USACO 心得

第一部分 动态规划

USACO 里可用 DP 解决的题目不少，计有：bigbrn、buylow、charrec、game1、inflate、milk4、money、nocows、nuggets、numtri、range、rectbarn、rockers、stamps、subsets、theme、tour、vans。

首先，有一类典型的题目是与“背包问题”及其变形有关的。

Inflate 是加权 01 背包问题，也就是说：每种物品只有一件，只可以选择放或者不放；而且每种物品有对应的权值，目标是使总权值最大或最小。它最朴素的状态转移方程是：f[k][i] = max{f[k-1][i]，f[k-1][i-v[k]]+w[k]}。f[k][i]表示前 k 件物品花费代价 i 可以得到的最大权值。v[k]和 w[k]分别是第 k 件物品的花费和权值。可以看到，f[k]的求解过程就是使用第 k 件物品对 f[k-1]进行更新的过程。那么事实上就不用使用二维数组，只需要定义 f[i]，然后对于每件物品 k，顺序地检查 f[i]与 f[i-v[k]]+w[k]的大小，如果后者更大，就对前者进行更新。这是背包问题中典型的优化方法。

题目 stamps 中，每种物品的使用量没有直接限制，但使用物品的总量有限制。求第一个不能用这有限个物品组成的背包的大小。（可以这样等价地认为）设 f[k][i]表示前 k 件物品组成大小为 i 的背包，最少需要物品的数量。则 f[k][i]= min{f[k-1][i],f[k-1][i-j*s[k]]+j}，其中 j 是选择使用第 k 件物品的数目，这个方程运用时可以用和上面一样的方法处理成一维的。求解时先设置一个粗糙的循环上限，即最大的物品乘最多物品数。

Money 是多重背包问题。也就是每个物品可以使用无限多次。要求解的是构成

一种背包的不同方案总数。基本上就是把一般的多重背包的方程中的 min 改成 sum 就行了。

Nuggets 的模型也是多重背包。要求求解所给的物品不能恰好放入的背包大小的最大值（可能不存在）。只需要根据“若 i、j 互质，则关于 x、y 的不定方程 i*x+y*j=n 必有正整数解，其中 n>i*j”这一定理得出一个循环的上限。

Subsets 子集和问题相当于物品大小是前 N 个自然数时求大小为 N*(N+1)/4 的 01 背包的方案数。

Rockers 可以利用求解背包问题的思想设计解法。我的状态转移方程如下：f[i][j][t]=max{f[i][j][t-1] , f[i-1][j][t] , f[i-1][j][t-time[i]]+1 , f[i-1][j-1][T]+(t>=time[i])}。其中 f[i][j][t]表示前 i 首歌用 j 张完整的盘和一张录了 t 分钟的盘可以放入的最多歌数，T 是一张光盘的最大容量，t>=time[i]是一个 bool 值转换成 int 取值为 0 或 1。但我后来发现我当时设计的状态和方程效率有点低，如果换成这样：f[i][j]=(a,b)表示前 i 首歌中选了 j 首需要用到 a 张完整的光盘以及一张录了 b 分钟的光盘，会将时空复杂度都大大降低。这种将状态的值设为二维的方法值得注意。

Milk4 是这些类背包问题中难度最大的一道了。很多人无法做到将它用纯 DP 方法求解，而是用迭代加深搜索枚举使用的桶，将其转换成多重背包问题再 DP。由于 USACO 的数据弱，迭代加深的深度很小，这样也可以 AC，但我们还是可以用纯 DP 方法将它完美解决的。设 f[k]为称量出 k 单位牛奶需要的最少的桶数。那么可以用类似多重背包的方法对 f 数组反复更新以求得最小值。然而困难在于如何输出字典序最小的方案。我们可以对每个 i 记录 pre_f[i]和 pre_v[i]。表示得到 i 单位牛奶的过程是用 pre_f[i]单位牛奶加上若干个编号为 pre_v[i]的桶的牛奶。这样就可以一步步求得得到 i 单位牛奶的完整方案。为了使方案的字典序最小，我们在每次找到一个耗费桶数

相同的方案时对已储存的方案和新方案进行比较再决定是否更新方案。为了使这种比较快捷，在使用各种大小的桶对 f 数组进行更新时先大后小地进行。USACO 的官方题解正是这一思路。如果认为以上文字比较难理解可以阅读官方程序或我的程序。

我将 bigbrn、range 和 rectbarn 划分成一类，它们都是要在 01 矩阵中寻找某种全为 0 或 1 的矩阵。

Bigbrn 最简单，需要在 01 矩阵中寻找最大的全为 0 的正方形，其状态转移方程为：f[i][j]=0（s[i][j]为 1）或 min{f[i-1][j-1] , f[i-1][j] , f[i][j-1]}+1（s[i][j]为 0）。f[i][j]表示以格子 s[i][j]作为右下角可以找到的最大的全为 0 的正方形。这个方程很好理解。

Range 并不是一个最优化问题，它的求解目标是统计各种大小的全为 1 的正方形的个数。我们完全可以利用 bigbrn 的方程求解。最后只需要进行简单统计处理即可。注意由于正方形是可以重叠的，f[i][j]=k 时表示以 s[i][j]为右下角可以找到大小分别为 1 至 k 的正方形。

