《L4-CSP-J 复赛进阶算法与数据结构》

Day07 动态规划-Dynamic Programming

线性动态规划

主讲人：小武老师

可达信奥_小武老师_keda.ac

线性DP

线性动态规划，是较常见的一类动态规划问题，其是在线性结构上进行状态转移。线性动态规划的目标函数为特定变量的线性函数，约束是这些变量的线性不等式或等式，目的是求目标函数的最大值或最小值。

可达信奥_小武老师_keda.ac

DP的三要素

状态、阶段和决策是构成DP算法的三要素，而子问题重叠性、无后效性和最优子结构性质是问题能用DP求解的三个基本条件。动态规划是解决多阶段决策过程最优化问题的一种方法。

状态

某一阶段的出发位置称为状态。通常一个阶段包含若干状态。

阶段

把问题分成几个相互联系的有顺序的几个环节，这些环节即称为阶段。

决策/转移方程

从某阶段的一个状态演变到下一个阶段某状态的选择。也就是从哪来要到哪去。前一阶段的终点就是后一阶段的起点，前一阶段的决策选择导出了后一阶段的状态，这种关系描述了由i阶段到i+1阶段状态的演变规律，称为状态转移方程。

DP的三个基本条件

状态、阶段和决策是构成DP算法的三要素，而子问题重叠性、无后效性和最优子结构性质是问题能用DP求解的三个基本条件。

子问题重叠性

首先，题目要满足具有相同子问题。DP算法把原问题视作若干个重叠子问题的逐层递进，每个子问题的求解过程都构成一个“阶段”。在完成前一个阶段的计算后，DP才会执行下一阶段的计算。

无后效性

为了保证DP中每一阶段的计算能够按顺序、不重复的进行，DP要求已经求解的子问题不受后续阶段的影响。

最优子结构

下一阶段的最优解能够由前面各阶段子问题的最优解导出。

DP的三板斧

我是谁？我从哪里来？我要到哪里去？

第一步：状态定义 （我是谁？我在哪？）

就是定义子问题，如何表示目标规模的问题和更小规模的问题，写出原问题和子问题，以及dp数组。例如常见的方法：定义状态 dp[n]，表示规模为 n 的问题的解，dp[n - 1] 就表示规模为 n - 1 的子问题的解。

第二步：状态转移 （我从哪里来？/我到哪里去？）

就是子问题之间的关系，例如定义好状态 dp[n]，此时子问题是 dp[n-1] 等，并且大规模的问题的解依赖小规模问题的解，此时需要知道怎样通过小规模问题的解推出大规模问题的解。

第三步：优化及实现

一般来说动态规划的优化分为空间优化和时间优化。对于空间我们最经常讨论能不能滚动数组？需不需要离散化？之类的。而对于时间优化，我们最常讨论的是能不能通过矩阵快速幂进行加速？之类的。

线性DP

线性动态规划是较常见的一类动态规划问题，其是在线性结构上进行状态转移。

线性动态规划的目标函数为特定变量的线性函数，约束是这些变量的线性不等式或等式，目的是求目标函数的最大值或最小值。

动态规划用一句话概括就是对各个状态维度进行**分阶段、有顺序、无重复、决策性**的遍历求解。其中对阶段进行划分后，每个阶段就是一个子问题。

- 线性DP: 具有线性阶段划分的 DP 问题
- 树形DP: 以节点的深度作为阶段的 DP 问题
- 图上DP: 以节点在图上的拓扑序作为阶段的 DP 问题

例题选讲

P0449 最长上升子序列

P0446 合唱队形

P0444 挖地雷

P0442 导弹拦截

P0440 数字金字塔

P0477 过河卒

P0449 最长上升子序列

设计状态

以$f[x]$表示“以$a[x]$结尾的上升子序列，最长有多长”！

那么，答案就是$f[1]$，$f[2]...f[n]$里面的最大值。

问题来了，如何求出f数组？提示：思考$f[x]$**从哪里来**。

$f[x]$表达的是“以$a[x]$结尾的最长的上升子序列长度”。这个最长的子序列，一定是把$a[x]$接在某个上升子序列尾部形成的！

P0449 最长上升子序列

$f[x]$ 表达的是“以$a[x]$ 结尾的最长的上升子序列长度”。这个最长的子序列，一定是把$a[x]$ 接在某个上升子序列尾部形成的！

数组a:[1,3,4,7,2,6,8,5]

考虑f[8]，它的来源有：

自己一个元素作为一个序列。长度为1.

