区间DP

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中的一种，它以**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包含于它的区间所代表的状态转移而来，因此区间DP的决策往往就是划分区间的方法。区间DP的初态一般就由长度为1的**元区间**构成。

6118 石子合并

题目描述

设有 N 堆石子排成一排，其编号为 $1, 2, 3, \cdots, N$。

每堆石子有一定的质量 m_i。现在要将这 N 堆石子合并成为一堆。每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和，合并后与这两堆石子相邻的石子将和新堆相邻。合并时由于选择的顺序不同，合并的总代价也不相同。试找出一种合理的方法，使总的代价最小，并输出最小代价。

输入格式

第一行，一个整数 $N(1 \leq N \leq 300)$。

第二行，N 个整数 $m_i(1 \leq m_i \leq 1000)$。

输出格式

输出文件仅一个整数，也就是最小代价。

样例输入

```
4
2 5 3 1
```

样例输出

```
22
```

Solution

这是区间dp的经典模型之一，我们设计状态$\mathrm{dp}[l][r]$为合并**下标**为$[l, l+1, \cdots, r]$的石子所需要的**最小代价**

考虑如何进行状态转移，我们枚举将区间$[l, r]$的石子合并成一堆的**最后一次**合并，该次合并的代价是一定的，即我们$sum(l, r) = m_l + m_{l+1} + \cdots + m_r$在该次合并前，区间会被某个位置$k$分为$[l, k], [k+1, r]$两个小区间，因此我们可以枚举$k$，并得到转移方程如下：

$$\mathrm{dp}[l][r] = \min(\mathrm{dp}[l][k] + \mathrm{dp}[k+1][r]) + sum(l, r)$$

Code

```
void solve(){
    int n; cin >> n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;
    for (int i = 1; i < n; i++) a[i] += a[i - 1];

    auto get = [&](int l, int r) {
        return a[r] - (l ? a[l - 1] : 0);
```

```
    };

    vector dp(n, vector(n, inf));
    for (int i = 0; i < n; i++) dp[i][i] = 0;

    for (int i = 1; i < n; i++) for (int j = 0; j + i < n; j++)
    for (int k = j; k < i + j; k++) {
        cmin(dp[j][i + j], dp[j][k] + dp[k + 1][i + j] + get(j, i + j));
    }

    cout << dp[0][n - 1] << endl;
}
```

152 石子合并2

题目描述

在一个圆形操场的四周摆放 N 堆石子，现要将石子有次序地合并成一堆，规定每次只能选相邻的 2 堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。

试设计出一个算法,计算出将 N 堆石子合并成 1 堆的最小得分和最大得分。

输入格式

数据的第 1 行是正整数 $N(1 \leq N \leq 100)$，表示有 N 堆石子。

第 2 行有 N 个整数，第 i 个整数 $a_i(0 \leq a_i \leq 10000)$ 表示第 i 堆石子的个数。

输出格式

输出共 2 行，第 1 行为最小得分，第 2 行为最大得分。

样例输入

```
4
4 5 9 4
```

样例输出

```
43
54
```

Solution

该题总体类似于普通的石子合并，唯二的区别在于需要求一个**最大得分**和**环状**

对于最大得分而言，只需要将最小得分的转移方程中的$\min$改成$\max$即可

$$\mathrm{dp}[l][r] = \max(\mathrm{dp}[l][k] + \mathrm{dp}[k+1][r]) + sum(l, r)$$

而对于**环状**而言，我们可以采用很经典的方法：破环为链，具体操作是将原数组复制一份，并加到原数组的**末尾**，这样，我们数组$[1, n]$就代表原数组的$[1, n]$区间，$[2, n+1]$则代表原数组$[2, n] + [1, 1]$区间，以此类推

我们在合并石子的时候，由于只进行了$n-1$次操作，则一定存在相邻的某两堆没有**直接合并**，按照如上方法破环为链后，不会影响答案的正确性

最后，我们在$\mathrm{dp}[i][n+i-1]$中间寻找相应的最大 / 最小值即可

Code

```
void solve(){
    int n, m; cin >> n;
    m = 2 * n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;
    for (int i = 0; i < n; i++) a.push_back(a[i]);
    for (int i = 1; i < m; i++) a[i] += a[i - 1];

    auto get = [&](int l, int r) {
        return a[r] - (l ? a[l - 1] : 0);
    };

    vector dpmin(m, vector(m, inf)); vector dpmax(m, vector(m, 0));
    for (int i = 0; i < m; i++) dpmin[i][i] = dpmax[i][i] = 0;

    for (int i = 1; i < n; i++) for (int j = 0; j + i < m; j++)
    for (int k = j; k < i + j; k++) {
        cmin(dpmin[j][i + j], dpmin[j][k] + dpmin[k + 1][i + j] + get(j, i + j));
        cmax(dpmax[j][i + j], dpmax[j][k] + dpmax[k + 1][i + j] + get(j, i + j));
    }

    int ansmin = inf, ansmax = 0;
    for (int i = 0; i < n; i++) {
        cmin(ansmin, dpmin[i][i + n - 1]);
        cmax(ansmax, dpmax[i][i + n - 1]);
    }

    cout << ansmin << endl << ansmax << endl;
}
```

45978 「POJ3186」Treats for the Cows

题目描述

约翰经常给产奶量高的奶牛发特殊津贴，于是很快奶牛们拥有了大笔不知该怎么花的钱。为此，约翰购置了 N 份美味的零食来卖给奶牛们。每天约翰售出一份零食。当然约翰希望这些零食全部售出后能得到最大的收益，这些零食有以下这些有趣的特性：

- 零食按照 $1, \ldots, N$ 编号，它们被排成一列放在一个很长的盒子里。盒子的两端都有开口，约翰每天可以从盒子的任一端取出最外面的一个。
- 与美酒与好吃的奶酪相似，这些零食储存得越久就越好吃。当然，这样约翰就可以把它们卖出更高的价钱。
- 每份零食的初始价值不一定相同。约翰进货时，第i份零食的初始价值为 V_i。
- 第i份零食如果在被买进后的第 a 天出售，则它的售价是 $V_i \times a$。

V_i 的是从盒子顶端往下的第i份零食的初始价值。约翰告诉了你所有零食的初始价值，并希望你能帮他计算一下，在这些零食全被卖出后，他最多能得到多少钱。

输入格式

第 1 行：一个整数，$N(1 \leq N \leq 2000)$。

第 2 行至第 $N+1$ 行：第 $i+1$ 行一个整数，奖励 $V_i(1 \leq V_i \leq 1000)$ 的价值。

输出格式

第 1 行：通过出售奖励，农夫约翰能够实现的最大收入。

样例输入

```
5
1
3
1
5
2
```

样例输出

```
43
```

Solution

我们假设$\mathrm{dp}[l][r]$为剩下零食区间为$[l,r]$时，能够获得的**最大收益**，那么，我们**上一次**拿走的零食，要么是$l-1$，要么是$r+1$

转移方程就很好得出来了：

$$\mathrm{dp}[l][r]=\max(\mathrm{dp}[l-1][r]+V_{l-1}\times t,\ \mathrm{dp}[l][r+1]+V_{r+1}\times t)$$

