DP入门 && 记忆化搜索

HEU ACM集训队 by 王志立

July 26, 2014

1 动态规划

DP：*Dynamic Programming*的缩写，中文意思动态规划。在讲解这个概念之前先来看一个例子，体会下什么是动态规划。

假设现在你有一个容量为V的背包和一堆石头，每个石头都有一个体积v_i和价值w_i,你需要向背包里面装一些石头使这些石头的价值之和最大，问你这个最大价值是多少？

你可能会想到首先挑价值最大的装，例如：$V=20$，有3个石头（以下石头都以$[\boldsymbol{v_i},\boldsymbol{w_i}]$形式表示），体积和价值分别是$[18,22],[15,30],[9,13]$,这时你选择一个价值最大的$[18,22]$，并且发现这就是最终的最大价值，因为这个包只能装一个石头，这个方法奏效了。

继续看 :$V=20$，有3个石头，体积和价值分别是$[10,22],[15,30],[9,13]$,仍然首先选价值最大的？那么你得到的答案就是30，因为当你装完这个石头（$[15,30]$）后其他的石头再也无法装入！事实上很显然这个答案应该是$22+13=35$，选择$[10,22],[9,13]$，$10+9=19<20$。所以这个思路是错的，因为受背包本身容量的影响。

那么优先选择“性价比”(w_i/v_i)最大的？容易验证这也是错的，例如：$V=10$，两个石头体积和价值分别是$[2,8],[5,10]$就是反例。

上面两种思路之所以错是因为他们都是贪心思想。显然对于背包问题如果我们一味的贪心很可能使我们在贪心的过程中丢失全局最优解。因为在后来的选

择中我们很可能需要去调整前面的决策方案以使我们达到全局的最优解。 因此，我们需要设一种状态，它表示并记录之前我们的决策方案，以便我们需要时去使用它。

定义：**$dp[i][j]$表示背包容量为i且在前j个石头中选取时的最大价值**。 显然,根据定义如果背包容量为v，石头个数为n，最后答案就是$dp[v][n]$。

那么我们如何来得到这个状态呢？换句话说就是这个状态可以由哪些状态转移而来？考虑对于任意一个石头我们有选或不选两种决策，因此取这两者中最大的即可！状态转移方程就是：

$$dp[i][j] = max(dp[i][j-1], dp[i-v[j]][j-1] + w[j])$$

其中$dp[i][j-1]$表示不选第j个石头，$dp[i-v[j]][j-1]+w[j]$表示选择第j个石头。

```
//C++ code for this problem
const int MAXN = ONE_CONST_VALUE;
int V,v[MAXN],w[MAXN],dp[MAXN][MAXN];
for(int i = 1;i <= V;i ++){
  for(int j = 1;j <= n;j ++){
    if(i < v[j]){
      dp[i][j] = dp[i][j-1];
      continue;
    }
    dp[i][j] = max(dp[i][j-1],dp[i-v[j]][j-1] + w[j]);
  }
}
```

容易看出，算法时间复杂度为$O(V*n)$

思考：空间复杂度能否再优化?

下面给一组数据来模拟一下，看看这个算法是怎么工作的

$\boldsymbol{V=15,\ 4}$个石头:分别为$\boldsymbol{[3,4],\ [5,8],\ [10,10],\ [6,9]}$

n \ v	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	4	4	4	4	4	4	4	4	4	4	4	4	4
2	0	0	4	4	8	8	8	12	12	12	12	12	12	12	12
3	0	0	4	4	8	8	8	12	12	12	12	12	14	14	18
4	0	0	4	4	8	9	9	12	13	13	17	17	17	21	21

1.1 概念

动态规划：是一种在数学、计算机科学和经济学中使用的，**通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。**

1.2 性质

1.最优子结构:如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质（即满足最优化原理）。最优子结构性质为动态规划算法解决问题提供了重要线索。

2.子问题重叠:子问题重叠性质是指在用递归算法自顶向下对问题进行求解时，每次产生的子问题并不总是新问题，有些子问题会被重复计算多次。动态规划算法正是利用了这种子问题的重叠性质，对每一个子问题只计算一次，然后将其计算结果保存在一个表格中，当再次需要计算已经计算过的子问题时，只是在表格中简单地查看一下结果，从而获得较高的效率。