Rectbarn 是第 6 章的题目，的确很难。它与 bigbrn 唯一的不同是这题中要求寻找的是面积最大的全为 0 的矩形而非正方形，这导致 bigbrn 的方法完全无法应用，但我们可以从另一个角度设计状态转移方程。设 h[i][j]表示以 s[i][j]为下端点可向上扩展的最大全为 0 的线段高度，l[i][j]表示此线段可向左扩展的最大宽度，r[i][j]表示此线段可向右扩展的最大宽度，则以 s[i][j]为底边上一点，先向上再向左右能扩展出的最大的矩形可以由以上三个值轻松求得。s[i][j]==0 时，有 h[i][j]=h[i-1][j]+1，l[i][j]=min{l[i-1][j] , lmax[i][j]}，r[i][j]=min{r[i-1][j] , rmax[i][j]}；其中 lmax[i][j]和 rmax[i][j]分别是 s[i][j]可向左右扩展出的最长全 0 线段宽度。s[i][j]==1 时，h[i][j]=0，l[i][j]=r[i][j]=∞。由于 h[i][j]、l[i][j]、r[i][j]等只是对 h[i-1][j]、l[i-1][j]、r[i-1][j]更新而求出，所以我们也只需要将它们保存为 1 维数组。另外，为了有效地随时求解 lmax[i][j]、

rmax[i][j]，也为了不超空间，我们也不应将整个矩阵保存成 s[i][j]这样的二维数组的形式，而是应该对每一行的 1 所在的位置建一张有序的表，进行离散化的处理。递推时保存当前点的最左边的 1 在表中的下标，以便顺利求解。

Buylow 的第一问是经典的最长下降子序列问题，其状态转移方程为：f[i]=max{f[j]}+1 (j<i , s[j]<s[i])，f[i]表示以 s[i]结尾的最长下降子序列的最大长度，其中第二个小于号若改变成大于、小于等于、大于等于就可以求出最长上升子序列、最长不上升子序列、最长不下降子序列。第二问是问不重复的最长下降子序列的个数。如果给定的序列中不包含重复的数字，那么我们可以轻易地想到这样的状态转移方程：n[i]=sum{n[j]} (j<i , s[j]<s[i], f[j]+1==f[i])。但当原序列包含重复的数字时，上述求解方法会对一些相同的子序列需重复计算，尽管子序列中数字在原序列的位置可能并不相同。我将上述方程修改成了这样：n[i]=sum{n[j]}–n[k] (j<i, s[j]<s[i], f[j]+1==f[i])，其中 k 为满足 k<i 且 f[k]==f[i]、s[k]==s[i]的最大的 k，这样就减去了可能重复统计的子序列。最后，由于结果可能很大，数组 n 的元素应该是高精度数。实现的一个技巧是在 s 序列的最后添上一个比序列的所有数都要小的数（本题中 0 就可以），这样可以简化编程。

将 charrec 这一有些复杂的题顺利解决的前提是将题目“抽象化”“模型化”，也就是从具体繁琐的题目叙述中跳脱出来。我设计的状态是：v[i][k]表示把第 i 行当成第 k 个字母的第 0 行时下面的最小差别，我们只需枚举这个字母占据了 19 行、20 行或 21 行时的情况即可，再用 l[i][k]纪录是这个状态的值达到最优时字母到底占据了几行用来输出最佳方案。具体的状态转移方程若写出来比较复杂，但很好设计，基本上就是枚举取哪个字母、哪一行重复了、哪一行遗漏了之类。

Game1 是很有趣的可用 DP 解决的博弈问题。设 f[i][j]表示先手面对 s[i]..s[j]这样一个序列时可以保证取到的最大值。那么有 f[i][j]=sum{s[i]..s[j]}–min{f[i+1][j], f[i][j-1]}。这个方程很好理解，我们有 s[i]和 s[j]两种选择，然后就是对手的先手，显然对方也会采取最优的策略拿到 f[i+1][j]或 f[i][j-1]的值，我们只需使 i 到 j 所有这些数中被对手拿到的最少即可。这有点像“极大极小搜索”的想法：将自己的所得极大化相当于将对手的极大所得极小化。

Nocows 这题真是很不好想。要求得出节点数为 N，高度为 K 的“家谱树”数目。如果设这个数目为 f[n][k]的话，那么实在是难以利用子问题得出它的数值。事实上这种设计状态的方法难以想出方程的原因可能就是因为“高度为 k”的条件太严格。如果我们将 f[n][k]定义为“节点数为 n，高度不大于 k”的家谱树数目，那么我们可以得到如下方程：f[n][k]=sum{f[l][k-1]*f[n-1-l][k-1]}。最后的答案就是 f[N][K]–f[N][K-1]。不妨认为这是一种“差分”的思想。

Theme 也是一道不太好想的题。定义 f[i][j]为以 s[i]和 s[j]开头的最长主题的音符数，则 f[i][j]=1（若 s[i]–s[j]!=s[i+1]–s[j+1]）或 f[i+1][j+1]+1（若 s[i]-s[j]==s[i+1]-s[j+1]）。具体实现时并不用开二维数组，只需根据状态间的依赖关系，枚举 i 与 j 的差不断更新当前值即可。

Tour 是一道有趣的题目，在这里我不写出它的状态转移方程。事实上在写这个程序前我也没有事先写出转移方程。我写这道题时的思考方式是“如果我们知道了某个状态，那么我们可以用它来更新哪些状态的值”。设 f[i][j]表示由起点到城市 i、j 走两条不相交路径时的最大城市总数。我们如果知道了 f[i][j]，那么我们可以用它更新 f[i][k]（如果 j、k 间有航线）和 f[k][j]（如果 i、k 间有航线）的取值，其中 k>i, j。最

后的答案是 f[N][N]–1（两条路线交汇于最后一个城市）。具体的实现方法可以看我的程序。我认为这是不同于“自顶向下”和“逐步递推”的第三种 DP 程序的实现方法，而且有时候更易于编程和思考。

