区间 DP

李淳风

长郡中学

2024 年 9 月 23 日

前言

到目前为止，我们介绍的线性 DP 一般从初态开始，沿着阶段的扩张向某个方向递推，直至计算出目标状态。区间 DP 也属于线性 DP 的一种，只不过是以“区间长度”作为 DP 的阶段，使用两个坐标（区间的左、右端点）描述每个维度。
在区间 DP 中，一个状态由若干个比它更小且被它包含的区间代表的状态转移而来。因此，区间 DP 的决策往往就是如何划分区间，初态一般就是由长度为 1 的“元区间”构成。

例题

石子合并

有 n 堆石子排成一排，其中第 i 堆石子的重量为 A_i。每次可以选择其中相邻的两堆石子合并成一堆，形成的新石子堆的重量以及消耗的体力都是两堆石子的重量之和。求把全部 n 堆石子合成一堆最小需要消耗多少体力。

$1 \leq n \leq 300$。

例题

石子合并

有 n 堆石子排成一排，其中第 i 堆石子的重量为 A_i。每次可以选择其中相邻的两堆石子合并成一堆，形成的新石子堆的重量以及消耗的体力都是两堆石子的重量之和。求把全部 n 堆石子合成一堆最小需要消耗多少体力。
$1 \leq n \leq 300$。

若最初的第 l 堆石子和第 r 堆石子被合并成一堆，则说明第 $l \sim r$ 堆之间的每堆石子也已经被合并。因此，在任意时刻，任何一堆石子都可以用一个闭区间 $[l, r]$ 来描述，表示这堆石子是由最初的第 $l \sim r$ 堆石子合并而成的，其重量为 $\sum_{i=l}^{r} A_i$。另外，考虑得到这堆石子的最后一次合并操作，一定存在一个 k，先把第 $l \sim k$ 堆石子和第 $k+1 \sim r$ 堆石子合并，然后再合并成 $[l, r]$。

石子合并

对应到动态规划当中，就意味着两个长度较小的区间的信息，向一个更长的区间发生了转移，划分点 k 就是转移的决策。自然地，应该把区间长度 len 作为 DP 的阶段。而我们在表示状态的使用应该尽量少地使用维度，所以我们可以只用左、右端点来表示 DP 地状态。
设 $f[l][r]$ 表示把最初的第 $l \sim r$ 堆石子合并成一堆，需要消耗的最少体力，我们可以写出状态转移方程：

$$f[l][r] = \min_{l \leq k < r} \{f[l][k] + f[k+1][r]\} + \sum_{i=l}^{r} A_i$$

初始时 $\forall l \in [1, n], f[l][l] = 0$，其余为正无穷，目标为 $f[1][n]$。

石子合并

在实现转移方程的时候，务必分清楚阶段、状态和决策，先枚举阶段，再枚举状态，最后枚举决策。

```
memset(f,0x3f,sizeof(f));
for(int i=1;i<=n;i++){
    f[i][i]=0;
    sum[i]=sum[i-1]+a[i];//前缀和
}
for(int len=2;len<=n;len++)//长度作为阶段
    for(int l=1;l<=n-len+1;l++){//左端点，状态
        int r=l+len-1;//右端点，状态
        for(int k=l;k<r;k++)//决策
            f[l][r]=min(f[l][r],f[l][k]+f[k+1][r]);
        f[l][r]+=sum[r]-sum[l-1];
    }
```

例题

Polygon

多边形是一个玩家在一个有 n 个顶点的多边形上的游戏。每个顶点用整数标记，每个边用符号 $+$（加）或符号 *（乘积）标记。
第一步，删除其中一条边。随后每一步选择一条边连接的两个顶点 V_1 和 V_2，用边上的运算符计算 V_1 和 V_2 得到的结果来替换这两个顶点。只有一个顶点时游戏结束。
给定一个多边形，计算最高可能的分数。
$1 \leq n \leq 50$，保证无论如何操作，顶点数字都在 $[-32768, 32767]$ 的范围内。

