

---

# 算法题笔记

## 一、基本输入输出

### 1. C 语言 printf 函数格式化输入输出

`printf("%-0m.na", f);`

若 `f` 是一个双精度浮点数:

-表示左对齐, 0 表示对齐补 0, `m` 表示对齐宽度。没有-表示右对齐, 没有 0 表示补空格

`n` 表示保留 `n` 位小数

`a` 表示双精度浮点数, 还有 `%d`, `%s`, `%e`, `%f` 之类的

[https://blog.csdn.net/qq\\_36513469/article/details/80002431](https://blog.csdn.net/qq_36513469/article/details/80002431)

## 二、枚举法

枚举法不等于模拟! 枚举法不等于模拟! 枚举法不等于模拟! 重要的事情说三遍!!!

所谓的枚举法, 是所有可能的解的解空间有限, 并且有验证解的正确性的快速算法, 从而可以使用将所有的解枚举一遍, 看看哪些是正确的, 这样的方法来解题。这也叫作暴力法, 和模拟有本质上的区别。模拟是按照人脑的解题思路模拟解答的过程, 最后得出结果。

### 1. 枚举法的应用场景:

最大的特征就是解空间有限!!!!!! 答案个数的组合不太大 (比如几百个几千个, 而且可以用循环来遍历), 那选择枚举是一个好的办法! 当然, 这边的解空间并不是说就是所有答案的组合, 可能可以通过一些限制来减少解空间。比如例 2 的等式限制, 例 3 的逻辑限制 (重复涂色一个地方等于没涂, 所以最多下笔  $n*n$  次)。

### 2. 枚举法举例 1: 假硬币

POJ1013, NOI15: <http://noi.openjudge.cn/ch0201/15/>

其中解唯一, 并且告诉你了只有 A-L+重或轻一共  $2*12=24$  种解的组合, 因此采用枚举法是效率最高的, 编写 `check` 函数来检查猜测的解是否满足所有的情况, 都满足则找到了解。而模拟要考虑的情况太多了, 十分繁琐。

### 3. 枚举法举例 2: 五户共井问题

POJ7623, NOI7623: <http://noi.openjudge.cn/ch0201/7623/>

枚举法最大的特征是解空间有限对吧, 但有时候解空间会非常大啊!!! 比如上面这题, 每户的绳子有 2000 个解, 总共就是 2000 的 5 次方的解空间! 太大了!!! 但回忆一下, 其实并不需要这么多 `for` 循环来遍历所有解, 想起那个买西瓜苹果橘子的题了不, 有的时候一个或几个变量确定了, 其他的变量也就确定了, 因此不需要那么多 `for` 循环!!! 比如这题, A 确定了其实 BCDE 就确定了, 遇到除不尽就说明不能成立, 循环下一个 A 即可。根本不用遍历 2000 的 5 次方这么多的解空间。

### 4. 枚举法举例 3: 画家问题

POJ1681, NOI1815: <http://noi.openjudge.cn/ch0201/1815/>

本题的关键有两个, 第一个是看出重复涂色一个地方是没有意义的做法, 因此最多涂色  $n*n$  次。第二个关键是在看出这个性质的基础上, 思考如何考虑下笔的顺序。我们假设决定是否涂色的顺序是从左到右, 从上到下的。那么第一行肯定是没什么规律的, 你无法决定一种很好的涂色方案, 因此要选出所有的组合 ( $2^n$ ) 这么多

种（具体如何选出所有组合，见小技巧的【二进制枚举法】）。而第一行涂完了第二行就简单了，因为第一行没有被涂成黄色的地方一定需要第二行来补全，因为只有第二行可以影响到第一行的某格。因此之后的涂色策略是，看上一行的对应格是否没有被涂成黄色，若没有被涂则涂这一行的对应格（必须涂，否则不可能全涂黄，且与第二行的涂色顺序无关，反正最后的状态是所有第二行涂色的叠加，第一行全为黄色，第二行就是对应涂色的叠加，无关顺序），否则不涂。涂完最后一行检查是否所有的格子都被涂成黄色即可，是就记录涂色次数并更新最小值，否则记录 `inf`。

#### 5. 枚举法举例 4：因子问题

NOI2723: <http://noi.openjudge.cn/ch0201/2723/>

这边直接暴力测试所有的 `a` 的值是不行的，会超时，因为复杂度是  $O(m)$ ，`m` 的值可能超过  $100w$ 。注意到 `a` 为 `n` 的因数，而求 `n` 的因数的复杂度只有  $O(\sqrt{n})$ ，因为小于  $\sqrt{n}$  的因子和大于  $\sqrt{n}$  的因子是成对出现的。所以先把 `n` 的因数 `a` 算好再判断它是否满足 `m-a` 整除 `n` 即可。复杂度就变为  $O(\sqrt{n})$  了！这道题告诉我们的是，枚举法除了能够使用一些等式的限制、逻辑的限制来减少解空间，还可以用一些更高效的算法来缩小解空间（比如这题使用了  $O(\sqrt{n})$  的算法来缩小解空间）。

2020.04.26 补充：你是怎么超时的？？？穷举只要枚举 `a` 从 1 到 `n` 就可以了，最多 1,000,000 次运算啊，测出来 20ms，咋超时的？！

#### 6. 枚举法举例 5：我家的门牌号

NOI7649: <http://noi.openjudge.cn/ch0201/7649/>

其实就是简单的枚举但是.....有个问题是，有的时候数学推导会出现小数，这个时候不要在 C++ 代码里把小数部分算出来参与计算.....不然就错了，最好是不要用除法，而是将等式两边同时乘个啥系数把除法消除，然后在判断计算结果除以那个系数是不是整数（模系数等于不等于 0），如果是的话就得到了正确的结果。像这道题的代码中就乘了系数 2.....

```
x = (m+1)*m/2 - n;
if(x>=2 && x<=2*m && x%2==0)
{
    cout << x/2 << " " << m;|
    return 0;
}
```

## 三、递归和自调用函数


说是递归和自调用函数，但是做了两题感觉更像是**递推**欸，就是通过递推式来推出最后的结果，嘛，不过递推也可以用递归函数来做啦，就是效率太低，一般直接 for 循环了。递推的精髓我觉得就是一定要考虑清楚所有的情况，不重不漏!!! 也就是利用像全概率公式一样的想法，所有情况的交集为空集，并集为全集！这样的递推才不会出错。



递推或者递归最大的特点是什么呢，做了几题现在肤浅地认为，求“有几种”的这种问题一般是用递推做的好像。最关键的点是找到把大问题分解成规模更小的问题的方法。

#### 1. 递推例子 1:

NOI9273: <http://noi.openjudge.cn/ch0202/9273/>

递推分析的关键是，第一块砖头用的是啥，或者说这整个走道的前面部分是什么样的，一共有三种情况：

第一种，前面就是一块竖的砖头：，对应的后面的种类数是  $F(n-1)$

第二种，前面是  $2 \times 2$  的架构，有两种可能： 和 ，注意不能有竖的，因为这样会和第一种有重复。因此对应的后面的种类数为  $2 \times F(n-2)$

由此得出  $F(n) = F(n-1) + 2 \times F(n-2)$

然后递推即可。这道题让我想起了原来做过的一道 dp 的题目，似乎有好几种状态，比这个复杂多了。

2. 递推例子 2（模板题!!! 就是  $n$  个相同的球放入  $m$  个相同的箱子的模型!!!）

NOI666: <http://noi.openjudge.cn/ch0202/666/>

这道题是箱子里的球数不限制，因此更难一点。

如果规定每个箱子里都要有球，那就和把  $n$  这个整数划分成  $m$  个大于 0 的整数集合，有多少种划分方法一样了（为什么是集合，就是因为集合不考虑顺序，有无序性）。假设把  $n$  个球放入  $m$  个同样的桶里的放法是  $P_m(n)$ ，记住下列递推公式：

$$P_m(n) = P_{m-1}(n-1) + P_m(n-m)$$

这个递推在大二的离散 2 里推过，就是解一个解降序排列的方程的解个数问题（ $m$  个未知数，等号右边为  $n$ ）。加号左边是最小的解=1 的情况，刚好移过去就变成了  $m-1$  个未知数，等号右边为  $n-1$  的情况，所以是  $P_{m-1}(n-1)$ 。加号右边是最小解大于 1，即  $x_i \geq 2$  的情况，那就是  $x_i - 1 \geq 1$  啦，把等号左边都-1，右边就减了  $m$ ，就变成了原问题  $m$  个未知数右边等于  $n-m$  的问题了，所以是  $P_m(n-m)$ 。

这边的递推其实也是用了“全概率”的想法，最小解=1 or 最小解>1，这就是全概率。所以万变不离其宗!!!!!!

回到允许有桶里不放球的情况，那其实就是 1 个桶里放球其他不放+2 个桶里放球其他不放+3 个桶里放球其他不放.....一直到  $m$  个桶里放球其他不放，全部加起来就是了咯。就是： $\sum_{i=1}^m P_i(n)$

关于这个的初始化问题.....先 memset 一下将  $p[][]$  全部初始化为 0。之后  $P_m(n)$  的计算，初始化的值应该是， $m=1$ （即一个桶）的  $P_1(n)$  全初始化为 1，而当  $m>n$ ，即桶更多时全初始化为 0，或者只需将  $P_m(1)$  初始化为 0 即可（ $m \geq 2$ ）。之后外层循环为  $m$ ，内层循环为  $n$ ，都从 2 开始即可。

3. 二叉树问题: <http://t.cn/Ai0Ke6l0>

注意下完全二叉树如果中序遍历并逐个标号，左儿子是父亲结点序号\*2，右结点是父亲节点\*2+1 哈。

这里的递归（分治）就是：树的节点数 = 1 + 左子树节点数 + 右子树节点数

二叉树的题肯定递归啊卧槽，你这都忘了??? 而且递归也很快，因为递归的出口是  $m>n$  的时候 return 0，而  $m$  一直\*2，以指数速度增长，所以不会造成爆栈，虽然递归调用栈也是以指数增长，但总体还是  $O(n)$  的。纯数学推导是  $O(\log n)$  啦，但是没有递归写起来快 qwq

4. NOI8758: <http://noi.openjudge.cn/ch0202/8758/>

这个是上交的复试题欸.....其实思想很简单啦就递归调用得到表达式的函数，然后最终组成一个完整的表达式字符串。很坑的地方有两个：第一个是你在分解一个数的时候用的  $\log_2(x)$ ，假如你直接用 math 库里的 log 函数+换底公式（因为这个 log 函数是以  $e$  为底数的，即  $\ln$ ），然后再强制类型转换 int，会出现  $\text{int}(\log(8)/\log(2))$  等

于 2 的情况（因为浮点数的准确度问题）。还有就是为什么分解数的时候要用  $\log$  啊.....直接用轮除分解成二进制的系数不香吗.....下次遇到要用整数  $\log$  的情况就直接自己写一个  $\log$  函数好了.....第二个地方是，你在计算完某个数的分解后对分解的数进行递归调用时，如果恰好有  $2^1$  的项，即有一个指数是 1，不应该递归调用 “2(”+get\_expression(1)+ “2)” 函数，因为这样返回的会是 2(2(0))，但我们所期待的是直接返回 2。但是又有一个矛盾，因为 get\_expression(1) 的确应该返回 2(0)。所以我们选择在递归调用的时候，遇到系数为 1 的直接返回 “2”，而不是去调用自身（2020.04.27 在重做这题的时候感觉这个才是重点 x 这个弄对了就秒做.....）。然后递归出口尽量设置得简单啊我操，不要搞一个 1 就得返回什么 2(0)，2 就得返回 2，直接一个 0 就返回 0 就好，因为你可以分析出来 0 返回 0+递归调用已经可以完成任务了，没必要多此一举。

## 四、分治

### 1. 找第 k 大的数

注意不是输出前 k 大的数，如果是输出前 k 大的数那就用大小为 k 的最小化堆来做，而找第 k 大的数可以用类似快排的分治方法做。

找第 k 大的就从大到小排，快排不是先选一个  $\text{pivot} = a[\text{low}]$  吗，然后从 high 到 low 扫描，遇到比 pivot 小的就正常，遇到大的就把它放到 low 的位置，然后从 low+1 开始往 high 扫，直到 low 和 high 重合。此时左边都是比 pivot 大的，右边都是比 pivot 小的。

若此时  $\text{low-ori\_low} = \text{high} = k-1$ ，那敢情好，pivot 就是需要输出的第 k 大的数

若此时  $\text{low-ori\_low} = \text{high} < k-1$ ，那说明第 k 大的数是右边数组第  $k-1-\text{low}$  大的数，

递归调用即可，参数 k 的位置写  $k-1-\text{low}$ ，不是这个啊!!! 是  $k-1-(\text{low-ori\_low})$

参数 high 和 low 写右边的，看你是怎么实现的，如果你是用数组的话，不改变下标，那么就要写  $k-1-(\text{low-ori\_low})$ ，low-ori\_low 表示你这一轮的循环偏移了多少，若此时  $\text{low} = \text{high} > k-1$ ，那说明第 k 大的数还在左边，递归调用并把参数 high 和 low 写左边的，k 还是 k

还有记住一下判断条件全是大于或者小于，没有  $\geq$  或者  $\leq$ 。

### 2. NOI 7622: <http://noi.openjudge.cn/ch0204/7622/>

本题是使用归并排序顺带计算逆序的题（我记得我还做过一题强逆序来着，好像是 SJTUOJ 的题目，但是做法差不多）。因为归并排序本来就是通过不断地交换逆序数对来达到排序的目的。具体思路是在归并（merge）的过程中，若是看到了左边数组的第 i 个数大于右边的第 j 个数，即出现逆序，此时全局变量 cnt 要加上左边数组从第 i 个开始到左边数组结束的个数（即  $\text{mid}-i+1$ ），这是因为左边数组后面的数都比第 i 个数大了，肯定也比右边数组的第 j 个数大。这样当所有数组都 merge 好了之后，cnt 也同时算好了。

