动态规划基础

硬币问题

你有无限多的硬币，面值为$1,5,10,20,50,100$，现在有一个待付款金额w，请问你**最少**要用多少枚硬币才能凑出金额

显然，对于这个问题，我们硬币的面值和我们人民币的面值是**一样的**，根据我们的生活经验，我们需要尽量用面值大的，面值大的无法满足了再用面值较小的，即一个贪心做法

例如：$w=666=6\times100+1\times50+1\times10+1\times5+1\times1$

然而，这样做一定是最优的吗?

考虑一组新的硬币面值$1,5,11$和一个新的$w=15$，按照我们刚刚的策略，我们会选用$1\times11+4\times1$这样的组合，一共使用**五枚**硬币完成我们的支付，然而，一个很显然的策略3×5，就可以只使用**三枚**硬币就完成我们的支付

这就是我们贪心策略的不足之处了，我们在选择面值为11的硬币的时候，并不知道我们在$w=4$的情况下需要四枚硬币才能完成我们的支付，只是**鼠目寸光**地尽可能降低我们支付的金额

那怎么样能避免我们贪心中的**鼠目寸光**呢?

强行枚举我们使用的硬币？复杂度是指数级的，在w较大的时候难以接受

我们观察一下性质，在$w=15$时，如果我们取了11，则我们需要面对$w=4$的情况，如果我们取了5，则我们需要面对$w=10$的情况，如果我们取了1，则我们需要面对$w=14$的情况

记凑出w需要的最少的硬币数量为$f(w)$，我们把上述三种情况的方程写下来

$$
\begin{aligned}
f(w)=&f(w-1)+1\\
f(w)=&f(w-5)+1\\
f(w)=&f(w-11)+1
\end{aligned}
$$

其中，后面的$+1$代表我们增加了一枚硬币

很显然可以看出，$f(w)$只与$f(w-1),f(w-5),f(w-11)$有关，在这三者中，由于我们要**最小化**硬币的数量，因此，我们得到$f(w)=min\{f(w-1),f(w-5),f(w-11)\}+1$

推广到硬币面值为$a_1,a_2,\cdots,a_n$的场景，$f(w)=min\{f(w-a_i)\}+1$

这个做法和贪心的区别在于，贪心会选择当前能够接受的，面值最大的硬币，而我们现在的做法，则会对**选用所有硬币**的方案都进行比较，在里面找出最优解

可以看出，求解$f(w-a_i)$和求解$f(w)$的过程是一模一样的，不过$f(w-a_i)$**问题的规模**更小，这个时候，我们称$f(w-a_i)$为$f(w)$的子问题

上述问题的求解方法，就是我们常说的**动态规划**

动态规划（DP, Dynamic Programming)

动态规划，就是将一个问题拆成几个子问题，分别求解这些子问题，即可推断出大问题的解。

那么，在什么情况下，我们可以使用动态规划进行问题的求解呢?

无后效性

已经求解的子问题，不会再受到后续决策的影响

例如上题中，一旦$f(n)$确定，**我们如何凑出**$f(n)$就再也用不着了。

要求出$f(15)$，只需要知道$f(14), f(10), f(4)$的值，而$f(14), f(10), f(4)$是如何算出来的，对之后的问题没有影响。**未来与过去无关**，这就是无后效性。