1.3 例题

1.(最大子段和) 给定一个初始序列$X=\{a_1,a_2,a_3,a_4,...,a_n\}$,求出$X$的一段**连续**子序列(子段)，使得这段子序列的和最大,求出这个最大和。例如序列$X=\{1,2,3,-10,5,6\}$,那么$\{1,2,3\},\{2,3,-10\},\{5,6\}$都是$X$的连续子序列,而$\{1,3\},\{2,5\}$不是。在$X$的所有连续子序列中$\{5,6\}$的和最大。

分析：考虑不管最终如何选择，**目标序列都会以序列X中的某一个元素结尾**，这是显然的。因此如果我们知道了以X中每个元素结尾的连续子序列的最大和时，我们就可以以$\boldsymbol{O(n)}$的时间复杂度从左到右扫描一遍整个序列得到最大值。

因此状态的设置就很简单了。**设$dp[i]$表示以元素a_i结尾的连续子序列的最大和**。状态转移方程怎么写?

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

上述方程的意思是，**$dp[i]$必须选择a_i(根据定义)，至于选不选$dp[i-1]$需要看它是不是大于0**。

```
1 //C++ code for this problem
2 const int MAXN = ONE_CONST_VALUE;
3 int dp[MAXN],ans = a[1];
4 dp[1] = a[1];  //init,dp[1] only can choose a[1]
5 for(int i = 2;i <= n;i ++){
6   dp[i] = max(a[i],a[i] + dp[i-1]);
7   ans = max(ans,dp[i]);
8 }
```

算法时间复杂度$\boldsymbol{O(n)}$

2.给定一个初始序列$X = \{a_1, a_2, a_3, a_4, ..., a_n\}$,求出它的最长上升子序列的长度$(LIS)$.

分析：**以某个数a_i结尾的上升子序列的长度肯定是以在它之前且比它小的数a_j结尾的上升子序列的长度+1**。因此如果知道了以$a_1 - a_{i-1}$结尾的最长上升子序列的长度，那么就可以通过枚举每个$j(1 <= j <= i-1)$来求得以i结尾的最长上升子序列的长度。

设：$dp[i]$表示以a_i结尾的最长上升子序列的长度，则有：

$$dp[i] = max\{dp[j] + 1 \mid 1 <= j < i,\ a_j < a_i\}$$

```
1  //C++ code for this problem
2  const int MAXN = ONE_CONST_VALUE;
3  int dp[MAXN] = {0},ans = 1;
4  dp[1] = 1;              //init
5  for(int i = 2;i <= n;i ++){
6    for(int j = 1;j < i;j ++){
7      if(a[i] < a[j]) continue;
8      dp[i] = max(dp[i],dp[j] + 1);
9    }
10   ans = max(ans,dp[i]);
11 }
```

算法时间复杂度为 $O(n^2)$, 有更快的算法吗？ view better solution

1.4 练习题

给定两个初始字符序列$X,Y(max(len(X),len(Y)) <= 10^3$),求出X,Y的最长公共子序列的长度(LCS) view solution

2 记忆化搜索

有了上面DP的基础知识后，我们开始学习记忆化搜索。记忆化搜索在形式上仍然是搜索的流程。我们知道朴素的搜索算法本身就是一种暴力的解法，通过枚举所有状态来得到解（当然可以通过剪枝来舍去一些无效状态），效率低的原因是因为它没有能够很好地处理重叠的子问题。**在求解问题的过程中我们可能要去求解若干子问题，然而这些子问题可能又会包含相同的子问题，如**

果我们只是简单的搜索那么有些子问题就会被我们重复计算很多次，这无疑造成时间的浪费，因此就产生了记忆化搜索，它采用搜索的形式和动态规划中递推的思想将这两种方法有机地综合在一起，扬长避短，简单实用，在信息学中有着重要的作用。用一个公式简单地说：**记忆化搜索=搜索的形式+动态规划的思想。记忆化搜索的大致思想是：我们在搜索的过程中把已经被搜索过的状态记录并保存起来，如果在以后的搜索过程中再遇到这个状态就不必再搜索下去，直接用即可。**