### 3. 注意一下分治（递归）的效率问题

例子：NOI2991: <http://noi.openjudge.cn/ch0204/2991/>

这道题其实是快速幂啦，你如果在每次的递归中都调用两次自身，递归深度为  $\log_2 n$ ，递归广度为 2，你最后的复杂度就是  $O(\text{pow}(2, \log_2 n)) = O(n)$  的级别了.....这显然会超时的。其实根据快速幂，你的  $\text{get\_ans}(s) * \text{get\_ans}(s)$  根本不用调用两次啊，直接调用  $\text{int tmp} = \text{get\_ans}(s)$ ,  $\text{return tmp} * \text{tmp}$  就行了.....这样就是  $\log_2 n$  的复杂度了。

- 
4. 涉及到有关统计的二分查找的问题（比如统计单词数，统计数出现的次数等），直接用 STL 的 map 来统计就好，因为它内部的数据结构是一颗平衡二叉树，查找效率是  $O(\log N)$ ，还是蛮高效的，就不要自己写二叉查找树了。

## 五、搜索

### （一）、广度优先搜索 BFS

广度优先搜索 BFS 通常是用来寻找最优解问题，其特征是解空间不大，有着每一步选择的特征，并且有的时候可以打表。其解法特征是需要构造 status 数据结构，用来记录每一步搜索时的一些状态特征（比如迷宫搜索时会记录当前的长度、横坐标、纵坐标，即  $\text{int cur\_len, x, y}$ ），并且构造以 status 类为元素的 queue。还有就是通常需要 vis 数组（除非后面的状态不可能与前面的重复，如例 2），vis 数组也可能用 set 之类的数据结构实现（如例 3）。

1. 书 P140, HUD2717: <http://acm.hdu.edu.cn/showproblem.php?pid=2717>

这题的最关键在于看出这是一个最短路的问题，且是三维的最短路（即每一步有三种选择），且每一次的花费均相等，可以用 BFS 来做。解空间似乎是  $3^n$ ，但是可以采用剪枝来把决策树剪短，从而减少层次遍历的代价，其实就是 BFS。每一次决定是否入队是看看是否有 visited 过（使用 visited 数组），然后看看下一个入队的位置是否合法  $0 \leq \text{pos} \leq \text{maxn}$  即可。注意每次出队时要设置  $\text{visited}[\text{current\_pos}] = \text{true}$ ，其实最好是在每次入队的时候就设  $\text{visited}[\text{入队的 pos}] = \text{true}!!!$

2. 书 P143, POJ1426: <http://poj.org/problem?id=1426>

这道题.....说是 100 位以内的数都可以接受，它还放到搜索这个专题里，搜索空间看起来像是  $2^{100}$  啊，这.....但是有一点好，它 n 的范围在 1 到 200，完全可以直接试试 BFS 搜索然后打表嘛 hhhhh 结果发现真的可以，可行解都在 long long 的范围内!!! 我服了。所以这题就是直接双出口的 BFS，先是 1，再是 10,11，再是 100, 101,110,111.....每一次都入队  $x*10$  和  $x*10+1$  即可。

3. 书 P145, 清华大学上机题: <http://t.cn/Ai0lUhJi>

这道题也有类似的“每一步进行决策”的感觉，每一次的决策就是这一次交换密码中的哪两个相邻位置，而且结果是和决策的顺序有关的。BFS 可以用来进行这样的决策，由于  $N \leq 13$ ，又只有 0、1、2 三个数字组成，因此所有的解空间应该没有特别大，可以进行广搜。值得注意的有两点，第一点是 visited 数组怎么去完成：由于状态改变的时候会 and 前面重复，而 visited 数组又不能单纯地用数组去完成（太大了），因此我采用的是 set（集合）数据结构来记录已经试过的密码，集合用的是红黑树结构，有  $O(\log N)$  的增查效率，因此还行，没超时。第二点是何时无解，我暂时的理解是只要有 0、1、2 并且大于 4 位都有解，毕竟交换可以交换出所有的排列.....不知道有没有错。

### （二）、深度优先搜索 DFS

深度优先搜索是使用栈结构来以深度优先原则扩展状态，通常用于寻径（判断问题是否有解）而不是找最优解。其本质可以理解为决策树的深度优先遍历。注意就是深度优先找解的时候，状态也有 visited 数组，但是注意，在递归调用 DFS 函数前把  $\text{visited}[\text{当前状态}]$  设为 true，但在搜索失败之后要重新将  $\text{visited}[\text{当前状态}]$  设为 false。再说个题外（题内？）话，做了这么久的搜索题，有两种题目很适合搜索，第一种是状态有限或者转态之间的转移关系非



常明显（比如说那个 1010 的题目），**第二种就是数据范围特别小，像什么 0 到 20 这样的数据，很可能就是用决策来深搜或者广搜，配合剪枝来做。**

1. 书 P149: Square

这题是判断给定一些木棍和其长度，判断能不能用这些木棍围成一个正方形。其思路是使用深搜来判断加入某根木棍是否可以最终到达四条边相同的状态。其最难的点在于状态的设计。何时可以返回 `true`? 旧状态如何向新状态转换? 这是很需要思考的问题。其实还是可以从决策的角度来出发想。当前状态是一根木棍（从没有 `visit` 过的木棍里遍历）：是否可以把这根木棍加入当前解? ——这里的是否就是根据递归来判断的。所以深搜一定是遍历木棍来深搜的，决定是否把这根棍子加入解中。那么什么时候停止搜索呢? 可以在状态元组中引入一个 `number` 变量，记录了当前是在凑正方形哪一条边，当 `number == 4` 的时候就可以 `return` 了，因为在凑第四条边的时候，前三条边已经凑完了，而所有木棍的长度之和又是 4 的倍数，因此最后一条边一定能凑出来。这样返回的条件就是 `number == 4` 了! 同时你需要状态元组中新加一个参数 `sum`，用来记录凑当前边时已经凑了多少长度了。而这样的搜索可能耗时比较久，有几个剪枝的策略：

- a) 开始计算所有木棍的长度和，不是 4 的倍数就输出 `no`
- b) 对所有木棍从大到小排序，最长的那根木棍超过总长度/4（即边长）就输出 `no`
- c) 非常重要的剪枝策略!!!**

在遍历木棍的时候，如果是在找正方形的同一条边上的木棍时，在递归调用函数时如果又从 0 开始遍历木棍决定放不放到解中，会产生一些无意义的计算。比如你在凑某条边时，某层调用中已经判断了第 3 根（其实就是前 3 根的意思，因为可以加入就找到解了，根本不会往后找）木棍不能加入解中，在遍历到第四根木棍时，递归调用自身，`sum` 参数设为 `sum+sticks[4]`，进入下一层递归时其实没必要再从第 1 根木棍开始 `for` 循环遍历了，因为我们已经知道加入前三根是不行了的了。因此我们在状态元组中再加入一个参数 `position`，用来表示，在凑当前边时前 `position-1` 根木棍都不要再试了，因此函数中便利木棍 `for` 循环的下标从 `position` 开始就好了。这样节省了大量的函数调用!!! 其实也正因如此才不超时的!!!!!! 没有这条剪枝策略就超时了，亲测!!!!!!

2. 书 P152, 习题 9.2: <http://t.cn/Ai0u0GUz>

这道题其实就是 `subset sum` 啊.....从一堆数中找出子集和为  $m = 40$  的物品子集有多少个，那就遍历所有子集就好了。一种办法是使用深搜，类似于二叉树的后序遍历，emmmm 好像也不能说是后序遍历，就是二叉树的遍历，第  $i$  层的左 or 右子树表示取 or 不取第  $i$  个物品加入子集。由于  $n$  个物品有  $2^n$  个子集，因此每条从根节点到叶节点的道路就是一种子集的情况。有一个剪枝的方法是，在某个中间结点中已经发现了 `sum == m` 时就可以不用再往下搜索了，因为后面的肯定不取了。因此搜索状态为二元组 `(position, sum)`，表示搜索到了第 `position` 个物品，当前的子集和是 `sum`，返回的条件为 `position > n` 或 `sum == m`。返回前判断 `sum` 是不是等于 `m`，是的话就给全局变量 `cnt` 加 1 即可。

第二种方法是使用二进制指数遍历来遍历所有的子集，但这种方法没有剪枝，所以时间比 DFS 慢了 10 倍。

**以及网上还有动态规划的解法，等学到 dp 那里再看看吧。**



dp 解法就是 0-1 背包，解数问题把 max 改成 sum 就行，只用 4ms:



3. NOI1805:碎纸机: <http://noi.openjudge.cn/ch0205/1805/>

这道题和上午做的 2 挺像，也是需要遍历出各种分割的方法。其实也可以用二进制指数遍历来做，只不过不好剪枝。而用 DFS 可以剪枝，剪枝的方法是如果当前想要试的下一个分割状态的 sum 大于了目标数，就不要调用函数了。有一点不一样的是这道题需要保存解的状态（即分割的方法，或者说 DFS 的路径）。并且还要求在有多组解时输出 rejected。我采用的方式是使用一个结构体：parts(vector<int>)来记录当前的分割方法，sum 来记录当前的和，num 来记录此解的个数。用 real\_ans 来保存 sum 最大的解，并在搜索时实时更新。最后查看它的 num 是否为 1，如果为 0 就输出 error，大于 1 就输出 rejected，等于 1 就正常输出 parts 这个数组。搜索的状态有前面的结构体、当前决定切不切的 position 以及剩余的 string。递归的返回条件是 position 超过范围。注意 position 等于输入数的长度时也是要切的，不能作为递归返回条件。好麻烦，还去复习了一下复制构造函数：（类名叫 ans）  
ans(const ans& ans2){.....}，里面写复制的代码。

**2020.05.25 更新:** ummmmm，好像二进制枚举挺好写的，而且这题数字位数不超过 6，完全可以直接枚举不会慢多少。

总结一下：深搜的状态就是递归的参数，而广搜的状态是一个 status 结构体!!!

### （三）、刷题

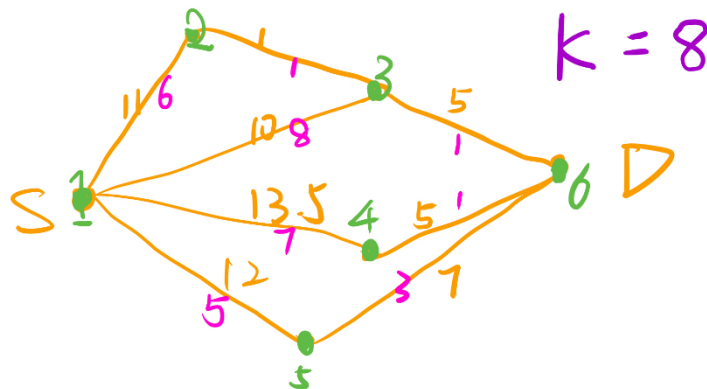
1. NOI6044:鸣人和佐助: <http://noi.openjudge.cn/ch0205/6044/> （变种广搜）

说个题外话：广搜一定要在 push 进队列之后就马上把 visited 数组设为 true 啊!!!  
不要等取出来之后在设为 true 啊!!! 会出问题的!!!!!!

这道题其实就是二维最短路，即有限制的最短路（查克拉不能为负数）。相对于一般的广搜加了限制，所以应该考虑到这个限制。考虑一下原来单纯的广搜，再只是单纯地加上一个查克拉不为负数的限制的情况：你的广搜是按照一定的顺序（比如上→下→左→右）进行的，你的确可以保证到达某点时所走的路是最短的，但可能这是因为这条路用了很多查克拉，而你把这个点的 `visited` 值直接设为 `true` 的话，说不定有相同长度查克拉却用得更少的路本可以经过此点，但现在因为 `visited` 被设为 `true` 就不行了，这样可能会带来潜在的错误，比如说最后到不了终点，而本来是可以到的。解决方法有两种：

一是：`visited` 数组变成三维的，`visited[行][列][剩余查克拉]`，表示剩余查克拉为某值时有没有走过这个点。这样的话就不会有使用过多查克拉的路占用某点的问题了，因为使用不同查克拉到达同一点的 `visited` 数组是分开的。这是最简单的方法，效率可能低一点点，但绝对正确。

二是使用优先级队列：既然前面分析过了可能存在使用更少查克拉到某点的情况，那我们不妨直接用优先级队列，当步数相同时所剩查克拉多的路线被试探。这样就根本不存在使用查克拉多的路占用某点的问题了。但是不知道为什么代码一直没 AC，一直 4/6 分然后 WA..... 似乎是这个算法本来就错了..... 因为你是以距离为主进行优先级队列的排序的，只有当距离一样的时候才会去比较 `cost`，然而会出现以下的情况，仍然导致 `vis` 数组被错误的路径占用：比如有一条路它虽然 `length` 是最短的，但是 `cost` 比较大，会导致之后无法经过这个点到达终点，但本算法仍然把它的 `vis` 设成 `true` 了。比较极端的例子就是，到达这个点的 `cost` 刚好等于 `max_cost` 了，但距离的确是当前最短的，于是这个点的 `vis` 被设成 `true`，但最佳路径本应该是通过另一条 `cost` 较小但 `length` 较长的，经过该点的路径的，但 `vis` 被设成 `true` 了导致这条路径找不到了。如下图：K 是 `max_cost`，紫色是边的 `cost`，蓝色是 `length`



2. CCF201403-4 无线网络: <http://118.190.20.162/view.page?gpid=T7>

这道题呢，也是一个二维最短路问题哈。可以用 SPFA，但我们现在还没复习到图论，所以先不用。这个的距离度量是相同的，所以可以用广搜哈。那么二维最短路就是记录每一个路由器有没有被访问到的同时还要再开一个维度，保存这个路由器经过某数量的增设路由器有没有被访问过。即这个数组是这样定义的：

```
bool vis[num_of_routers][num_of_extra_routers]
```