其严格定义是：如果给定某一阶段的状态，则在这一阶段以后过程的发展不受这阶段以前各段状态的影响

最优子结构

大问题的最优解可以由小问题的最优解推出

例如上题中，我们对$f(n)$的定义是，凑出n需要的最少的硬币数量，在定义中，我们已经蕴含了**最优**

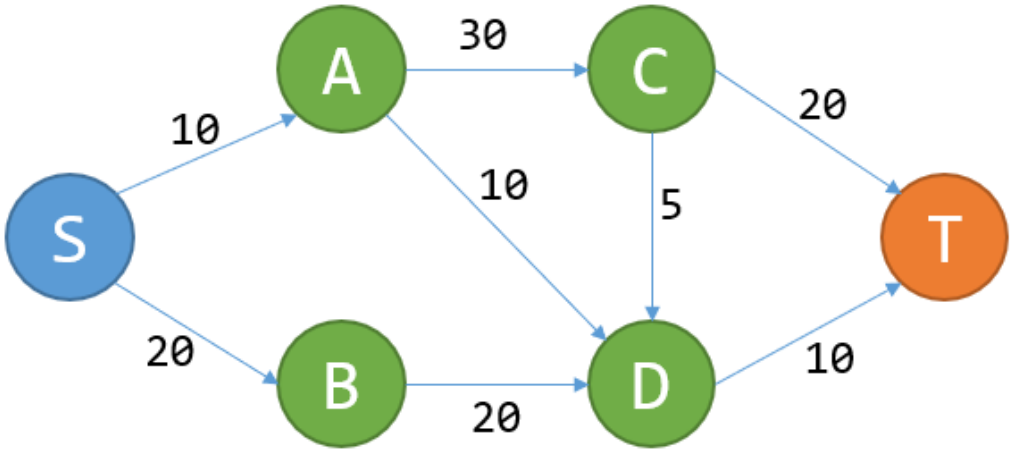
（可选）子问题重叠

如果有大量的重叠子问题，我们可以用空间将这些子问题的解存储下来，避免重复求解相同的子问题，从而提升效率。

例如在上题中，我们可以把$f(10)$的值存下来，以供求解$f(11), (15), f(21)$的时候使用，这样就不用**多次求解**$f(10)$

DAG最短路

给定一个城市的地图，所有的道路都是单行道，而且不会构成环。每条道路都有过路费，问您从S点到T点花费的最少费用。



思考：能不能使用动态规划求解？

记S到某个点P的最少费用为$f(P)$

想要到达点T，要么经过点C，要么经过点D

$$f(T) = \min(f(C) + 20, f(D) + 10)$$

回过头看是否满足我们dp所要求的两个性质：

1. 无后效性：对于某个点P，一旦$f(P)$确定，以后就不关心怎么去的，且$f(P)$的确定不会对点P之前的点的$f(X)$有影响（如果有影响，则图中存在**环**，与题干中有向无环图的定义相悖）
2. 最优子结构：对于点P，我们当然只关心到P的最小费用，即$f(P)$，并以此来解决之后的问题

这两个性质都满足，我们可以使用dp进行求解，状态我们已经设计出来了，而转移方程如下：

$$f(P)=\min\{f(R)+w_{R\to P}\}$$

其中，R为有路通到P的所有的点，$w_{R\to P}$为R到P的过路费

DP为什么会很大程度上地优于搜索?

DP枚举的是所有**有希望成为答案的解**，而搜索枚举的是**所有可能**的解，例如在上文的硬币问题中，$f(15)=2\times5+5\times1$就从来没有被考虑过，因为这不能从任意一个子问题$f(4),f(10),f(14)$转移过来

DP的核心

尽可能缩小**可能解**的空间

DP的操作过程

大事化小，小事化了

将一个大问题转化成几个小问题；求解小问题；推出大问题的解

一般来讲，我们在使用DP求解问题时，分为以下三个步骤：

1. 将原问题划分为若干**阶段**，每个阶段对应若干个子问题，提取这些子问题的特征（称之为**状态**）
2. 寻找每一个状态的可能**决策**，或者说是各状态间的相互转移方式（用数学的语言描述就是**状态转移方程**）
3. 按顺序求解每一个阶段的问题

以硬币问题为例：

状态就是我们设计的，凑出w需要的最少的硬币数量$(f(w))$

转移方程，或者说**决策**，就是我们在推广到硬币面值为$a_1,a_2,\cdots,a_n$的场景时给出的方程
$f(w)=min\{f(w-a_i)\}+1$

而按顺序求解问题时，我们一般希望，求解到某个$f(w)$时，它**所有的子问题**已经求解完毕，对于本题而言，我们**从小到大枚举**w即可保证解决该问题

[# 384] 斐波那契数列

题目描述

斐波那契数列$0,1,1,2,3,5,8,13,21,34\cdots$，从第三项起，每一项都是紧挨着的前两项的和。写出计算斐波那契数列的任意一个数据项递归程序。

输入格式

所求的项数。

输出格式

数据项的值，保证数据项在int范围之内

样例输入

```
10
```

样例输出

```
34
```

Solution

状态很明显，我们可以定义$f[i]$为斐波那契数列的第i项

而转移方程也很明显，$f[i]=f[i-1]+f[i-2](f[1]=0,f[2]=1)$

由于我们求解$f[i]$时，子问题$f[j]$都有$i>j$，因此顺序枚举即可

而枚举的项数，我们一方面可以使用**打表**，估算什么时候能够超过int，并且留一定的**冗余量**；另一方面，也可以直接利用long long存储，并且和宏INT_MAX进行比较即可

Code

```
vector<ll> f = {0, 1};

void init(){
    while (f.back() < INT_MAX)
        f.push_back(f[f.size() - 1] + f[f.size() - 2]);
}

void solve() {
    int n; cin >> n;
    cout << f[n - 1] << '\n';
}
```

[# 8134] 最长上升子序列（输出序列）

题目描述

一个给定序列的子序列是在该序列中删去若干元素后得到的序列。确切地说，若给定序列$X=x_1,x_2,\cdots x_n$，则另一序列$Z=z_1,z_2\cdots z_k$是X的子序列是指存在一个严格递增的下标序列$i_1,i_2\cdots i_k$使得对于所有$j=1,2,\cdots k$有：

$$X_{i_j}=Z_j$$

例如，序列$Z=B,C,D,B$是序列$X=A,B,C,B,D,A,b$的子序列,相应的递增下标序列为$2,3,5,7$。给定两个序列X和Y，当另一序列Z既是X的子序列又是Y的子序列时，称Z是序列和的公共子序列。例如，若$X=A,B,C,B,D,A,B$和$Y=B,D,C,A,B,A$，则序列$Z_1=B,C,A$是和的一个公共子序列，序列$Z_2=B,C,B,A$也是和的一个公共子序列。而且，后者是A和B的一个最长公共子序列．因为和没有长度大于4的公共子序列。

给定一个整数序列$a_1,a_2\cdots,a_n$。求它的一个递增**子序列**，使子序列的元素个数尽量多，元素不一定要求连续。

输入格式

第一行：一个整数$n(1 \leq n \leq 5000)$，表示序列中元素的个数.

