线性DP/区间dp

动态规划的基本术语

- 动态规划：运筹学中的一个分支，是求解决策过程最优化的数学方法。
- 阶段：把所给求解问题的过程恰当地分成若干个相互联系的阶段。
- 状态：状态表示每个阶段开始面临的自然状况或客观条件。
- 决策（转移）：一个阶段的状态给定以后，从该状态演变到下一阶段某个状态的一种选择称为决策。
- 策略：每阶段都做一个决策，一系列决策的集合。
- 边界：初始集合。

动态规划的性质

- 最优子结构：一个最优化策略的子策略总是最优的，反过来，我们可以通过最优的子策略，推出最优策略。
- 无后效性：当我们通过一系列策略到达了某一阶段的某一状态时，下一步决策不受之前的一系列策略影响，仅由当前状态决定。
- 子问题重叠：算法计算的过程中会反复地求解相同的一定量的子问题，而不是不断生成没有见过的新问题。也就是说子问题空间不大，或是状态空间不大，我们可以通过存储状态的答案加快计算速度。

动态规划的步骤

- 设计状态，要知道自己的每一维度代表什么，当前这个状态表示什么
- 确定初始状态，我们手动给出最基本的子状态，才能转移
- 思考决策，设计状态转移式，也可以想成决策的具体实现

线性dp

线性dp是用来解决一些 **线性区间上的最优化问题**

经典的题目和变种题有很多，比如`LIS，LCS，LCIS`等等

举几个例子

LIS-最长上升子序列

1. 例如`5，1，3，2，7，4，6`这个序列
2. 首先我们设计好状态，`dp[i]`表示以`a[i]`结尾的最长上升子序列长度
3. 状态设计好，我们就可以定义初始状态了，`dp[i]=1（1<=i<=n）`，最少也能得到本身的1个长度
4. 然后我们来设计状态转移方程，每次拿出`a[i]`去比对它前面的数，如果有小于它的数，那么我们就更新状态，这是容易想到的（这部分可以理解为策略），那么状态的转移方程可以设计为，因为它当前的状态都是由比它小的最优子结构更新上来的，且可能更新多次，所以设计为`dp[i]=max(dp[i],dp[j]+1)`，`dp[i]`是因为可能更新多次，要保持最优，`dp[j]+1`是尝试，看看这个序列加上之后能否比前面的最优长度长
5. 最后我们按照设计好的来模拟一遍序列
6. 这是`n^2`的解法

1. 还是例如`5，1，3，2，7，4，6`这个序列

2. `nlogn`复杂度的解法本质上是个贪心+二分优化
3. 因为尾部的数越小，可能更新的序列长度就越长，那我们每次只要判断，如果大于就添加到尾部，小于就找到前面第一个大于等于它的数替换掉，为什么可以用二分来进行优化，因为这个序列必定是一个有序增长序列
4. 这种写法得到的`LIS`是正确的，但序列不一定正确，例如`5, 1, 3, 2, 7, 6, 8, 4`，这个最后求出来的序列就是不正确的，因为我们维护的数组是维护一个最小的可能序列，使得后续的操作尽可能的加更多的数.
5. 这是`nlogn`的解法

lower_bound与upper_bound

- lower_bound会找出序列中第一个大于等于x的数
- upper_bound会找出序列中第一个大于x的数
- **使用：**`lower_bound(a+1,a+1+n,x,cmp)`，默认是升序，当然比较器默认也是<
- `bool cmp(const int& a,const int& b){return a > b;}`
- 也可以`lower_bound(a+1,a+1+n,x,greater<int>())`
- `greater<int>()`是`c++`友情提供的大于函数，懒人必备
- 这两个函数的存在的意义就是为了让我们更方便的偷懒
- `lower_bound(a+1,a+1+n,x)`的返回值是你查找到的值的指针，提供两种用法
- **第一种**，指针受难者福音：`int p=lower_bound(a+1,a+1+n,x)-a`用我们得到的指针减去数组开头的指针(也就是数组名)，就能得到查找到的数的下标.
- **第二种**，指针好，指针妙，指针真奇妙：`*p=lower_bound(a+1,a+1+n,x)`，这样*p就是我们要找的值，

  也可以更直接一点`*lower_bound(a+1,a+1+n,x)=y`

LCS-最长公共子序列

- `dp[i][j]`表示第一个串的第i个字符，第二个串的第j个字符，这个范围内的最大LCS
- 思考决策，相等就更新，不相等就继承，设计状态转移式
- $$\begin{cases} dp[i][j]=max(dp[i][j]\ ,\ dp[i-1][j-1]+1) & \text{if } a[i]=b[j] \\ dp[i][j]=max(dp[i-1][j]\ ,\ dp[i][j-1]) & \text{if } a[i]!=b[j] \end{cases}$$
- 可以推个状态表辅助理解，dp的精髓就在于状态表，表对了才有信心能A掉题.

区间dp

- 顾名思义：区间dp就是在区间上进行动态规划，求解一段区间上的最优解。主要是通过合并小区间的最优解进而得出整个大区间上最优解的dp算法。
- 区间dp的状态设计比较简单，大部分都是`dp[i][j]`表示`i-j`这个区间的最值，最优解等等
- 区间dp的主要变化就是在第三层for和决策以及区间转移式的思考.
- p3146 区间dp模板题
- 给定一个1*n的地图，在里面玩2048，每次可以合并相邻两个（数值范围1-40），问最大能合出多少。注意合并后的数值并非加倍而是+1，例如2与2合并后的数值为3。

基本模板

```
//#pragma GCC oplmize(2)
#include <bits/stdc++.h>
#define ll long long
#define sc(x) scanf("%lld",&x)
#define scs(x) scanf("%s",x)
```

```
#define pr(x) printf("%lld\n",x)
#define prs(x) printf("%s\n",x)
using namespace std;
const int maxn=1e3+5;
const int mod=998244353;
const double pi=acos(-1.0);
const double eps = 1e-8;
ll n,dp[300][300],ma=-0x3f3f3f3f;

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    sc(n);
    for(int i=1;i<=n;i++){
        sc(dp[i][i]);
        ma=max(ma,dp[i][i]);
    }
    for(int i=2;i<=n;i++){    //枚举区间长度
        for(int j=1;j<=n-i+1;j++){    //枚举左端点
            for(int k=j;k<i+j-1;k++){       //枚举断点
                if(dp[j][k]==dp[k+1][i+j-1] && dp[j][k]) {
                    dp[j][i+j-1]=max(dp[j][i+j-1],dp[j][k]+1);
                    ma=max(ma,dp[j][i+j-1]);
                }
            }
        }
    }
    pr(ma);
    return 0;
}
```