这样我们就不会出现某个点被错误的路径占用的情况了。值得注意的是这边用的是邻接表的数据结构来保存的邻居，即 `vector<int> neighbours[m+n]`，`neighbour[i]`表示序号为 `i` 的路由器的邻居们，是个 `vector`。

3. NOI1789:算 24: <http://noi.openjudge.cn/ch0205/1789/>

这道题思路其实非常明确，就是对于 4 个数去遍历所有的计算可能。计算流程为，首先在四个数中取出两个数，这两个数进行一共 6 种运算( $x1+x2$ ,  $x1-x2$ ,  $x2-x1$ ,  $x1*x2$ ,  $x1/x2$ ,  $x2/x1$ )，得到的数与剩下的 2 数得到 3 个数。这 3 个数再取两个进行运算，得到两个数，两个数再运算得到一个数，与 24 比较即可。因此我们很容易想到用递归来做，数的个数为 1 时停止递归，判断是否为 24。否则取两数进行运算然后把结果再放入函数中递归。

主要是有几点很恶心的点需要注意下：

- a) 除 0 问题：当中间结果有 0 的时候要注意判断除数是否为 0，因为 `double` 的除 0 是不会自动报错的！
- b) `double` 的计算问题：因为是 `double`，所以相等要用 `fabs` 与一个小数(0.0001 或者  $1e6$  这样)进行比较。而且中间计算值也全部得是 `double`.....不要手贱习惯性地写成 `int` 了
- c) 如何选两个数的问题：我发现我是真的傻逼，还用下标数组去记录选的是哪两个.....真是被二进制指数遍历法给荼毒了 QAQ。其实和找逆序的暴力方法一样啊，只要两个 `for` 循环就可以找出所有两个的下标组合了.....
- d) 效率问题：使用 `vector` 效率很低，但还是有优化方法的。比如我们在选定了两个数后，剩下的数是不变的，在本次的计算中仅仅是算出的结果不同，因此可以共用一个 `vector`，不要每次调用函数之前都创建一个新的 `vector` 然后 `push` 全部的数。剩下的数先 `push` 进 `vector` 里，然后每次计算后再把计算的那个数更新一下（`vector` 里最后一个数更新），而不是创建新的 `vector` 再全部 `push`。而且其实用数组会更好一点！

最后就是，其实这题是可以直接打表的.....因为它是 4 个 1~9 的整数，所以其实数据范围不超过 9999。把 0-10000 的计算结果全部打印出来得到一个 10000 长度的数组 `ans`，然后直接根据数组值输出结果即可。需要注意的是不能只打印含有 1~9 数字的组合，因为中间会有缺失数导致下标无法寻找。比如 1119 之后是 1121，就没法按照输入值直接根据下标找到答案了。所以还是需要把 0-10000 的所有答案都打印出来。

4. NOI1998:寻找 Nemo: <http://noi.openjudge.cn/ch0205/1998/>

这道题其实就是 BFS+优先队列的问题.....虽然它不是二维最短路，但是它其实比二维最短路还简单。回忆一下二维最短路，我们的优先原则是在走过的路径长度相同时，第二维的消耗（或者说长度）越小越好。而这道题的优先原则只有一个，就是能不使用门就不使用门，因此使用门的个数小的状态优先扩展。这个使用一个优先级队列就可以了，使用门数小的 `status` 排在队首。最终找到了 Nemo 的时候出队的门数一定是最少滴。比较难的其实是建图啦，我用的是建立以每个 `room` 为元素的二维数组。每个 `room` 有上下左右四个参数，表示上下左右是墙壁(0)、门(1)还是空气(2)。根据输入的墙和门的坐标、类型、长度来更新各个 `room` 的状态。

有一个超级坑的点啊 woc，就是 Nemo 居然不一定在迷宫里？？？然后 Nemo 的坐标可能会比 199 要大.....这样搞数据不是有病吗？？？所以判断出 Nemo 不在迷宫里的时候直接输出 0，不然疯狂 WA 到死.....

5. NOI6266:取石子游戏: <http://noi.openjudge.cn/ch0205/6266/>

这道题其实引出了一个很有趣的问题：取石子游戏的博弈论

这道题不是经典的三种博弈之一，题设是每次只能在多的那堆石子中取少的那堆石子的整数倍。但是题目的提示已经把必胜状态告诉你了：两堆石子为  $a$  和  $b$ ,  $a > b$ , 必胜状态为  $a \geq 2 * b$ 。而当  $a < 2 * b$  时只有一种取法。所以很容易想到递归，当  $a \geq 2 * b$  时直接把 `flag` 设为 `true`（表示先手必赢），并退出递归程序。而当  $a < 2 * b$  的时候，只有一种取法，递归进入下一个状态即可。

有两个需要注意的地方，一个是最好使用 `long long`，怕越界。第二个是递归的出口不能只是  $a \geq 2 * b$ ，因为只有这样的话当  $a = b$  这种情况先手也应该赢，而没有体现出来。因此递归出口应该是  $a \geq 2 * b \parallel a \% b == 0$  这样。

### 补充：三种取石子的博弈：

- a) 巴什博弈(Bash Game)，有 1 堆含  $n$  个石子，两个人轮流从这堆物品中取物，规定每次至少取 1 个，最多取  $m$  个。取走最后石子的人获胜。

判断方法为，看现在的石子是否有  $n \% (m+1) == 0$ ，若有，则先手必败，反之先手必胜。后手获胜的方法为， $n \% (m+1) == r$ ，拿  $r$  个石子把这个状态留给先手。因为后手之人若按这样的方法拿，到最后轮到先手之人的时候，必然剩  $m+1$  个石子，此时先拿的人无论拿多少个，后拿的均可以拿完（获胜）。

- b) 尼姆博弈(Nimm Game)，有  $k$  堆各  $n$  个石子，两个人轮流从某一堆取任意多的物品，规定每次至少取一个，多者不限。取走最后石子的人获胜。

这个有点玄学，不过结论还挺好记的。结论是当前每堆石子的个数转为二进制，按位异或之后若等于 0 则先手必败。反之先手必胜。获胜的方法为在某一堆（通常是最大的那堆吧？）里取某些石子，使其取完之后所有个数二进制按位异或为 0。通常就把剩下堆按位异或一下得  $a$ ，然后在某堆中取使得剩下  $a$  个。

- c) 威佐夫博弈(Wythoff Game)，有 2 堆各若干个石子，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取 1 个，多者不限。取走最后石子的人获胜。POJ1067

这个就更玄学了，它的奇异状态（先手必败状态）是：(a, b)，当且仅当  $a$  和  $b$

满足  $(a, b) == (a_k, b_k)$ ，其中  $a_k$  是从零开始的  $\frac{1+\sqrt{5}}{2}$  的倍数（向下取整）。而

$b_k = a_k + k$ 。至于如何从非奇异态转化到奇异态.....太复杂了懒得记。

POJ1067 的代码见同目录下的 `stone3.cpp`

## 六、字符串

### 1. 字符串匹配：(KMP 算法)

就是模板题好吧，背代码就好了.....

几个注意点：（我背的这版代码，数据结构书上的）

- a) `get_nxt` 的过程中，先把 `nxt[0]` 设为 -1。
- b) 下标  $i=1$ ;  $i <$  子串长度;  $i++$ , `nxt[i]` 表示的是  $i$  这个位置的最长重合前后缀的前缀的最后一个字符的下标，在后面用的时候， $j$  这个位置没有匹配上，要重新从 `nxt[j-1]+1` 这个位置匹配，而不是 `nxt[j]`（所以这个数组是不是不要叫 `nxt` 比较好.....）
- c) 预处理到第  $i$  个位置时，先把  $j$  设置为  $i-1$ ，再  
`while(j>0 && pattern[nxt[j]+1] != text[i]) j = nxt[j];`  
然后判断  $j$  是否小于 0，若小于 0 则 `nxt[i] = -1;`

- 
- 否则 `nxt[i] = nxt[j] + 1;` //符合刚刚我们说的 `nxt[i]` 的定义
- d) 最后的判断, 先设 `i=0; j=0;` 然后 `while(i<textlength && j<patternlength)`  
若 `text[i] == pattern[j]` 就两者都+1  
否则若 `j=0;` 表示要从头匹配了, 直接 `i++;`  
再否则就刚刚说的从 `nxt[j-1]+1` 的位置匹配, 即 `j = nxt[j-1]+1`。  
循环退出后 `if(j==m)` 就 `return i-j;` 即第一次匹配的下标 (+1 就是位置)
2. KMP 算法在多次匹配场景下的应用: (即一个 `text` 不是返回第一次匹配成功的位置, 而是返回匹配的个数, 且匹配串在 `text` 中可能有重叠)
- 此时修改代码, 在匹配代码的外层加一个 `while(i<n)`, 然后里面判断语句:  
`if(j == m)` 时, 不要 `return`, 而是 `cnt++; j = nxt(j-1)+1;` 这样相当于考虑了重叠的那部分, 因为匹配成功之后不是重新开始一轮匹配, 而是从匹配子串有可能重叠的地方开始新一轮的匹配。若题目有说不会重叠, 则可以设 `j` 为 0。  
具体见 HDU1686。

## 七、线性数据结构

1. 队列:
  - a) 循环队列可以用普通队列来模拟, 每次移动队列就把头部的 `pop` 出来然后 `push` 到尾部。可以用于解决约瑟夫环问题。
2. 栈:
  - a) 括号匹配: 括号匹配问题, 遇到左括号则进栈, 遇到右括号则看看栈里有没有左括号, 若有则 `pop` 出来并匹配。若无则右括号匹配失败。最终栈里的左括号都是匹配失败的。
  - b) 计算表达式: 运用两个栈, 一个运算符栈一个运算数栈。
    - i. 读到乘方、左括号直接入栈, 读到数字直接入栈, 乘方是右结合的, 所以必须入栈。
    - ii. 读到乘号除号, 前面的直到遇到左括号的乘方和乘除操作可以做, 取运算数两个 (先取的为右运算数) 做运算即可
    - iii. 读到加号减号, 前面直到遇到左括号的所有的运算都可以做, 按栈的顺序不断地取出、运算、结果放回.....即可
    - iv. 读到右括号, 一直到读到左括号为止所有的操作都可以做 (按栈的顺序)
    - v. 最后读完之后若运算符栈不为空, 要依次运算把它清空。

## 八、数学问题

### (一)、进制转换

1. 十进制二进制转换:
  - a) 十进制转二进制:  
`while(deci):` 把 `dec%2` 放入 `vector`, `dec /= 2`。之后 `vector` 里倒序输出就是二进制的值。
  - b) 二进制转十进制:  
`rtv = 0;`  
`for(最高位到最低位):` `rtv += 当前位的数字; rtv *= 2;`

最后 `rtv` 要再除以 2（因为最后多乘了个 2）。

注意这种题目中可能会出现 `deci` 是个很大的数的情况，要用 `string` 来做。可以不用高精度，但要自己写字符串高精度整数的乘法（和一个小数据相乘，本题为 2）、除法（和一个小数据相除，本题为 2），和加法（两个字符串相加）。

注意几个易错点（本来打算写在易错点里面的，但是想想还是直接写到这里好了，到时候再复制过去得了）：

**第一，**在写两个字符串的加法（高精度加法）时，先不是要得出两个字符串哪个大嘛，然后在小的那个前面补 0。这个时候先赋两个变量（我们先假设 `s1` 比 `s2` 更长），`int cha = s1.size() - s2.size();` 然后 `int len = s1.size();` 再循环 `cha` 次给 `s2` 补 0.....不要在循环时写 `i <= s1.size() - s2.size()`，因为这时候 `s2.size()` 是在一直增加的.....这个错误犯了两次了不要再犯了啊。

**第二，**在写乘法时，字符变数字是 `c - '0'`，而数字变字符是 `char(x + '0')`，要强制类型转换哈，不然还是 `int`。

**第三，**在除法中最后除去头部的 0 时，是 `if(rtv[i] == '0')` 而不是 `if(rtv[i] == 0)!!!!`

## 2. 任意进制（m）转任意进制（n）

就先把 `m` 进制转为 10 进制，再把 10 进制转为 `n` 进制。转换方法和 10 转 2 及 2 转 10 是一样的，不同的是转 `n` 进制时模和除的是 `n` 而不是 2，`m` 转 10 的时候乘的是 `m` 而不是 2，最后也是除 `m`。还有就是任意进制可能有用字母表示的，因此一般不超过 36 进制（即 A 表 10，Z 表 35）。写字母和 `int` 互相转换的函数然后调用可以快速解决这个问题。然后刚刚说的可能转 10 进制的时候很大，需要写高精度的操作也是一样，不同的是乘法可能要稍微改一下，就乘两位数变成：

（乘个位）+（乘 10 位补一个 0）这样。

10 进制转任意进制要注意要自己写模函数，思路是取出被模数的最后 8 位（如果小于 8 位就保持原数），然后用自己写的 `str2int` 转化为 `int` 之后再取模。

## （二）、GCD, LCM

### 1. 最大公因数：

函数参数 `a` 和 `b`，`while(a % b != 0)` `a` 赋值为 `b`，`b` 赋值为 `a % b`（即 `a` 除 `b` 的余数）最后返回 `b` 即可。

### 2. 最小公倍数：`a * b / gcd(a, b)` 即是最小公倍数

## （三）、质数相关

### 1. 质数判定：

小于 2 的肯定不是素数

从 `for i: 2 ≤ i ≤ bound` 注意是小于等于，看看 `i` 能不能整除 `n`，若能则返回 `false` 注意先算出 `bound = sqrt(n)`，防止 `sqrt` 反复调用

### 2. 素数筛法：

用一个 `bool` 数组储存是否为素数初始化为全 `true`，先把 0、1 设为非素数，然后从 2 开始遍历，若为 `false`，即 `i` 不是素数，则 `continue`；若是素数，那么在 `prime` 这个 `vector` 中 `push_back(i)`，并且从 `i * i` 开始直到 `bound` 把 `i` 的倍数设成 `false`（每次循环 `+i` 即为 `i` 的倍数，不要做乘法）。