第二到$n+1$行：每行一个整数$x(-1000 \leq x \leq 1000)$，第$i+1$行表示序列中的第$i$个元素。

输出格式

第一行：一个整数k，表示最长上升子序列的长度。

第二行：k个用单个空格分开的整数，表示找到了最长上升子序列。输出的是序列中的元素，并非下标。如果有多个长度等于k的子序列，则输出最靠前的那个。

样例输入

```
8
1 3 2 4 3 5 4 6
```

样例输出

```
5
1 3 4 5 6
```

Solution

我们使用$f[i]$记录**最后一个元素是$a[i]$的最长上升子序列的长度**

很明显，我们可以在$j \in [1, i-1]$里面枚举所有可能的转移值，如果$a[j] < a[i]$，就代表我们可以将$a[i]$**接到**$a[j]$后面，不断使用$f[j]+1$更新$f[i]$即可，即：

$$f[i] = \max(f[j]+1)(a[j] < a[i])$$

同时，在转移的时候，我们可以利用一个$b[i]$数组，记录$f[i]$是由哪个$f[j]$转移而来，在最后递归 / 迭代输出序列即可

bonus：思考$O(n\log n)$的做法（不用输出序列）

我们用一个vetor用来维护一个序列，该序列**长度**等于我们的最长上升子序列的长度

我们遍历原序列中的所有元素，对于元素$a[i]$，我们分两种情况考虑

1. $a[i]$大于当前的末尾元素，我们直接将其push_back至vector末尾即可
2. $a[i]$小于当前的末尾元素，我们可以找到**最小的大于等于**$a[i]$的元素，将其替换为$a[i]$，这样的话，实际意义是将最长上升子序列的某一位替换成$a[i]$，在$a[i]$后面所有元素被替换之前，我们用**被替换前**的元素仍然能够得到长度和该序列相等的的上升子序列，在所有元素被替换之后，其本身就可以成为长度和该序列相等的的上升子序列

显然，vector里面的元素是**单调**的，因此2中的替换可以使用二分，整体的时间复杂度降到了$O(n\log n)$

Code

```
// O(n^2)
void solve() {
    int n; cin >> n;

    vector<int> a(n), dp(n, 1), f(n, -1), ans;
    for (auto& v: a) cin >> v;

    for (int i = 0; i < n; i++) for (int j = 0; j < i; j++)
        if (a[i] > a[j] && dp[j] + 1 > dp[i]) {
            f[i] = j; dp[i] = dp[j] + 1;
        }
    auto maxx = *max_element(all(dp));
    for (int i = 0; i < n; i++) if (dp[i] == maxx) {
        while (~i) {ans.push_back(a[i]); i = f[i];}
        break;
    }

    reverse(all(ans));
    cout << ans.size() << '\n';
    for (auto v: ans) cout << v << ' ';
}
```

```
// O(nlogn)
void solve() {
    int n; cin >> n;
    vector<int> a(n), b;
    for (auto& v: a) cin >> v;
    for (auto v: a) {
        if (!b.size() || v > b.back()) b.push_back(v);
        else *lower_bound(all(b), v) = v;
    }
    cout << b.size() << endl;
}
```

[# 4599] 最长公共子序列(输出序列)

题目描述

给定两个序列$X = x_1, x_2 \cdots x_n$和$Y = y_1, y_2, \cdots, y_m$要求找出和的一个最长公共子序列。

输入格式

共有两行。每行为一个**由大写字母构成**的长度不超过1000的字符串，表示序列X和Y。

输出格式

第一行为一个非负整数。表示所求得的最长公共子序列大小。

第二行为最长公共子序列本身，如果存在多个长度等于L的子序列，输出在X序列中位置最靠前的一个。若不存在公共子序列，则不输出。

样例输入

```
ABCBDAB
BDCABA
```

样例输出

```
4
BCBA
```

Solution

如果说，以$f[i][j]$表示前串X的前i个元素和串Y的前j个元素，能够构成的，最长公共子序列的长度，我们很容易考虑到转移方程

$$dp[i][j] = max(dp[i][j], dp[i-1][j])$$
$$dp[i][j] = max(dp[i][j], dp[i][j-1])$$
$$dp[i][j] = max(dp[i][j], dp[i-1][j-1])(a[i] == b[j])$$

其中，第一个方程的现实意义是**不使用**$X[i]$，第二个方程的现实意义是**不使用**$Y[j]$，第三个方程的现实意义是，**将**$a[i]$**作为公共子序列的一部分**

