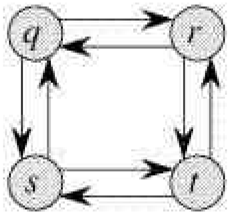
动态规划（Dynamic Programming, DP）

DP大致简述

能用动态规划解决的问题，需要满足三个条件：最优子结构，无后效性和子问题重叠。

最优子结构，举个例子，最短路,下图边权均为1



如图:q->t的最短路满足最优子结构，他由q->s的最短路转移过来

但是最长路并不是最优子结构，例如q->t的最长路实际上是由q->s的最短路转移过来的，不满足最优子结构，因为q->s的最长路显然是q->r->t->s

DP用图论的概念我们可以抽象成一张有向无环图(DAG)，边相当于决策方案，求解的过程相当于在做一个拓扑排序

无后效性，就是已经求解的子问题，不会再受到后续决策的影响，从DAG中的理解就是无环，例如状态a可以推出b，b可以推出c，c可以推出a，这个就不满足无后效性,因为后续的决策c会对a产生影响

子问题重叠性,大量的重叠子问题，我们可以用空间将这些子问题的解存储下来，避免重复求解相同的子问题，从而提升效率。

DP能做什么

dp能写的东西其实很多，一些贪心的题目，你用DP也能够去解决，因为某种意义上dp相当于做了最优的决策

然后还有一大类DP的题目，比较常见的有LCS(最长公共子序列),LIS(最长上升子序列),LCP(最长公共子串),记忆化搜索，背包DP，区间DP，树形DP，状压DP，数位DP，计数DP，概率/期望DP等，涉及范围较广。

除此之外还有一些DP的优化算法，本质是利用DP函数的单调性等优化决策转移的速度

比赛中出现的频率也是很高的那种，签到到牌区的题都有。

今天我们介绍下几个简单的例子入门LCS,LIS,LCP,背包DP

LCS,LIS,LCP

首先区分下两个概念，子序列和子串。举个例子，例如一个字符串$abcdefghijk$

我们从头和尾部删除一部分，留下中间连续的一部分内容，这部分被称之为子串

然后我们按照顺序依次拿出$p_1 < p_2 < p_3 ... < p_m$（$1 \le p_i \le n$）,这些位置组成的序列被称为子序列

LCS(最长公共子序列)

```
abcfbc
abfcab
```

按照这个例子，我们可以发现最长公共子序列是abfc

我们可以设计DP方程$dp[i][j]$代表第一个字符串的$[1,i]$的子串和第二个字符串$[1,j]$的子串的LCS，当$s_1[i]$和$s_2[j]$位置相同时，我们可以产生一个贡献，他由$dp[i-1][j-1]$转移，如果两者不同，我们要从最大的那个转移，就是$max\{dp[i-1][j],dp[i][j-1]\}$,例如$dp[3][2]$这个状态，由于$s_1[3]$和$s_2[2]$不同，我们要么选择$s_1$少一个或者$s_2$少一个的状态转移。

$$dp[i][j]=\begin{cases}dp[i-1][j-1]+1,\ s_1[i]=s_2[j]\\max\{dp[i-1][j],dp[i][j-1]\},\ s_1[i]\neq s_2[j]\end{cases}$$

初始态显然是$dp[0][0]=0$

LCP(最长公共子串)

子串和子序列的区别实际上就是必须要连续，所以在$s_1[i]\neq s_2[j]$时，贡献会被清零，我们定义$dp[i][j]$代表第一个字符串的$[1,i]$的子串和第二个字符串$[1,j]$的子串的LCP

$$dp[i][j]=\begin{cases}dp[i-1][j-1]+1,\ s_1[i]=s_2[j]\\0,\ s_1[i]\neq s_2[j]\end{cases}$$

从转移方程可以看出，$dp[n][m]$不一定是最优解，所以要对全部的状态取$max\{dp[i][j]|1\leq i\leq n,1\leq j\leq m\}$

LIS(最长上升子序列)

我们定义$dp[i]$为已i结尾的LCS，那么我们$dp[i]$如果要从$dp[j]$转移的话，首先要满足上升子序列的这一个性质(基于题目，可以取到等于，看是否是严格递增)，$a_i>a_j$满足上升这一个性质，然后此时能够转移的状态应该有很多位置满足这一个性质，这时候要让当前状态最优，就要取一个最大的$dp[j]$进行转移，所以我们需要用$O(n)$的时间便利数组实现转移