Numtri 一题难度很小，vans 一题状态转移方程的难度太大我还不能完全理解，均不做具体分析。

总结一下：做 DP 题时，第一步是设计状态，这一般不难；下一步是设计能够依据已求出的状态求出新状态的状态转移方程，在这个过程中可以广泛联想借鉴经典的方程；然后考察得出的方程的求解过程是否满足时间空间的要求，考虑对其进行降维之类的优化。

第二部分　图论

USACO 中涉及到的图论知识有以下一些：遍历与连通性、最小生成树、最短路、欧拉路、最大流与最小割、二分匹配。

遍历与连通性

判断图的连通性的最常用方法是深度优先遍历(DFS)，其步骤是递归的：首先访问起点，对于每一个被访问的点，访问其所有未被访问的邻接点。要判断当前 A、B 两点之间的连通性，只需以 A 为起点做一次深度优先遍历，然后查看 B 点是否在遍历过程中被访问到。在使用边表保存边集的情况下，图中一次 DFS 的时间复杂度是 Θ(V+E)。（复杂度中的字母 V、E 分别代表图的顶点数和边数，下同。）

另外，如果需要得到图中所有顶点对之间的连通性信息（即所谓“传递闭包”），可以采用在最短路部分提到的 Floyd 算法。

Race3 是一道需要用到连通性的题。其第一问就是删掉之后起点与终点不再连通的点，其第二问是删掉之后图中出现两个互不连通的块（即它能到达与能到达它的两个点的集合之间没有任何边）的点。

图的 DFS 是强大的线性算法，除简单的求连通性外，还可以用来求割顶、桥、强连通子图，但 USACO 中并没有必须这样求解的题目。Schlnet 这题的两问的答案分别是没有入边的强连通子图的个数以及没有出边和没有入边的强连通子图数目中较大的一个。但这题的数据规模太小了，用 Floyd 都能过。

拓扑排序的算法与 DFS 非常像，时间复杂度也是线性的，可以用来解决 frameup 这题。

最小生成树

可使整个图连通的边的子集中权值和最小的称为最小生成树。最小生成树有两种常用算法：Kruskal 算法和 Prim 算法。Kruscal 是每次选择最小的边，如果不会造成环（用并查集判断）就加入解中；Prim 则是从所有和当前的树连接的边中选择最小的边加入解中。它们都属于贪心法，适用范围、效率、编程复杂度等都接近。我个人比较喜欢 Kruskal，它似乎更具灵活性和可扩充性。

USACO 里关于最小生成树的题只有一道，就是 agrinet，只需求出最小生成树的权值和。随便用以上两种方法中的一种都可以。

最短路

最短路是图论里值得详细说的一个话题，USACO 里有很多风格不同的考察最短路算法的题目。先介绍三种适用范围不同的常用算法：

Dijkstra 算法。这个算法可以求出图中某一个点到其它所有点的最短距离以及路径，但前提是图中没有权为负的边。它的大致思路是：每次找到当前距离最短的一个点，将它的距离固定下来，并用这个距离更新它的邻接点的距离，直至所有点的距离都被固定。采用最朴素实现的 Dijkstra 的时间复杂度是 Θ(V^2)。有一种高效的实现方法是使用一个最小堆保存所有顶点的当前距离，但这造成了需要更新当前距离时的不便，导致了实际中编程的复杂性，一般不考虑使用（事实上我还从来没有写过这种 Heaped Dijkstra）。

SPFA 算法。（我不打算提 Bellman-Ford 算法，因为完全可以把 SPFA 当作 Bellman-Ford 的一种优化、变形以及竞赛中的替代品。）同样是用来解决单源最短路，图中可以有负权的边。使用一个队列，首先使源点入队，然后每次出队一个顶点，

用这个顶点的当前距离更新它的所有邻接点的距离，所有距离实际上被更新且未在队列中的点入队。重复以上过程直至队列空。另外，当一个点入队次数超过图的顶点数时，表明图中存在负权环。SPFA 的最坏情况，也就是图中有负权环时的时间复杂度是 Θ(V*E)，但在实际应用中若没有负权环时会非常快，甚至可以认为大约与 O(E)同阶。我在用到 SPFA 时一般都采用边表来来存边，这样可以最大限度发挥 SPFA 的优越性。

Floyd 算法可以比较高效地求出图中所有顶点两两之间的最短距离，有负权边也没问题。它的基本思想是枚举每一个顶点，试图用“松弛操作”将它加到最短路径中去。具体的算法不太好用纯语言来描述，所以我就不描述了，保证看代码能一看就懂。它的一大优点是程序非常短且非常不容易写错。所以说，对于一些简单题，图的顶点比较少，特别是在还会有负权边的时候，哪怕需要求的是单源最短路，写一个 Floyd 也是明智的。Floyd 的时间复杂度是雷打不动的 Θ(V^3)。

不管使用什么算法，如果要求得到最短路的具体路径，可以对每个顶点增加一个 pre 域（在 Floyd 中，pre 需要是一个二维数组），以保存到达这个顶点的最短路的上一个顶点的编号，这样就可以沿着 pre 提供的信息，逆向找到整条最短路。

Butter 一题虽然是需要求所有的顶点两两之间的最短距离，但我仍然是写了 SPFA，对图的每个顶点执行一次。因为图的顶点可以达到 800 个，而边数最多只有 1450，是典型的稀疏图。800 个顶点的 Floyd 在 USACO 的评测机上据说刚好超时。对于稀疏图来说，最主要的优化方法就是采用边表而非邻接矩阵保存图的边，Floyd 算法无法采用这种方法进行优化，故 Floyd 对于稀疏图是不适用的。采用了边表的 SPFA 在本题这种稀疏图的情况下非常快。另外一种 AC 此题的方法是 Heaped Dijkstra，不过编程复杂度显然在 SPFA 之上。这道题可以看出 SPFA 的优越性和