我们可以用某个状态数组$s[i][j]$标记我们$dp[i][j]$是由哪种转移最终确定的，从$s[n][m]$一直不断回溯，就能找到我们的最长公共子序列

Code

```
void solve(){
    string a, b; cin >> a >> b;
    int n = a.length(), m = b.length();
    vector dp(n, vector<int>(m));
    vector f(n, vector<int>(m));
    for (int i = 0; i < n; i++) for (int j = 0; j < m; j++) {
        if (a[i] == b[j]) {
            cmax(dp[i][j], 1);
            if (i && j && dp[i - 1][j - 1] + 1 > dp[i][j])
                    f[i][j] = 3, dp[i][j] = dp[i - 1][j - 1] + 1;
        }
        if (i && dp[i - 1][j] > dp[i][j]) f[i][j] = 1, dp[i][j] = dp[i - 1][j];
        if (j && dp[i][j - 1] > dp[i][j]) f[i][j] = 2, dp[i][j] = dp[i][j - 1];
    }

    int maxx = dp[n - 1][m - 1];
    string ans;
    int i = n - 1, j = m - 1;
    while (~i && ~j) {
        if (!f[i][j]) break;
        if (f[i][j] == 1) i--;
        else if (f[i][j] == 2) j--;
        else {ans += a[i]; i--; j--;}
    }
    ans += a[i];
    reverse(all(ans));
    cout << maxx << endl << ans << endl;
```

```
}
```

[洛谷P1439] 【模板】最长公共子序列

题目描述

给出 $1, 2, \ldots, n$ 的两个排列 P_1 和 P_2 ，求它们的最长公共子序列。

输入格式

第一行是一个数 $n(1 \leq n \leq 10^5)$。

接下来两行，每行为 n 个数，为自然数 $1, 2, \ldots, n$ 的一个排列。

输出格式

一个数，即最长公共子序列的长度。

样例输入

```
5
3 2 1 4 5
1 2 3 4 5
```

样例输出

```
3
```

Solution

我们刚才所讲的，朴素的最长公共子序列，其时空复杂度均为$O(n^2)$，很明显不足以通过此题，~~因此这题的题目是假的~~。

仔细阅读题面可以得到，序列P_1, P_2中的元素都是**两两不同**，且$P_1(P_2)$中的元素在$P_2(P_1)$中会且仅会出现**一次**。另外，在我们的最长公共子序列中，元素在**原序列**中的下标是单调增加的。

因此，如果我们将P_2中的元素，替换成P_1中的元素的**下标**，这时候我们得到了一个新的A序列，且数组中元素两两不同。显然，A序列中的最长上升子序列，就代表着某个P_1, P_2的下标都逐渐增加的最长公共子序列

Code

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

    for (auto& v: a) {cin >> v; v = mp[v];}

    for (auto v: a) {
        if (!b.size() || v > b.back()) b.push_back(v);
```

```
        else *lower_bound(all(b), v) = v;
    }
    cout << b.size() << endl;
}
```

[# 6107] 最长公共上升子序列(加强)

题目描述

熊大妈的奶牛在小沐沐的熏陶下开始研究信息题目。

小沐沐先让奶牛研究了最长上升子序列，再让他们研究了最长公共子序列，现在又让他们研究最长公共上升子序列了。

小沐沐说，对于两个数列A和B，如果它们都包含一段位置不一定连续的数，且数值是严格递增的，那么称这一段数是两个数列的公共上升子序列，而所有的公共上升子序列中最长的就是最长公共上升子序列了。

奶牛半懂不懂，小沐沐要你来告诉奶牛什么是最长公共上升子序列。

不过，只要告诉奶牛它的长度就可以了。

输入格式

第一行包含一个整数$N(1 \leq N \leq 3000)$，表示数列A, B的长度。

第二行包含N个整数，表示数列A。

第三行包含N个整数，表示数列B。

其中，数列A, B的元素范围为$-2^{31} \leq a_i, b_i \leq 2^{31} - 1$

输出格式

输出一个整数，表示最长公共子序列的长度。

样例输入

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

样例输出

```
2
```

Solution

类似最长公共子序列问题，我们假设$dp[i][j]$为使用数列A的前i个元素，数列B的前j个元素，并且结尾为$b[j]$的最长上升公共子序列的长度

很容易得到我们的转移方程

$$dp[i][j] = \max(dp[i][j], dp[i-1][j])$$
$$dp[i][j] = \max(dp[i][j], dp[i-1][k] + 1)(a[i] == b[j] \text{ and } b[k] < b[j])$$

方程一的现实意义是不使用$a[i]$，方程二的现实意义是，在$a[i] == b[j]$的情况下，在前面去找一个最长的，并且能够把$b[j]$**接到末尾**的公共上升子序列

此时我们很容易就可以写出一个时间复杂度为$O(n^3)$的算法，但很明显不能通过此题，此时时间复杂度的瓶颈在于方程二

注意到方程二中，$a[i] == b[j]$才会有如此转移，我们可以等价把$b[k] < b[j]$替换成$b[k] < a[i]$，并且在枚举j的过程中不断地去更新一个$maxx = \max(dp[i-1][j])(b[j] < a[i])$，这样就可以将方程二改为

$$dp[i][j] \quad = \max(dp[i][j], maxx + 1)(a[i] == b[j])$$

此时时间复杂度将为了$O(n^2)$，足以通过此题

Code

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

    vector dp(n + 1, vector<int>(n + 1));

    for (int i = 1; i <= n; i++) {
        int maxx = 0;
        for (int j = 1; j <= n; j++) {
            dp[i][j] = dp[i - 1][j];
            if (a[i] == b[j]) cmax(dp[i][j], maxx + 1);
            if (b[j] < a[i]) cmax(maxx, dp[i - 1][j]);
        }
    }

    cout << *max_element(all(dp[n])) << endl;
}
```

[# 6106] 杨老师的照相排列

题目描述

有N个学生合影，站成左端对齐的k排，每排分别有$N_1, N_2, \cdots N_k$个人（$N_1 \geq N_2 \geq N_3 \cdots \geq N_k$）

第1排站在最后边，第N排站在最前边。

学生的身高**互不相同**，把他们从高到低依次为$1, 2, 3, \cdots N$。

在合影时要求每一排从左到右身高递减，每一列从后到前身高也递减。

问一共有多少种安排合影位置的方案?

