直角三角形解题报告

首先可以注意到 70%的数据 N<=10, 可以直接搜索每根木棍的取舍情况, 复杂度 O(T*4^N)

对于 100%的数据, 注意到 L<=20, 所以合法情况下, 最长边的长度<=150. 于是我们可以使用 dp 来判断可行性. 设 f[i][a][b][c]表示考虑了前 i 根木棍, 是否可以组成(a, b, c)这种长度.复杂度 O(150^3*N*T), 枚举的时候保证 a<=b<=c, 可以减少很多无用状态. 另外, 带比较好的剪枝的搜索可以更快速地通过本题的数据.

字符串解题报告

30%做法：将所有子串取出后直接排序。

60%做法：用基排或 trie

100%做法：将字符串的所有后缀排序，将排在第 i 位的后缀记为 SA(i)，然后求出每一个 SA(i)与 SA(i-1)的 LCP 值，记为 h(i),（其实就是个后缀数组），可以发现找第 K 大子串等价于找所有后缀的第 K 大前缀，可以发现相对于 SA（i-1)，SA(i)有 SA（i)的长度-h(i)个不同的前缀，并且这些新的前缀严格大于 SA（i-1)的长度。逐个累加统计后输出即可。

可靠道路网络

非官方 solution　　---Edited by CD

【暴力】

2^(n*n)枚举全部边选或不选，然后再在图上枚举全部边删边后原图是否连通。

预计得分 5%。复杂度为 O(2^(n*n)*m^2)

【优化 1】

把判断网络是否可靠的方法改成求割边。预计复杂度为 O(2^(n*n)*m)，预计得分 10%。

【优化 2】

把盲目的搜索换成迭代加深搜索，预计得分 30%.

【题目模型】

发现题目的模型是给你一幅连通图，你需要加尽量少的边使得原图为一个边双连通分支(如果有一个边集合，删除这个边集合以后，原图变成多个连通块，就称这个点集为**割边集合**。一个图的**边连通度**的定义为，最小割边集合中的边数。如果一个无向连通图的边连通度大于 1，则称该图是**边双连通的(edge biconnected)**，简称双连通或重连通)

于是可以用传统的缩边双连通的方法，把原图缩成一棵树，发现每次加一条边就是在把树上的一条路径缩成一个点。

【错误贪心】

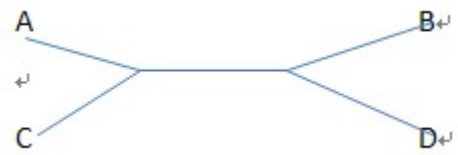
很容易认为每次在缩完点的树上找直径，每次把直径缩成一个点后继续重复至只剩下一个点为止。这样的贪心是错误的，详见以下样例：

9 8
1 2
2 3
3 4
1 5
1 6
1 7
7 8
7 9

答案应为 2，可贪心结果为 3.

【正确贪心】

其实路径的长度并不是本质，本质是叶子数，叶子数能很好的说明树的分叉情况，而不是路径长度决定树的分叉情况。而把原树缩成一个点的操作数又是取决于树的分叉情况的。可以轻易地证明当叶子数量大于 3 时，可以找到一条路径使得叶子数量减少 2.（自己画画就知道了，证明过于繁琐在此不赘述）寻找的方法就是提取树上两条相交的路径：



上图中 A-D,B-C 这两条路径就是相交的，可以把 A-D 或 B-C 缩成一个点，这样叶子的个数必定减少 2.对于叶子数<4 的情况，可以 2 步完成，推得答案数公式为叶子树除二上取整。

于是当叶子数大于 n 每次在树上选取形如 B-C,A-D 的路径。时间复杂度为 O(m+n^2)，预计得分 100%。笔者实在不知道 45%的部分分如何水掉。欢迎补充。

【更优的算法】

在对树上的遍历时可以方便地得出答案。

在 DFS 遍历时，如遍历完 x 的子树后 x 的各个子树可以保证是一个双连通分支，对每个节点的工作就是把一棵层数为2的树缩成一个点每次只要随便取两个在x的不同子树上的点连边，直到剩下 2 个或 1 个点向上连（这是为了保证 x 与其父亲的连边不是割边，而儿子之间随便连边的正确性显然，因为儿子中的叶子数大于 2 时路径相交的节点就是 x）。

