动态规划

关注Akie秋绘谢谢喵

目录

一. 动态规划原理

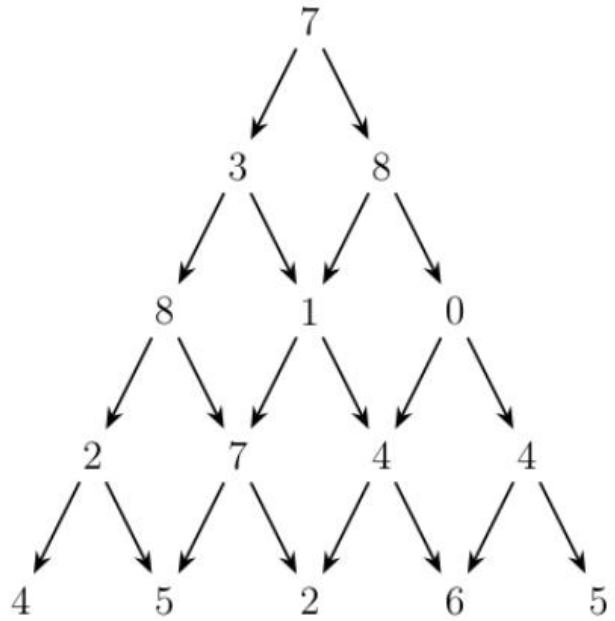
- 什么是动态规划?
- 人生阶段：小学->初中->高中->大学->...
- 在最初的求学阶段，父母总会根据每阶段表现， 为你选择之后的求学路程。
- 动态规划：将待求解的问题划分为若干个阶段，即若干个互相联系的子问题，然后按自底向上的顺序推导出原问题的解。

一. 动态规划原理

- 先看一个问题：洛谷P1216
- 金字塔行数r<=1000

观察下面的数字金字塔。

写一个程序来查找从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以走到左下方的点也可以到达右下方的点。



洛谷

一. 动态规划原理

- 暴力搜索？O(2^r), TLE!
- 不难发现，搜索过程中很多个点被重复走了很多次，非常浪费时间。可以想到记搜可以解决这个问题。（方法一）
- 可以发现，如果我们知道前r-1行的答案，就可以推出第r行的答案。

一. 动态规划原理

- 定义 f[i][j] 为经过第i行第j列时数字和的最大值
- 状态转移方程为：

$$f_{i,j} = max(f_{i-1,j-1}, f_{i-1,j}) + a_{i,j}$$

一. 动态规划原理

- 最优子结构：问题可拆分
- 无后效性：子问题不影响后续决策
- 子问题重叠：牺牲空间储存子问题的解

二. 线性DP

- 典例：求斐波那契数列前n项(n<=1e7)

$$f_i = f_{i-2} + f_{i-1}$$

二. 线性DP

- 最长上升子序列(LIS)：[洛谷B3637]

给出一个由 $n(n \leq 5000)$ 个不超过 10^6 的正整数组成的序列。请输出这个序列的**最长上升子序列**的长度。

最长上升子序列是指，从原序列中**按顺序**取出一些数字排在一起，这些数字是**逐渐增大**的。

二. 线性DP

- 令f[i]为 以第i个元素(a[i])为LIS的结尾的长度。
- 考虑如果知道了f[1…i-1]，怎么推出f[i]。
- a[i]能接在某个序列的后面当且仅当它大于序列的最后一个元素。
- 所以f[i]能从f[j]转移过来的充要条件是a[i]>a[j]

$$f_i = \max_{j=1}^{i-1}(f_j + 1)[a_i > a_j]$$

- 答案就是所有f[i]的最大值。

二. 线性DP

- 如果我把数据加大，n<=1e6，阁下又该如何应对？
- 另一种设状态的方式：f[i][j]为枚举到a[i]，长度为j的序列，最后一个元素最小是f[i][j]。
- 为什么这么设？贪心思想：小的末尾可以接纳更多的数接在后面。
- 转移就很显然了：

$$f_{i,j} = \begin{cases} min(f_{i-1,j}, a_i) & a_i > f_{i-1,j-1} \\ f_{i-1,j} & otherwise \end{cases}$$

二. 线性DP

- 优化1：可以发现，i那一维是完全没必要的，只需要令f[]为长度为j的序列末尾元素的最小值即可。
- 优化2：显然，f[]是单调递增的，于是转移那一步可以通过二分实现。

```
vector<int> a(n+1);
for(int i=1; i<=n; i++)
{
    cin>>a[i];
}
vector<int> f;
for(int i=1; i<=n; i++)
{
    int t=lower_bound(f.begin(),f.end(),a[i])-f.begin();
    if(t==f.size())//比序列中任何元素都大
        f.push_back(a[i]);
    else f[t]=a[i];
}
cout<<f.size();
```