下面的一排三角矩阵给出了当$N = 6, k = 3, N_1 = 3, N_2 = 2, N_3 = 1$时的全部16种合影方案。注意身高最高的是1，最低的是6。

```
123 123 124 124 125 125 126 126 134 134 135 135 136 136 145 146
45  46  35  36  34  36  34  35  25  26  24  26  24  25  26  25
6   5   6   5   6   4   5   4   6   5   6   4   5   4   3   3
```

输入格式

输入包含多组测试数据。

每组数据两行，第一行包含一个整数表示总排数。

第二行包含个整数，表示从后向前每排的具体人数。

当输入的数据时，表示输入终止，且该数据无需处理。

输出格式

每组测试数据输出一个答案，表示不同安排的数量。

每个答案占一行。

样例输入

```
1
30
5
1 1 1 1 1
3
3 2 1
4
5 3 3 1
5
6 5 4 3 2
2
15 15
0
```

样例输出

```
1
1
16
4158
141892608
9694845
```

Solution

假设我们从1到N依次安排，显然，我们在安排第i个人的站位时，他的整个**左上方**的区域的一定已经没有空位了，因为如果有空位，那么该空位安排的人一定比i矮，就不符合题设了

我们定义$\mathrm{dp}[i_1][i_2][i_3][i_4][i_5]$为第$x$排安放了$i_x$个人，一共安排了前$\sum_{x=1}^{5} i_x$个人的安排方式数

考虑最后一个人安排在哪里，很明显，只能安排在第x排的i_x位置，那么，如果安排在第一排的i_1位置，那么就需要第二排人数$i_2 < i_1$，否则$i_2 \geq i_1$时，对于站在第二排i_2位置的人而言，在排到他的站位时，他的左上方区域就相当于存在**空位**了

如果说排数**不足五排**时，其它排相当于限制只有**0个人**

因此，我们就得到了转移方程

$$
\begin{aligned}
&\mathrm{dp}[i0][i1][i2][i3][i4] += \mathrm{dp}[i0-1][i1][i2][i3][i4] \text{ where } i_0 > i_1\\
&\mathrm{dp}[i0][i1][i2][i3][i4] += \mathrm{dp}[i0][i1-1][i2][i3][i4] \text{ where } i_1 > i_2\\
&\mathrm{dp}[i0][i1][i2][i3][i4] += \mathrm{dp}[i0][i1][i2-1][i3][i4] \text{ where } i_2 > i_3\\
&\mathrm{dp}[i0][i1][i2][i3][i4] += \mathrm{dp}[i0][i1][i2][i3-1][i4] \text{ where } i_3 > i_4\\
&\mathrm{dp}[i0][i1][i2][i3][i4] += \mathrm{dp}[i0][i1][i2][i3][i4-1] \text{ where } i_4 > 0
\end{aligned}
$$

第j个方程的现实意义是，将第$\sum_{x=1}^{5} i_x$个人安排在第j排

Code

```
ll dp[maxm][maxm][maxm][maxm][maxm];

void solve() {
    int k; cin >> k;
    if (!k) exit(0);

    vector<int> a(k);
    for (auto& v: a) cin >> v;
    while (a.size() < 5) a.push_back(0);

    ll ans = 0;
    dp[0][0][0][0][0] = 1;

    for (int i0 = 0; i0 <= a[0]; i0++) for (int i1 = 0; i1 <= min(i0, a[1]);
i1++)
    for (int i2 = 0; i2 <= min(i1, a[2]); i2++) for (int i3 = 0; i3 <= min(i2,
a[3]); i3++)
    for (int i4 = 0; i4 <= min(i3, a[4]); i4++) {
        auto& v = dp[i0][i1][i2][i3][i4];
        if (i0 && i0 - 1 >= i1) v += dp[i0 - 1][i1][i2][i3][i4];
        if (i1 && i1 - 1 >= i2) v += dp[i0][i1 - 1][i2][i3][i4];
        if (i2 && i2 - 1 >= i3) v += dp[i0][i1][i2 - 1][i3][i4];
        if (i3 && i3 - 1 >= i4) v += dp[i0][i1][i2][i3 - 1][i4];
        if (i4) v += dp[i0][i1][i2][i3][i4 - 1];
        cmax(ans, v);
    }
    cout << ans << endl;

    for (int i0 = 0; i0 <= a[0]; i0++) for (int i1 = 0; i1 <= min(i0, a[1]);
i1++)
    for (int i2 = 0; i2 <= min(i1, a[2]); i2++) for (int i3 = 0; i3 <= min(i2,
a[3]); i3++)
    for (int i4 = 0; i4 <= min(i3, a[4]); i4++) dp[i0][i1][i2][i3][i4] = 0;
}
```