其中$t=n-(r-l+1)$

Code

```
void solve(){
    int n; cin >> n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;

    vector dp(n + 1, vector(n + 1, 0));

    for (int len = n - 1; len > 0; len--) for (int l = 0; l + len - 1 < n; l++) {
        int r = l + len - 1;
        if (r < n - 1) cmax(dp[l][r], dp[l][r + 1] + (n - len) * a[r + 1]);
        if (l) cmax(dp[l][r], dp[l - 1][r] + (n - len) * a[l - 1]);
    }

    int ans = 0;
    for (int i = 0; i < n; i++) cmax(ans, dp[i][i] + a[i] * n);
    cout << ans << endl;
}
```

27749 「dpN」Slimes

题目描述

有N个史莱姆排成一行。最初，从左边数起第i个史莱姆的大小为a_i。

太郎正在尝试将所有的史莱姆合并成一个更大的史莱姆。他将反复执行以下操作，直到只剩下一个史莱姆：

选择两个相邻的史莱姆，并将它们合并成一个新的史莱姆。新史莱姆的大小为$x+y$，其中x和y是合并前史莱姆的大小。这里，会产生$x+y$的成本。合并史莱姆时，史莱姆的位置关系不变。 找出可能发生的最小总成本。

输入格式

第一行一个整数$N(2 \leq N \leq 400)$，表示史莱姆的数量

第二行N个整数$a_1, a_2, \cdots a_n (1 \leq a_i \leq 10^9)$，表示史莱姆的大小

输出格式

打印可能发生的最小总成本。

样例输入

```
4
10 20 30 40
```

样例输出

```
190
```

Solution

~~双倍经验？但凡测子样例就会发现需要使用~~long long~~的数据类型~~

注意平时的练习代码中尽量不要出现`#define int ling long`之类的字段

Code

```
void solve(){
    int n; cin >> n;
    vector a(n, 0ll);
    for (auto& v: a) cin >> v;
    for (int i = 1; i < n; i++) a[i] += a[i - 1];

    auto get = [&](int l, int r) {
        return a[r] - (l ? a[l - 1] : 0);
    };

    vector dp(n, vector(n, INF));
    for (int i = 0; i < n; i++) dp[i][i] = 0;

    for (int i = 1; i < n; i++) for (int j = 0; j + i < n; j++)
    for (int k = j; k < i + j; k++) {
        cmin(dp[j][i + j], dp[j][k] + dp[k + 1][i + j] + get(j, i + j));
    }

    cout << dp[0][n - 1] << endl;
}
```

49802 删数

题目描述

有N个不同的正整数$x_1, x_2, \cdots x_n$排成一排，我们可以从左边或右边去掉连续的$i(1 \leq i \leq n)$个数（只能从两边删除数），剩下$N-i$个数，再把剩下的数按以上操作处理，直到所有的数都被删除为止。

每次操作都有一个操作价值，比如现在要删除从 位置到 位置上的所有的数。操作价值为$|x_i - x_k| \times (k - i + 1)$，如果只去掉一个数，操作价值为这个数的值。 问如何操作可以得到**最大值**，求操作的最大价值。

输入格式

第一行为一个正整数$N(3 \leq N \leq 100)$;

第二行有N个用空格隔开的正整数$(1 \leq x_i \leq 1000)$。

输出格式

一行，包含一个正整数，为操作的最大值

输入样例

```
6
54 29 196 21 133 118
```

输出样例

```
768
```

Solution

虽然题目限制只能从左边或者右边取，但是我们查看最后的序列，会被若干个**断点**分成一根根独立的线段，我们删除这些线段的先后顺序不会影响最后的结果

我们考虑使用$\mathrm{dp}[l][r]$表示区间$[l,r]$内的数全部删完所能得到的最大值，显然，我们的转移分为两个大类，第一个大类是我们直接将$[l,r]$区间删去，所花费的代价是$|x_l - x_r| \times (r-l+1)$；第二个大类是枚举中间点$k$，对于点$k$的左右区间分别取其最大值

$$\mathrm{dp}[l][r] = \max(\mathrm{dp}[l][r], |x_l - x_r| \times (r-l+1))$$
$$\mathrm{dp}[l][r] = \max(\mathrm{dp}[l][r], \mathrm{dp}[l][k] + \mathrm{dp}[k+1][r])$$

Code

```
void solve() {
    int n; cin >> n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;

    vector dp(n, vector(n, 0));
    for (int i = 0; i < n; i++) dp[i][i] = a[i];

    for (int len = 1; len < n; len++) for (int i = 0; i + len < n; i++) {
        for (int k = i; k < i + len; k++) {
            cmax(dp[i][i + len], dp[i][k] + dp[k + 1][i + len]);
        }
        cmax(dp[i][i + len], abs(a[i] - a[i + len]) * (len + 1));
    }

    cout << dp[0][n - 1] << endl;
}
```

368 戳西瓜

题目描述

有 n 个西瓜，编号为0 到 $n-1$，每个西瓜上都标有一个数字，这些数字存在数组 a_i 中。

现在要求你戳破所有的西瓜。每当你戳破一个西瓜 i 时，你可以获得 $a_l \times a_i \times a_r$ 个硬币。 这里的 l 和 r 代表和 i 相邻的两个西瓜的序号。注意当你戳破了西瓜 i 后，西瓜 l 和西瓜 r 就变成了相邻的西瓜。

求所能获得硬币的最大数量。

输入格式

共两行。

第一行：一个正整数$n(0 \leq n \leq 500)$，代表西瓜个数。

第二行：共n个数字$a_1, a_2 \cdots a_n (0 \leq a_i \leq 100)$，表示西瓜上标的数字。

输出格式

一行，一个整数，获得硬币的最大数量。

样例输入

```
4
3 1 5 8
```

样例输出

```
167
```

Solution

我们用$\mathrm{dp}[l][r]$代表将区间$[l, r]$内的所有西瓜戳破所能获得的最大收益，我们枚举**最后一个**被戳破的西瓜，假设下标是k，那么我们能获得的收益是

$$\mathrm{dp}[l][r] = \max(\mathrm{dp}[l][k-1] + \mathrm{dp}[k+1][r] + a_{l-1} \times a_k \times a_{r+1})$$

从小到大枚举区间即可

Code

```
void solve() {
    int n; cin >> n;
    vector a(n + 2, 0);
    for (int i = 1; i <= n; i++) cin >> a[i];
    a[0] = a[n + 1] = 1;

    vector dp(n + 2, vector(n + 2, 0));
    for (int i = 1; i <= n; i++) dp[i][i] = a[i] * a[i - 1] * a[i + 1];

    for (int len = 1; len < n; len++) for (int i = 1; i + len <= n; i++)
    for (int k = i; k <= i + len; k++) {
        cmax(dp[i][i + len], dp[i][k - 1] + dp[k + 1][i + len] + a[k] * a[i - 1] * a[i +
len + 1]);
    }

    cout << dp[1][n] << endl;
}
```

