Dynamic Programming

Jinning Li（15）

Introduction

- 规划
  - 规划是比较全面的长远的发展计划----现代汉语词典
- 动态规划主要针对最优化问题(决策问题)
- 动态规划
  - 通过多阶段决策逐步找出问题的最终解，并且每个阶段的决策都是需要全面考虑各种不同的情况分别进行决策。这样，当各阶段采取决策后，会不断决策出新的数据，直到找到最优解。
  - 每次决策依赖于当前状态，又随即引起状态的转移，一个决策序列就是在变化的状态中产生出来的，故有“动态”的含义。
  - 这种多阶段最优化决策解决问题的过程称为动态规划。

Introduction

- 状态
  - 状态用于表示和存储某个局面。
  - 状态可以理解为一个多元函数（映射） f(x1, x2, x3...) = k
  - 状态通常可以使用数组存储。例如f[i][j]。
  - 状态也可以用其他方式存储，例如二哥找宝箱中，我们用二进制串表示目前已找到的宝箱的情况。
- 状态的转移
  - 一个状态可以由一个或多个其他的状态转移得到。
  - 状态转移方程。例如长成这样： f[i][j] = f[i-1][j] + f[i-1][j-1]
  - 状态的转移也可以是其他方式。例如二哥找宝箱中，我们通过修改二进制串不同位上的值表示已找到宝箱的情况发生了变化

The Triangle

```
1
2   3
4   5   6
7   8   9   10
11 12 13 14 15
```

请编一个程序输出从顶至底的所有方案。

- 每一步可延直线向下或右斜线向下走；
- 1<三角形行数<=10

搜索

Solution

请编一个程序输出从顶至底的所有方案。

```
1
2   3
4   5   6
7   8   9   10
11 12 13 14 15
```

- 每一步可延直线向下或右斜线向下走；
- 1<三角形行数<=10

- DFS:

```
int path[11];
void dfs(int i, int j)
{
        path[i] = a[i][j];
        if(i==n)
                for(int k = 1; k <= n; k++)
                { cout<<path[k]; return;}
        dfs(i+1, j);
        dfs(i+1, j+1);
}
```

搜索：

此题中我们表示状态的方法就是函数的自变量和记录路径的数组：

`void dfs(int i, int j)`和数组path

i，j表示当前搜索到i行j列的状态。Path记录到现在为止经过哪些点

而状态的转移就是搜索到下一行时,把i变为i+1，并把当前经过的点加入path数组中。

在搜索问题中，每个状态是由之前的各个状态通过组合得到的。例如本题对于到第k行的所有可能路线，是从顶部到第k-1行所有状态的组合。

所以搜索其实是一种比较暴力的方法。遇到需要枚举所有可能的路径方案这类问题时，就需要用到搜索。

The Triangle

```
1
1 1
1 1 1
1 1 1 1
1 1 1 1 1
```

请编一个程序计算从顶至底一共有几种方案。

- 每一步可延直线向下或右斜线向下走；
- 1<三角形行数<=10

递推

Solution

- 设sum[i][j]为从三角形顶部走到第i行j列位置的方案数之和。则有
- 初始状态 sum[1][1] = 1
- 递推条件 sum[i][j] = sum[i - 1][j] + sum[i - 1][j - 1] (i > 1; j>=1)
- 递推：不需要进行最优决策。每一个状态都可以由之前的状态直接得出。
- 递推并不像搜索一样需要枚举和储存每一个过程中的每种情况。所以递推常常用于解决“有几种”、“和是多少”的问题。（斐波那契数列）

代码

```
int sum[11][11];
sum[i][j]的值置为0;
sum[1][1] = 1;
for(int i = 2; i <= n; i ++)
{
	for(int j = 1; j <= n; j ++)
	{
		sum[i][j] = sum[i-1][j] + sum[i-1][j-1];
	}
}
int ans = 0;
for(int j = 1; j <= n; j ++)
{
	ans += sum[n][j];
}
```