这样即筛出了小于 `bound` 的全部素数。

为什么是从  $i*i$  开始呢，因为  $i*2, i*3 \dots i*(i-1)$  都被 2、3、5... 之类的素数更新过了，用再去判断一遍了。

### 3. 分解质因数：

给出  $n$  要求  $n$  的质因数，只要先筛出  $\sqrt{n}$  以内的素数，遍历这些素数，不断试除（即只要  $n$  能被当前数整除就一直除它，并更新  $n$  为  $n/\text{factor}$ ），试除几次则这个因子的指数为多少。最后判断若  $n > 1$  则还有一个大于  $\sqrt{n}$  的素数因子（就是当前的  $n$ ），其指数必为 1。为什么呢，是因为  $n$  只有可能有一个大于  $\sqrt{n}$  的素因子。

**例题 1、整除问题：** <http://t.cn/Aip7eHBD>

这道题还是蛮难的：给定  $n, a$  求最大的  $k$ ，使  $n!$  可以被  $a^k$  整除但不能被  $a^{(k+1)}$  整除。其中  $n$  和  $a$  都在 2 到 1000 的范围内。

$n!$  是个极大的数，显然不能暴力求解。那如何判断  $n!$  能不能被  $a^k$  整除呢？这就涉及到质因数分解的问题了：

**两个数  $a$  和  $b$ ，若  $a|b$ ，即  $a$  能整除  $b$ ，那么  $a$  的质因数分解必定是  $b$  的一个子集。**

即， $b$  的质因数都能在  $a$  的质因数中找到，并且  $b$  的系数都小于等于  $a$  的相应系数。所以这道题只要求出  $n!$  和  $a$  的质因数分解（用 map 保存），然后即可按  $k$  的递增来判断是否满足  $a^k$  可整除  $n$ 。对于某个  $k$ ， $a^k$  的质因数其实就是  $a$  的质因数系数都乘  $k$ 。找到第一个不满足的  $k$ ， $k-1$  即是所求。

## （四）、快速幂和矩阵快速幂

### 1. 快速幂，通常是和 mod 结合起来的，背一个模板好了

int fastExp(int a, int b, int mod)

```
{
    int ans = 1;
    while(b!=0)
    {
        if(b % 2 == 1)
        {
            ans *= a;
            ans %= mod;
        }
        b /= 2;
        a *= a;
        //上面这句如果用 int 有可能会越界，可以用 ull 也可以写成
        //a = (a%mod)*(a%mod);
        a %= mod;
    }
    return ans;
}
```

**精简版：**

int fastExp(int a, int b, int mod)

```
{
    int ans = 1;
    while(b)
```



```

    {
        if(b % 2 == 1) ans = (ans*a) % mod;
        b /= 2;
        a = (a%mod)*(a%mod) % mod;
    }
    return ans;
}

```

例 1、root(N, k): <http://t.cn/AipAw4B1>

这道题好难 QAQ，一通数学分析分析出最后的结果是  $Nr$  的话，那么有：

$Nr = N \pmod{k-1}$ ，所以只要快速幂求  $x^y \pmod{k-1}$  的值就好了，注意为 0 的时候返回的是  $k-1$  而不是 0。啊啊啊啊啊啊这个真的难，这样的数学分析我在考场上估计是想不出来的……

## 2. 矩阵快速幂：

和数的快速幂一样啦，不一样的地方就是 `ans` 最开始是一个单位阵，其他都一样啦。最好写一个矩阵类，然后重载一下乘法，会更方便的啦！

## 3. 高精度整数：

主要是注意以下几点易错点吧：

- 构造函数中记得 `memset` 把 `digit` 都设为 0，然后 `length` 不要忘记设为 0
- 注意最好要写一下复制构造函数，因为不写的话默认传指针可能会出问题，如果是用过的数之后不会再用了好像没关系，但是为了不改变原数而赋值到一个 `tmp` 上这种情况就不得不写了。emmmmm 好像实验出来这种静态的数组是会深拷贝的？？？似乎不用写了？？？
- 减法函数里面记得如果当前算出的 `current` 不是小于 0，`carry` 要设为 0。
- 加法也要注意 `carry` 每次都要设为 `current/10`
- 加减法都要注意循环是 `for(int i=0; i<length || i<b.length; i++)`
- 重载小于等于的时候记得是从 `i=length-1` 开始往下比（从高位开始）
- 除法记得最开始要初始化 `remain` 为 0，然后在每一次处理的时候如果此时上一次处理的 `remain` 为 0（即 `remain.length==1 && remain.digit[0]==0`）就不要进行移位操作，直接把 `remain` 的最低位设为当前处理位。
- 模操作的代码直接 `copy` 除法就是了，不用 `answer` 变量然后返回的是 `remain`。当然，如果只需要其中的一种操作（比如加法、减法，或者是高精度\*整数，高精度/整数，高精度%整数）这样的，也可以直接写字符串啦。

还有就是，在 `codeblocks` 里调试的时候，如果开的 `digit` 数组太大，很可能会崩掉，不是因为代码写错而是内存不能开那么大。所以在调试的时候可以把 `digit` 数组的大小开小一点，等提交之前再按照给定的规模去设置 `digit` 数组的大小。

# 九、非线性数据结构

## 1. 二叉树：

注意一下二叉树的几个性质： $n_0 = n_2 + 1$

完全二叉树： $\log_2(n) + 1$ ， $2k$  和  $2k+1$

- 书例 10.1：特殊的前序建树：<http://t.cn/AiKuUtlX>

**注意一下!!!**

①、就是树节点这个 `class`，不是有左儿子右儿子两个指针嘛。注意一下在开辟一个新的儿子节点的时候，一定要 `new` 啊!!! 不然不会分配储存空间的。

②、不是说只知道前序就不能建树哈，如果前序约定好了一个特殊的字符，如 '@' 或者 '#' 啥的当做空树标志，即保证每一个非叶子结点的度都为 2，那么只知道前序遍历（或者中序、后序）是可以建出这棵树的（即返回这棵树的根节点，根节点已经连好各子树），具体的函数也写成递归的，当输入的是特殊符号时返回 `NULL`，否则创建结点，`data` 为读到的符号，左右子树为递归调用返回的节点。

b) 书例 10.2：已知前中遍历序列建树：<http://t.cn/AiKgDfLU>

emmmmmm，我记得当年学数据结构的时候写了老半天，似乎是因为不能用 `STL` ??? 然后发现用 `string` 的话快的一批，也是写成递归函数，输入是 `pre` 和 `post`，如果 `pre` 和 `post` 长度为 0 那就返回 `NULL`，否则创建一个节点，值为 `pre[0]`，然后分别切出左右子树的 `pre` 和 `post` 序列，这个节点的左孩子和右孩子是递归调用的返回，即可。（不知道当年为什么写了那么久.....破案了，似乎是因为老师规定了接口，而且要实现模板类，所以很难搞）

噢噢，还有一个就是 `string` 的 `substr` 函数，如果取的字串超过原串，会直接取到末尾，不会报错哈。

## 2. 二叉查找树：

二叉查找树也叫二叉排序树，或者叫二叉搜索树。其特点是非空树的根节点左边的各节点值都小于根节点，右边的各节点值都大于根节点（不能有相等的）。

注意一个点！建树的时候，如果采用的无返回值的 `void` 函数，必须要把参数传成引用传递，即 `treeNode* &t`，这样才能真正改变 `t` 的值。

## 3. 优先级队列：

还是注意一下 `STL` 里优先级队列需要你重载小于号，默认是最大化堆。所以如果你想用最小化堆的话，在重载小于号的时候写大于号的逻辑即可。换句话说，在重载的时候如果你希望是越小越好，就写 `return` 自己大于对方，如果你希望越大越好，就写 `return` 自己小于对方。

以及如果不是重载，是用本来的数据类型，比如说 `int, double` 这样，想用最小化堆：

```
priority_queue<double, vector<double>, greater<double>> > q;
```

要这样写，中间那个是定义底层容器，由于要写第三个参数所以第二个必写。

a) 哈弗曼树权重计算：<http://t.cn/AiCuGMki>

Emmmmm，这道题主要是告诉我们哈夫曼树只要算权重的话，可以直接简单地用一个优先级队列维护，每次出队维护时，把最终权重值 `ans` 加上当前最小次小的两个元素的和，再把此和放回队列。这样实际上第 `i` 层的那些值就加了 `i` 次，即乘以了结点的路径长度权重 `i`。

4. 散列表 (`map`)，见《`STL` 笔记》。值得一提的是，这边的 `map` 底层采用的是红黑树实现的，而真正的散列表为 `unordered_map`，至于这两个数据结构孰优孰劣，还是难以下一个定论的。

# 十、贪心算法

## 1. 区间贪心：

a) 最多分离区间数：书例题 7.4：

贪心的策略是按照节目结束时间排序，优先选择节目结束时间早的节目。这是因为在给定的时刻，剩下的时间是一定的。这个时候选择的第一个节目一定是当前可选节目中结束时间最早的那个，若选择结束时间迟的，可能导致后面有节目不能选，而选择一个都是给最终答案+1，既然要求最大化最终答案，不如选择对后面更有利的结束时间最早的。

b) 区间连接问题：（小岛搭桥）：书例题 7.5:

贪心的方法是，先计算出各个 interval 可以使用的桥的区间[minlen, maxlen]，然后按照桥从短到长去选择满足哪个区间。优先选择 maxlen 小的区间。这是因为我们是按桥从短到长选的区间，所以先满足了 maxlen 小的区间，之后更长的桥可以去满足 maxlen 更长的区间，而如果先去满足了 maxlen 大的，可能之后 maxlen 小的区间就无法被满足了。

2. 刷题:

a) To fill or not to fill:

<https://www.nowcoder.com/practice/f7eba38f7cd24c45982831e0f38518f9?tpId=63&tid=29602&tpage=2&ru=%2Fkaoyan%2Ftest%2F9001&qru=%2Fta%2Fzju-kaoyan%2Fquestion-ranking>

这到题问的是给定一个目的地，出发点到目的地之间的线段上有一些加油站，每个加油站的油价不同。给定车的油箱容量、每单位油能走的路，要求出从出发点到目的地的最低加油价格。

贪心的思路大概如下：状态（在某一个加油站，并且油箱里有油 x 单位）。对于当前状态，需要决定的是在这个加油站加多少油？下一个到达的加油站应该是哪个？→按加油站的距离升序考虑：

- i. 若该加油站开始，往后最大行驶距离（即满油行驶距离）中：①、有加油站的价格比当前便宜，则当前加油站只需要从当前油量把油加到，能够行驶到第一个比当前加油站便宜的加油站/所需要的油量即可。②、都比当前加油站油价贵，那么只需要从当前油量把油加满即可，然后行驶到的下一个加油站应该是最大行驶距离内，油价最便宜的那一个加油站。③、若没有加油站了，则判断当前最大行驶距离是否能达到目的地。
- ii. 重复 i 过程，累加 cost，即可得到最终的答案。

我采用了一个分割区间的方法，即以各个加油站位置，加油站位置+满油行驶距离为本加油站可以覆盖的区域，以区域的端点划分线段为好多多个小区间，每个小区间都有一些加油站可以覆盖到。每个小区间都选取这个区间覆盖到的油价最低的那个即可。若发现没有油价（即没有覆盖到），则该区间的左端点就是最大行驶距离。

b) NOI 1799:最短前缀: <http://noi.openjudge.cn/ch0406/1799>

草，这道题想法非常简单啊，就是对于某一个字符串，从长度为 1 开始遍历它长度为 i 的前缀，然后检查这个前缀是否是唯一的或者就是它本身（精准匹配），若是，则这个就是它的最短前缀了。因为不能再短（有二义性）也不能再长（违反最短原则）。但是注意一下：查找某一字符串 pre 是否是另一个字符串 s 的前缀，不是 `s.find(pre) != string::npos`，而是 `s.find(pre) == 0`，如果这个是 true 则表示是前缀啊啊啊啊啊！坑死人了!!!!

c) NOI 2469:电池的寿命: <http://noi.openjudge.cn/ch0406/2469/>

草，真就想题 1 小时，代码 5 分钟呗，我佛了。这道题其实想明白就很简单啊，就是 Interval Partitioning（区间分割）的一个变种，不一样的地方在于它

---

可以把某个区间再切成小块放到另一个分区中,限制是不能和本分区的剩下的这个区间有重合。因此一同分析之后得出结论,只有当时间最长的那个电池的寿命大于剩余所有电池寿命之和的时候才不能完全利用电池,这时最长使用时间为剩余电池寿命之和。否则就是所有电池寿命和除以 2。啊啊啊啊啊,就这么简单,居然想了一个小时 15 分钟.....