例题

Polygon

多边形是一个玩家在一个有 n 个顶点的多边形上的游戏。每个顶点用整数标记，每个边用符号 $+$（加）或符号 *（乘积）标记。
第一步，删除其中一条边。随后每一步选择一条边连接的两个顶点 V_1 和 V_2，用边上的运算符计算 V_1 和 V_2 得到的结果来替换这两个顶点。只有一个顶点时游戏结束。
给定一个多边形，计算最高可能的分数。
$1 \le n \le 50$，保证无论如何操作，顶点数字都在 $[-32768, 32767]$ 的范围内。

在枚举完第一步删除哪条边之后，这道题就与石子合并非常类似。由于每次是对两个相邻的元素进行某种运算来合成一个新元素，我们把被删除的边逆时针第一个节点看作第一个节点，很容易想到用 $f[l][r]$ 表示把原本的第 $l \sim r$ 个顶点合成一个顶点后，顶点上的数最大是多少。

Polygon

然而，如果简单使用这样的做法，就完全忽视了动态规划问题的“最优子结构性质”。因为负数的存在，进行乘法运算时，大区间 $[l, r]$ 的最大数值不能简单由区间 $[l, k]$ 和 $[k+1, r]$ 合成的两个顶点的最大数值推出——因为如果能够合成出两个很小的负数，乘积反而会变大。
因此，如果我们再仔细想想，发现我们只需要同时记录能得到的最大值和最小值，就能保证“最优子结构性质”了。最大值的来源只可能是两个最大值相加、相乘，或者两个最小值相乘（负负得正），或一个最大值和一个最小值相乘。最小值的来源也同理。

Polygon

然而，如果简单使用这样的做法，就完全忽视了动态规划问题的“最优子结构性质”。因为负数的存在，进行乘法运算时，大区间 $[l,r]$ 的最大数值不能简单由区间 $[l,k]$ 和 $[k+1,r]$ 合成的两个顶点的最大数值推出——因为如果能够合成出两个很小的负数，乘积反而会变大。
因此，如果我们再仔细想想，发现我们只需要同时记录能得到的最大值和最小值，就能保证“最优子结构性质”了。最大值的来源只可能是两个最大值相加、相乘，或者两个最小值相乘（负负得正），或一个最大值和一个最小值相乘。最小值的来源也同理。
因此，我们可以设 $f[l][r][0]$ 表示把第 $l \sim r$ 个节点合并成一个后，数值最大是多少，$f[l][r][1]$ 表示数值最小是多少。枚举区间的划分点 k，我们就可以列出转移方程：

$$f[l][r][0] = \max_{l \le k < r} \left\{ \max \begin{cases} f[l][k][0] + f[k+1][r][0] & op = + \\ f[l][k][p] * f[k+1][r][q] & p, q \in \{0,1\}, op = * \end{cases} \right\}$$

$f[l][r][1]$ 的转移同理。初始时 $f[l][l][0] = f[l][l][1] = A_l$，其余为正/负无穷，目标是 $f[1][n][0]$。

Polygon

上述算法的时间复杂度为 $O(n^4)$。实际上我们还可以优化掉第一步枚举删除哪条边耗费的时间。在最开始，我们任意选择一条边删除，然后把剩下的“链”复制一倍接在末尾。
在这个长度为 $2n$ 的链上，$\forall i \in [1, n]$，把长度为 n 的区间 $[i, i+n-1]$ 合并成一个顶点，就对应原游戏的第一步删除第 i 个顶点逆时针一侧的边，然后把剩余的部分合并成一个顶点。因为区间长度是 DP 的阶段，我们只需要对前 n 个阶段进行 DP，每个阶段只有不超过 $2n$ 个状态，总复杂度降低为 $O(n^3)$。最后的答案就是 $\max_{1\le i\le n}\{f[i][i+n-1]\}$。
这种“选择任意一个位置断开，复制一遍形成 2 倍长度的链”的方法，是解决 DP 中环形问题的常用手段之一。