[# 6108] 分级

题目描述

给定长度为N的序列A，构造一个长度为N的序列B，满足：

1. B非严格单调，即$B_1 \leq B_2 \leq \cdots \leq B_n$或$B_1 \geq B_2 \geq \cdots \geq B_n$
2. 最小化$S = \sum_{i=1}^{n} |A_i - B_i|$

只需要求出这个最小值S。

输入格式

第一行包含一个整数$N(1 \leq N \leq 2000)$。

接下来N行，每行包含一个整数$A_i(0 \leq A_i \leq 10^9)$。

输出格式

输出一个整数，表示最小的S值。

样例输入

```
7
1
3
2
4
5
3
9
```

样例输出

```
3
```

Solution

很明显，存在某个最优解，使得B中的所有元素，都在A中出现过，下面我们来证明这一点：

假设存在一段$B_i, B_{i+1}, \cdots B_j$，使得$B_i = B_{i+1} = \cdots = B_j$，设有
$A_x \leq B_i$ or $A_y \geq B_i (i \leq x, y \leq j)$，其中，$A_x$为小于$B_i$的最大值，$A_y$为大于$B_i$的最小值，此时将$B_i$改为$A_x$或$A_y$，我们的$S$不会改变，故得证

那么，我们定义$dp[i][j]$为考虑序列前j个数，序列的结尾为$C[i]$的S的最小值，其中，C是序列A排序后的结果

转移方程如下

$$
\begin{aligned}
dp[i][0] =& abs(a[0] - c[0]) \\
dp[i][j] =& min(dp[i][j-1], minn[j-1]) + abs(a[j] - c[i])
\end{aligned}
$$

其中，$minn[j]$表示序列前j个数，序列的结尾为$C[j]$的S的最小值

之所以要用$minn$数组单独存储最小值，因为我们需要保证我们的序列中不出现
$C[i], C[i], C[j], \cdots C[i], C[i] (j < i)$的情况

因此，我们需要在一个$C[i]$遍历完所有的位置之后，再将$minn$数组的值赋给$dp[i]$数组

递增序列和递减序列的做法类似，只是体现在序列C是升序排序或降序排序

Code

```
void solve() {
    int n; cin >> n;
    vector<ll> a(n), b, minn(n, INF);
    for (auto& v: a) cin >> v;
    b = a; sort(all(b));

    ll ans = INF;

    // dp[i][j] means a_1, \cdots , a_j, with b[i] tail
    vector dp(n, vector(n, INF));

    for (int i = 0; i < n; i++) {
        dp[i][0] = abs(a[0] - b[i]);
        for (int j = 1; j < n; j++) {
            dp[i][j] = min(dp[i][j - 1], minn[j - 1]) + abs(a[j] - b[i]);
        }
        cmin(ans, dp[i][n - 1]);
        for (int j = 0; j < n; j++) cmin(minn[j], dp[i][j]);
    }

    fill(all(minn), INF);
    for (auto& vec: dp) fill(all(vec), INF);

    reverse(all(b));
    for (int i = 0; i < n; i++) {
        dp[i][0] = (a[0] - b[i]) * (a[0] - b[i]);
        for (int j = 1; j < n; j++) {
            dp[i][j] = min(dp[i][j - 1], minn[j - 1]) + (a[j] - b[i]) * (a[j] -
b[i]);
        }
        cmin(ans, dp[i][n - 1]);
        for (int j = 0; j < n; j++) cmin(minn[j], dp[i][j]);
    }

    cout << ans << endl;
}
```

[# 6109] 移动服务

题目描述

一个公司有三个移动服务员，最初分别在位置$1, 2, 3$处。

如果某个位置（用一个整数表示）有一个请求，那么公司必须指派某名员工赶到那个地方去。

某一时刻只有一个员工能移动，且不允许在同样的位置出现两个员工。

从i移动一个员工到j，需要花费$c(i, j)$。

这个函数不一定对称，但保证$c(i,i)=0$。

给出N个请求，请求发生的时间依次为$p_1,p_2,\cdots p_N$。

公司必须按顺序依次满足所有请求，目标是最小化公司花费，请你帮忙计算这个最小花费。

注意：员工移动一次所需时间是1，即员工只能直接通过$c(i,j)$的花费从i移动到j

输入格式

第1行有两个整数$L,N(1\leq L\leq 200,1\leq N\leq 1000)$，其中$L$是位置数量，$N$是请求数量，每个位置从1到$L$编号。

第2至$L+1$行每行包含L个非负整数，第$i+1$行的第j个数表示$c(i,j)(1\leq c(i,j)\leq 2000)$

最后一行包含N个整数，是请求列表。

输出格式

输出一个整数M，表示最小花费。

样例输入

```
5 9
0 1 1 1 1
1 0 2 3 2
1 1 0 4 1
2 1 5 0 1
4 2 3 4 0
4 2 4 1 5 4 3 2 1
```