最后就是些许细节，根是叶子的情况和不是叶子的情况分别考虑：是叶子就把根与到根时剩下的未取的叶节点连边；否则，若剩下两个叶子，直接连边；剩下一个叶子，与根连边。

Solution of digit

对于 30%的数据，可以直接枚举 1～n 的每个数，每次暴力统计每个数字出现的次数。

对于另外 70%的数据，我们需要寻求更高效的算法。

【思路 1】

不难发现，当 n 递增时，每个数字出现过的次数都是不递减的。所以可以先二分答案，把问题转化为判定性问题，只要每次统计 1～X 里各个数字个数就可以了。

如何统计 1～X 每个数字出现的个数？

首先考虑一个问题： $1\sim10^i-1$中各个数字会出现多少次？不难通过计算

或找规律发现：1～9 每个数字会出现 $i*10^{i-1}$次,0 会出现 $i*10^{i-1}-\frac{10^{i-1}-1}{9}$ 次。

接下来只要枚举每一位，计算前 i 位确定的情况下每个数字出现的次数就可以解决这个问题了。

由于需要高精度运算，该算法的时间复杂度是 O（D^3）的。比较难通过所有

数据。（$D=\log_{10}n$）

【思路 2】

观察统计的过程，每次统计的时候都是统计前 i 位确定的情况下的出现次数。于是可以想到，我们可以直接枚举每一位，一位一位确定。如果 n 的前 i 位已经确定，第 i 位为 k，假设 n 的后面每一位都是 9 的情况下各个数字的出现次数。如果出现次数恰好大于要求的次数，就可以确定第 i 位为 k 了。一位一位确定最后就可以把 n 的每一位都确定下来了。

另外，要判断一下无解的情况。

本算法算上高精度运算的时间复杂度是 O（D^2）的。可以通过所有数据。

另外，其实确定每一位的时候我们可以直接考虑所有数字出现次数的和，这样时间复杂度的常数就大大降低了。

【思路 3】

再提供一种实现比较方便的方法。我们把所有数字出现次数和加起来一起考

虑。先确定 n 的位数，不难推出$10^{i-1}\sim10^i-1$中，会出现$(9*10^{i-1}-1)*i$个

数字，利用这个就可以方便算出 n 的位数了。

接下来，我们可以计算出 n 是 D 位数里的第几个数。先把 1～D-1 位数出现过的次数减去，用剩下的出现次数除以 D 就可以得到，n 是 D 位数里的第几个数了。就可以得到 n 了。

计算出n的值还要注意n有可能是不合法的,也就是说原问题可能是无解的。可以再做一次统计,判断每一个数字的出现次数是否完全和读入的每一个数字出现次数相符。不相符则输出-1。

Solution “magicka”

By Polo

Solution A

首先由於 N<=16，非常的小，非常的和諧，又由於這是個“統計方案數”的問題，以及每個魔法元素 0 和 1 的兩種狀態，所以種種條件引導我們往狀壓 DP 上思考。

由於每個元素之間互不影響，這可以啓示我們用位運算表示使用一次魔法。

由題目容易發現，當 Ai = 0 時第 i 個元素使用的是 or 運算，Ai = 1 時第 i 個元素使用的是 xor 運算。而它們都符合結合律。

我們可以很容易地想到這樣一個狀態:設 dp[state][k]爲當前所有魔法元素的二進制狀態和爲 state，共用了 K 次魔法後有多少鍾方法數，則

$$dp[state][k] = \sum_{j=1}^{M} dp[Trans(state, b[j])][j-1]$$

其中 Trans(x, y)表示 x 和 y 兩個狀態經過轉換後的新狀態，b[j]表示第 j 個魔法所能影響的魔法元素的二進制狀態。

這個算法的時間複雜度是$O(2^N * K * M)$的，可以過 30%的數據。

實際上考場上我這麼寫可以過 40%....