Floyd 的一个局限性。

Camelot 这题也需要先求一个所有点对间的最短路以便后面的枚举，但这题的顶点数不多，所以在就选择编程最简单的 Floyd 了。

Comehome 是“单汇最短路”，不过由于无向图，其实还是单源最短路。我写的是 Dijkstra。

Cowtour 需要枚举添加的边，求好多次所有点对间的最短路，我采用的是 Floyd。

Maze1 可以求两次单源最短路，每个顶点的最终“距离”是两次求得的值中较小的一个，采用 SPFA 或者 Dijkstra 估计都没问题。但这题还有更更好一些的解法：在 SPFA 的开始，把迷宫的两个出口都加入队列，最终求出的每个点的最短距离肯定就是该点到两个出口中较近一个的距离。如果想用 Dijkstra 求解事实上也可以用同样的技巧只求一次单源最短路。

另外还有一点，对 Floyd 算法稍作改动可以求图的最小圈。Fence6 一题我就是这样写的（然而，前面的构图过程相当麻烦）。具体方法是：在每次未用 k 结点更新 j 与 i 之间的最短路时检查 k->j->i->k 这一圈，其中 k 与 j 及 i 与 k 有直接的边连接。还有一种求最小圈的方法是：对于图中的每一条边 i、j，删掉这条边再求 i、j 间的最短路，就得到了一个圈的长度，fence6 的标程就是这样写的，这显然不如前述方法效率高。

欧拉路

欧拉路是图论中很特别的一个算法，这是因为它似乎完全不跟图论中的其它算法有什么联系。首先要知道图中有欧拉路的充要条件：对于无向图，所有顶点的度数为偶（欧拉回路）或有且仅有两个顶点的度数为奇（欧拉路径）；对于有向图，所

有顶点的入度与出度相等（欧拉回路）或者是仅添一条边就可以满足前述条件（欧拉路径）。当然，充要条件还应包括整个图是（强）连通的。

求欧拉回路 / 欧拉路径的一种算法是类似于 DFS 的递归算法：从任意一个顶点 / 奇数顶点开始，每次访问图中的一个顶点，对于它的每条边（出边），先删掉这条边再对边的另一个端点进行访问，直至该点没有邻接边（出边），每个顶点的上述过程进行完毕后记录一次这个点（事实上，每个顶点可能被记录多次），最后逆序输出记录的顶点即为欧拉回路 / 路径。

用上述算法可以解决 fence。

最大流与最小割

在有向图中，规定一个源点、一个汇点，给每条边规定一个“最大流量”，求从源点到汇点最多能输送多少流量，要保证除源和汇外每个顶点的输入和输出流量必须相等，当然每条边上的流量也不能超过最大流量。这个问题就是最大流问题。

求最大流的一类常用算法属于增广路方法，即每次找到源点至汇点的一条路径，这条路径上所有的边都减去最小的那条边的权，所有的反向边加上相同的值，直至源点与汇点不联通。找这条增广路可以采用 BFS、DFS 等。我一般采用 BFS 找增广路的方法，即所谓的 Edmondes-Karp 算法，时间复杂度是 O(VE^2)。

在图中删去一些边可以将这个图分成分别包含源点和汇点的两个不连通的部分，最少需要删去权值和为多少的边的问题就是最小割问题。根据最大流最小割定理，最大流的流量值等于最小割的值。可见最大流和最小割密切相关。

USACO 中与最大流及最小割有关的题目有：ditch、milk6、telecow。

Ditch 就是最纯粹的的最大流模型。基本上用什么算法都能过。就是用来检验你

写的最大流算法的正确性的。

Milk6 则是最明显的最小割模型。第一问要求输出最小割的数值自然很简单，由前述定理可知它等于最大流的数值即可。然后要求最小的割集，而且要求边数最小且字典序最小，这一问就相对麻烦。如果是需要求任意一个权值最小的割集，需要考察每一条边，如果它在残留网络中一个端点与源点连通，另一个端点与汇点连通，就选出来。如果是仅仅要求“边数最小”，可以将每条边的权值乘以顶点数再加一，得到的割集肯定就是边数最小的。但是本题中要求“字典序最小”的割集我还没有想到足够好（与最大流的复杂度同阶）的方法。本题我采用的方法是先将边集按照权值由大到小、再字典序由小到大的方法排序，求出初始最大流后，按照前述顺序每次去掉一条边，如果流量的减少值恰等于这条边的权值，就将这条边加到所求割集里，否则把这条边再恢复。

Telecow 要求的是最少去掉几个顶点能使源和汇不连通，粗略一看与最小割模型不符合，但我们可以采用“拆点为边”的方法进行转换。对于每个点 A，拆成两个点 A 和 A’，添加权值为 1 的边 AA’。对于原图中每条边 AB，再新图中以流量为∞的边 A’B 表示。最终求出的最大流的数值即为第一问的答案。在转换了的图中，第二问也就相当于求和上题一样的字典序最小的割集，我使用的方法也是一样的。

二分匹配

给定两个顶点集合 X 与 Y 以及一些 X 中某顶点与 Y 中某顶点“匹配”的关系，求最多能找到多少个互不相交的匹配的顶点对，这就是二分匹配问题。USACO 中的 stall4 就是单纯的二分匹配问题。

解决二分匹配问题，可以将它转换成最大流问题求解，方法是：添加源点 S 和

汇点 T，对于 X 集合中的每个顶点，添加从 S 出发的一条边，对于 X 集合中的每个顶点，添加指向 T 的一条边，对于 X 与 Y 的每个匹配关系，添加一条由 X 中顶点指向 Y 中顶点的边，以上边的权值均为 1。在这个图中求出的最大流的数值即与原最大匹配的数值相等。