样例输出

```
5
```

Solution

显然，在i时刻，我们肯定**有且仅有**一名员工处于p_i，那么，我们可以使用$dp[i][j][k]$代表时刻i，三名员工分别处于j,k,p_i的最小花费

那么，从时刻i到时刻$i+1$，由于我们只能有一名员工移动一次，因此，我们只有**至多**三种派遣方式，即分别将j,k,p_i派遣到p_{i+1}，转移方程可以很简单地表示如下：

$$\begin{aligned}dp[i+1][j][k]=dp[i+1][k][j]=dp[i][j][k]+c[p[i]][p[i+1]]\\dp[i+1][j][p[i]]=dp[i+1][p[i]][j]=dp[i][j][k]+c[k][p[i+1]]\\dp[i+1][k][p[i]]=dp[i+1][p[i]][k]=dp[i][j][k]+c[j][p[i+1]]\end{aligned}$$

其中，方程一代表将位置处于p_i的员工派到p_{i+1}，方程二代表将位置处于k的员工派到p_{i+1}，方程三代表将位置处于j的员工派到p_{i+1}

时间复杂度$O(NL^2)$，空间复杂度$O(NL^2)$，注意到我们$dp[i+1]$只依赖于$dp[i]$的值，因此可以使用**滚动数组**将空间复杂度优化至$O(L^2)$

Code

```
void solve() {
    int n, m; cin >> n >> m;
    vector dis(n, vector(n, 0));
    vector dp(2, vector(n, vector(n, inf)));
    for (auto& vec: dis) for (auto& v: vec) cin >> v;

    vector<int> pos(m);
    for (auto& v: pos) {cin >> v; v--;}

    if (pos[0] != 0 && pos[0] != 1) dp[0][0][1] = dis[2][pos[0]];
    if (pos[0] != 0 && pos[0] != 2) dp[0][0][2] = dis[1][pos[0]];
    if (pos[0] != 1 && pos[0] != 2) dp[0][1][2] = dis[0][pos[0]];

    int now = 1;
    for (int i = 1; i < m; i++) {
        if (pos[i] == pos[i - 1]) continue;
        for (int j = 0; j < n; j++) for (int k = j + 1; k < n; k++) dp[now][j]
[k] = inf;
        for (int j = 0; j < n; j++) for (int k = j + 1; k < n; k++) {
            // j, k, pos[i - 1]
            if (j == pos[i - 1] || k == pos[i - 1]) continue;
            // j, k, pos[i]
            cmin(dp[now][j][k], dp[now ^ 1][j][k] + dis[pos[i - 1]][pos[i]]);
            // pos[i - 1], k, pos[i]
            cmin(dp[now][min(pos[i - 1], k)][max(pos[i - 1], k)], dp[now ^ 1][j]
[k] + dis[j][pos[i]]);
            // pos[i - 1], j, pos[i]
            cmin(dp[now][min(pos[i - 1], j)][max(pos[i - 1], j)], dp[now ^ 1][j]
[k] + dis[k][pos[i]]);
        }
        now ^= 1;
    }

    now ^= 1;
    int ans = inf;
    for (int i = 0; i < n; i++) for (int j = i + 1; j < n; j++) cmin(ans,
dp[now][i][j]);
    cout << ans << endl;
}
```

[# 6110] 传纸条

题目描述

小渊和小轩是好朋友也是同班同学，他们在一起总有谈不完的话题。一次素质拓展活动中，班上同学安排坐成一个 m 行 n 列的矩阵，而小渊和小轩被安排在矩阵对角线的两端，因此，他们就无法直接交谈了。幸运的是，他们可以通过传纸条来进行交流。纸条要经由许多同学传到对方手里，小渊坐在矩阵的左上角，坐标 $(1,1)$，小轩坐在矩阵的右下角，坐标 (m,n)。从小渊传到小轩的纸条只可以向下或者向右传递，从小轩传给小渊的纸条只可以向上或者向左传递。

在活动进行中，小渊希望给小轩传递一张纸条，同时希望小轩给他回复。班里每个同学都可以帮他们传递，但只会帮他们一次，也就是说如果此人在小渊递给小轩纸条的时候帮忙，那么在小轩递给小渊的时候就不会再帮忙。反之亦然。

还有一件事情需要注意，全班每个同学愿意帮忙的好感度有高有低（注意：小渊和小轩的好心程度没有定义，输入时用 0 表示），可以用一个 $[0,100]$ 内的自然数来表示，数越大表示越好心。小渊和小轩希望尽可能找好心程度高的同学来帮忙传纸条，即找到来回两条传递路径，使得这两条路径上同学的好心程度之和最大。现在，请你帮助小渊和小轩找到这样的两条路径。

输入格式

第一行有两个用空格隔开的整数 m 和 n，表示班里有 m 行 n 列。

接下来的 m 行是一个 $m \times n$ 的矩阵，矩阵中第 i 行 j 列的整数表示坐在第 i 行 j 列的学生的好心程度。每行的 n 个整数之间用空格隔开。