...第二次想了半个小时 QAQ

- d) NOI 712: Magic of David Copperfield: <http://noi.openjudge.cn/ch0406/712/>

他妈的,又是一道想一小时代码五分钟的题目(而且我还煞笔到并没有写出 5 分钟的代码而是写了 2 小时的代码.....)这道题其实最关键的观察是,国际象棋棋盘中,走奇数步必到不同颜色的格子,偶数步必到相同颜色的格子。而大卫科波菲尔的这个魔术,实际上就是利用了这一点,理论上每次都让观众走奇数步,然后删掉不一样颜色的那个格子就好了。但是要考虑到格子的连通性的问题,所以不能瞎删。我这个傻逼想的删除方法是每次去判断图是否连通然后删除尽可能多的点,但是这样根本没意义啊我操,它题目只是让你输出一种删除的方法让最后剩一个,并没有说最少删除多少次啊????所以这道题和贪心有什么卵关系吗???!迷惑。因此最简单的方法就是每次走奇数步,然后删除左上角的那条对角线的元素,最后留下的就是右下角那个格子.....就这么简单暴力。

- e) NOI2986: 拼点游戏: <http://noi.openjudge.cn/ch0406/2986/>

我 tm.....看了题解都不太懂为什么。真的干脆索性就记答案好了!就已知我方  $n$  张卡牌的战斗,敌方  $n$  张卡牌的战斗,要求最大化我方得分(胜利得分最多,平局得分第二多,失败得分最少)。不能重复使用。贪心的策略如下:首先按照牌的大小从小到大排好序。

- i. 若我方当前最大卡 > 敌方当前最大卡, 则比较, 否则 ii
- ii. 若我方当前最小卡 > 敌方当前最小卡, 则比较, 否则 iii
- iii. 若我方当前最小卡 = 敌方当前最大卡, 则比较, 否则 iv
- iv. 将我方当前最小卡与敌方当前最大卡比较。

- f) NOI8469: 特殊密码锁: <http://noi.openjudge.cn/ch0406/8469/>

这道题感觉和贪心没啥太大关系啊,很像枚举法那个专题里面做过的一道涂色的题目,那道题是二维的就是了,是涂了一个地方上下左右都会翻转颜色。这道题是一维的。有两个观察很重要:一个是重复按一个地方两次等于没按,第二个是按按钮的顺序无关紧要,我们关心的只是哪些地方被按了。因此只需要判断第一个位置翻转 or 不翻转两种情况即可。第一个位置确定了,按顺序看后面的,若前一个位置和目标不一样,那就要翻转当前位置,否则当前位置不翻转。这样可以确定所有的地方有没有翻转。最后看如果最后一个位置不一样,则 impossible,否则就是当前翻转次数。两种情况取最小的那个即是答案。

- g) NOI2384: Cell Phone Network: <http://noi.openjudge.cn/ch0406/2384/>

emmmm, 这道题其实是求连通图的最小支配集(别问什么是最小支配集,问就是你忘了,忘了就去查)。似乎好多人都是用树状 dp 做的(因为还没有复习到的 dp 所以我也不知道什么是树状 dp)。贪心策略感觉好难想啊,而且看了题解也没有太理解,不过既然是经典的题目那就记下结论好了:

①、首先,选取一个结点为根节点进行 DFS,构建出图的一个生成树序列,并给每个节点打上一个 DFS 遍历编号(用 newPos 数组实现)。这边要注意的是,在 DFS 之前要先把 vis[1] = true,不然会重复遍历到 1(根节点)。

②、接着，按照 DFS 遍历编号逆序处理每一个节点：若此节点没有被支配集的点 cover，那么进行如下处理：a. 若此结点的父节点不在支配集中，将父节点加入支配集，ans++。b. 将此节点、此节点的父节点、此节点的爷爷节点对应的 cover 数组地方都设成 true。

③、输出 ans 即可。

虽然不知道为什么要这样.....但是至少记下了结论了，下次遇到最小支配集的题目也就会做了。

还有就是关于邻接表和链式前向星的，看“小技巧”笔记。

h) NOI1768:最大子矩阵: <http://noi.openjudge.cn/ch0406/1768/>

怎么感觉做的题越来越不像贪心了??? 这道题主要是前缀和+最大子段和结合的题解。最大子段和用的是 dp 来解。前缀和是用于算每一列纵向压缩的和，这样在遍历  $n(n-1)/2$  个纵向段的时候可以避免把计算复杂度升到  $O(n^4)$ ，而是  $O(n^3)$ 。对于每一次遍历，用 dp 算这次遍历的一维最大子段和即可。总体的复杂度是  $O(n^3)$ 。

当然由于这道题数据很弱，直接前缀和+枚举也是可以过的。

具体题解可以参考 [https://blog.csdn.net/da\\_kao\\_la/article/details/80755613](https://blog.csdn.net/da_kao_la/article/details/80755613)。

值得注意的是一维的最大子段和的 dp 状态是，必须包含 i 的前向子段最大值为  $C[i]$ ，那么  $C[i+1] = \max(C[i] + a_i, a_i)$ ，而  $ans = \max(C[i])$ ,  $i=0,1,2,\dots$

这个 dp 的正确性主要是在于，若一段子段和为负数，那么最终答案必然不会以这个子段开头或结尾。而  $\max(C[i] + a_i, a_i)$  其实正是体现了这一点，当前向子段和为负数的时候其实就抛弃了那一段了。其实这个线性时间的算法也可以写成普通的 for，在数据结构书上也有，但是就是麻烦了点，而 dp 非常方便易懂。可以参考: [https://blog.csdn.net/weixin\\_40170902/article/details/80585218](https://blog.csdn.net/weixin_40170902/article/details/80585218)

i) NOI2404:Ride to Office: <http://noi.openjudge.cn/ch0406/2404/>

草了，这道题真的是我白痴了.....其实很好分析，时间在 0 之前出发的，男主追上也没啥，追不上也没啥影响，所以不用考虑。之后就是，男主最后到终点时“搭乘”的那辆自行车一定是最先到终点的.....所以输出满足  $t_{start} > 0$  里  $t_{start} + 4500/v$  最小的那个就好了.....

## 十一、图算法

### (一)、例题

1. 无向图的连通分支数:

在“经典算法笔记”里有写，这里是一道例题:

NOI1526:宗教信仰: <http://noi.openjudge.cn/ch0403/1526/>

注意一下我在写代码的时候有一个小问题，就是在 `for(int i=0; i<num_nodes; i++)` 的那个外层循环里，是 `if(!vis[i]) {cc++; DFS(i);}` 就 `cc++` 要写在里面.....我脑残写到外面了就每次都是 `num_nodes` 的 `cc` 数了.....

当然，算连通分支数也可以直接用并查集.....

2. 最小生成树（还复习了一下并查集）:

NOI253:丛林中的路: <http://noi.openjudge.cn/ch0403/253/>

我用的 Kruscal 算法，其中有一步是判断想要插入的边是否会在图中形成回路，此时的思路很简单，只需要利用并查集的 find 函数，对两个端点都 find 一下它们属



于哪个类，若两个端点不在同一个类中，则可以插入这条边，并且利用并查集的 union 函数把两个类合起来。

并查集的模板代码如下：

```
1 class disjointset{
2 public:
3     int *parent;
4     disjointset(int n)
5     {
6         parent = new int[n];
7         for(int i=0; i<n; i++) parent[i] = -1;
8     }
9     ~disjointset() {delete [] parent;}
10    int disfind(int x)
11    {
12        if(parent[x] < 0) return x;
13        return parent[x] = disfind(parent[x]); //路径压缩
14    }
15    void disunion(int root1, int root2)
16    {
17        if(parent[root1] > parent[root2]) parent[root2] += parent[root1], parent[root1] = root2;
18        else parent[root1] += parent[root2], parent[root2] = root1;
19    }
20 };
```

其主要思想是，成员是一个 parent 数组，用来记录每一个节点的父节点，而父节点为负数的就是根节点，这个节点的下标表示这个类的编号。负数的绝对值表示类树的规模。find 操作会递归的把每个节点的父节点调整为根节点，若输入是根节点直接返回其下标。union 操作会看哪棵树规模大（root 的 parent 值更小），就更新这个节点的 parent 值并且把另一个 root 的 parent 设成这个 root。

Kruscal 算法主体部分采用的数据结构是边列表，定义一个 edge 类，有 from、to、cost 三个字段，再 vector<edge> es; 并且比较函数是 e1.cost < e2.cost 的 sort 一下，逐步取出边判断加入是否有环路。加入 n-1 条边即可。

### 3. 二部图的匹配算法之最大匹配数量：

例子：NOI1538:Gopher II: <http://noi.openjudge.cn/ch0403/1538/>

匈牙利算法，其核心在于利用递归的思想找到增广路，具体见“经典算法笔记”。这道题大意是有 m 个鼹鼠和 n 个洞，老鹰来抓鼠了，现在已知鼹鼠和鼹鼠洞的坐标以及鼹鼠的移动速度 v，在 s 秒内不能跑回某一洞穴的鼹鼠就得死（一个洞只能容纳一个鼹鼠）。抽象出来就是 sv 是最大距离，在最大距离范围内的鼹鼠和洞可以连一条边，抽象成二分图，求最大匹配数量。

代码模板见“经典算法笔记” 嗽

### 4. 二维最短路，权值非 1（带条件的最短路）：

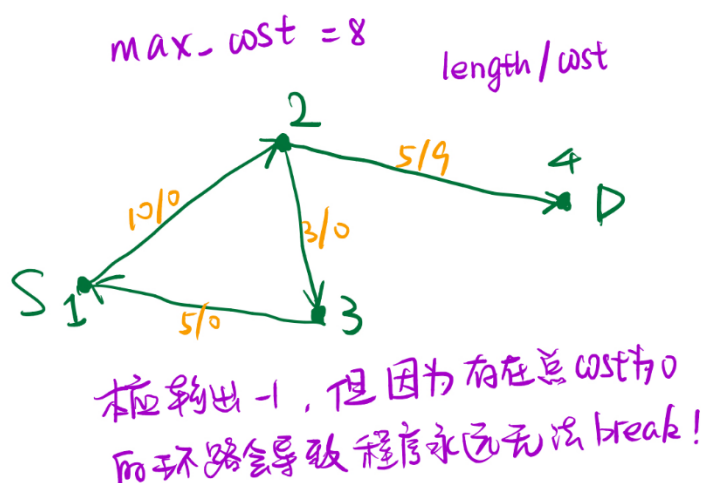
如果权值是 1，可以用二维 bool 数组的 BFS 来做，但这题是权值非 1

例子：NOI726:ROADS: <http://noi.openjudge.cn/ch0403/726/>

最暴力的解法：直接使用优先级队列来做。

状态为(int node\_num, int len, int cost)，优先级队列是把 len 最小的放在队头。然后首先把第 1 号节点入队，即 q.push(status(1, 0, 0)); 然后 while 队列非空时，取出队头，判断是否是目的节点，若是直接输出 len，若不是则访问它的后序节点，当当前 cost 加上当前节点到后序节点的 cost <= 最大 cost 时，就把它入队。也不用考虑什么松弛距离的问题和 vis 的问题了，因为你用的是优先级队列，当前队头一定是符合 cost 限制的最短的路径，目的节点第一次出队时一定是符合 cost 最短的路径。当然没有用 vis 是可能会走环路的，不过由于 cost 的限制最终一定能够跳出 while 循环的。值得注意的是，若不是因为到达目的点，而是因为没有点符合 cost 了而跳出 while 循环，说明到不了目的地，输出-1。

但是这种做法会存在翻车的风险！那就是当存在总 cost 为 0 的环路，并且本应该输出 -1 的情况时，程序永远不会跳出 while 循环：



这道题似乎是恰好因为没有 -1 的输出才.....可以这样过的。

目前还没有想到解决办法（解决了!!!），本来想着加一个二维的 vis 数组，vis[num][cost] 用来表示使用花费 cost 到 num 这个节点有没有被访问到，但是 WA 了，暂时不知为什么，然后在网上找到一个二维的 dijkstra，就是 dis 数组开二维的，dis[num][cost]，然后 vis 也是二维的，松弛操作为  $dis[v][cost + node\_v.cost] > dis[v][cost] + node\_v.len$  且  $cost + node\_v.cost < max\_cost$  就更新，最后从 1 到 max\_cost 找出最小的 length 就是答案，可以解决上面的问题，但是似乎 TLE 了.....

总结一下现在权值非 1 的二维最短路有两种做法：

- ①、直接用最暴力的做法，但无法解决总 cost 为 0 的环路问题，无向图尤其要注意
- ②、用二维的 dijkstra，但是容易 TLE.....

③、好的，终于找到了解决方法：那就是!!! 二维 SPFA!!!!!! SPFA 牛逼!!!!!!

二维 SPFA 就是，将 dis 数组和 vis 数组开成二维的，dis[num][cost] 表示使用 cost 这个花费到达 num 这个点的最小路径长，vis[num][cost] 表示 (num, cost) 这个状态是否在队列中。我们首先把初始节点的状态 (int num, int len, int cost) 放入队列中，初始化  $dis[1][0] = 0$ ，然后不断取出节点，对后继节点 v 们松弛：

if (tmpu.cost + costv <= max\_cost && dis[v][tmpu.cost + costv] > dis[u][tmpu.cost] + lenv)

就把 status(v, dis[u][tmpu.cost] + lenv, tmpu.costv) 入队并设 vis 为 false

最后再从  $i \in [0, k]$  中选取  $\min(dis[n][i])$  即可。

5. 有向图的树判定：

NOI310: Is It A Tree?: <http://noi.openjudge.cn/ch0308/310/>

注意有向图要判断 3 点：

- a) 图中没有环（指无向图的环）：可以用并查集来做
- b) 图中只有一个连通分量（这道题似乎是认为没有孤立的节点——废话，给的数据是边啊，哪来的孤立节点.....），所以只要用并查集看看 parent[i] < -1 的有几个就好了，大于 1 就不行了
- c) 只有一个结点的入度为零（此结点就是根节点）→ 可以直接用输入给定的边集来统计入度为 0 的点的个数
- d) 似乎只需要判断是否只有一个节点的入度为 0，其余的入度均为 1 就可以了???

6. Floyd 算法:

闭包传递: NOI3529:Professor John: <http://noi.openjudge.cn/ch0308/3529/>

就三个 for 之后, `if(adjmatrix[i][k] && adjmatrix[k][j]) adjmatrix[i][j] = 1;`

7. 二重最短路: (区别于二维最短路!!!)