然而，使用一般的最大流算法求解二分匹配问题的效率是比较低的，由于最大流的代码一般都不短，前述方法在实际编程中的“性价比”不高。事实上，还有一种易于编程且效率很高的专门求解二分匹配问题的算法：Hungary 算法（也就是“匈牙利算法”）。它的基本思想是通过 DFS 在二分图中找“交错轨”。但事实上，我认为掌握这个算法甚至根本不需要理解“交错轨”这个概念，它似乎和“决策树”“隐式图”类似，只是为了理解算法的本质而抽象出来的一种东西，在代码中不会出现，也不影响对代码的浅层理解和记忆。

算法的核心是 bool find(int v)这个函数，它的作用是：寻找顶点 v 可能匹配的顶点。对于每个可以与 v 匹配的顶点 j，假如它未被匹配，可以直接使用 j 与 v 匹配；假如 j 已与某顶点 x 匹配，那么只需调用 find(x)来求证 x 是否可以与其它顶点匹配，如果返回 true 的话，仍可以使 j 与 v 匹配；这就是一次 DFS。每次 DFS 时，要标记访问到的顶点（vis[j]=true），以防死循环和重复计算；每次 DFS 开始前所有的顶点都是未标记的。主过程只需对每个左侧的顶点调用 find，如果返回一次 true，就对最大匹配数加一；一个简单的循环就求出了最大匹配的数目。

附：图的表示

在图论问题中，保存顶点之间的相连关系有两种常用的方法：邻接矩阵与邻接表。邻接矩阵即定义二维数组 s，将顶点 i 与顶点 j 之间的权值保存在 s[i][j]中。邻接

表即对于每个顶点 i，维护一个与它连通的边的列表。

邻接矩阵是最直观、最容易理解的方法，有些算法（如 Floyd）要求必须采用这种数据结构才能实现。但它也有致命的缺点：占用空间比较大，是 Θ(V^2)的。对于所谓“稀疏图”，这类图中的边数相对于顶点数的平方而言非常少，若用邻接矩阵保存则会有很多空间根本没有利用，而且在竞赛中的内存限制可能不允许开那样一个二维数组，这时就只能用邻接表了。

邻接矩阵的另一个缺点是，若算法中有很多“对于与顶点 A 相连的每一条边如何如何”这样的操作，每一次操作的复杂度均为 Θ(V)，会严重影响程序的时间效率。例如，我们都知道深度优先遍历的时间复杂度是 Θ(V+E)，是优秀的线性算法。但是你还应该知道，得到这个线性复杂度的前提是边需要采用邻接表保存，如果采用邻接矩阵，那么复杂度就变成了 Θ(V^2)。

上面的论述也并不表明邻接表就是完美无缺的，它也有它的缺点。例如，在邻接表中，如果我们只知道两个顶点的编号，我们无法快速的定位这两个顶点间的边，这是邻接矩阵可以轻易做到的，这也是 Floyd 算法不能应用于邻接表的原因。其次就是邻接表的编程复杂度以及出错的可能性显然要比邻接矩阵高。没有两全其美的事情，OI 的世界里亦是如此。

对于邻接表，我看到的很多资料里都将它描述成一组“链表”。而我在前面对于临界表的定义时却用了相对模糊的“列表”一词。这是因为，在实际应用中，我很少用链表这种动态分配内存的数据结构实现邻接表。我采用的方法，一般是对于每个顶点有一个足够大的数组保存它的邻边，再记录它的邻边的实际数量，有时也会用 C++ 中的 basic_string（即使在中国的 OI 比赛中，这也是不被禁用的）。的确，现在的竞赛中的内存限制并不都像 USACO 上那么严格，一般都是 64M 以上。在这种环境下，

采用邻接表的主要原因是为了提高算法的速度而非节省内存。不过用链表保存边集也并不是用不到的东西，纯动态的结构毕竟更灵活，借助于“构造函数”，在 C++中可以将链表形式的边表实现得非常美观。

有时候，在空间允许的情况下，我们可以尝试将邻接表与邻接矩阵进行“杂交”，以实现它们各自的优点。例如，我们可以在邻接矩阵的 s 这个二维数组的基础上增加二维数组 next，next[i][j]表示 i 至 j 的这条边（如果有的话）的后一条 i 的出边是什么。使用这种方法优化原来用邻接矩阵写成的 Edmondes-Karp，解决 stall4 这题的效率大约提高了十倍，因为每次 BFS 的速度都大大提升了。另外这样一来反向边也容易找到，只是要注意一开始要把反向边也串进 next 数组所定义的模拟链表里。

还有一个关于邻接矩阵实现中的细节问题。在 USACO 的很多题目中，给定的图并不是简单图，可能两点之间有多条边。这时，在读入时要加倍注意，要根据题目的类型分别处理。如果是最短路，肯定是在读入重复的边时选择权值最小的保存；如果是欧拉回路，可以在邻接矩阵中保存这条边出现的次数；如果是最大流，则应该将重复的边的权值加在一起。

还有一种比较特殊的情况可能以上两种图的表示方法都会不太适用：点很多，不可能邻接矩阵；每个顶点的边数没有上限，不能用数组模拟的邻接表；由于某种原因需要避免动态内存分配也不能用链表实现的邻接表。这时就是“前向星”隆重登场的时候了。前向星很简单，就是将所有的边放到一个数组里面，所有的边输入完之后将边按照起点排一下序，记下每个顶点的边表的开始位置，前向星就构造完成了。除了不能像邻接矩阵一样直接根据起点和终点定位边，它几乎是完美的，但一般还是只在其它表示方法都不可能应用的前提下再去考虑它，毕竟它的预处理是略显复杂的。