//由于递推并不需要把每一种路线都枚举一遍，所以只求方案数时用递推比搜索要快了不少。

The Triangle

```
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

请编一个程序计算从顶至底的某处的一条路径，使该路径所经过的数字的总和最大。

- 每一步可延直线向下或右斜线向下走；
- 1<三角形行数<=100
- 三角形中的数字为整数0,1,…，99

动态规划

Solution

- 设a[i][j]为第i行第j列的数字
- 状态：从顶部到i行j列的数字的最大总和。
- 状态用数组表示：opt[i][j]记录从顶部到第i行第j列的最大总和
- 初始条件（边界条件）：

  opt[1][1] = a[1][1]

- 状态转移：i行j列的最大总和等于（从它正上方的点到这点的总和）与（从它左上方到这点的总和）取最大值。
- 用转移方程表示：

  opt[i][j] = a[i][j] + opt[i-1][j] for j = 1

  opt[i][j] = a[i][j] + max(opt[i-1][j], opt[i-1][j-1]) for j > 1

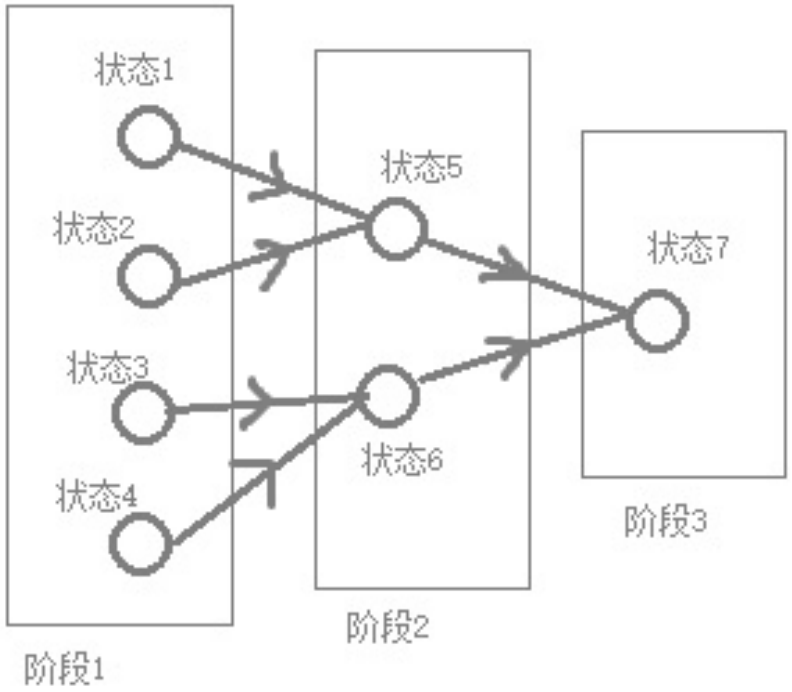
- 结果：到达第N行每个点的最大总和取其最大max{opt[N][i], 1 <= i <= N}

核心代码