153 能量项链

题目描述

在 Mars 星球上，每个 Mars 人都随身佩带着一串能量项链。在项链上有 N 颗能量珠。能量珠是一颗有头标记与尾标记的珠子，这些标记对应着某个正整数。并且，对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。因为只有这样，通过吸盘（吸盘是 Mars 人吸收能量的一种器官）的作用，这两颗珠子才能聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。如果前一颗能量珠的头标记为 m，尾标记为 r，后一颗能量珠的头标记为 r，尾标记为 n，则聚合后释放的能量为 $m \times r \times n$（Mars 单位），新产生的珠子的头标记为 m，尾标记为 n。

需要时，Mars 人就用吸盘夹住相邻的两颗珠子，通过聚合得到能量，直到项链上只剩下一颗珠子为止。显然，不同的聚合顺序得到的总能量是不同的，请你设计一个聚合顺序，使一串项链释放出的总能量最大。

例如：设 $N = 4$，4 颗珠子的头标记与尾标记依次为 $(2,3)(3,5)(5,10)(10,2)$。我们用记号 $\oplus$ 表示两颗珠子的聚合操作，$(j \oplus k)$ 表示第 j,k 两颗珠子聚合后所释放的能量。则第 4，1 两颗珠子聚合后释放的能量为：

$(4 \oplus 1) = 10 \times 2 \times 3 = 60$。

这一串项链可以得到最优值的一个聚合顺序所释放的总能量为：

$(((4 \oplus 1) \oplus 2) \oplus 3) = 10 \times 2 \times 3 + 10 \times 3 \times 5 + 10 \times 5 \times 10 = 710$。

输入格式

第一行是一个正整数 N（$4 \leq N \leq 100$），表示项链上珠子的个数。第二行是 N 个用空格隔开的正整数，所有的数均不超过 1000。第 i 个数为第 i 颗珠子的头标记（$1 \leq i \leq N$），当 $i < N$ 时，第 i 颗珠子的尾标记应该等于第 $i+1$ 颗珠子的头标记。第 N 颗珠子的尾标记应该等于第 1 颗珠子的头标记。

至于珠子的顺序，你可以这样确定：将项链放到桌面上，不要出现交叉，随意指定第一颗珠子，然后按顺时针方向确定其他珠子的顺序。

输出格式

一个正整数 E（$E \leq 2.1 \times 10^9$），为一个最优聚合顺序所释放的总能量。

样例输入

```
4
2 3 5 10
```

样例输出

```
710
```

Solution

设$\mathrm{dp}[l][r]$为将区间$[l,r]$的所有珠子合并成**一颗**所得到的最大能量

很显然，我们在将区间$[l,r]$合并成一颗之前，一定会经历**二合一**的状态，因此，我们枚举中间点k，即珠子的状态是$[l,k-1]$合并，$[k,r]$合并，转移方程如下：

$$\mathrm{dp}[l][r] = \max(\mathrm{dp}[l][k-1] + \mathrm{dp}[k][r] + v[l] \times v[k] \times v[r+1])$$

其中，$v[i]$代表i的头标记

由于能量项链是一个环，所以说我们需要**破环成链**，只需要将项链**延长一倍**即可

Code

```
void solve() {
    int n; cin >> n;
    vector a(n, make_pair(0, 0));
    for (auto& [f, s]: a) cin >> f;
    for (int i = 0; i < n; i++) {
        a[i].second = a[(i + 1 + n) % n].first;
        a.emplace_back(a[i]);
    }

    vector dp(2 * n, vector(2 * n, 0));
    for (int len = 1; len < n; len++) for (int i = 0; i + len < 2 * n; i++)
    for (int k = i + 1; k <= i + len; k++) {
        cmax(dp[i][i + len], dp[i][k - 1] + dp[k][i + len] + a[i].first * a[k].first * a[i
+ len].second);
    }

    int ans = 0;
    for (int i = 0; i < n; i++) cmax(ans, dp[i][n - 1 + i]);
    cout << ans;
}
```

[# 8553] 祖玛

题目描述

Mirko 将N颗珠子排成一排，依次编号为$1, 2 \cdots, N$。i号珠子的颜色为c_i。他发现，如果他触摸$\geq T$颗连续的珠子，且这些珠子的颜色相同，魔法会使这些弹子消失；此后，这T颗珠子前面的珠子便与这T颗珠子后面的珠子相邻。

Mirko 家里有很多珠子，他想在这N颗珠子之间（也可以在开头的珠子前面或末尾的珠子后面）插入尽可能少的珠子，使得这N颗珠子和插入的所有珠子消失。

注意：本题Mirko需要触碰不少于T个同颜色的彩球，它们才能全部消失，这意味着初始的彩球只要大于T个，可以自动消失，当然也可以不消失。而且，在某一段彩球消失以后，左右两边的彩球碰到一起，即使他们的是同颜色且超过T个，也可以不消失。

输入格式

第一行：$N(1 \leq N \leq 100), T(2 \leq T \leq 5)$分别表示彩珠串里包含彩珠的个数和能够神奇消失的最少彩珠数。

第二行：$c_1, c_2 \cdots, c_n (1 \leq c_i \leq 100)$，表示彩珠的颜色。（数字相同即表示颜色相同）

输出格式

输出一个整数，表示Mirko需要插入的最少彩珠数。

样例输入

```
10 4
3 3 3 3 2 3 1 1 1 3
```

样例输出

```
4
```

Solution

该题思维难度较大，我们可以一步步地推

观察到对于某一个点而言，添加珠子的数量至多为$T-1$个，因此，我们定义$\mathrm{dp}[l][r][k]$为在l前面**已经**添加了k颗珠子的情况下，完全消去区间$[l,r]$所需的最小添加珠子数

首先是很显然的边界条件

$$\mathrm{dp}[i][i][k]=T-k-1$$

代表着对于单点构成的区间，我们需要补齐剩下的$T-k-1$颗珠子才能完全消去

首先，我们对于某个状态$\mathrm{dp}[l][r][k]$而言，如果在l前添加一颗珠子，即可以视为从$\mathrm{dp}[l][r][k+1]+1$转移而来，这样，我们就得到了第一个转移方程

$$\mathrm{dp}[l][r][k]=\min(\mathrm{dp}[l][r][k],\mathrm{dp}[l][r][k+1]+1)$$

特殊的，如果$k=T-1$了，我们此时相当于能够直接消去l及其之前的所有的珠子，因此我们得到了第二个转移方程

$$\mathrm{dp}[l][r][k]=\min(\mathrm{dp}[l][r][T-1],\mathrm{dp}[l+1][r][0])$$

接着，对于某个l，有着$c_l==c_{l+1}$而言，我们$\mathrm{dp}[l][r][k]$实际上和$\mathrm{dp}[l+1][r][k+1]$等价，因此我们得到了第三个转移方程

$$\mathrm{dp}[l][r][k]=\min(\mathrm{dp}[l][r][k],\mathrm{dp}[l+1][r][k+1])$$

最后，对于某个位置x，如果有$x\in(l+1,r]$且$c_l==c_x$，我们可以选择先将区间$[l+1,x)$内的珠子消去，再将l及其前面的珠子和区间$[x,r]$合并，再进行求解，因此我们得到了最后一个转移方程

$$\mathrm{dp}[l][r][k]=\min(\mathrm{dp}[l][r][k],\mathrm{dp}[l+1][x-1][0]+\mathrm{dp}[x][r][k+1])$$