第三部分　搜索

USACO 中需要使用搜索解决的题目不少，我认为十分有价值的包括：1latin、betsy、checker、cryptcow、fence8、msquare、prime3。

统计可行方案的深度优先搜索

若需要统计可行方案的数量或输出所有可行方案，基本框架就是利用 DFS 枚举出所有方案来判断是否可行。

1latin 是典型的统计方案数型的题，这显然要用 DFS 解决，因为你要枚举出所有的方案。我采用的优化：1.将第一列同样固定成 1~n，结果乘以(n-1)!即可；2.最后一行不用搜，一定有解；3.前两行可以当作一个置换群，当置换群同构时，下面的方案数一定相等，这就可以采用记忆化的方法加以优化。

Betsy 要求统计矩阵中起点和终点一定的 Hamilton 路的个数，所以只能 DFS，不剪枝会超时得很惨，但剪枝条件不够多同样会很惨。我使用的剪枝：1.记录每行、每列被访问的格子数，若当前下面一行或左面一列全被填满了，则剪枝。2.若当前在第一列或最后一列，而上面的那个格子还未访问过，则剪枝。3.若当前在第一行且左面的那个格子未访问过，或当前在最后一行且右面的那个格子未访问过，则剪枝。4.若左右两个格子都被访问过，而上下两个格子都未被访问过，或对称的情况，则剪枝。5.未访问当前格子时，若上下左右格子中存在“死点”（即未被访问且周围有三个以上格子已被访问的顶点，目的地任何时候不算死点），则剪枝。6.访问了当前格子后，若上下左右格子中存在死点，则只考虑访问那个点的情况。通过合理组织程序，可以将上面的剪枝方便快捷地实现。

Checker 要求统计 n 皇后问题的方案数。在随时记录每列及每条斜线是否被占用的前提下已经没有什么剪枝的余地。可以采用的优化方法是用链表将 1~n 串起来，每次放皇后时都将列的编号去掉，下面枚举放在哪一列时只用枚举链表中的数，这样大大提高了效率。如果用 C++编程的话，仅仅加上这一个优化就可以卡着时间 AC 了。另外的优化手段是利用减少统计的范围：若 n 为偶数，只统计第一行的皇后放在左半边的情况，最后的结果乘以二；若 n 为奇数，需要统计的是第一行的皇后放在最中间和左半边的结果，答案就是前者加上后者乘以二。这道题还有一种很剑走偏锋的使用位运算优化的方法，不过我还没有透彻地理解。

Cryptcow 是典型的这类问题。基本的搜索框架很好想：枚举原字符串中的 C、O、W 进行转换，最后判断是否转换到了目标字符串。但是不加优化和剪枝会超时无比严重。我采用的优化：将字符串用链表保存，这样就可以将每次转换用 Θ(1)时间完成。我采用的剪枝比较多（专门为剪枝写了一个函数）：1.第一个“特殊字符”（指 C、O、W，下同）不是 C 或最后一个不是 W，剪枝。2.当前字符串第一个特殊字符之前的部分必须是目标字符串的前缀，最后一个特殊字符之后的部分必须是目标字符串的后缀。3.当前字符串中每个不包含特殊字符的子串必须是目标字符串的子串，采用一个 Suffix Tree（即加入了目标字符串所有后缀的 Trie）来判断。4.不处理冲突的乱写的 hash 函数判重。另外，这题有类似“构造法”的解法，在 USACO 官方题解的第二个程序里，但是编程复杂度极高，我没透彻地弄懂。

Prime3 这道题是我所有的 USACO 程序中最大的一个，约有 7k。比较特别的地方：1.搜索顺序是先对角线、再边缘、再中间。2.主要的剪枝依靠预处理得到的 V 数组，它能告诉你哪怕不完整的一行一列有没有可能是素数。3.将“两边两数已确定且没有 0”和“两边两数已确定且仅有 1379 组成”的东西建一张表，在搜索边缘的时候可

以大大加快枚举速度。

Milk3、sprime、wissqu、zerosum 等是这一类题目中较简单的，不用多说。

广度优先搜索

事实上，BFS 比 DFS 的适用范围要窄，USACO 中的题目也不多。它主要适用于求“步骤最少的方案”。它的实现方法不用递归，需要用到一个队列：将初始状态入队，每次出队一个状态对它能扩展成的状态入队，直至达到目标状态。

Msquare 是一道典型的 BFS 题目。它的特点是状态数很少，仅 8!=40320 种，这道题最主要的剪枝手段是避免重复扩展同一个状态，但在队列中顺序查找一遍的方法显然不现实，所以考虑使用 hash 判重，hash 函数采用康托展开。这是广搜中典型的优化方法。

Shopping 一题我也采用 BFS 解决。其实我写的时候总在感觉我好像在写 SPFA——目的是找到一条由所有物品都没买的状态到买齐所有物品的状态的最短路径，路径的边就是单买某种物品或某个 profit。需要用一个龌龊的六维数组来保证没有状态在队列中重复出现。由于并不是所有的边权均为一，所以与一般的 BFS 不同，并不是第一次找到目标状态时就是最优解。其实这题采用自顶向下的记忆化搜索也是可以顺利的。

BFS 最主要的优化方式是所谓“启发式搜索”，例如 A*方法。启发式搜索的特点是对于待扩展的结点进行“估价”，选择最有希望尽快得到最优解的状态进行扩展。但我在所有的 USACO 训练题中并没有用到启发式的搜索方法。