```
int opt[101][101];
把opt[i][j]置为-inf;
Opt[1][1] = a[1][1];
for (int i = 1; i <= n; ++i)
for (int j = 1; j <= i; ++j)
    if (j == 1)//因为不可能从左上角到达当前点
        opt[i][j]=a[i][j]+opt[i-1][j];
    else
        opt[i][j]=a[i][j]+max(opt[i-1][j],opt[i-1][j-1])
```

最后对第n行即opt[n][j]取最大值。

- 简而言之，动态规划可以理解为是一种有选择的递推，这体现在状态转移时总是选择最优的方法进行转移，从而得到这个状态的局部最优。
- 通过这样不断的转移最终可以得到全局的最优解。
- 动态规划的过程可以理解为一个有向无环图(树)。例如：



每个阶段的状态都由它之前阶段的一个或多个状态取最优得到。这时的状态是局部最优的。

最终能够达到全局最优。

动态规划的基本思想

- 最优化原理（最优子结构性质）：不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有最优子结构性质。也就是说，某个每个阶段的最优状态可以从之前某个阶段的某个或某些状态直接得到。
- 无后效性：将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。例如在01背包问题中，dp[i][v]到dp[i+1][v]时，前i个物品怎么选的对于第i+1个物品怎么选没有直接的影响，而我们需要知道的只是dp[i][v]的值。
- 子问题的重叠性：动态规划将原来具有指数级时间复杂度的搜索算法改进成了具有多项式时间复杂度的算法。其中的关键在于解决冗余，这是动态规划算法的根本目的。动态规划实质上是一种以空间换时间的技术，它在实现的过程中，不得不存储产生过程中的各种状态，所以它的空间复杂度要大于其它的算法。(将子问题的答案存储下来并重复利用)

动态规划算法的基本步骤

- 设计一个动态规划算法，通常可按以下几个步骤进行：
- （1）分析最优解的性质，并刻划其结构特征。（找到符合动态规划三种性质的状态的定义）
- （2）递归地定义最优值。（找到状态之间转移的规律，写出状态转移方程）
- （3）以记忆化方法计算出最优值（用数组等方法来存储状态，递推地算出最优值）

最长递增子序列（LIS）

- 给定一个数列，长度为N，
  求这个数列的最长上升（递增）子数列（LIS）的长度.
- 例如：
  1 7 2 8 3 4
  这个数列的最长递增子数列是 1 2 3 4，长度为4；
  次长的长度为3，包括 1 7 8; 1 2 3 等.
- 问题转化为：
- 给定一个数列，长度为N，
  设Fk为：以数列中第k项结尾的最长递增子序列的长度.
  求 F1…FN中的最大值.
- 这里的Fi就是所谓的状态。为什么要这样定义状态？因为这样的定义可以保证无后效性以及最优子结构性质。
- 无后效性是说这里的F1…Fk-1具体选了哪些数对于Fk没有影响，有影响的只是它们的值。最优子结构是说这里的Fk最优时，F1…Fk-1一定是最优的。
- 现在我们来思考一下怎么转移。

最长递增子序列（LIS）

- 给定一个数列，长度为N，
  求这个数列的最长上升（递增）子数列（LIS）的长度.
- 转化为：给定一个数列，长度为N，
  设F_k为：以数列中第k项结尾的最长递增子序列的长度.
  求 F1…FN中的最大值.
- 边界条件：$F_1 = 1$
- 状态转移方程：$F_k = max(F_i + 1 | A_k > A_i, i \in (1..k-1))$
- 也就是说，以第k项结尾的LIS的长度是：取遍i小于k的所有值，保证第i项比第k项小的情况下，以第i项结尾的LIS长度加一的最大值。
- 这样转移之后对最后得到的F_1~F_N取最大值就是答案。

导弹拦截

- 某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。输入导弹依次飞来的高度（雷达给出的高度数据是不大于30000的正整数），计算这套系统最多能拦截多少导弹。
- 样例输入
- 389 207 155 300 299 170 158 65
- 样例输出
- 6

Solution

- 最长递减子序列
- 令opt[i]记录处理到前i个导弹，并且第i枚导弹被拦截时的最大拦截导弹数
- 转移方程：opt[i]=max{1, opt[j] + 1 (j < i and a[j] > a[i])}
- 时间复杂度：$O(n^2)$

核心代码

```
for (int i = 1; i <= N; ++i)   //计算opt[i]
{
     for (int j = 1; j < i; ++j)   //j为i之前某个高度更高的导弹
      if (a[j] > a[i])
        opt[i]=max(opt[j]+1, 1);
}
```

最长公共子序列（LCS）

- 一个序列的子序列是在该序列中删去若干元素后得到的序列。
- 给定两个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列X和Y的公共子序列。
- 给定两个序列X和Y，要求求出X和Y的最长公共子序列的长度。
- 设X的长度为m，Y的长度为n。
- 例：X=<A,B,C,B,D,A,B> Y=<B,D,C,A,B,A>
- LCS=<B,C,B,A>，长度为4

Solution

- 相当于是最长递增子序列的二维升级版
- c[i][j]记录X的前i个字符构成的串与Y的前j个字符构成的串的LCS长度

$$c[i][j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1][j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i][j-1],c[i-1][j]) & \text{if } i,j>0 \text{ and } x_i \mathrel{!}= y_j \end{cases}$$

01背包问题

- 问题：有N件物品和一个容量为V的背包。第i件物品的费用是c[i]，价值是w[i]。求解将哪些物品装入背包可使价值总和最大。每种物品仅有一件，可以选择放或不放。
- 状态： f[i][v]表示前i件物品恰放入一个容量为v的背包可以获得的最大价值。
- 状态转移方程： f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}
- 考虑“将前i件物品放入容量为v的背包中”（即f[i][v]）这个子问题：
- 1.如果不放第i件物品，那么问题就转化为“前i-1件物品放入容量为v的背包中”，价值为f[i-1][v]；
- 2.如果放第i件物品，那么问题就转化为“前i-1件物品放入剩下的容量为v-c[i]的背包中”，此时能获得的最大价值就是f[i-1][v-c[i]]再加上通过放入第i件物品获得的价值w[i]。

核心代码