注意边界情况即可

Code

```
void solve(){
    int n, t; cin >> n >> t;
    vector a(n + 1, 0);
    for (auto& v: a) cin >> v;

    vector dp(n, vector(n, vector(t, inf)));

    for (int i = 0; i < n; i++) for (int j = 0; j < t; j++) dp[i][i][j] = t - j - 1;

    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++)
    for (int k = t - 1; k >= 0; k--) {
        int r = l + len;
        if (k == t - 1) cmin(dp[l][r][k], dp[l + 1][r][0]);
        else cmin(dp[l][r][k], dp[l][r][k + 1] + 1);
        if (a[l] == a[l + 1]) cmin(dp[l][r][k], dp[l + 1][r][min(t - 1, k + 1)]);
        for (int x = l + 1; x < r; x++) if (a[l] == a[x + 1])
            cmin(dp[l][r][k], dp[l + 1][x][0] + dp[x + 1][r][min(t - 1, k + 1)]);

    }

    cout << dp[0][n - 1][0] << endl;
}
```

31895 「CF1312E」Array Shrinking 数组缩减

题目描述

给你一个长度为 $n(1 \leq n \leq 500)$ 的数组 a，每次你可以进行以下两步操作：

1. 找到 $i \in [1, n)$，使得 $a_i = a_{i+1}$；
2. 将 **它们** 替换为 $a_i + 1$。

每轮操作之后，显然数组的长度会减小 1，问剩余数组长度的最小值。

输入格式

第一行包含一个整数 $n(1 \leq n \leq 500)$，表示数组的原长。

第二行包含 n 个整数 $a_1, a_2, \cdots a_n(1 \leq a_i \leq 1000)$，表示原数组 a。

输出格式

共一行仅一个整数 num，表示进行许多轮操作之后，a 剩余长度的最小值。

样例输入

```
5
4 3 2 2 3
```

样例输出

```
2
```

Solution

显然，对于一个区间而言，假设其缩减后长度为1，则其剩下的数无论如何缩减，都是一定的

考虑区间dp，定义$\mathrm{dp}[l][r]$为将区间$[l, r]$进行若干次操作后，剩余长度的最小值

很明显，如果区间超过两个数$a_1, a_2, \cdots, a_x$，我们**没有必要**将其和后面的区间合并，因为我们区间大小是**从小到大枚举**的，如果需要和后面的区间合并，例如$a_x = a_y, a_{x-1} = a_x + 1 \cdots$的情况已经能被覆盖到

我们使用一个辅助数组$w[l][r]$，记录区间$[l, r]$有$\mathrm{dp}[l][r] = 1$时的合并结果

那么初始化就很明显了，$\mathrm{dp}[i][i] = 1, w[i][i] = a[i]$

考虑枚举区间中点k，将区间分为$[l, k], [k + 1, r]$则得到转移方程如下：

$$\mathrm{dp}[l][r] = \min(\mathrm{dp}[l][r], \mathrm{dp}[l][k] + \mathrm{dp}[k + 1][r])$$

特别的，如果$\mathrm{dp}[l][k] = \mathrm{dp}[k + 1][r] = 1$，且有$w[l][k] = w[k + 1][r]$时，我们可以直接将这两个数合并，此时$\mathrm{dp}[l][r] = 1, w[l][r] = w[l][k] + 1$

Code

```
void solve() {
    int n; cin >> n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;

    vector dp(n, vector(n, inf)), w = dp;
    for (int i = 0; i < n; i++) dp[i][i] = 1, w[i][i] = a[i];

    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++)
    for (int k = l; k < l + len; k++) {
        int r = l + len;
```

```
            cmin(dp[l][r], dp[l][k] + dp[k + 1][r]);
            if (dp[l][k] == 1 && dp[k + 1][r] == 1 && w[l][k] == w[k + 1][r])
                dp[l][r] = 1, w[l][r] = w[l][k] + 1;
        }

        cout << dp[0][n - 1] << endl;
    }
```

155 括号配对

题目描述

Hecy 又接了个新任务：BE 处理。BE 中有一类被称为 GBE。

以下是 GBE 的定义：

1. 空表达式是 GBE
2. 如果表达式 `A` 是 GBE，则 `[A]` 与 `(A)` 都是 GBE
3. 如果 `A` 与 `B` 都是 GBE，那么 `AB` 是 GBE

下面给出一个 BE，求至少添加多少字符能使这个 BE 成为 GBE。

输入格式

输入仅一行，为字符串 BE，保证$|BE| \leq 100$，且BE只由字符 `()[]` 构成。

输出格式

输出仅一个整数，表示增加的最少字符数。

样例输入

```
[])
```

样例输出

```
1
```

Solution

该题中，GBE的定义实际上已经把转移方程列举出来了

我们定义$\mathrm{dp}[l][r]$为将区间$[l,r]$转换成GBE的最小操作次数

初始化很简单，对于单个字符而言，我们需要添加一个字符就能将其转换成GBE，即$\mathrm{dp}[i][i]=1$

对于GBE定义二，我们转移方程如下：

$$\mathrm{dp}[l][r]=\min(\mathrm{dp}[l][r],\mathrm{dp}[l+1][r-1])\ \text{where a[l]a[r]} = \text{'()'} \mid \text{'[]'}$$

如果$a[l]$和$a[r]$能够直接匹配，我们只需要将其匹配就好了

而对于GBE定义三，我们枚举$[l,r]$中间的所有位置k即可，转移方程如下

$$\mathrm{dp}[l][r]=\min(\mathrm{dp}[l][r],\mathrm{dp}[l][k]+\mathrm{dp}[k+1][r])$$

Code

```
void solve() {
    string str; cin >> str;
    int n = str.size();
    vector dp(n, vector(n, inf));
    for (int i = 0; i < n; i++) dp[i][i] = 1;

    auto get = [&](int l, int r) {
        if (l > r) return 0;
        if (l < 0 || r < 0 || l >= n || r >= n) return 0;
        return dp[l][r];
    };

    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++)
    for (int k = l; k < l + len; k++) {
        int r = l + len;
        cmin(dp[l][r], dp[l][k] + dp[k + 1][r]);
        if (str[l] == '(' && str[r] == ')') cmin(dp[l][r], get(l + 1, r - 1));
        if (str[l] == '[' && str[r] == ']') cmin(dp[l][r], get(l + 1, r - 1));
    }

    cout << dp[0][n - 1] << endl;
}
```

49894 「UVa11584」Partitioning by Palindromes 划分回文串

题目描述

我们说一个字符序列是回文的，如果它正向和反向写都是一样的。例如，“racecar”就是一个回文，但是“fastcar”不是。

一个字符序列的*划分*是指一个由一个或多个不相交的、非空的、连续字符组成的列表，这些字符组拼接起来可以得到原始序列。例如，（‘race’, ‘car’）是“racecar”的划分成两组的一个例子。