求解最优方案的深度优先搜索

对于求解最优方案类的问题，可以使用深度优先搜索枚举出所有的方案，就可以找到最优的一个。当然，在实际应用中远不止这么简单。

Cowcycle 的基本框架是很简单的枚举，只需在最内层的排序做一点优化就可以 AC。不过恼人的是这题目的数据范围很有故弄玄虚的感觉——没有极限数据，如果有极限数据的话我就不知道可以怎么弄了。

Fence8 是我认为很有价值的题目。它的基本框架可以采用 ID-DFS，即迭代加深搜索，每次增加试图切的 rails 的数目，直至不能再切出更多的 rails。优化有：1.对 rails 都进行排序，因为显然切小的 rails 更划算。2.对 fences 进行排序，这个似乎优化不大，但可以方便下面的剪枝。剪枝有：1.对于重复的 fences 或 rails 可以剪枝，避免重复搜索。2.如果某个 fence 剩下的可用值与当前 rail 完全相同，不再考虑将当前 rail 放入其它 fence 中的方案。但是这样似乎还是会略微超时，所以需要二分答案，注意一定要在事先尽最大可能缩小答案的上限，否则可能会死得更惨。

Snail 一题看上去很难，因为矩阵可能达到 120x120 的大小。但事实上写一个最朴素的 DFS 就可以解决，这是因为路径的数量事实上收敛得非常快。

Clocks、hamming、hostain 等题也属于这个类型，但没什么好说的。毕竟它们真的很简单，只是为了让你练习 DFS 的框架而已，剪枝都不怎么用加。

剪枝

剪枝，就是去掉搜索树中肯定不通向问题的最优方案或可行解的枝条，分为最优性剪枝和可行性剪枝。

我认为 camelot 不是搜索题（在我看来，“搜索题”是一定要用到保存状态的栈或

队列的），但能说明最优性剪枝的方法。需要枚举的是所有人的汇合点以及骑士如何背上国王。在枚举过程中，如果仅仅是所有人在该点汇合而不考虑骑士可以背上国王这个条件所得的步数就已经超过了当前最优解，那么可以剪枝，也就是不再枚举这个汇合点下由哪个骑士背国王以及在哪里背上国王。可以在所有的枚举开始之前就用所有骑士的中心点以及国王所在点分别当作汇合点尝试一下，则可以在一开始就得到一个比较好的答案的上界，十分有利于完整的枚举过程中的剪枝。

至于可行性剪枝，上面提到的 betsy 和 cryptcow 是这方面的典型题目，可以仔细体会。在写 betsy 这个题目时，我采用的一个技巧是让程序把一些没有找到任何可行解的中间状态输出到 log 里，分析其共性就可以总结出一些有用的可行性剪枝的方法。

一般的剪枝思路是剪去搜索树中多余的枝条，也就是“判断一定不可行的状态”，但有时也可以从不太一样的角度考虑，例如上述的 betsy 中第 6 个剪枝和 fence8 中第 2 个剪枝，都是从“减少当前可选择的策略数”的角度出发的。

搜索题目很多变。由于“剪枝”事实上极具技巧性，故它的特点是写出一个“正确”的程序不难，但对于难题来说写一个能够得满分或一次 AC 的程序还是相当有难度的。

记得初学搜索时觉得是一种很神秘的东西。那么，搜索到底是什么？我现在认为，搜索的本质不过就是枚举而已，枚举所有的策略、所有的路径、所有的可能或别的什么东西。所以搜索这个话题虽然外延很大，但它的内涵其实很简单。

第四部分　其它类型

有一类题目可以在程序中用循环的方式穷尽所有变量取值的可能性，我认为这属于枚举的题目。本类的题目有：calfflac、camelot、castle、crypt1、dualpal、palsquare、preface、ratio、runround、spin。这一类里没有什么难题。

对于只要将题目中描述的操作转换成正确程序语言叙述就可以的问题，我归类为“模拟”。USACO 中此类题目不少，靠前的章节尤甚，计有：beads、friday、gift1、milk2、namenum、packrec、preface、ride、runround、spin、transform、ttwo、

milk2 值得说一下。也许你会认为纯模拟的算法不可能 AC，但是我的确就是用纯模拟算法 AC 的，最大数据也只需要 0.024s。我像所有用模拟做这道题的人一样，开了一个 1000000 的 bool 数组，只不过我在填充这个数组时使用了 memset 函数。由于 memset 函数的调用在编译时可以转换成一条汇编指令，有时可以认为是在常数时间内完成的，就大大加快了模拟的速度。看来模拟也是需要一点点小技巧的。当然，更“标准”的方法是使用区间加法，先要将所有的区间按起点排序，每次试图用后面的区间和前面的区间合并，就可以找到最大的区间。

USACO 中有一类统计性的问题，往往需要借助某种方便的数据结构解决。

Trie 是一种很好用的数据结构，在字符串的查找中用处很大。它是每个节点表示一个字母的树，每个节点代表自根到该节点的路径上所有字母连成的单词。我使用 trie 作为程序的核心解决的题目有 preface、contact。

Preface 是用 trie 保存所有待查找的字符串，对于每一个已经验证合法的前缀试图利用 trie 向后扩充，如果 trie 树中不存在合法的分支就停止。

Contact 要保存的是 01 串。如果建立 trie 的话它就是严格二叉的。这让我联想到了二叉堆的保存方法。将空串当成根，每个节点的左儿子表示在最后加上一个 0，右儿子表示加上一个 1 这种结构很便于添加与统计。

有两道题可以使用矩形切割算法，它主要用来解决一组矩形在坐标系上堆叠放置构成的面积问题。具体算法就是对于放在下面的矩形不断的“上浮”，每次遇到阻碍它的矩形，就将原矩形切割成几块让未被阻碍的部分继续上浮，具体可以看 rect1 和 window 的代码。