```
for i=1..N
    for v=0..V
       if(v<c[i]) f[i][v] = f[i-1][v];
        f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]};
```

```
for i=1..N
    for v=V..0
       if(v<c[i]) continue;
        f[v]=max{f[v],f[v-c[i]]+w[i]};
```

这段代码与上面的是等价的。注意剩余背包容量的v取值要从V取到0而不能反过来。
想想为什么？

f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程
f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}
因为现在的f[v-c[i]]就相当于原来的f[i-1][v-c[i]]

采药（NOIp’05 PJ）

- 有N种草药可供采集（每种只有一棵），采集第i种草药需要花费ci的时间，采集后可以得到收益vi，问花费不超过L时间采药，能采到的草药总收益最大为多少？
- 例：
- L=70 N=3
- c_1=71 v_1=100
- c_2=69 v_2=1
- c_3=1 v_3=2
- 输出为3

Solution

- 设c[i]为采集i花费的时间，v[i]为i的价值，L为总时间限制，N为草药的数量
- opt[i][j]记录只考虑前i种草，花费j时间所能取到的最大价值。
- 转移方程：opt[i][j] = max(opt[i-1][j], opt[i-1][j-c[i]]+v[i])
- 边界条件：opt[0][0] = 0

  opt[0][i] = -inf 1<=i<=L
- 解：max{opt[N][i], 0<=i<=L}

代码

```
memset(a, 0, sizeof(a));        //初始化
 n=0;//n为当前最大可能花费的时间，初始为0
   for (i = 1; i <= m; ++i)
   {
      n = min(n + w[i], t);     //只需要计算opt[i][0]到opt[i][n]
      for (j = 0; j < w[i]; ++j)
         a[i][j] = a[i-1][j];
      for (; j <= n; ++j)
         a[i][j] = max(a[i-1][j], a[i-1][j-w[i]]+v[i]);
   }
   ans = 0;
   for (i = 0; i <= n; ++i)
      ans = max(ans, a[m][i]);   //ans为所求答案
```

完全背包问题

- 问题：有N种物品和一个容量为V的背包。第i种物品的费用是c[i]，价值是w[i]。求解如何将物品装入背包可使价值总和最大。每种物品有无穷多件。
  - 只要将01背包一维数组的方法中，剩余体积v改成从0枚举到V就可以了