给定一个字符序列，我们总是可以创建这些字符的一个划分，使得划分中的每一组都是回文的！基于这个观察，很自然会问：

给定一个字符串，使得每一组都是回文的，需要的最小组数是多少？

例如：

• “racecar”已经是一个回文，因此它可以被划分成一组。

• “fastcar”不包含任何非平凡的回文，所以它必须被划分为（‘f’, ‘a’, ‘s’, ‘t’, ‘c’, ‘a’, ‘r’）。

• “aaadbccb”可以被划分为（‘aaa’, ‘d’, ‘bccb’）。

输入格式

输入开始于测试案例的数目n。

每个测试案例由一行由1到1000个小写字母组成，中间没有空白。

输出格式

对于每个测试案例，在一行中输出将输入划分成回文组所需要的最小组数。

输入样例

```
3
racecar
fastcar
aaadbccb
```

输出样例

```
1
7
3
```

Solution

我们可以使用**线性dp**的知识，很容易地推出状态$\mathrm{dp}[i]$表示前i个字符划分为回文组所需要的最小组数，而转移方程如下：

$$\mathrm{dp}[i] = \min(\mathrm{dp}[i], \mathrm{dp}[j-1]+1) \text{ where str[j, i] is Palindrome}$$

代表着，如果字符串$str_j, str_{j+1}, \cdots str_i$是回文串时，$\mathrm{dp}[i]$就可以由$\mathrm{dp}[j-1]$转移而来

那么，我们如何确定某个字符串$str[j,i]$是不是回文串呢?

我们能很轻易地想到一个$O(n^3)$的暴力，但在本题会超时

考虑到如果$str[j,i]$不是回文串，$str[j-1,i+1]$也一定不是回文串，而$str[j,i]$是回文串的情况下，如果$str[j-1] = str[i+1]$，那么$str[j-1,i+1]$就是回文串了

观察上述表达式，发现**大区间的解依赖于小区间的解**，因此可以用区间dp处理

假设$is_p[l][r]$表示区间$[l,r]$是否为回文串，转移方程也很好想：

$$is\_p[l][r] = is\_p[l+1][r-1] \ \&\& \ str[l] == str[r]$$

Code

```cpp
void solve(){
    string str; cin >> str;
    int n = str.length();

    vector dp(n, inf);
    vector is_p(n, vector(n, 0));

    for (int len = 0; len < n; len++) for (int l = 0; l + len < n; l++){
        int r = l + len;
        if (str[l] == str[r] && (r - l <= 1 || is_p[l + 1][r - 1])){
            is_p[l][r] = 1;
        }
    }

    dp[0] = 1;
    for (int i = 1; i < n; i++) for (int j = 0; j <= i; j++) {
        if (is_p[j][i]) cmin(dp[i], (j ? dp[j - 1]: 0) + 1);
    }

    cout << dp[n - 1] << endl;
}
```

49895 「UVa11151」Longest Palindrome 最长回文串

题目描述

一个**回文**是指一个字符串从左读和从右读是相同的。例如，"I"、"GAG"和"MADAM"是回文，但"ADAM"不是。这里我们也把*空字符串*当作一个回文。

对于任何非回文的字符串，你总能通过去除一些字母得到一个回文子序列。例如，给定字符串"ADAM"，你移除字母"M"后会得到一个回文"ADA"。

编写一个程序来确定从一个字符串中你能得到的最长的回文长度。

输入格式

输入的第一行包含一个整数$T(1 \leq T \leq 60)$。接下来的T行中，每行是一个字符串$str(|str| \leq 1000)$

对于$\geq 90\%$的测试案例，字符串长度$|str| \leq 255$。

输出格式

对于每个输入字符串，你的程序应该输出通过移除零个或多个字符可以得到的最长的回文长度。

输入样例

```
2
ADAM
MADAM
```

输出样例

```
3
5
```

Solution

我们假设$\mathrm{dp}[l][r]$是区间$[l,r]$通过移除某些字符能得到的**最长回文长度**，有两个显然的转移方程如下：

$$\mathrm{dp}[l][r] = \max(\mathrm{dp}[l][r], \mathrm{dp}[l+1][r])$$
$$\mathrm{dp}[l][r] = \max(\mathrm{dp}[l][r], \mathrm{dp}[l][r-1])$$

分别代表着**直接移除**str_l或str_r

剩下一种转移也比较显然：

$$\mathrm{dp}[l][r] = \max(\mathrm{dp}[l][r], \mathrm{dp}[l+1][r-1]+2) \text{ where } str_l = str_r$$

代表着当$str_l = str_r$时，可以将中间的内容**包**住

边界条件是$\mathrm{dp}[i][i] = 1$

特别的，对于本题而言，可能会出现**空字符串**的情况，需要特判一下

Code

```
char str[1010];

void solve() {
    cin.getline(str, 1010);
    for (int i = 0; i < 1010; i++) if (str[i] == '\r' || str[i] == '\n') str[i] = 0;
    int n = strlen(str);
    vector dp(n, vector(n, 0));
    for (int i = 0; i < n; i++) dp[i][i] = 1;
```

```
    auto get = [&](int l, int r) {
        if (l > r) return 0;
        if (l < 0 || r < 0 || l >= n || r >= n) return 0;
        return dp[l][r];
    };

    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++) {
        int r = l + len;
        if (str[l] == str[r]) cmax(dp[l][r], get(l + 1, r - 1) + 2);
        cmax(dp[l][r], get(l + 1, r));
        cmax(dp[l][r], get(l, r - 1));
    }

    if (!n) cout << 0 << endl;
    else cout << dp[0][n - 1] << endl;
}
```

49896 「Hdu4632」Palindrome subsequence

题目描述

在数学中，子序列是指通过删除另一个序列中的某些元素而得到的序列，删除元素时不改变剩余元素的顺序。例如，序列<A, B, D>是<A, B, C, D, E, F>的一个子序列。

给定一个字符串，你的任务是找出有多少个不同的子序列是回文的。其中，两个子序列不同，当且仅当所有的字符在原序列中的下标**不全相同**

输入格式

第一行只包含一个整数$T(1 \leq T \leq 50)$，即测试用例的数量。每个测试用例包含一个字符串$str, 1 \leq |str| \leq 1000$且只包含小写字母。

输出格式

对于每个测试用例，首先输出用例编号，然后输出给定字符串的不同子序列的数量，答案应该对`10007`取模。

输入样例

```
4
a
aaaaa
goodafternooneveryone
welcometoooxxourproblems
```

输出样例

```
Case 1: 1
Case 2: 31
Case 3: 421
Case 4: 960
```

Solution

我们定义$\mathrm{dp}[l][r]$为字符串区间$[l, r]$的回文子序列数量

和**二维前缀和**类似，我们能很轻易地想到转移方程：

$$\mathrm{dp}[l][r] = \mathrm{dp}[l+1][r] + \mathrm{dp}[l][r-1] - \mathrm{dp}[l+1][r-1]$$

同时，在$str_l = str_r$时，我们就可以同时选用str_l和str_r，这样的**额外**方案数就是$\mathrm{dp}[l+1][r-1]+1$，其中$+1$代表着**只选用**str_l和str_r