最短路径问题: <http://t.cn/AilPbME2>

题意大致是, 给定一张无向图, 每条边有 `length` 和 `cost`, 要求出从源点到目标点的最短路径长, 在有多条最短路径长一样的时候, 输出总 `cost` 最小的那一条。

解法就是把 `dijkstra` 扩展成二重的, 即加一个 `costs` 数组, 当做第二重 (第二重要) 的 `length`, 优先级队列重载一下, 然后松弛操作的判定条件也要改为:

```
if(dis[v]>dis[u]+len_uv || (dis[v]==dis[u]+len_uv && costs[v]>costs[u]+cost_uv))
{
    dis[v]=dis[u]+len_uv;
    costs[v]=costs[u]+cost_uv;
    q.push(node(v, dis[v], costs[v]));
}
```

这样。可以解决重边的问题哦。

8. 拓扑排序问题:

HDU3342: Legal or Not: <http://acm.hdu.edu.cn/showproblem.php?pid=3342>

其实就是用拓扑排序来判断有没有环啦, 思路很简单, 就是设置 `vis` 和 `indegree` 数组, 然后在输入数据的时候维护 `indegree` 数组, 在拓扑排序前把 `vis` 数组设为 0。

接着, 首先把所有入度为 0 的节点都入队并设 `vis` 为 `true`, 然后 `while` 队非空, 出队并对其所有后继节点: 入度-1, 若入度为 0 入队并设 `vis` 为 `true`。其实就和 BFS 有点像啦。最后检查所有节点的 `vis` 情况, 若存在没有 `vis` 过的节点, 那么原图就不是一个 DAG, 存在有向环。

其原理就是如果存在有向环, 不可能使其上元素入度为 0, 就不可能 `vis` 到啦。

9. AOE 问题 (关键路径问题):

HUD4109: Instruction Arrangement: <http://acm.hdu.edu.cn/showproblem.php?pid=4109>

这道题关键是能把它建模为 AOE 问题!!! 就是一些指令一定要在另外一些指令之后一段确定的时间执行, 这样完成全部的指令就只和最长的那条指令路径长度有关。其实就是一个 AOE 工期的问题, 然后要记得加上源节点到那些入度为 0 的结点上 (使得只有源节点的入度为 0), 然后要把那些出度为 0 的结点加到汇节点上。之后跑 AOE 的最长路算法即可。

其实 AOE 问题的几个特点:

①、有明显的活动先后顺序

②、求最早完成时间 or 求关键活动。

重做了一遍, 妈的被坑死了

关于 `vector` 实现的邻接表.....如果是需要用到多次 (即有多个 case 这种情况)

不要用 `vector<Type> adjList[MAXN]!!!!!!!`

不要用 `vector<Type> adjList[MAXN]!!!!!!!`

不要用 `vector<Type> adjList[MAXN]!!!!!!!`

因为很容易忘记每次 case 之前清空邻接表, 而且也不好清空! 要用 `for` 循环加 `clear` 去清空, 很麻烦的!!!

这种情况直接用 `vector<Type>* adjList` 然后每个 case 去 `new` 就好了! 还省空间!!

然后就是只需把入度为 0 的加在源节点之后...不需要全部加!!!

## (二)、刷题

1. 畅通工程: <http://t.cn/AiOvBHj9>  
就是问现在已经有  $n$  个节点和  $m$  条路, 最少再加几条路可以使图变成连通图。(其实就是问连通分支数-1 是多少嘛.....) 无向图, 直接上并查集即可。需要注意的是并查集的 parent 要 new[n+1], 然后下标从 1 开始。因为输入的点是从 1 开始的。
2. 找出直系亲属: <http://t.cn/AiOzQMBH>  
Emmmmm...这题想考的就是建有向图然后给定初始节点来寻找目标结点吧, 用 BFS 就可以了, 而且由于这个图的子图(树)的搜索问题, 所以不用 vis 数组也可以。比较要注意的是这道题需要找出层数(也可以说是步数啦), 所以要用一个 pair 来作为搜索状态, 而且输出的时候.....**审题啊!!!!!!** 他要输出 child 的.....你怎么能只输出 parent.....
3. SJTU 复试: 最短路径:  
<https://www.nowcoder.com/practice/a29d0b5eb46b4b90bfa22aa98cf5ff17?tpId=62&tpQId=29464&tPage=1&ru=%2Fkaoyan%2Ftest%2F2002&qru=%2Fta%2Fquestion-ranking>  
乍一看, 似乎是高精度+dijkstra 的问题, 我也的确是这样去做了, 虽然 AC 了, 但是花了一个小时.....而且时间复杂度和空间复杂度都挺高。仔细想想就会发现不对劲的地方! 首先你知道的一点是,  $2^n$  是比  $\sum_{k=0}^{n-1} 2^k$  还要大的, 也就是说, 你在添加边的过程中, 如果发现当前添加的边的两个端点在同一并查集里, 那么这条边必定比之前所有的边加起来都大, 所有不可能有最短路径是用到这条边的, 因此舍弃。这样我们每一次添加边, 都是添加两个端点在两个不同并查集里的边, 而不会出现环路。所以最后我们得到的应该是一棵生成树。而众所周知的是, 生成树的两点间只有唯一路径.....因此在添加边的过程中可以直接取模了, 最后再用普通的最短路跑一下就好了(可以直接用 BFS, 因为是有唯一路径).....不愧是我们学校, 一点套路都没有!!! (叫最短路径真是太...具有误导性了) 注意答案输出的时候也要取模...因为这个 WA 了草。
4. HDU1285:确定比赛名次: <http://acm.hdu.edu.cn/showproblem.php?pid=1285>  
草??? 这道题思路及其简单, 就是一个拓扑排序+优先级队列, 确保在不破坏拓扑排序顺序的同时编号小的先输出, 比较草的是它要求最后一个输出之后没有空格, 然后我就用 string rtv 来保存输出最后取 rtv.substr(0, rtv.size()-1) 然后一直 WA??? 没搞懂是为什么, 本身以为是重边问题, 但似乎邻接表本来就不会有重边的问题。之后采用了如果是第一个就输出本身, 不是第一个就输出空格+本身这样的操作结果就 AC 了?????? 莫名其妙, 迷惑。

## 十二、动态规划

### (一)、例子

1. 最大连续子序列和:  
dp[i]表示的是以 a[i]为结尾的连续子序列的最大和。那么状态转移方程是:  
$$dp[0] = a[0]; dp[i] = \max\{dp[i-1] + a[i], a[i]\}$$

即，是在前面的序列上再加上  $a[i]$  还是另起炉灶再开一个序列（单个  $a[i]$ ），最终的答案是  $dp[0]$  到  $dp[n-1]$  中最大的那个。算法的正确性其实是，当  $dp[i-1] < 0$  时选的肯定是  $a[i]$ ，因为我们知道负序列肯定不会包含在答案的头或尾的，而当  $dp[i-1] > 0$  时我们就选前面一个，无论如何都比  $a[i]$  自己另起炉灶更大（当然  $a[i]$  此时  $> 0$  就更好了）。

## 2. 最大递增（递减，非增，非减）子序列：

$dp[i]$  表示的是以  $a[i]$  为结尾的递增序列的最大长度。那么状态转移方程是：

$dp[0] = 1; dp[i] =$  从小于  $i$  的  $j$  里找  $a[j] < a[i]$  的，在所有满足条件的  $j$  里，找出  $dp[j]$  最大的，此  $dp[j] + 1$  就是  $dp[i]$  的值。这样的复杂度是  $O(n^2)$ 。

当然，递减就是找  $a[j] > a[i]$  的，非递增就是找  $a[j] \geq a[i]$  的，非递减就是  $a[j] \leq a[i]$  的。可以用树状数组优化，以最大递增子序列为例，由于树状数组的 `tsearch` 函数返回的是小于等于  $x$  的区间的最大值，而我们要找的就是  $a[j] < a[i]$  的里面  $dp[j]$  最大的那个，那么我们可以用  $a[j]$  作为树状数组的下标， $dp[j]$  作为树状数组的值。这样 `tsearch` 函数 `tsearch(a[i]-1)` 【注意这个减一!!!】就可以找出比  $a[i]$  小的里， $dp$  值最大的那个了。一次寻找的复杂度是  $O(\log(\max n))$ ，其中  $\max n$  是  $a[i]$  的最大值。总复杂度为  $O(n \log(\max n))$ 。注意这里 `update` 函数只要写普通的 `tadd` 函数就可以了，因为我们在更新  $(a[i], dp[i])$  对的时候，我们希望的恰好是在有重复的  $a[i]$  值时， $dp[i]$  小的那一个不要更新原有的最大值。普通的 `tadd` 函数刚好有这样的特点（虽然这在需要点修改影响到区间最值的时候是错误的）。//具体可以见模板代码的 `treeArray.cpp`，里面有注释

eg: 合唱队形: <http://t.cn/AiYNYHPe>

这题是最大上升再下降序列，其实就是把最大上升序列的  $dp1[i]$  和最大下降序列的  $dp2[i]$  全都算出来就好了，根据  $dp1[i]$  的意义是以  $a[i]$  为结尾的最大上升序列， $dp2[i]$  的意义是以  $a[i]$  为开头的最大下降序列（或者说反向的以  $a[i]$  为结尾的最大上升序列），其也反着求就好了。然后答案就是  $dp1[i] + dp2[i] - 1$  中最大的那个了。

注意就是...这道题算的是出队人数的最小值，所以要用 `n-ans`.....

【2020.07.03】我用树状数组又实现了一遍，可以看 ShuaTi 文件夹，需要注意的是 `tsearch` 的时候如果是求单调增 or 单调减要用 `tsearch(a[i]-1)` 啊!!! 因为不能取小于等于 or 大于等于!!!

## 3. 最长公共子序列: max common sequence length

$a$  串和  $b$  串， $dp[i][j]$  表示的是  $a$  里面  $1 \sim i$  的子串与  $b$  中  $1 \sim j$  的子串的最长公共子序列。显然  $dp[0][x] = dp[x][0] = 0$ 。状态转移方程：

$$dp[i][j] = dp[i-1][j-1] + 1 \quad \text{if } a[i] == b[j]$$

$$dp[i][j] = \max(dp[i-1][j], dp[i][j-1]) \quad \text{if } a[i] != b[j]$$

最后  $dp[a.size()][b.size()]$  就是答案了（注意如果你在  $a, b$  前面添了一个字符好让  $a, b$  下标从 1 开始，那你要先记录  $m = a.size(), n = b.size()$  再最后返回  $dp[m][n]$ ）

**注意！**

“最长公共子序列”和“最长公共子串”不一样！后者要求子串是连续的，也是用 `dp` 去解，留给你复习的时候思考怎么做（虽然你就是我）。

【2020.07.03】好吧，其实就是  $dp[i][j]$  表示以  $a[i]$  和  $b[j]$  为结尾的最大公共子串长度，那么  $dp[i][j] = 0$  if  $a[i] != b[j]$  else  $dp[i-1][j-1] + 1$ ;

这里注意一下字符串也是从下标 1 开始的，因为如果从 0 开始遇到  $a[1] = b[0]$  这种不好去递推。然后 `dp` 的初始化就  $dp[0][xx] = dp[xx][0] = 0$  就好了。

## 4. 0-1 背包：



0-1 背包的意思就是，一样东西有它的“重量”和“价值”，背包有它的“容量”，要在背包的容量限制下选择价值和最大的东西。一样东西要么不拿要么拿一件（这就是为什么是 0-1 背包）。

$dp[i][j]$  表示用前  $i$  样东西( $i=0$  表示没有东西)装容量为  $j$  的背包的最大价值。

显然有  $dp[xx][0] = 0$ ;  $dp[0][xx] = 0$ ; 状态转移方程:

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-\text{weight}[i]] + \text{value}[i])$$

前一项表示不装第  $i$  个东西，那么就是用前  $i-1$  个东西去装容量为  $j$  的背包，后一项表示装第  $i$  个东西，那么就是用前  $i-1$  个东西去装容量为  $j-\text{weight}[i]$  的最大值+当前东西的  $\text{value}[i]$ 。这两者间大的就是  $dp[i][j]$  的值。

观察发现每一轮的  $dp[i][j]$  都之和  $dp[i-1][xx]$  有关，所以可以把二维的  $dp$  压缩到一维节省空间（好像也没什么卵用）。也就是新的  $dp[j]$  更新  $n$  轮，每一轮要从  $j=\text{容量}$  开始更新，直到  $j < \text{weight}[i]$ 。为什么要从  $j=\text{容量}$  开始  $j--$  呢，因为这是不想让先更新的  $j-\text{weight}[i]$  影响到后面  $j$  的更新（因为这里的  $dp$  相当于存着上一轮的数组，而我们知道上一轮的数组在二维的  $dp$  中是不变的，从后往前更新才能体现出这种不变，因为  $dp[j]$  的更新 ( $k < j$ ) 不会影响到之后更小的  $dp[k]$  的更新，而从小到大更新会影响到）。

eg: <http://t.cn/AiYlwchD> 最小邮票数

这道题，其实不算是装包问题，但是可以用类似的思想。仔细想想，这种“第  $i$  个物品用还是不用”的问题似乎都可以  $dp$ ，而且可以用 DFS 爆搜（如果数据小）。我们用  $dp[i][j]$  表示用前  $i$  张邮票刚好凑出  $j$  的价格所花费的邮票数， $\text{inf}$  表示凑不出，则我们有：

$$dp[i][j] = \min(dp[i-1][j], dp[i-1][j-\text{value}[i]] + 1)$$

前半表示不用第  $i$  张，后半表示用。同样可以压缩成一维。然后  $dp[xx][0] = 0$ ;  $dp[0][xx] = \text{inf}$ ，但是  $dp[0][0] = 0$ ；压缩后初始化： $dp[0] = 0$ ；其他  $dp[j] = \text{inf}$ ，用类似的  $dp$  就可以求解。