Solution B

我們可以發現每次只枚舉一個新的魔法相當的慢，而且每次枚舉的本質都是相同的。

對於這種情況，一個比較常用的方法是將 K 拆成二進制，用倍增的思想算出所有 k = 2^x 的 dp[][k]值。

定義 dp2[state][k]爲當前所有魔法元素的二進制狀態爲 state，用了2^k次魔法的方法數。

則$dp2[state][k] = \sum_{j=0}^{2^N-1} dp2[Trans(state, j)][k-1] * dp2[j][k-1]$

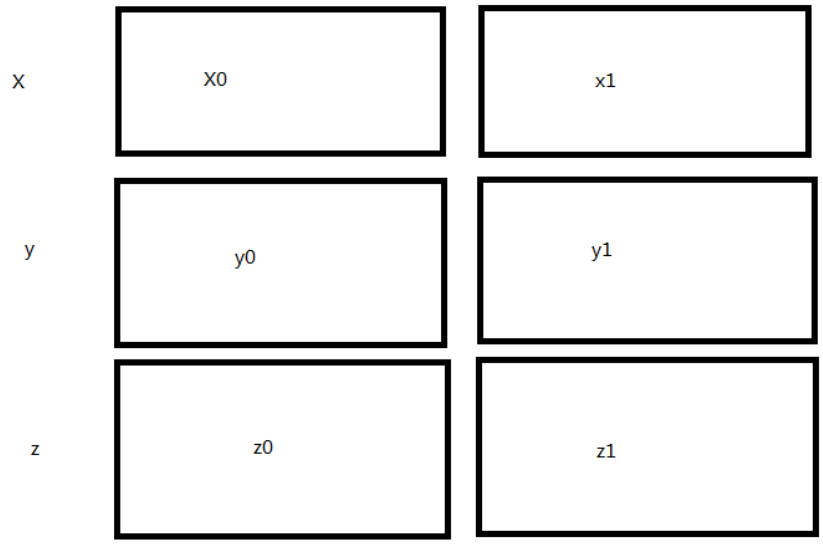
時間複雜度$O(4^N * \log K)$。

Solution C

我們仍然發現我們做了大量的重複工作….如果整體來求會不會好些呢?

定義一個函數 Solve(x, y, z)表示用 x 和 y 兩個數組來更新 z。

由於位與位之間是獨立的，我們可以按最高位的 01 性來分開做。
和諧的是，根據 01 性，我們可以把這三個數組都分成長度等長的兩部分。
分別記作 x0， x1, y0, y1, z0, z1。



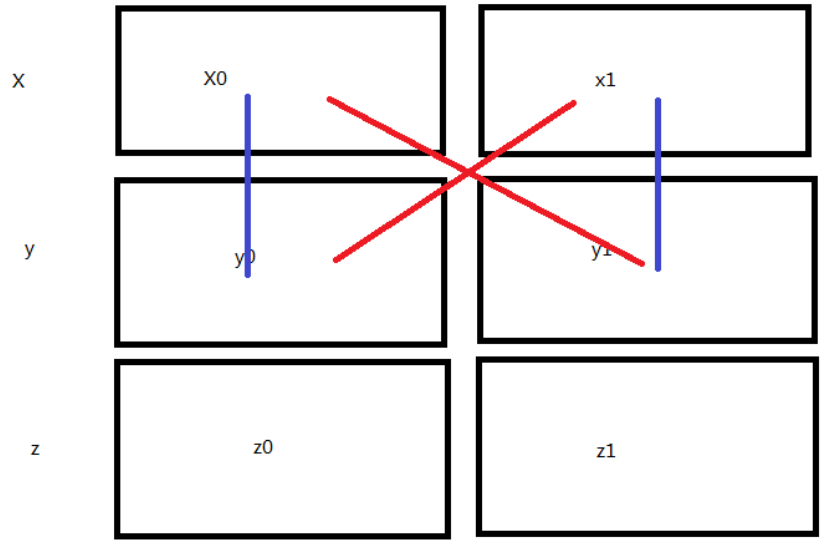
Z0 對應最高位爲 0，z1 對應最高位爲 1。
若最高位的操作爲 or，則 z0,z1 可以這樣求得：

$$Solve(x0,y0,z0)$$

$$Solve(x0,y1,z1),Solve(x1,y0,z1),Solve(x1,y1,z1)$$

這樣的遞歸式的複雜度是$f(n)=\begin{cases}1, & n=1\\ 4f\left(\frac{n}{2}\right)+O(n), & n\geq 2\end{cases}=O(n^2\log n)=O(4^N*N)$

顯然 TLE..
不過其實 z1 也可以等於 Solve(x0 + x1, y0 – y1, z1) – Solve(x0, y0, z1)
爲什麼呢….因爲這些運算都符合結合率和交換率….所以可以這麼搞
這樣我們只需調用兩次 Solve 就可以算出 z0 和 z1 了。
對於 xor，情況複雜一些。



其中藍線表示要算進 z0 中的 Solve(),紅線表示要算進 z1 中的 Solve()

我們定義 a*b=c 表示$C[x] = \sum_{i=0}^{n} A[i] * B[trans(i,x)]$,a+b=c 表示 C[x]=A[x]+B[x]，則

Z0 = x0 *y0 + x1 * y1, Z1 = x0 * y1 + x1 * y0(就是上圖)
則(x0 + x1) * (y0 + y1) = x0 * y0 + x0 * y1 + x1 * y0 + x1 * y1
(x0 – x1) *(y0 – y1) = x0 * y0 – x0 * y1 – x1 *y0 + x1 * y1
兩式相加得 2(x0 * x0 + x1 * x1) = 2z0!!!!
而 z1 = (x0 + x1) *(y0 + y1) – z0
所以對於 xor 運算我們同樣只用兩次 Solve 和加減就可以得出 z0 和 z1 了。

時間複雜度爲$O(2^N * N * \log(K))$

可以切掉
That’s it~