Code

```
void solve(){
    static int t = 1;
    cout << "Case " << t++ << ": ";
    string str; cin >> str;
    int n = str.length();

    vector dp(n, vector(n, 0));
    for (int i = 0; i < n; i++) dp[i][i] = 1;

    auto get = [&](int l, int r) {
        if (l > r) return 0;
        if (l < 0 || r < 0 || l >= n || r >= n) return 0;
        return dp[l][r];
    };

    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++) {
        int r = l + len;
        add(dp[l][r], dp[l + 1][r]);
        add(dp[l][r], dp[l][r - 1]);
        dec(dp[l][r], get(l + 1, r - 1));
        if (str[l] == str[r]) add(dp[l][r], get(l + 1, r - 1) + 1);
    }

    cout << dp[0][n - 1] << endl;
}
```

154 凸多边形的划分

题目描述

给定一个具有n个顶点的凸多边形，将顶点从1至n标号，每个顶点的权值都是一个正整数。将这个凸多边形划分成$n-2$个**互不相交**的三角形，试求这些三角形顶点的权值乘积和至少为多少。

输入格式

输入第一行为顶点数$n(1 \le n \le 50)$

第二行依次为顶点1至顶点n的权值$a_1, a_2, \cdots, a_n (1 \le a_i \le 10^9)$。

输出格式

输出仅一行，为这些三角形顶点的权值乘积和的最小值。

样例输入

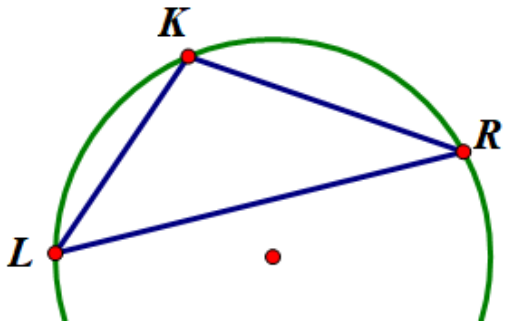
```
5
121 122 123 245 231
```

样例输出

```
12214884
```

Solution

我们假设$\mathrm{dp}[l][r]$为**以**(l,r)**为底边**，点集$[l,r]$划分三角形后权值乘积的最小值，那么我们就可以枚举三角形Δ_{lkr}，如图



显然，我们的转移方程如下：

$$\mathrm{dp}[l][r]=\min(\mathrm{dp}[l][k]+\mathrm{dp}[k][r]+a_l\times a_r\times a_k)$$

其中前两项代表我们区间$[l,k]$和区间$[k,r]$划分三角形后权值乘积的最小值，后项代表三角形Δ_{lkr}的权值

由于凸多边形的顶点序列构成了一个**环**，因此我们需要用到破换成链的方法

Code

```
void solve(){
    int n; cin >> n;
    vector _a(n, 0);
    for (auto& v: _a) cin >> v;
    vector<__int128_t> a;
    for (auto v: _a) a.push_back(v);
    for (auto v: _a) a.push_back(v);

    vector dp(2 * n, vector<__int128_t>(2 * n, 1e30));

    for (int i = 0; i < 2 * n; i++) for (int j = i; j < 2 * n && j < i + 2; j++)
        dp[i][j] = 0;

    for (int len = 2; len < n; len++) for (int l = 0; l + len < 2 * n; l++)
    for (int k = l + 1; k < l + len; k++) {
        int r = l + len;
        cmin(dp[l][r], dp[l][k] + dp[k][r] + a[l] * a[k] * a[r]);
    }

    auto print = [](__int128_t res) {
        vector<int> ans;
        while (res) {
            ans.push_back(res % 10);
            res /= 10;
        }
        reverse(all(ans));
        if (!ans.size()) ans.push_back(0);
        for (auto v: ans) cout << v;
    };

    auto ans = dp[0][n - 1];
    for (int i = 1; i < n; i++) cmax(ans, dp[i][i + n - 1]);
    print(ans);
}
```

5772 多边形（Polygon）

题目描述

多边形是一种单人游戏，开始时在一个具有N个顶点的多边形上，就像图1中所示的那样，其中$N = 4$。每个顶点都标有一个整数，每条边上都标有加号（+）符号或乘号（*）符号。边从1到N编号。

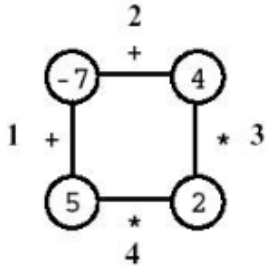


Figure 1. Graphical representation of a polygon

第一步移除其中一条边。随后的移动包括以下步骤：

- 选择一条边E和由E连接的两个顶点V_1和V_2；并用一个新顶点替换它们，新顶点的标签是对V_1和V_2的标签执行E中指示的操作的结果。

当没有更多边时，游戏结束，其得分是剩下的单个顶点所标的整数值。

考虑图1的多边形。玩家首先移除了边3。之后，玩家选择了边1，然后是边4，最后是边2。得分是0。

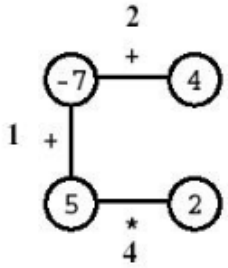


Figure 2. Removing edge **3**

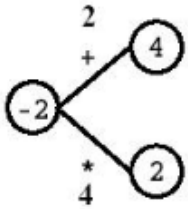


Figure 3. Picking edge **1**

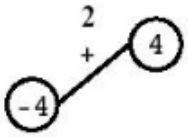


Figure 4. Picking edge **4**

Figure 5. Picking edge **2**

编写一个程序，给定一个多边形，计算可能得到的最高得分，并列出所有的边，如果在第一步移除，可以得到该得分的游戏。

输入格式

输入描述一个有$N(3 \le N \le 50)$个顶点的多边形。它包含两行。

第一行是数N。

第二行包含边$1, 2, \cdots N$的标签，与顶点的标签交错（首先是边1和2之间的顶点的标签，然后是边2和3之间的顶点的标签，依此类推，直到边N和1之间的顶点的标签），所有这些都由一个空格分隔。边的标签要么是字母 `t`（代表 `+`），要么是字母 `x`（代表 `*`）。

保证在任意操作序列中，任意一个顶点上的数均在区间$[-2^{31}, 2^{31} - 1]$内

输出格式

在第一行上，你的程序必须写出输入多边形可以获得的最高得分。

在第二行上，它必须写出所有边的列表，如果在第一步移除，可以达到那个得分的游戏。边必须以**递增**的顺序写出，中间用一个空格隔开。

样例输入

```
4
t -7 t 4 x 2 x 5
```

样例输出

```
33
1 2
```

Solution

我们令$\mathrm{dp}[l][r]$为区间$[l,r]$合并成一个节点后，该节点元素的最大值

考虑最后一次合并$[l,k],[k+1,r]$，如果k和$k+1$之间的符号是**加号**，我们能很简单地得到转移方程

$$\mathrm{dp}[l][r]=\max(\mathrm{dp}[l][r],\mathrm{dp}[l][k]+\mathrm{dp}[k+1][r])$$

但如果k和$k+1$之间的符号是**乘号**，我们的转移方程就没有这么简单了，因为**两个负数乘积为正**，因此，我们同时还要用一个$\mathrm{dp2}[l][r]$数组，维护区间$[l,r]$合并成一个节点后，该节点元素的最**小**值