记忆化搜索的大致流程如下：

```
XXX dfs(status){
  if(recursion should be over) return ;
  if(status have been searched before) return ;
  record status have been searched;
  dfs(next status);
}
```

2.1 例子

1.先看一个最简单的例子：斐波那契数列（so easy ？ !），相信大家都写过，一个for循环就解决

```
for(int i = 3;i <= n;i ++)
  fib[i] = fib[i-1] + fib[i-2]
```

现在从另一个角度解决这个问题，用递归的方法写一次看看(also so easy ？ !)，通过这个例子简单体会下记忆化搜索。

```
int dfs(int n){
  if(n == 1 || n == 2) return 1;
  return dfs(n-1) + dfs(n-2);
}
```

你知道这样写会有多浪费时间吗？下面是在我的计算机上运行出结果的时间(单位：秒)：

N	10	15	20	25	30	35	40	45	46
T	0.0000	0.0000	0.0000	0.0000	0.0000	0.0600	0.6200	6.8800	16.7400

之所以这么慢是因为我们重复计算了很多的子问题，上述代码中求$dfs(n)$的时候首先会去求$dfs(n-1)$，程序就跳到下一轮递归，dfs(n-1)算完之后程序会回溯，然后又去计算$dfs(n-2)$，显然在求$dfs(n-1)$的过程中$dfs(n-2)$就已经求出来了，只是我们没有记录和保存它，导致了我们又要重新算一次$dfs(n-2)$，造成了时间的浪费。

如果采用记忆化搜索的思想，把被计算过的状态保存下来，那么时间会大大降低:

```
1 //C++ code for this problem
2 long long int dp[50] = {0};
3 long long int dfs(int n){
4     if(n == 1 || n == 2) return 1;
5     if(dp[n]) return dp[n];
6     return dp[n] = dfs(n-1) + dfs(n-2);
7 }
```

这样，**每个状态只被计算一次,时间复杂度为$O(n)$**，运行时间为：

N	10	15	20	25	30	35	40	45	46
T	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

2.：Michael喜欢滑雪百这并不奇怪， 因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael想知道载一个区域中最长底滑坡。区域由一个二维数组$mat[R][C]$给出行数R和列数$C(1 <= R, C <= 100)$。数组的每个数字代表点的高度。下面是一个例子:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 16 & 17 & 18 & 19 & 6 \\ 15 & 24 & 25 & 20 & 7 \\ 14 & 23 & 22 & 21 & 8 \\ 13 & 12 & 11 & 10 & 9 \end{bmatrix}$$

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为24-17-16-1。当然25-24-23-...-3-2-1更长。事实上，这是最长的一条。

分析：假设我们知道了从某个点$[i,j]$周围的四个点开始滑雪的最大长度$\boldsymbol{h_{1m},h_{2m},h_{3m},h_{4m}}$,那么我们就可以得到从该点滑雪的最大长度$\boldsymbol{h[i][j]_m}$，根据动态规划思想：

$$\boldsymbol{h[i][j]_m = max(h[i][j]_m, h_{km} + 1)} \qquad \boldsymbol{(1 <= k <= 4)}$$

如果该点高度大于第k个点的高度

因此采用记忆化搜索，对于每个没有被搜索过的点进行搜索，在搜索的过程中如果发现某个点已被搜索就不要再向下搜索，直接返回结果即可。

小提示:在一张用二维矩阵表示的图（区别于图论中的图）上进行搜索时，我们往往会涉及到上下左右的概念，搜索这四个方向，初学者很容易手写四行代码来枚举四个方向，这样代码看起来较乱且不易于维护。可以用一个二维数组表示四个方向：

```
1 int dir[4][2] = {1,0,0,1,-1,0,0,-1};
```

这样代码看起来较简洁，不易出错。

如果还有斜着的方向，共八个方向，同样也可以表示：

```
1 int dir[8][2]={1,0,0,1,-1,0,0,-1,1,1,1,-1,-1,1,-1,-1};
```

```cpp
//C++ code for this problem
const int MAXN = 111;
int dp[MAXN][MAXN],mat[MAXN][MAXN],R,C;
int dir[4][2] = {1,0,0,1,-1,0,0,-1};
memset(dp,-1,sizeof dp);    //init
bool inMap(int x,int y){
  return x >= 0 && x < R && y >= 0 && y < C
}
int dfs(int x,int y){
  if(~dp[x][y]) return dp[x][y];
  dp[x][y] = 0;
  for(int i = 0;i < 4;i ++){
    int xx = dir[i][0] + x, yy = dir[i][1] + y;
    if(inMap(xx,yy) && mat[x][y] > mat[xx][yy])
      dp[x][y] = max(dp[x][y],dfs(xx,yy) + 1);
  }
  return dp[x][y];
}
```