#### 5. 前 $k$ 优背包：

eg: <https://www.luogu.com.cn/problem/P1858>

其思路是在 0-1 背包（或者完全背包啦）的基础上再加一维状态，下标  $i$  表示第  $i$  好的答案。初始化的时候要注意是装满还是啥，如果是装满那  $dp[0][1 \sim k]$  就要设为负无穷了，只有  $dp[0][0] = 0$ 。然后状态转移的时候，在处理第  $i$  轮容量为  $j$  的时候，要  $\text{int } t \text{ from } 0 \text{ to } k-1$  对比  $dp[j][\text{head1}]$  与  $dp[j-\text{C}[i]][\text{head2}] + V[i]$  哪个大，哪个更大就赋值给  $\text{tmp}[t]$  哪个，其实就是归并的思想。然后  $t$  的循环结束后再把  $dp[j][t]$  用  $\text{tmp}[t]$  更新一轮。为什么要  $\text{tmp}$  呢，因为在归并的时候不希望影响到  $dp[j][xx]$  的值。还有就是写了  $i, j, k$  的三维  $dp$ ...本来以为更稳结果还 WA 了，不知道为什么...不管了，记一下这个题型的结论好了。

**我知道为什么错了!!!!!!**

如果你硬要用  $i, j, k$  三个状态（就是不做空间复杂度的优化），那么你不能进行这样的操作：for( $j=\text{C}[i]$ ;  $j \leq V$ ;  $j++$ ) 或者是 for( $j=V$ ;  $j \geq \text{C}[i]$ ;  $j--$ )

因为如果有遇到  $\text{C}[i]$  是大于  $j$  的，根本不会执行这个 for 循环，但是其实应该有  $dp[i][j][k] = dp[i-1][j][k]$  的，在不用  $i$  这个维度的时候，默认是不修改  $dp[j][k]$  的值的，其实隐式地达到了这样的效果，但是使用  $i$  这个维度的时候，如果不执行上述语句  $dp[i][j][k]$  是**没有被更新的!!!** 这样就会导致错误了!!! 因此要  $j=0 \text{ to } V$ ! 然后在里面判断 if( $j \geq \text{C}[i]$ )

## (二)、刷题

1. NOI6049: 买书 <http://noi.openjudge.cn/ch0206/6049/>

2020-04-08

这道题就是完全背包的变形，要求输出方案数。所以很简单咯，把 max 改为 sum 就行！注意初始化不要出错！初始化  $dp[0]=1$ ，其他都是 0，因为用前 0 本书装满容量为 0 的背包的方案数是 1，不是 0！其他的都是 0。

易错点 1：要注意这是完全背包，所以是顺序遍历  $j$ （可以从  $j=C[i]$ ）开始

易错点 2：其实就是上面黄红字体的那个啦，如果用  $i$  这个下标要 from  $j=0$  to  $V$ ，在循环里判断  $if(j \geq C[i])$

2. NOI 4978: 宠物小精灵之收服: <http://noi.openjudge.cn/ch0206/4978/>

2020-04-08

就是很简单的二维背包.....不过要注意它要得到的是，抓到小精灵数量最多的时候，若有多个解输出体力消耗最少的那个，以及剩余体力。其实就是对 dp 数组的理解。

$dp[i][j]$  表示最多使用  $j$  个精灵球， $i$  的体力能抓到的小精灵的最大数量。因此最终的最大抓获小精灵数量一定是  $dp[m][n]$  ( $m$  是最大体力， $n$  是精灵球数量)，因此我们从  $x$  from 0 to  $m$  遍历  $dp[x][n]$ ，找到的第一个  $dp[x][n] == dp[m][n]$  的  $x$  就是最小消耗的体力了。这是因为 dp 数组其实是考虑了所有可能的情况的，因此不去除完爆的去 dp，最后得到的也是最小的体力消耗。

妈的这题太坑了，居然  $n$  是精灵球数量... $k$  是小精灵数目，搞得我在写代码的时候不小心写错好多变量名 QAQ。下次遇到这种变量名很迷惑的题还是自己起一下名字吧!!!

3. NOI8467: 鸣人的影分身: <http://noi.openjudge.cn/ch0206/8467/>

2020-04-08

说的是鸣人有  $m$  的查克拉，分到  $n$  个相同的影分身上有几种分法（影分身可以分 0 查克拉）。典型的组合数学里  $m$  个相同的球分到  $n$  个相同的桶里的问题.....

这边的思路是  $x_1+x_2+\dots+x_n = m, x_1 \geq x_2 \geq \dots \geq x_n \geq 0$  哦

$dp[m][n]$  表示  $m$  球入  $n$  桶的组合数

可以先考虑  $x_1 \geq x_2 \geq \dots \geq x_n \geq 1$ ，然后考虑  $x_n=1$  和  $x_n>1$  两种情况

一种可以得到是  $dp[m-1][n-1]$ ，另一种是  $dp[m-n][n]$

所以  $dp[m][n] = dp[m-1][n-1] + dp[m-n][n]$

比较需要注意的是初始化问题：

由于要求  $x_i \geq 1$ ，所以  $dp[0][xx]$  都等于 0（0 个球怎么投），然后  $dp[\geq 1][1]$  都等于 1（只有一个桶，只有一种放法）。实际代码 memset 为 0 之后只要更新  $dp[\geq 1][1]$  的值为 1 就好了。

然后 for 循环  $i$  从 1 开始， $j$  从 2 开始就 ok

如果是  $x_1 \geq x_2 \geq \dots \geq x_n \geq 0$ ，有两种方法，一种是一开始就让  $m=m+n$ ，你可以通过  $y=x+1 \geq 1$  来推。第二种是  $i$  从 1 到  $n$  对  $dp[m][i]$  求个和，表示有些桶（影分身）可以不放球（查克拉）。

以前做的“放苹果”也是这题型，这次再复习一下啦。

【2020.07.04】也可以直接用  $x_n=0$  or  $x_n>0$  两种情况直接算递推式啦。

4. NOI8471: 切割回文: <http://noi.openjudge.cn/ch0206/8471/>

2020-04-09

---

一开始想的 dp 是  $dp[i][j]$  表示  $s[i, j]$  最少切多少刀，然后初始化  $dp[x][x] = 0$ ，之后状态转移  $dp[i][j] = \min\{dp[i][k] + dp[k+1][j] + 1\}$  如果  $s[i, j]$  不是回文，否则就是 0。但是这样是  $O(n^3)$ .....直接 TLE。

上网查了之后发现可以只用一维的 dp， $dp[i]$  表示前  $i$  个字符最少切几刀，要做一个预处理，算出  $r[i][j]$  表示  $s[i, j]$  是否是回文。然后状态转移方程：

$dp[i] = \min\{dp[j] + 1 \mid r[j+1][i] == \text{true}\}, j \text{ from } 0 \text{ to } i-1$ ，当然  $r[0][i]=1$  的时候  $dp[i]=0$

**预处理：** $O(n^2)$  算法我也没想到.....就是选一个中心字符下标  $x$ ， $x$  从 0 到  $n-1$ ，在不越界的情况下向两边扩张来更新  $r[x-len][x+len]$  ( $len$  从 0 开始)，这时更新的是长度为奇数的子串的  $r[i][j]$  的值，为了保证长度为偶数的子串也被更新，也要更新  $r[x-len][x+len-1]$  ( $len$  从 1 开始)。

**正确性：**这个 dp 为什么正确呢，你可能会考虑，只考虑  $r[j+1][i] = \text{true}$  的情况会不会漏掉那些  $r[j+1][i] = \text{false}$ ，但是最优解是  $dp[j]$  加上  $s[j+1, i]$  中间再切几刀的情况。但其实这样的 dp 不会漏掉这些情况，因为这些情况其实  $dp[i-1]+1$  都能解决.....因为如果你后面不是回文的话，最后的  $dp[i-1]$  其实都包括  $s[j+1, i]$  中间再切几刀的情况了，不必担心它被漏掉.....所以你只要关心后面是回文的，因为它不用被切了，只要+1 就行了。

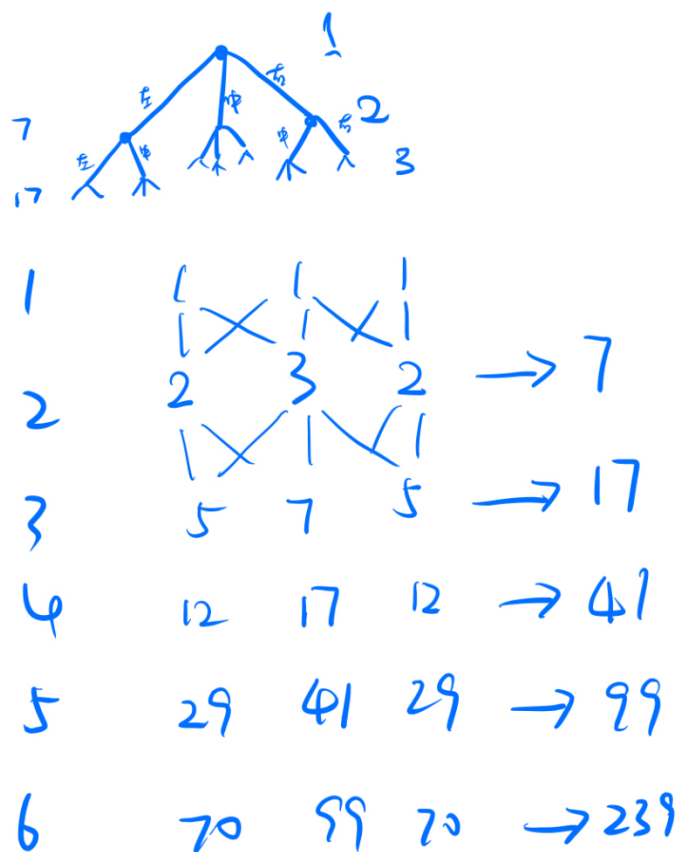
**dp 好难.....**

5. NOI9271:奶牛散步：<http://noi.openjudge.cn/ch0206/9271/> (2020-04-11)

题意是在无限大的方格棋盘上只能往上、左、右走一个方格，问你走  $n$  步不重复的走法有多少种。

观察：只要当前走的步不是上一次的反方向，就永远不会走重复的方格（因为不能往下走，所以想换左右方向必须先往上走一步，这样就不会重复了）。

所以就是一个简单的递推：



但是很坑爹的是它居然答案要 mod12345 啊!!!! 而且题目还没说!!! 草!

6. NOI7627:鸡蛋的硬度: <http://noi.openjudge.cn/ch0206/7627/> (2020-04-11)

状态设计很快出来了.....状态转移搞了好久!!! 这一次扔鸡蛋也要+1 啊!! 以后不要犯这种错误了 QAQ。其实就是简单的, 因为我害怕最坏情况的发生, 所以我每次都假设是最坏情况, 于是第一次扔在第几楼, 我就用  $\max+dp$  子问题来决定这一楼是碎还是不碎。然后再求最坏情况下我扔在第几楼是次数最小的, 所以要用  $\min$ 。即内层用的  $\max$ , 外层用的  $\min$ 。

【2020.07.05】初始状态也很坑啊.....只有一层楼的时候,  $dp[1][xx]=1$ , 但是没有鸡蛋了  $dp[xx][0]$ 要等于  $\inf$ ,  $dp[1][0]$ 也要是  $\inf$ !!!! 而且只有 0 层楼高时, 按照题意鸡蛋的硬度都为 1, 所以  $dp[0][xx]=0$ !!!! 太恶心了。

7. NOI2728:摘花生: <http://noi.openjudge.cn/ch0206/2728/> (2020-04-12)

这道题是一道比较简单的  $dp$ , 由于只能往右下角走, 所以你可以考虑每个状态是由从左边走来和从上边走来两种情况得到的。于是  $dp[i][j]$ 表示 Hello Kitty 走到位置为  $(i, j)$ 的这个点的最多花生数, 那么  $dp[i][j] = \max(dp[i][j-1], dp[i-1][j]) + a[i][j]$ 。其实从这道题如果数据小的确是可以“分治”(深搜)的, 就是每一个位置都有向右和向下两种情况, 即  $\text{return } \max(\text{dfs}(i, j+1), \text{dfs}(i+1, j))$ ; 但是这样时间复杂度太大了!!!  $O(2^{2n})$ 的时间复杂度。这是因为“分治”重复计算了好多子问题。

**这也可以引出何时应该用  $dp$  的思考:**

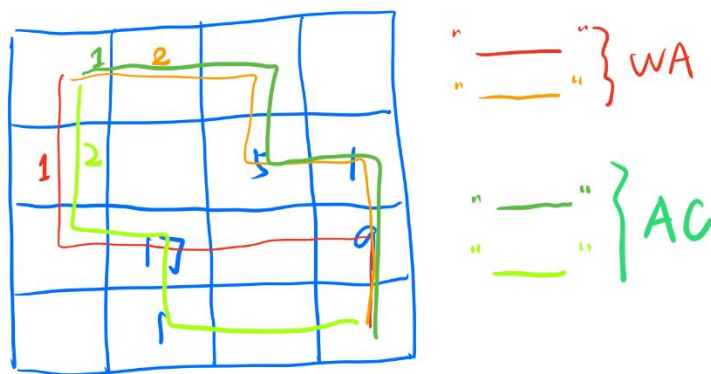
有类似于“每一步都有几种选择”这样的问题时, 并且搜索的时间复杂度很高, 那十有八九是  $dp$  没跑了!

【2020.07.06】妈的太可怕了.....初始化的时候又出错了, 一直横着走或者一直竖着走是没有办法用  $dp$  更新到的.....因为会越界!!! 所以  $dp$  的状态转移方程写出

来之后，初始化的时候不仅要考虑特殊情况，还得考虑能不能用转移方程更新！即会不会越界之类!!!