$dp[i]=max\{dp[j]|(1\leq j\leq i-1)\wedge(a_i>a_j)\}+1$

背包DP

背包DP有三种比较简单的模型

01背包

一般的模型就是有n个物品，每个物品重w_i，有v_i的价值，你有一个背包容量为V，求最大价值，每个物品只有取或者不取两种状态

我们一般会设置$dp[i][j]$为当前考虑到第i件物品，背包容量为j的最大价值，那么对于第i个物品我们有两种决策方式，第一种取，那么如果要取第i件物品，第$i-1$的状态时，他的体积必须是$j-w_i$才能够放下这件物品，并且我们获得了v_i的额外价值，如果不取，那么直接从$dp[i-1][j]$转移即可，然后对于两种决策我们要取到一个max，达到最优决策，所以dp方程就比较容易理解为

$dp[i][j]=max\{dp[i-1][j],dp[i-1][j-w_i]+v_i\}$

然后讲个比较简单的优化，也是常用到的，假如$n=10000$，$V=20000$,那么此双重循环时间实际上是够的，但是$2e8$的空间一般是开不下的，然后我们发现上述的dp方程，其实第一维并不是很重要，我们可以有两种方式把他优化掉

第一种，直接舍弃掉他，然后再做循环枚举的时候要注意顺序，假如我们从小开始枚举，我们把前面的数组更新了就会对后面的产生影响，所以要从后往前更新，因为这个式子大的体积的答案并不会影响小的

第二种，我们可以使用滚动数组优化，因为当前状态只和前一个状态有关，更早之前的其实并不影响，所以只需要记录前一个数组的信息即可

完全背包

完全背包和01背包的本质区别就是，每个物品是无限的

我们设计$dp[i]$代表容量为i时的最大价值，那么当前如果选择放置重w_j，j价值为v_j的物品，那么就要从$dp[i-w[j]]$转移过来，显然，这里有一个条件是$w[i] \leq i$，然后此时你可以选择放置n个物品中的任意一个，你有n种决策方式，从中选择最优解即可

所以，$dp[i] = max\left\{dp[i-w[j]] + v[j] | (1 \leq j \leq n) \wedge (w[j] \leq i)\right\}$

多重背包

现在给你一个容量为V的背包，有N个物品，其中第i件物品的重量为w_i，价值为v_i，第i件物品一共有s_i件，问在有限的容量内，最多可以拿到多少价值的物品。这是一个比较典型的多重背包例子，和01背包的区别就是每个物品多了一个件数的条件。

一般来说他会把$\sum s_i$的总和开的很大，如果你按照01背包的做法时间就会变成$O(V * \sum s_i)$会被卡tle

二进制拆分优化

假如当前物品有s_i件，我们可以根据二进制可以表示任意数字的思想，将其捆绑成整体做01背包

特殊地，若s_i+1不是2的整数次幂，则需要在最后添加一个由$s_i - 2^{\lfloor log_2(s_i+1) \rfloor - 1}$个捆绑的物品

举个例子

$7 = 1 + 2 + 4$这个就是刚好的

$6 = 1 + 2 + 3$这个最后补4就多了，所以吧剩余的3补上

$18 = 1 + 2 + 4 + 8 + 3$

$31 = 1 + 2 + 4 + 8 + 16$

```
index = 0;
for (int i = 1; i <= m; i++) {
  int c = 1, p, h, k;
  cin >> p >> h >> k;
  while (k > c) {
    k -= c;
    list[++index].w = c * p;
    list[index].v = c * h;
    c *= 2;
  }
  list[++index].w = p * k;
  list[index].v = h * k;
}
```

在二进制拆分之后你只需要把拆分后的数组做01背包即可，总的时间复杂度被优化到了$O(V * \sum log(s_i))$

DP路径记录

dp的路径记录一般是利用前驱数组记录的，因为一个状态可能会转移给很多状态，但是由于最优决策，只有一个最优的前驱会向当前状态转移，所以我们只需要在转移时记录其前驱即可，然后用递归的方式还原路径

```
int pre[N];//前驱数组
vector<int>ansid;//路径记录
void getans(int pos){//递归记录路径
    if(pre[pos])getans(pre[pos]);
    ansid.push_back(pos);
    return;
}
```