接在a[1] 后面。长度为f[1]+1=2

接在a[2] 后面。长度为f[2]+1=3

接在a[3] 后面。长度为f[3]+1=4

接在a[5] 后面。长度为f[5]+1=3

$$f[x] = \max_{p<x, a[p]<a[x]} \{f[p]+1\}$$

P0449 最长上升子序列

```cpp
#include<bits/stdc++.h>
using namespace std;
int a[10001];
int dp[10001];
int main(){
    int n, maxn = 0;
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i];
    for(int i = 1; i <= n; i++){
        dp[i] = 1;
        for(int j = 1; j < i; j++){
            if(a[j] < a[i]) dp[i] = max(dp[j]+1, dp[i]);
        }
        maxn = max(maxn, dp[i]);
    }
    cout << maxn;
    return 0;
}
```

$$f[x] = \max_{p<x, a[p]<a[x]} \{f[p] + 1\}$$

P0442 导弹拦截

问题1：导弹拦截的高度只能和上一次的拦截高度相同或者更低，故最长不上升子序列的长度就是问题1的答案。

设计状态

d[i]表示以i结尾的最长不上升子序列的长度，答案：ans= max(d[i]);

状态转移

$$dp[x] = \max_{p<x, a[p] \geq a[x]} \{dp[p] + 1\}$$

P0442 导弹拦截

问题2：一个不下降子序列需要一套系统，最少的不下降子序列数量就是问题2的答案。最少不下降子序列的数量=最长上升子序列的长度，问题转换为求最长上升子序列。（每高度增加）

设计状态

d[i]表示以i结尾的最长上升子序列的长度，答案：ans= max(d[i]);

状态转移

d[i] = max(d[j]) +1; (j<i&& a[j]>= a[i])。

$$f[x] = \max_{p<x, a[p]<a[x]} \{f[p] + 1\}$$

P0442 导弹拦截

```cpp
#define M 1010
int dp_down[M], dp_up[M], a[M];
int ans_down = -1, ans_up = -1;
int main() {
    int n = 1;
    while (cin >> a[n]) {
        dp_down[n] = dp_up[n] = 1;
        n++;
    }
    n--;
    for(int i = 2; i <= n; i++)
        for(int j = 1; j < i; j++) {
            if (a[j] < a[i])
                dp_up[i] = max(dp_up[i], dp_up[j] + 1);
            if (a[j] >= a[i])
                dp_down[i] = max(dp_down[i], dp_down[j] + 1);
        }
    for(int i = 1; i <= n; i++) {
        ans_down = max(ans_down, dp_down[i]);
        ans_up = max(ans_up, dp_up[i]);
    }
    cout << ans_down << endl << ans_up;
    return 0;
}
```

P0446 合唱队形

题目分析：

枚举中间点 `ti`，分别求左边的最长上升子序列和右边的最长递减子序列。求右边的最长递减子序列的时候，可以转化成从右到左的最长上升子序列。把两边出队的人数相加就得到了全部的出队人数。

$f[x]$ 表达的是“以$a[x]$ 结尾的最长的上升子序列长度”。这个最长的子序列，一定是把$a[x]$ 接在某个上升子序列尾部形成的！

P0446 合唱队形

题目分析：

枚举中间点 `ti`，分别求左边的最长上升子序列和右边的最长递减子序列。求右边的最长递减子序列的时候，可以转化成从右到左的最长上升子序列。把两边出队的人数相加就得到了全部的出队人数。