8. NOI8786:方格取数: <http://noi.openjudge.cn/ch0206/8786/> (2020-04-12)

这道题和前一题非常像，不一样的地方是这道题可以走两次，每次取了数字会把方格里的数字清 0。比较 naïve 的想法是贪心，第一次用 dp 取最多的数字，然后根据路径更新取走数字的方格为 0。第二次再取当前最多的。但这样的贪心是错误的，因为策略具有后效性，第一次取走数字对第二次 dp 有影响。可能会出现例如如果两次都取局部最优无法将数字全部拿走，但是第一次不取最优第二次可以把数字全部拿走的情况：



上网查之后发现这叫多线程 dp，就是多加状态去同时模拟两个人的行为。

$dp[i][j][k][p]$  表示第一个人走到  $(i, j)$  这个位置，第二个人走到  $(k, p)$  这个位置时两人的收益之和的最大值，则状态转移方程为：

$$dp[i][j][k][p] = \max(dp[i][j-1][k][p-1], dp[i][j-1][k-1][p], dp[i-1][j][k][p-1], dp[i-1][j][k-1][p]) + a[i][j] \text{ if } (i, j) == (k, p) \text{ else } (a[i][j] + a[k][p])$$

语义解释就是四种移动情况：

- 1、第一个人从左边过来 + 第二个人从左边过来
  - 2、第一个人从左边过来 + 第二个人从上边过来
  - 3、第一个人从上边过来 + 第二个人从左边过来
  - 4、第一个人从上边过来 + 第二个人从上边过来
- 以及两种当前情况：

- 1、两个人当前在同一格（只能  $+a[i][j]$ ）
- 2、两个人当前在不同格（ $+a[i][j] + a[k][p]$ ）

9. NOI 9268:酒鬼: <http://noi.openjudge.cn/ch0206/9268/>

一开始想是不是和最长递增子序列数一样  $dp[i]$  表示第  $i$  瓶酒必喝的情况.....结果发现状态转移找不出来.....或者说找出来的是错的!!!最后用了个错的状态转移（没有后效独立性，换句话说有后效性），但是还骗到了 6/9 分.....

然后又想是不是和背包类似， $dp[i]$  表示第  $i$  瓶酒喝还是不喝.....但是状态转移方程一直想不出来!!! 思维局限在了第  $i$  瓶酒上！就不知道怎么去表达不能连续喝三瓶酒的这个规则了.....然后也没想出来。

正解看一下就很简单 QAQ

不能连续喝三瓶酒的这个规则其实可以用喝不喝第  $i-1$  瓶酒得到.....所以状态转移方程为：

$$dp[i] = \max(dp[i-1], a[i] + dp[i-2], a[i] + a[i-1] + dp[i-3])$$

分别表示：



- a) 不喝第  $i$  瓶酒，就是只用前  $i-1$  瓶酒能达到的最大体积
- b) 喝第  $i$  瓶但是不喝第  $i-1$  瓶，那么第  $i-2$  瓶理论上是可以喝的，所以就是第  $i$  瓶的体积加上用前  $i-2$  瓶酒能喝到的最大体积
- c) 第  $i$  瓶和第  $i-1$  瓶都喝，那按照规则就不能喝第  $i-2$  瓶了，所以只能用第  $i$ 、 $i-1$  瓶的体积加上用前  $i-3$  瓶酒所能喝到的最大体积

上述三者的最大值就是  $dp[i]$  了.....

然后  $dp[1,2,3]$  都很好算，就是初始状态了！

**一定要灵活运用 dp 数组的意义去阐释题目的规则啊.....**

10. NOI9282:B 君的多边形: <http://noi.openjudge.cn/ch0206/9282/>

这道题.....如果考试考我是不可能做的哈！

答案是一个叫做超级卡特兰数的东西（第一眼看过过去我就觉得像卡特兰数），递推公式如下：

$$F_n * (n + 1) = (6 * n - 3) * F_{n-1} - (n - 2) * F_{n-2}$$

然后就是这道题要求 mod 1000000007 嘛，正好也给我复习了两个知识点：

1、快速幂：

fast\_exp(a, b):

ans = 1;

while(b){if(b&1) ans = ans\*a% mod; b/=2; a=a\*a%mod; }

return ans;

2、带有 mod 的 dp（递推）方程的处理：

因为带有 mod，递推方程中间如果有减号并涉及到两个不同的状态，可能相减之后会小于 0，这时需要用  $(xxx+mod) \% mod$  来处理：

$$F[i] = (( (6*i-3)*F[i-1]) \% mod - ((i-2)*F[i-2]) \% mod + mod) \% mod * inv \% mod;$$

11. NOI2988:计算字符串距离: <http://noi.openjudge.cn/ch0206/2988/>

其实就是 edit distance，以前 dp 课上都学过!!!

具体就是， $dp[i][j]$  表示匹配  $s1$  的前  $i$  个字符和  $s2$  的前  $j$  个字符的 cost，那么针对字符  $s1[i]$  和  $s2[j]$ ，有三种选择：

- a) 匹配  $i$  和  $j$  的字符，若匹配，cost 为  $dp[i-1][j-1]$ ，不匹配就再+1
- b)  $s2[j]$  复制一个给  $s1$ ，用  $s1$  的前  $i$  个字符再去和  $s2$  的前  $j-1$  个字符匹配，即有  $dp[i][j-1]+1$
- c)  $s1[i]$  复制一个给  $s2$ ，用  $s2$  的前  $j$  个字符再去和  $s1$  的前  $i-1$  个字符匹配，即有： $dp[i-1][j]+1$

$dp[i][j]$  就是三者中最小的。

12. NOI4979:海贼王之伟大航路(OnePiece): <http://noi.openjudge.cn/ch0405/4979/>

这道题是一种新的 dp 思路: **状态压缩 bp**

状压 dp 的一个特点是，算法是指数级的，但是比搜索（穷举）效率要高，比如说这倒题搜索的复杂度最高是  $14!$ （中间 14 座岛的排列数），而状压 dp 的复杂度最高是  $2^{*16}$ ，可以接受。

另一个特点是，状态数很多，通常是指数级的，且可以用二进制表示。如这道题，状态有  $2^{*16}$  种，用 16 位二进制数可以表示，第  $i$  位为 0 表示经过了第  $i$  座岛（最低位为第 1 位）。

具体的思路可以看代码 F:\POJ\DynamicProgramming\OnePiece\main.cpp，里面的注释都非常清楚了。（直接在资源管理器打这个位置就行）

下面说一下为什么这种 dp 是正确的，以及如何想到这样的 dp。

首先明确, **dp** 最重要的一点就是无后效性, 即后面的问题调用之前保存的子问题, 且这些问题不会影响到后续的问题。

拿这道题来说, 这道题是通过枚举状态作为最外层循环, 即  $k$  从 1 到  $(1 \ll n) - 1$ , 每轮循环的  $k$  对应于一种状态, 而对于每一个合法的岛  $i$  ( $k$  这个状态经过了  $i$ ), 检查每一个合法的岛  $j$  ( $k$  状态经过了岛  $j$ ), 看看能不能用 “ $dp[j][k]$  把  $k$  的第  $i$  位置 0” 这个状态 +  $dis[j][i]$  到  $i$  岛来更新  $dp[i][k]$  的值。这样做的正确性在于, 我们用来更新 **dp** 数组的  $dp[j][k]$  把  $k$  的第  $i$  位置 0 的第二维一定是比  $k$  小的 (因为把 1 置零了), 这样在前面的循环中已经把小于  $k$  的状态都更新过了, 因此前面的更新不影响后面, 后面的更新也不影响前面, 具有无后效性。

至于如何想到这样的 **dp**.....我也不知道

然后是一些注意点:

a) 初始化一定要做好啊!!!

b) 移位是用 1 去移啊!!! 虽然等价于 2 的  $i$  次方但是别写成用 2 去移啊魂淡!

最后给出一些有用的帖子:

关于状压 **dp** 中位操作的好帖子 (只要看它的位操作部分就好了):

<https://blog.csdn.net/u011077606/article/details/43487421>

还有一个讲状压 **dp** 的帖子:

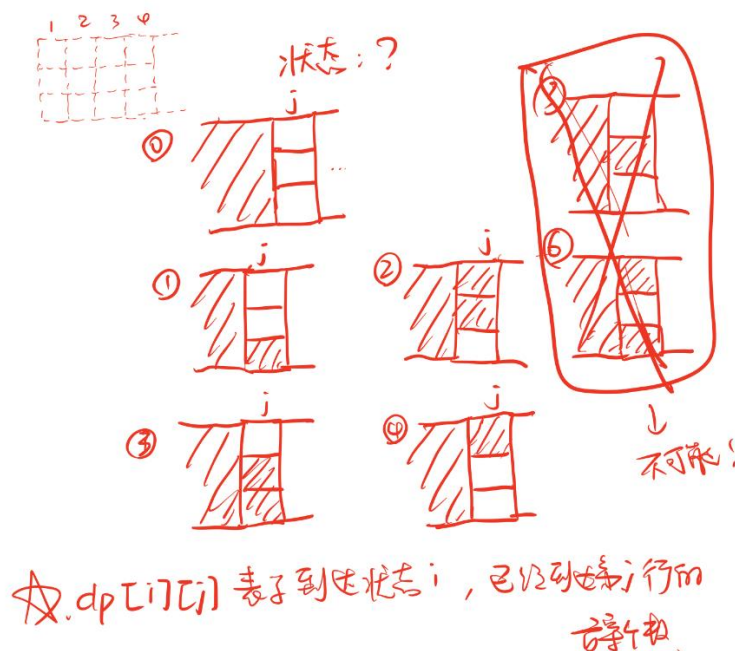
<https://blog.csdn.net/u013377068/article/details/81054112>

13. NOI 1665:完美覆盖: <http://noi.openjudge.cn/ch0405/1665/>

这道题, 理论上也可以是状压 **bp**, 但是可以做得更简单。之前做过! 当时没想出来, 这次还是自己想出来了! 有进步哈!!!

前面的状压 **dp** 也解决了我当时想不明白这道题的状态到底怎么想出来的的问题...

其实状态就是第  $j$  列哪里有被覆盖到的组合, 理论上  $2^3$  种(这道题一行只有 3 行), 但是可以通过等价和不可能情况排除三种, 最后剩下 5 种:



然后写出每种状态的转移方程就行了, 需要注意的是状态 0 除了可以由①、④转移, 还可以通过 0 自身转移, 即:

$$dp[0][j] = dp[1][j-1] + dp[4][j-1] + dp[0][j-2]$$

$dp[0][j-2]$ 对应用三根横着的木块覆盖两列到达  $dp[0][j]$ 的情况。

然后其他的转移方程比较容易，但注意不要写重复了。

14. NOI 2454:雷涛的小猫: <http://noi.openjudge.cn/ch0405/2454/>

这道题好像不是什么套路？没搜过反正自己做出来的。

$dp[i][j]$ 表示在第  $i$  棵树的高度  $j$  上能吃到的最多柿子数，状态转移就是：

a) 本棵树高度-1 即:  $dp[i][j-1] + shizi[i][j]$  ——1 种

b) 跳到第  $k$  棵树上，即:  $dp[i][j-\delta] + shizi[i][j]$  —— $n-1$  种

这  $n$  种情况的最大值

所以发现应该以  $j$  为最外层循环然后一层一层向上推出最终答案。

刚开始这样写 for j, for i, for k max(...)

这样复杂度是  $O(h*n^2)$ ，TLE.....

后来发现最里层的 for k 其实没有必要写，因为可以在计算某一层  $j$  的时候用 record 数组记录第  $j$  层的最大柿子数是几（因为我们并不关心是哪棵树是最大的），这样最里层的 for k 就可以省略成 record[xxx]

即状态转移变成了  $dp[i][j] = \max(dp[i][j-1], record[j-\delta]) + shizi[i][j]$  了，然后再同时保存下 record[j]即可。这样复杂度就是  $O(hn)$ 了，简单的优化。

**这里说一下计算复杂性和时间的估算：**

**1秒可执行语句的范围为 $10^6 - 10^8$ 。**

对应 $O(n)$ 的范围为 $10^6 - 10^8$ ;

对应 $O(n^2)$ 的范围为 $10^3 - 10^4$ ;

对应 $O(n^3)$ 的范围为 $10^1 - 10^2$ 。。。。。

以后就把 1s 的操作数当成  $10^7$ ，就是一千万次基本计算，像这道题就是 10s，大概是最多做  $10^8$  次计算，顶破天  $10^9$  次计算。而  $O(h*n^2)$ 的算法有  $2000^3$  就是  $8*10^9$  次计算，显然是必超时的.....

15. NOI9272:偶数个数字 3: <http://noi.openjudge.cn/ch0206/9272/>

**【tmd 这题要模 12345.....】**

这道题.....居然想了好久，而且还是用歪门邪道的方法做出来的。一开始想用 3 填在不同的格子来分割子问题，但是发现会有重复，然后一直想着怎么去重，以及怎么解决第一位是不是 0 的问题.....陷入了思维的怪圈，没有 think out of the box。后来受到“全概率公式”的启发，想到用第一位是 0、1、2、3.....9 来区分状态，即  $dp\_even/odd[i][j]$ 表示长度为  $i$  且第一位为  $j$  ( $j$  可为 0) 的数里含 3 的个数为偶/奇的数的个数，然后状态转移就很简单。

看了别人的题解感觉自己是个 sb.....想了好久还用这种歪门邪道解法做了。

better 题解：

$dp[i][0]$ 表示 3 的个数为偶数的合法  $i$  位数的个数， $dp[i][1]$  表示 3 的个数为奇数的合法  $i$  位数的个数。那么我们从最后一位是不是 3 来分类讨论（这样既解决了合法不合法的问题，因为子问题一定是合法的，又只有两类，类似全概率公式的  $P$  和  $\bar{P}$ ）。

所以  $dp[i][0] = dp[i-1][0]*9 + dp[i-1][1]$ ,  $dp[i][1] = dp[i-1][1]*9 + dp[i-1][0]$

---

以第一个转移方程为例，就是说，如果最后一位不填 3，那么前面  $i-1$  位数含 3 的个数就应该是偶数，且可以填 0、1、2、4...9，所以是  $dp[i-1][0]*9$ ，如果最后一位填 3，那么前面  $i-1$  位数含 3 的个数就应该是奇数，所以是  $dp[i-1][1]*1$ 。