2.2 练习题

给定一个字符串s,可以删除其中的一些(或0个)字符，有多少删除种方法把这个字符串变成回文串？(回文串的定义：一个串反转之后和原串相同，则称它为回文串，比如aba。) view solution

3 Solutions

返回

设：$\boldsymbol{dp[len]}$**表示最长上升子序列长度为**len**时的末尾元素最小值，也就是这个最长上升子序列的最大元素的最小值。**则对于a_i,如果$a_i > dp[len]$,则说明这个序列还可以在末尾添加一个a_i使整个序列长度+1,即:$\boldsymbol{dp[len = len + 1] = a[i]}$,否则查找$dp$数组中第一个大于或等于$a_i$的数的下标$idx$,来更新$\boldsymbol{dp[idx] = a[i]}$。总的时间的复杂度降为$\boldsymbol{O(n * log_2 n)}$.

```
1  //C++ code for LIS
2  const int INF = 0x7fffffff;
3  const int MAXN = ONE_CONST_VALUE;
4  int dp[MAXN], ans(0);
5  int binarySearch(int target, int r, int l = 1){
6    while(l <= r){
7      int mid = (l + r) >> 1;
8      if(target > dp[mid]) l = mid + 1;
9      else r = mid - 1;
10   }
11   return l;
12 }
13 void work(){
14   for(int i = 1;i < MAXN;i ++) dp[i] = -INF;
15   for(int i = 1;i <= n;i ++){
16     if(dp[ans] < a[i]) dp[++ans] = a[i];
17     else{
18       int idx = binarySearch(a[i],ans);
19       dp[idx] = a[i];
20     }
21   }
22 }
```

返回

设：$dp[i][j]$表示X的前i个字符和Y的前j个字符的最长公共子序列的长度，状态转移方程如下：

$$if(X[i] == Y[j])\ \ dp[i][j] = dp[i-1][j-1] + 1$$
$$else\ \ dp[i][j] = max(dp[i-1][j], dp[i][j-1])$$

```
1  //C++ code for LCS
2  const int MAXN = ONE_CONST_VALUE;
3  int lenX = strlen(X),lenY = strlen(Y);
4  int dp[MAX][MAXN],ans(0);
5  for(int i = 1;i <= lenX;i ++){
6    for(int j = 1;j <= lenY;j ++){
7      if(X[i] == Y[j]) dp[i][j] = dp[i-1][j-1] + 1;
8      else dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
9      ans = max(ans,dp[i][j]);
10   }
11 }
```

时间复杂度$O(lenX * lenY)$

返回

设：$dp[i][j]$表示从第i个字符到第j个字符的删除方法数，则:如果$S[i]$ <> $S[j]$那么就必须至少删除$S[i], S[j]$中至少一个，如果$S[i]$ == $S[j]$,则可以至少删除$S[i], S[j]$中一个，也可以都保留。状态转移方程:

$$dp[n][m] = dp[n+1][m] + dp[n][m-1] + 1 \quad (s[i] == s[j])$$

$$dp[n][m] = dp[n][m-1] + dp[n+1][m] - dp[n+1][m-1] \quad (s[i] != s[j])$$

```
1  //C++ code for palindrome
2  const int MAXN = ONE_CONST_VALUE;
3  long long dp[MAXN][MAXN];
4  long long dfs(int n, int m){
5    if(n > m) return 0;
6    if(n == m) return 1;
7    if(dp[n][m]) return dp[n][m];
8    if(str[n] != s[m])
9      dp[n][m] = dfs(n,m-1)+dfs(n+1,m)-dfs(n+1,m-1);
10   else dp[n][m] = dfs(n+1,m)+dfs(n,m-1) + 1 ;
11   return dp[n][m];
12 }
```