-1

图三-4-1中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。




图三-5-1

有了一维的基础，对于二维的情况，其实也是一样的，如图三-5-1，图中红色的框为(1, 1)这个像素点的(2r+1)矩形区域，橙色的框则是(1, 2)的，它们的差别其实只是差了两列；同样的，橙色框和黄色框也差了两列，于是，我们可以从左向右枚举，每次将这个矩形框向右推进一格，然后将"离开"框的那一列数据从树状数组中删除，将"进入"框的那一列数据插入到树状数组中，然后统计中位数。

当枚举到右边界时，将矩形框向下推进一格，然后迂回向左，同样按照之前的方案统计中位数，就这样呈蛇字型迂回前进（具体顺序如图所示的红、橙、黄、绿、青、蓝、紫），这样就得到了一个O( $n^3$lognlogn )的算法，比朴素算法下降了一个数量级。

**6、多维树状数组模型**

**【例题6】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。

## 四、树状数组题集整理

| | | |
|---|---|---|
| Enemy Soilders | ★☆☆☆☆ | 树状数组基础 |
| Stars | ★☆☆☆☆ | 降维 |
| Tunnel Warfare | ★★☆☆☆ | 二分模型 |
| Apple Tree | ★★☆☆☆ | |
| Mobile phones | ★★☆☆☆ | 二维PUIQ |
| Minimum Inversion Number | ★★☆☆☆ | 树状数组求逆序数 |
| Ultra-QuickSort | ★★☆☆☆ | 求逆序数，经典树状数组 |
| Cows | ★★☆☆☆ | 区间问题转化成逆序求解 |
| MooFest | ★★☆☆☆ | 转化成求和 |
| Ping pong | ★★☆☆☆ | |
| Conturbatio | ★★☆☆☆ | 其实并不需要树状数组，直接静态求和就行O(1) |

| | | |
|---|---|---|
| Cube | ★★☆☆☆ | 三维树状数组 |
| Japan | ★★★☆☆ | 逆序模型，交叉统计问题 |
| Life is a Line | ★★★☆☆ | 逆序模型，几何上的树状数组 |
| Magic Ball Game | ★★★☆☆ | 树上的统计 |
| Super Mario | ★★★☆☆ | 降维思想 |
| Median Filter | ★★★☆☆ | 模糊算法 |
| Disharmony Trees | ★★★☆☆ | 统计问题 |
| Find the non sub | ★★★☆☆ | 动态规划+树状数组 |
| Traversal | ★★★☆☆ | 动态规划+树状数组 |
| KiKi's K-Number | ★★★☆☆ | 二分模型/K大数 |
| Matrix | ★★★☆☆ | 二维IUPQ |
| Sum the K-th's | ★★★☆☆ | K大数 / 二分模型 |
| Kiki & Little Kiki 1 | ★★★☆☆ | 二分模型 |
| The *k*-th Largest Group | ★★★☆☆ | 树状数组 + 并查集 |
| Inversion | ★★★★☆ | 逆序数 |
| Harmony Forever | ★★★★☆ | 二分模型 |