Picture 一题也与坐标系上堆叠放置的矩形有关，但它要求的是最终形成的图形的周长。一种做法是先将所有的边离散化再利用线段树统计，但这种方法的编程实在很繁琐。我采用的方法编程很简单，代码也简短，主要的统计过程只用了一个一维数组，但也可以很快 AC。具体是这样的：预处理（可以在输入时就处理）需要将每个矩形的四条边分开，横向的与纵向的放到两个数组里分别统计，还需要对每条边标出是开始的（左边、上边）的边还是结束的边。以统计纵向的边为例，将这些边按照横坐标从小到大排序，从左向右扫描，每次遇到一条线段将它所覆盖的纵坐标上的区间所对应的数组中的值加一（开始边）或减一（结束边），每次改变值时由 0 变成 1 或由 1 变成 0 的次数加到最后的结果里。具体实现时也有一点技巧，可以看我的程序。

贪心是一种重要的算法设计思想，它的核心是在每一步选择当前最优的策略，最终可以得到整体最优的结果。当然并不是所有的题目都可以这样做，贪心策略的正确性还是需要证明的。我在 USACO 中用到贪心策略的题目有：barn1、job、milk。

Job 这个题目值得一提。第一问很简单，采用十分简单的贪心策略（对每一个任

务，选择能最快完成当前任务的机器）就能得出正确答案。第二问我起初考虑的多种贪心策略都找到了反例。事实上，如果从“策略”的角度来想，也就是说每一个从 A 机器出来的东西要放到哪个 B 机器里，会完全无从下手。可以将两问分开来，也就是先假设两个任务是彼此独立的。根据时间的对称性，将费时长的 A 任务与费时短的 B 任务配对就可以了。

计算几何是 OI 中一类比较独立的问题。它的基本思想是利用向量的叉积、点积来判断相交、方向等。USACO 中有 fence4 和 fc 两道属于计算几何。

Fence4 用到的主要算法是判断线段是否相交、求交点等。只需要判断由观察者和某条线段的端点及中点构成的线段与某线段的交点与观察者最近，就找到了一条可以被看见的线段。

Fc 是凸包问题，我采用的是 Graham’s scan 算法。但是我认为 USACO 的课文里对于这种算法的实现并不好，又难理解又难记。《算法导论》里面的实现很好，我现在的程序就是按照那里面的伪代码写的。它的大致方法是这样的：首先，找到所有点中最左边的（y 坐标最小的），如果 y 坐标相同，找 x 坐标最小的；以这个点为基准求所有点的极角（在 C 语言里面就是 atan2(y-y0,x-x0)），并按照极角对这些点排序，前述基准点在最前面，设这些点为 P[0]..P[n-1]；建立一个栈，初始时 P[0]、P[1]、P[2]进栈，对于 P[3..n-1]的每个点，若栈顶的两个点与它不构成“向左转”的关系，则将栈顶的点出栈，直至没有点需要出栈以后将当前点进栈；所有点处理完之后栈中保存的点就是凸包了。如何判断 A、B、C 构成的关系不是向左转呢？如果 B－A 与 C－A 的叉积小于 0 就不是。这种方法的巧妙之处是它在预处理时能保证 P[n-1]、P[0]、P[1]一定是凸包的顶点，这样就省掉了很多繁琐的步骤。

解决题目 hidden 需要用到最小后缀算法。基本思想是这样的：定义 v[1..n]，其中 v[i]=k 表示 s[i..i+k-1]在 s 的所有长度为 k 的字串中最小。我们求出每个 v[i]最大的取值，再找到所有的 v[i]中最大的一个，s[i..n]就是问题的答案。求解 v 的过程是：先将所有的 1..n 都保存在一个链表里。每次扫描列表里所有的 i，若其 v[i+v[i]]>0，则可以将 i+v[i]从链表里删去，v[i]+=v[i+v[i]]；若其 v[i+v[i]]==0，则 v[i]++当且仅当 s[i+v[i]]在所有 s[k+v[k]]中最小（其中 k 是当前列表里的值）。然后删去所有链表中不等于 v[i]最大值的 v[i]。这基本上是一个不断合并/扩大区间的过程。当所有 v[i]值不再变化时结束。将原字符串复制两遍，求其最小后缀即可。

Cowxor 一题的算法的思想与此题有些类似。都是不断地将不可能成为最优解的编号排除掉，并将两个区间的结果合并的过程。

Kimbits 与 twofive 这两道题的思想有些相似，它们可以认为都是康托展开的加强版本，基本思想是相同的。

先说 kimbits，我们可以用组合数学的知识计算出当前几位都确定之后下面一位填 1 之后还有多少种方案，如果当前的方案序号加上后面的方案数比所求的序号要大，所以下面一位应该是 0。

Twofive 的基本思路和上题是一样的，只是确定下面还有多少方案数时要用到搜索，而且还要用记忆化的方法进行优化。

Fence3 很有意思，它是在连续的二维空间里搜索某个最优的点。可以采用多次调整并不断缩小调整的尺度的方法。每次如果在当前的尺度下已无法调整为更优就缩小尺度，直到达到所需的精度要求。

USACO 是我认为最适合初学者的题库。他的特色是题目质量高，循序渐进，还配有不错的课文和题目分析。做完了 USACO 以后，我感觉“内功”有了很大提高，特别是知识的系统性、对算法认识的深刻性得到了加强。USACO Training 就像一个私人教练，指导你的 OI 征程。如果你在 OI 的世界里徘徊，犹疑于努力的方向，沉下心来一道道攻克 USACO，一定是不错的选择。