转移方程，就是所有情况的穷举即可

对于**加法**而言，转移方程如下

$$\begin{aligned}\mathrm{dp}[l][r]&=\max(\mathrm{dp}[l][r],\mathrm{dp}[l][k]+\mathrm{dp}[k+1][r])\\\mathrm{dp2}[l][r]&=\min(\mathrm{dp}[l][r],\mathrm{dp2}[l][k]+\mathrm{dp2}[k+1][r])\end{aligned}$$

而对于**乘法**而言，转移方程如下

$$\begin{aligned}maxx&=\max(\mathrm{dp}[l][k]\times\mathrm{dp}[k+1][r],\mathrm{dp2}[l][k]\times\mathrm{dp}[k+1][r],\mathrm{dp}[l][k]\times\mathrm{dp2}[k+1][r],\mathrm{dp2}[l][k]\times\mathrm{dp2}[k+1][r])\\minn&=\min(\mathrm{dp}[l][k]\times\mathrm{dp}[k+1][r],\mathrm{dp2}[l][k]\times\mathrm{dp}[k+1][r],\mathrm{dp}[l][k]\times\mathrm{dp2}[k+1][r],\mathrm{dp2}[l][k]\times\mathrm{dp2}[k+1][r])\\\mathrm{dp}[l][r]&=\max(\mathrm{dp}[l][r],maxx)\\\mathrm{dp2}[l][r]&=\min(\mathrm{dp}[l][r],minn)\end{aligned}$$

同时，对这道题而言，我们**第一次删去的边**，相当于我们**破坏为链**的那一条边，只需要简单将数组重复一遍，再进行求解即可

Code

```
void solve(){
    int n; cin >> n;
    struct s {
        char op;
        int val;
    };

    vector a(n, s());
    vector dp(2 * n, vector(2 * n, make_pair(-inf, inf)));

    for (auto& [o, v]: a) cin >> o >> v;
    for (int i = 0; i < n; i++) a.push_back(a[i]);
    for (int i = 0; i < 2 * n; i++) dp[i][i] = {a[i].val, a[i].val};

    for (int len = 1; len < n; len++) for (int l = 0; l + len < 2 * n; l++)
```

```
    for (int k = l; k < l + len; k++) {
        int r = l + len;
        if (a[k + 1].op == 't') {
            cmax(dp[l][r].first, dp[l][k].first + dp[k + 1][r].first);
            cmin(dp[l][r].second, dp[l][k].second + dp[k + 1][r].second);
        }
        else {
            vector<int> res;
            res.push_back(dp[l][k].first * dp[k + 1][r].first);
            res.push_back(dp[l][k].first * dp[k + 1][r].second);
            res.push_back(dp[l][k].second * dp[k + 1][r].first);
            res.push_back(dp[l][k].second * dp[k + 1][r].second);
            sort(all(res));
            cmax(dp[l][r].first, res.back());
            cmin(dp[l][r].second, res.front());
        }
    }

    int ans = -inf;
    for (int i = 0; i < n; i++) cmax(ans, dp[i][i + n - 1].first);
    cout << ans << '\n';
    for (int i = 0; i < n; i++) if (dp[i][i + n - 1].first == ans) cout << i + 1 << ' ';
}
```

49946 区间DP1

题目描述

诚诚将在万圣节之夜参加多个聚会，并相应地更换不同的服装，他将多次更换服装。所以，为了使事情变得稍微简单一些，他可能会把一件服装套在另一件服装上面（也就是他可能会在穿着超人服装的外面，再穿上邮差的制服）。

在每个聚会前，他可以脱掉一些服装，或者穿上新的。也就是说，如果他穿着邮差制服在超人服装上面，而想要以超人的服装去参加一个聚会，他可以脱掉邮差制服，或者他可以再穿一件新的超人制服。

但是，请记住，诚诚不喜欢在不清洗的情况下穿衣服，所以，在脱掉邮差制服后，他不能在万圣节当晚再次穿上它，如果他需要再穿邮差服装，他将不得不使用一件新的。

他可以脱掉任意数量的服装，如果他脱掉了k件服装，那么将是最后的k件（例如，如果他在服装A前穿上了服装B，要脱掉A，他必须先脱掉B）。

给定聚会和服装，找出诚诚在万圣节当晚所需的最少服装数量。

输入格式

第一行包含一个整数$n(1 < n < 1000)$表示聚会的数量。

第二行包含n个整数，其中第i个整数c_i表示他将在第i个聚会上穿的服装。他将首先参加聚会1，然后是聚会2，以此类推。

输出格式

输出一个整数，表示所需的最少服装数量。

输入样例

```
7
1 2 1 1 3 2 1
```

输出样例

```
4
```

Solution

对于某两场聚会i, j而言，如果其所需的服装是相同的，有一种显然的策略是：我们在i聚会上，将衣服穿在**最内层**，随后只需要在**外层**增减衣服即可，在第j场聚会时，只需要将所有**外层**的衣服都脱掉

我们令$\mathrm{dp}[l][r]$为参加区间$[l, r]$的聚会，所需服装的最小数量

转移方程如下：

$$\mathrm{dp}[l][r] = \min(\mathrm{dp}[l][r], \mathrm{dp}[l][k] + \mathrm{dp}[k+1][r] - (dp[l] == dp[k+1]))$$

其中，两者之和的部分很显然，而是否减去**后式**，则代表我们l这件衣服能否在区间$[k+1, r]$上复用

Code

```
void solve() {
    int n; cin >> n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;

    vector dp(n, vector(n, inf));
    for (int i = 0; i < n; i++) dp[i][i] = 1;

    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++) {
        int r = l + len, ad = 1;
        for (int k = l + 1; k <= r; k++) {
            if (a[l] == a[k]) cmin(dp[l][r], dp[l][k - 1] + dp[k][r] - 1);
            cmin(dp[l][r], dp[l][k - 1] + dp[k][r]);
        }
    }

    cout << dp[0][n - 1] << endl;
}
```

49947 区间DP2

题目描述

农夫约翰忘记修补农场围栏上的一个洞，他的N头牛已经逃跑并开始狂闹！每头牛在围栏外面每分钟会造成一美元的损失。约翰必须访问每头牛，安装一个磐头来平息牛的暴动并停止损失。