二. 线性DP

- 最长公共子序列（LCS）
- 给定一个长度为 n 的序列 A 和一个长度为 m 的序列 B，求出一个最长的序列，使得该序列既是 A 的子序列，又是 B 的子序列。
- n, m<=5000

二. 线性DP

- 令 f[i][j] 为 A 枚举到 a[i], B 枚举到 b[j]，LCS最长的长度。
- 大家试着自己写一下转移方程。

二. 线性DP

- Bonus：给定两个长度为 n 的排列 P1, P2，求LCS。
- 排列的定义：如果一个长度为n的序列包括了 1 到 n 的每一个整数，那它就是排列（这意味着序列中的元素两两不同）。
- n<=1e6

二. 线性DP

- 思考1：可以换一种方法设状态吗？
- 思考2：怎么优化？
- 这题留作作业

三. 背包问题之0/1背包

- 有 n 个物品和一个**容量**为 W 的背包，每个物品有**重量** w[i] 和**价值** v[i] 两种属性，要求选若干物品放入背包使背包中物品的**总价值最大**且背包中物品的**总重量不超过背包的容量**。
- 上述问题对物品的取与不取分别对应二进制中的0和1，所以被称为0/1背包问题。
- 洛谷P2871

三. 背包问题之0/1背包

设 DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值。

考虑转移。假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为 $f_{i-1,j}$；当其放入背包时，背包的剩余容量会减小 w_i，背包中物品的总价值会增大 v_i，故这种情况的最大价值为 $f_{i-1,j-w_i}+v_i$。

由此可以得出状态转移方程：

$$f_{i,j}=\max(f_{i-1,j},f_{i-1,j-w_i}+v_i)$$

由于对 f_i 有影响的只有 f_{i-1}，可以去掉第一维，直接用 f_i 来表示处理到当前物品时背包容量为 i 的最大价值，得出以下方程：

$$f_j=\max\left(f_j,f_{j-w_i}+v_i\right)$$

三. 背包问题之0/1背包

- 需要注意的是，去掉 i 那一维之后，j 必须倒着枚举，这样才能保证转移过来的状态是 i-1 那一维的。不然就会导致一个物品被选多次。

```
for(int i=1; i<=n; i++)
{
    for(int j=W; j>=w[i]; j--)
        f[j]=max(f[j],f[j-w[i]]+v[i]);
}
```

三. 背包问题之0/1背包

- 洛谷P1049加强版
- 有N个物品，空间为W的背包，每个物品占用的空间为w[i]，求在不超过背包空间的情况下背包剩余空间最小是多少。
- N, W<=50000

三. 背包问题之0/1背包

- 如果不考虑时间复杂度，显然可以背包做。
- f[i] 为是否存在一种方案是的选择物品空间和为 i。
- 转移: $f_j = f_j | f_{j-w_i}$
- 时间复杂度：O(NW)

三. 背包问题之0/1背包

- Bitset优化：std::bitset 是标准库中的一个存储 0/1 的大小不可变容器。换句话说，它是含有 n 个 bool 变量的容器，可以像 int 型一样做位运算，可以像数组一样查看某一位是几。
- Bitset的时间复杂度：$O(\frac{n}{w})$，其中 ω 是计算机位数，现在一般取 64。
- https://oi-wiki.org/lang/csl/bitset/

三. 背包问题之0/1背包

```
bitset<N> s;
void O_o()
{
    int n,V;
    cin>>V>>n;
    s[0]=1;
    for(int i=1; i<=n; i++)
    {
        int x;
        cin>>x;
        s|=s<<x;
    }
    for(int i=V; i>=0; i--)
    {
        if(s[i]==1)
        {
            cout<<V-i;
            break;
        }
    }
}
```

三. 背包问题之完全背包

- 有 n 个物品和一个**容量**为 W 的背包，每个物品有**重量** w[i] 和**价值** v[i] 两种属性，**每个物品有无限个**，要求选若干物品放入背包使背包中物品的**总价值最大**且背包中物品的**总重量不超过背包的容量**。
- 它和0/1背包的区别是每件物品可以无限选。
- 洛谷P1616

三. 背包问题之完全背包

- 还记得把0/1背包 j 那维正着枚举会发生什么吗?
- 现在，某个物品被选多次正和我意。

```
for(int i=1; i<=n; i++)
{
    for(int j=w[i]; j<=W; j++)
    {
        f[j]=max(f[j],f[j-w[i]]+v[i]);
    }
}
```