状态定义

$f[x]$ 表达的是“以$a[x]$ 结尾的最长的上升子序列长度”。

状态转移

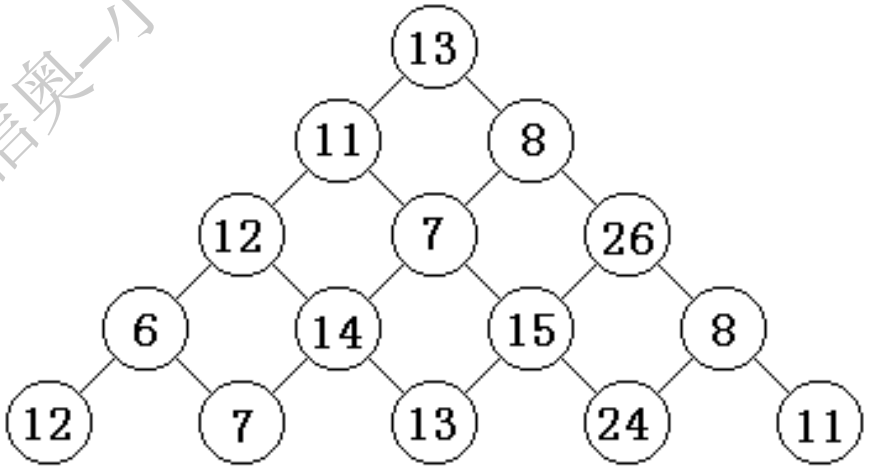
$$f[x] = \max_{p<x, a[p]<a[x]} \{f[p] + 1\}$$

P0446 合唱队形

$$f[x]=\max_{p<x,a[p]<a[x]}\{f[p]+1\}$$

```
#include<bits/stdc++.h>
#define MAXN 110
#define INF 0x3f3f3f3f
using namespace std;
int
h[MAXN],dp_a[MAXN],dp_b[MAXN],m=-INF;
int main(){
    int n;
    cin>>n;
    for(int i=1;i<=n;i++)cin>>h[i];
    dp_a[1]=1;
    dp_b[n]=1;
    //正着求最长上升子序列
    for(int i=2;i<=n;i++){
        dp_a[i]=1;
        for(int j=1;j<i;j++){
            if(h[j] < h[i]) dp_a[i] = max(dp_a[i],dp_a[j]+1);
        }
    }
    //倒着求最长上升子序列
    for(int i=n-1;i>=1;i--){
        dp_b[i]=1;
        for(int j=n;j>i;j--){
            if(h[j] < h[i]) dp_b[i]=max(dp_b[i],dp_b[j]+1);
        }
    }
    for(int i=1;i<=n;i++) m = max(dp_a[i]+dp_b[i]-1,m);
    cout<<n-m;
    return 0;
}
```

P0440 数字金字塔



0	0	0	0	0	0
0	13				
0	11	8			
0	12	7	26		
0	6	14	15	8	
0	12	7	13	24	11

P0440 数字金字塔

题目分析：

dp[i][j] 代表从上往下到达 i 行 j 列这个点所能达到的和的最大值。从上往下到达（i，j）这个点所能达到的和的最大值，即为其上方和左上方两个数中较大的那个加上它本身的值。

状态定义

dp[i][j] 代表从上往下到达 i 行 j 列这个点所能达到的和的最大值。

状态转移

```
dp[i][j]=max(dp[i-1][j-1],dp[i-1][j])+a[i][j];
```

P0440 数字金字塔

```cpp
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;
#define N 1000 + 1
int a[N][N], n, dp[N][N], ans = -1;
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++) cin >> a[i][j];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
            dp[i][j] = max(dp[i - 1][j - 1], dp[i - 1][j]) + a[i][j];
    for (int i = 1; i <= n; i++) ans = max(ans, dp[n][i]);
    cout << ans << endl;
    return 0;
}
```

课后习题与实验

Talk is cheap, show me the code !

下节课见啦！