幸运的是，这些牛分散在农场外的一条直线上的道路上，每头牛都在不同的位置上。约翰知道每头牛的位置P_i是相对于门（位置0）的，也就是约翰开始的位置。

约翰每分钟移动一个距离单位并且可以立刻安装磐头。请确定约翰应该按什么顺序访问牛，这样他就可以将损失的总成本最小化；你应该计算出在这种情况下的最低总损失成本。

输入格式

第1行：牛的数量，$N(1 \le N \le 1000)$。

第2行到第$N+1$行：第$i+1$行包含整数$P_i(-5 \times 10^5 \le P_i \le 5 \times 10^5)$。

输出格式

第1行：损失的最低总成本。

样例输入

```
4
-2
-12
3
7
```

样例输出

```
50
```

Solution

一个比较显然的结论是，我们处理完某个**连续区间的牛**后，位置一定位于这个连续区间的两端之一

因此，我们在处理输入时，还需要对牛按照**位置**排序

我们令$\mathrm{dp}[l][r][0]$为处理完区间$[l,r]$后，约翰位于位置l的最小花费，$\mathrm{dp}[l][r][1]$为处理完区间$[l,r]$后，约翰位于位置r的最小花费，转移方程只需要枚举**上一步所处位置**和**这一步到达位置**即可

$$\mathrm{dp}[l][r][0]=\min(\mathrm{dp}[l][r][0],\mathrm{dp}[l+1][r][0]+res\times abs(a[l]-a[l+1]));$$
$$\mathrm{dp}[l][r][0]=\min(\mathrm{dp}[l][r][0],\mathrm{dp}[l+1][r][1]+res\times abs(a[l]-a[r]));$$
$$\mathrm{dp}[l][r][1]=\min(\mathrm{dp}[l][r][1],\mathrm{dp}[l][r-1][1]+res\times abs(a[r]-a[r-1]));$$
$$\mathrm{dp}[l][r][1]=\min(\mathrm{dp}[l][r][1],\mathrm{dp}[l][r-1][0]+res\times abs(a[r]-a[l]));$$

其中，约翰处理某头牛的代价是，**还没有处理的牛的数量**和**处理牛的时间**之积

边界条件比较简单，$\mathrm{dp}[i][i][0]=\mathrm{dp}[i][i][1]=P_i\times n$，代表处理第一头牛的代价

Code

```
void solve() {
    int n; cin >> n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;
    sort(all(a));

    vector \text{dp}(n, vector(n, vector(2, INF)));
    for (int i = 0; i < n; i++) dp[i][i][0] = dp[i][i][1] = abs(a[i]) * n;

    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++) {
        int r = l + len, res = n - len;
        cmin(dp[l][r][0], dp[l + 1][r][0] + res * abs(a[l] - a[l + 1]));
        cmin(dp[l][r][0], dp[l + 1][r][1] + res * abs(a[l] - a[r]));
        cmin(dp[l][r][1], dp[l][r - 1][1] + res * abs(a[r] - a[r - 1]));
        cmin(dp[l][r][1], dp[l][r - 1][0] + res * abs(a[r] - a[l]));
    }

    cout << min(dp[0][n - 1][0], dp[0][n - 1][1]) << endl;
}
```

49904. 区间DP3

题目描述

因为对标准的二维艺术品感到厌倦（也对其他人抄袭她的作品感到沮丧），伟大的奶牛艺术家毕可素（Picowso）决定转向更简约的一维风格。她最新的画作可以用一个长度为N的一维颜色数组来描述，每种颜色由范围$1, 2, \cdots, N$在内的一个整数指定。

让毕可素大为痛心的是，她的竞争对手莫内（Moonet）似乎已经想出了如何复制这些一维画作！莫内会用单一颜色绘制一个区间，等它干了之后，再绘制另一个区间，如此这般。莫内可以根据需要多次使用这N种颜色中的任何一种（可能一次也不用）。

请计算莫内复制毕可素最新的一维画作所需的最小笔触次数。

输入格式

输入的第一行包含$N(1 \leq N \leq 300)$。

下一行包含N个整数，范围在$1, 2, \cdots, N$内，指示毕可素最新的一维画作中每个单元的颜色。

输出格式

输出复制画作所需的最小笔触次数。

输入样例

```
10
1 2 3 4 1 4 3 2 1 6
```

输出样例

```
6
```

Solution

显然，当一段区间**左右端点**颜色相同时，我们可以在涂**左端点**时**顺便**涂右端点，反之亦然，这样不会对中间造成任何负面的影响

我们令$\mathrm{dp}[l][r]$为区间$[l, r]$颜色涂抹正确所需的最小笔触次数

于是我们得到了第一种转移方程：

$$\mathrm{dp}[l][r] = \min(\mathrm{dp}[l][r], \min(\mathrm{dp}[l+1][r], \mathrm{dp}[l][r-1]) \text{ where } a_l = a_r$$

一般情况下的转移方程，我们就可以将区间拆分成两个小区间合并讨论了

$$\mathrm{dp}[l][r] = \min(\mathrm{dp}[l][r], \mathrm{dp}[l][k], \mathrm{dp}[k+1][r])$$

Code

```cpp
void solve() {
    int n; cin >> n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;

    vector dp(n, vector(n, inf));
    for (int i = 0; i < n; i++) dp[i][i] = 1;
    for (int i = 0; i < n; i++) for (int j = 0; j < i; j++) dp[i][j] = 0;

    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++) {
        int r = l + len;
        if (a[l] == a[r]) cmin(dp[l][r], min(dp[l][r - 1], dp[l + 1][r]));
```

```
        for (int k = l + 1; k <= r; k++) cmin(dp[l][r], dp[l][k - 1] + dp[k][r]);
    }

    cout << dp[0][n - 1] << endl;

}
```

49905 区间DP4

题目描述

给出一个长度为n的正整数序列$a_1, a_2, \cdots, a_n$

一次可以从序列中选择一段连续的个相同的元素，把它们删除，并获得的奖励。

求把整除此序列删除的最大奖励值。

输入格式

第1行：一个整数$n(1 \le n \le 300)$

第2行：n个正整数，表示序列$a_1, a_2, \cdots, a_n (1 \le a_i \le n)$

输出格式

第1行：一个整数，表示最大奖励

输入样例

```
6
3 2 2 1 2 2
```

输出样例

```
18
```

Solution

我们令$\mathrm{dp}[l][r][k]$为将区间$[l, r]$消去，同时l前面有k个和l相等的数的最大奖励

初始状态很简单，$\mathrm{dp}[i][i][k] = (k+1)^2$

第一种转移，就是将l及其前面的数全部消去：

$$\mathrm{dp}[l][r][k] = \max(\mathrm{dp}[l][r][k], \mathrm{dp}[l+1][r][0] + (k+1)^2)$$

而第二种转移，则是将l接到区间内某个和l**相同**的数的前面：

$$\mathrm{dp}[l][r][k] = \max(\mathrm{dp}[l][r][k], \mathrm{dp}[l+1][mid][0] + \mathrm{dp}[mid+1][r][k+1]) \text{ where } a_l = a_{mid+1}$$

Code

```
void solve() {
    int n; cin >> n;
    vector a(n, 0);
    for (auto& v: a) cin >> v;

    vector dp(n, vector(n, vector(n + 1, 0)));
    for (int i = 0; i < n; i++) for (int j = 0; j < n; j++)
        dp[i][i][j] = (j + 1) * (j + 1);
```

```
    for (int len = 1; len < n; len++) for (int l = 0; l + len < n; l++) {
        int r = l + len;
        for (int k = 0; k < n; k++) {
            cmax(dp[l][r][k], dp[l + 1][r][0] + (k + 1) * (k + 1));
            for (int mid = l + 1; mid <= r; mid++) if (a[l] == a[mid]) {
                cmax(dp[l][r][k], dp[l + 1][mid - 1][0] + dp[mid][r][k + 1]);
            }
        }
    }

    cout << dp[0][n - 1][0] << endl;
}
```