例题

金字塔

金字塔由若干房间组成，房间之间连有通道。
如果把房间看作节点，通道看作边的话，整个金字塔呈现一个有根树结构，节点的子树之间有序，金字塔有唯一的一个入口通向树根。
并且，每个房间的墙壁都涂有若干种颜色的一种。
探险队员打算进一步了解金字塔的结构，为此，他们使用了一种特殊设计的机器人，这种机器人会从入口进入金字塔，之后对金字塔进行深度优先遍历。
机器人每进入一个房间（无论是第一次进入还是返回），都会记录这个房间的颜色。最后，机器人会从入口退出金字塔。
显然，机器人会访问每个房间至少一次，并且穿越每条通道恰好两次（两个方向各一次），然后，机器人会得到一个颜色序列。
但是，探险队员发现这个颜色序列并不能唯一确定金字塔的结构。
现在他们想请你帮助他们计算，对于一个给定的颜色序列，有多少种可能的结构会得到这个序列。
因为结果可能会非常大，你只需要输出答案对 10^9 取模之后的值。
颜色序列 S 的长度不超过 300。

金字塔

我们知道，一棵树的每棵子树都对应着这棵树 DFS 序中的一个区间。本题中虽然记录的不是 DFS 序，但是也满足这个性质。因此，这道题目在“树形结构”和“字符串”之间建立了联系。结合之前对区间 DP 的分析，不难想到使用 $f[l][r]$ 表示子串 $S[l \sim r]$ 对应着多少种可能的树形结构。

接下来我们考虑对区间的划分。由于每次到达一个节点，就会记录一个字符，因此若 $S[l \sim r]$ 对应一棵子树，那么 $S[l+1], S[r-1]$ 两个字符是进入和离开时产生的。除此之外，$[l, r]$ 包含的每棵更深的子树都对应着一个子问题，会产生 $[l, r]$ 中的一段。相邻两端之间还有途径树根产生的一个字符。因为 $[l, r]$ 包含的子树个数可能不止两个，因此如果我们暴力枚举 $S[l \sim r]$ 划分点的数量和所有划分点的位置，时间复杂度会变得非常高。

那么，如果我们把子串 $S[l \sim r]$ 分成两部分，每部分可以由若干棵子树组成呢？这样就可能会产生重复计数。如果每段可以由多棵子树构成，那么方案"A|BAB|A|B|A" 和"A|B|A|BAB|A" 中的"BAB" 都能产生"B|A|B"，最终归为同一结果。

金字塔

实际上，为了保证方案不重不漏，我们可以只考虑子串 $S[l\sim r]$ 的第一棵子树是由哪一段构成的。枚举划分点 k，令子串 $S[l+1\sim k-1$ 构成 $S[l\sim r]$ 的第一棵子树，$S[k\sim r]$ 构成剩余部分。这样如果 k 不相同，那么第一棵子树的大小就不相同，就不会产生重复计算了。
当然前提条件是 $S[l]=S[r]=S[k]$。

$$f[l][r]=f[l+1][r-1]+\sum_{l+2\le k\le r-2,S[l]=S[k]}f[l+1][k-1]*f[k][r]$$

初始时 $\forall l\in[1,n],f[l][l]=1$，其余均为 0。目标为 $f[1][n]$。
这道题告诉我们，对于方案计数类的动态规划问题，一个状态之间的各个决策满足加法原理，而每个决策划分出来的几个子状态之间满足乘法原理。在设计状态的决策和转移时，一个状态之间的所有决策必须具有互斥性，才能保证不会重复计算。

金字塔

在具体的程序编写中，区间 DP 不仅可以通过递推（若干个循环）来实现，还可以通过递归（记忆化搜索）来实现。把子问题的求解过程写成一个函数 $solve(l,r)$，并建立一个全局舒服 f，在第一次计算完 $solve(l,r)$ 之后把值保存在 $f[l][r]$ 中，之后再调用就之间返回 $f[l][r]$。这样带有记忆化的搜索就保证了每个区间只会被求解一次，时间复杂度任然是 $O(n^3)$。

```
int solve(int l,int r){
    if(l>r) return 0;
    if(s[l]!=s[r]) return 0;
    if(l==r) return 1;
    if(f[l][r]!=-1) return f[l][r];
    f[l][r]=0;
    for(int k=l+2;k<=r;k++)
        f[l][r]=(f[l][r]+1ll*solve(l+1,k-1)*solve(k,r))%MOD;
    return f[l][r];
}
```