```
for i=1..N
    for v=0..V
        if(v<c[i]) continue;
         f[v]=max{f[v],f[v-c[i]]+w[i]};
```

首先想想为什么01背包中**一维数组方法**要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次。
而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-c[i]]，所以就可以并且必须采用v=0..V的顺序循环。

下课！

可以去看看 背包九讲 喔~

其他题目：

Little Shop of Flowers(IOI'99)

- 有F束不同品种的花束，同时有至少同样数量的花瓶被按顺序摆成一行。这些花瓶的位置固定于架子上，并从1至V顺序编号，V是花瓶的数目，从左至右排列，则最左边的是花瓶1，最右边的是花瓶V。
- 花束可以移动，并且每束花用1至F间的整数唯一标识。标识花束的整数决定了花束在花瓶中的顺序，如果I < J，则令花束I必须放在花束J左边的花瓶中。如果花瓶的数目大于花束的数目。则多余的花瓶必须空置，且每个花瓶中只能放一束花。
- 当各个花瓶中放入不同的花束时，会产生不同的美学效果，并以美学值（一个整数）来表示，空置花瓶的美学值为零。
- 你必须在保持花束顺序的前提下，使花束的摆放取得最大的美学值。如果有不止一种的摆放方式具有最大的美学值，则其中任何一直摆放方式都可以接受，但你只要输出任意一种摆放方式。

Solution

- 解一：

  opt[i][j]记录处理到前i束花，其中第i束花插在第j个花瓶里，所能取得的最大美学值。

  转移方程：opt[i][j] = max{opt[i-1][k] + a[i][j], i – 1 <= k < j}

  边界条件：opt[0][i] = 0 , 0 <= i <= v

- 解二：

  opt[i][j]记录前i个花瓶放了前j束花，此时取得的最大美学值。

  转移方程：opt[i][j]=max(opt[i-1][j-1]+a[j][i], opt[i-1][j])

  边界条件：opt[i][0] = 0, 0 <= i <= v

乌龟棋(NOIp’10 TG)

- 乌龟棋的棋盘是一行N个格子，每个格子上一个分数（非负整数）。棋盘第1格是唯一的起点，第N格是终点，游戏要求玩家控制一个乌龟棋子从起点出发走到终点。
- 乌龟棋中M张爬行卡片，分成4种不同的类型（M张卡片中不一定包含所有4种类型的卡片，见样例），每种类型的卡片上分别标有1、2、3、4四个数字之一，表示使用这种卡片后，乌龟棋子将向前爬行相应的格子数。游戏中，玩家每次需要从所有的爬行卡片中选择一张之前没有使用过的爬行卡片，控制乌龟棋子前进相应的格子数，每张卡片只能使用一次。
- 游戏中，乌龟棋子自动获得起点格子的分数，并且在后续的爬行中每到达一个格子，就得到该格子相应的分数。玩家最终游戏得分就是乌龟棋子从起点到终点过程中到过的所有格子的分数总和。
- 已知棋盘上每个格子的分数和所有的爬行卡片，求最多能得到多少分

Naïve Solution

- opt[L][a][b][c][d]表示走到第L个格子，四个卡片分别用了a、b、c、d张，能获得的最大分数
- 转移方程：opt[L][a][b][c][d] =

  Score[L]+max{

  opt[L-1][a-1][b][c][d] if a > 0 and L > 1

  opt[L-2][a][b-1][c][d] if b > 0 and L > 2

  opt[L-3][a][b][c-1][d] if c > 0 and L > 3

  opt[L-4][a][b][c][d-1] if d > 0 and L > 4

  }

- 数组大小：350*40*40*40*40 = 896 000 000

Solution

- 注意到L=(1,2,3,4)$(a,b,c,d)^T$，故方程中L维可以优化
- 数组大小：40*40*40*40 = 2 560 000
- opt[a][b][c][d]=

  Score[a+2b+3c+4d]+max{

  opt[a-1][b][c][d] if a > 0

  opt[a][b-1][c][d] if b > 0

  opt[a][b][c-1][d] if c > 0

  opt[a][b][c][d-1] if d > 0

  }

Post Office(IOI'00)

- n个村庄分布在一个数轴上，给出每个村庄的坐标位置，在这n个村庄中建m个邮局，每个村庄都只去最近的邮局。问所有村庄去邮局的距离和最小为多少？
- 例：n=10，m=5
- 坐标依次为1 2 3 6 7 9 11 22 44 50
- 此时距离总和最小为9，邮局分别建在坐标为2 7 22 44 50的村庄里

Solution

- 设opt[i][j]表示安排前i个邮局给前j个村庄使用，通过某种安排手段，使这j个村庄分别到这i个邮局中最近的一个距离之和为所取到的最小值。
- 令设NewCost[A][B]为：分配A至B这（B-A+1）个村庄使用一个新的邮局，通过某种安排新的邮局的手段，使A至B这（B-A+1）个村庄分别到这个新邮局的距离之和为所取到的最小值。
- 转移方程：opt[i][j] = min{opt[i-1][k] + NewCost[k+1][j], i-1 <= k <= j}
- Claim:取到NewCost[A][B]时，新建的邮局一定为 A到B之间最中间的村庄。

拓展

- 顺序DP
- 区间DP
- 树型DP/图型DP
- 状态压缩DP

- 滚动数组
- 单调队列优化
- 斜率优化
- 四边形不等式优化

总结

- 需要动态规划的题的特点：
  - 所需复杂度为低次多项式时间
  - 题目所求为最优值（最大or最小）
  - 可以按阶段划分状态
  - 无后效性
- 动态规划的解题流程：
  - 写出状态的表示方法（一维or二维or高维，opt[i][j]表示什么）
  - 写出状态转移方程
  - 处理边界条件
  - 估算时间复杂度