三. 背包问题之多重背包

- 有 n 个物品和一个**容量**为 W 的背包，每个物品有**重量** w[i] 和**价值** v[i] 两种属性，**每个物品有t[i]个**，要求选若干物品放入背包使背包中物品的**总价值最大**且背包中物品的**总重量不超过背包的容量**。
- 这次的限制变成的可以选t[i]个。
- 大家有什么想法?
- 洛谷P1776

三. 背包问题之多重背包

- 方法一：把所有物品都拆成 t[i] 个，做0/1背包
- 时间复杂度 $O(W\sum t_i)$
- 显然容易超时。

三. 背包问题之多重背包

- 观察这个序列：1, 2, 4, …, 2^n
- 证明：这些数通过相加（每个数只能选一次）可以表示小于等于 2^(n+1)-1 的所有非负整数。
- 提示：看成二进制

三. 背包问题之多重背包

- 二进制优化:
- 根据刚刚证明的结论，我们可以将 t[i] 进行二进制拆分:
- t[i]= 2^0+2^1+…+2^p+(t[i]-(2^(p+1)-1))
- 这样，每个物品就被我拆分成了O(log t[i]) 个物品。这样做0/1背包的时间复杂度是 $O(W\sum log\, t_i)$
- 其实还有单调队列优化，但那个太难了。

四. 区间DP

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。

令状态 $f(i,j)$ 表示将下标位置 i 到 j 的所有元素合并能获得的价值的最大值，那么 $f(i,j)=\max\{f(i,k)+f(k+1,j)+cost\}$，$cost$ 为将这两组元素合并起来的价值。

四. 区间DP

- 区间 DP 有以下特点:
- **合并**：即将两个或多个部分进行整合，当然也可以反过来;
- **特征**：能将问题分解为能两两合并的形式;
- **求解**：对整个问题设最优值，枚举合并点，将问题分解为左右两个部分，最后合并两个部分的最优值得到原问题的最优值。

四. 区间DP

- 洛谷P1775石子合并（弱化版）
- N 堆石子排成一排，每堆石子的质量为m[i]。你可以合并相邻两堆石子，代价为两堆石子的质量之和。
- 你需要把所有石子合并成一堆，问最小代价。
- N<=300

四. 区间DP

- 考虑我们已经知道合并区间 [l, x] 和 [x+1, r] (l<=x<=r)的最小代价，如何求合并区间 [l, r] 的最小代价。
- 如果我们选择把[l, x]合并成一堆，[x+1, r] 合并成一堆，那么再把这两堆合起来就是合并 [l, r] 了。那么代价就是这次合并的代价，加上之前合并的代价。
- 设 f[l][r] 为合并区间 [l, r] 的代价，那么可以写出转移方程

$$f_{l,r} = \min_{x=l}^{r}(f_{l,x} + f_{x+1,r}) + \sum_{i=l}^{r} m_i$$

- 后面的求和显然可以用前缀和优化。
- 时间复杂度O(N^3)

四. 区间DP

- 思考：如果这 N 堆石子围成一个环，其他条件不变，怎么办？

四. 区间DP

- 思考：如果这 N 堆石子围成一个环，其他条件不变，怎么办？
- 方法一：枚举断点，将环拆成一条链。时间复杂度O(N^4)
- 方法二：将序列复制一遍接在末尾，正常做，答案就是长度为 n 的区间。时间复杂度O(N^3)
- 为什么方法二是对的？
- 无论从哪个位置开始往后面数 N 个，就是方法一中拆出来的一条链。所以这样就涵盖了所有情况。

四. 区间DP

- 例题：洛谷P4170
- 给你一个长度为 n 的颜色序列。每次你可以选择一个区间[l,r]给它涂上同一种颜色。问将初始全部无色的序列变成颜色序列需要多少步？
- n<=50

四. 区间DP

- 例题: 洛谷P1063
- n 颗珠子围成一个环，每颗珠子可以用二元组 (x, y) 表示，并且保证 $y_i = x_{i+1}$，$y_n = x_1$。你现在可以对两颗相邻的珠子进行聚合，聚合时你可以获得的能量为 $x_i * y_i * x_{i+1}$。问你最多能获得多少能量。

四. 区间DP

- 例题: 洛谷P3205
- 给定一个理想序列，求有多少种初始序列经过特定规则排列之后会变成理想序列。
- 构造规则:
- 第一个数直接加入序列。
- 如果第 i 个数小于序列中第一个数，则它加入序列左边
- 如果第 i 个数大于序列中第一个数，则它加入序列右边