输出格式

输出文件共一行一个整数，表示来回两条路上参与传递纸条的学生的好心程度之和的最大值。

样例输入

```
3 3
0 3 9
2 8 5
5 7 0
```

样例输出

```
34
```

Solution

从小渊传给小宣，再从小宣传给小渊的过程，可以简化成**小渊传了两张纸条给小宣**

这样，我们就可以得到一个很简单的状态表示方法：$\mathrm{dp}[i][j][k][l]$，代表第一张纸条传到坐标(i,j)，第二张纸条传到坐标(k,l)时，我们所能收获的最大的好心程度

而转移过程也很简单，我们枚举所有的上一步，即：

$$(i-1,j),(i,j-1)$$
$$(k-1,l),(k,l-1)$$

两两组合形成的，**这轮移动前**的状态，然后加上$h[i][j]+h[k][l]$（$h[i][j]$表示坐标是(i,j)的同学的好心值）即可

题目中的约束**只会帮他们一次**，代表着我们从$\mathrm{dp}[i][j][k][l]$进行转移的时候，$(i,j)\neq(k,l)$即可

此时时空复杂度均为$O(n^4)$

注意到$i+j=k+l$，考虑优化空间复杂度，使用$\mathrm{dp}[step][i][j]$，代表在经过$step$次传递后，第一张纸条传到$(i,step-i+1)$，第二张纸条传到$(j,step-j+1)$时，能收获的最大的好心程度，转移方程和上文类似：

$$\begin{aligned}
\mathrm{dp}[i][j][k] &= \min(\mathrm{dp}[i][j][k], \mathrm{dp}[i-1][j][k]) \\
\mathrm{dp}[i][j][k] &= \min(\mathrm{dp}[i][j][k], \mathrm{dp}[i-1][j-1][k]) \\
\mathrm{dp}[i][j][k] &= \min(\mathrm{dp}[i][j][k], \mathrm{dp}[i-1][j][k-1]) \\
\mathrm{dp}[i][j][k] &= \min(\mathrm{dp}[i][j][k], \mathrm{dp}[i-1][j-1][k-1])
\end{aligned}$$

在转移的时候判断一下坐标$(i, step-i+1), (j, step-j+1)$的合法性（$i \neq j;\ i, j \leq m, \text{ect.}$）即可

Code

```
const vector<pair<int, int>> d = {{0, -1}, {-1, 0}};

void solve() {
    int m, n; cin >> m >> n;
    vector val(m, vector(n, 0));
    for (auto& vec: val) for (auto& v: vec) cin >> v;

    int step = n + m - 2;
    vector dp(step + 1, vector(m, vector(m, 0)));

    auto valid = [&](int x, int y) {
        return x >= 0 && x < m && y >= 0 && y < n;
    };

    for (int i = 1; i <= step; i++) for (int j = 0; j <= min(i, m - 1); j++)
    for (int k = 0; k <= min(i, m - 1); k++) {
        if (j == k && i != step) continue;
        int vj = i - j, vk = i - k;
        if (!valid(j, vj) || !valid(k, vk)) continue;
        for (auto [dxj, dyj]: d) for (auto [dxk, dyk]: d) {
            int xj = dxj + j, yj = dyj + vj;
            int xk = dxk + k, yk = dyk + vk;
            if (xj == xk && i != 1) continue;
            if (!valid(xj, yj) || !valid(xk, yk)) continue;
            cmax(dp[i][j][k], dp[i - 1][xj][xk] + val[j][vj] + val[k][vk]);
        }
    }

    cout << dp[step][m - 1][m - 1] << endl;
}
```

[# 6112] 饼干

题目描述

圣诞老人共有M个饼干，准备全部分给N个孩子。

每个孩子有一个贪婪度，第i个孩子的贪婪度为g_i。

如果有a_i个孩子拿到的饼干数比第i个孩子多，那么第i个孩子会产生$a_i \times g_i$的怨气。

给定N, M和序列g，圣诞老人请你帮他安排一种分配方式，使得每个孩子至少分到一块饼干，并且所有孩子的怨气总和最小。

输入格式

第一行包含两个整数$N, M(1 \leq N \leq 30, N \leq M \leq 5000)$。

第二行包含N个整数表示$g_1, g_2, \cdots g_N(1 \leq g_i \leq 10^7)$。

输出格式

第一行一个整数表示最小怨气总和。

第二行N个空格隔开的整数表示每个孩子分到的饼干数，若有多种方案，输出任意一种均可。

样例输入

```
3 20
1 2 3
```

样例输出

```
2
2 9 9
```

Solution

一个显然的贪心思路是，贪婪度越高的孩子，拿到饼干等数量应该越多，因此，我们将孩子按照**贪婪度**从大到小排序后，我们每个孩子拿到饼干的数量，应该是**单调不减**的

我们用$\mathrm{dp}[i][j]$表示前i个孩子，拿了j块饼干所能达到的**最小怨气总和**。由于孩子们产生怨气的来源是**相对大小**，因此，如果每个孩子拿到的饼干数量都**减少一个**，此时我们的**最小怨气总和**是不变的，这样，我们就得到了第一个转移方程

$$\mathrm{dp}[i][j] = \min(\mathrm{dp}[i][j], \mathrm{dp}[i][j-i])$$

由于在排好序的序列中，孩子拿到饼干数量是**单调不减**的，因此，比第i个孩子饼干更多的孩子，只会出现在$[1, i-1]$之间。

我们枚举在$[1, i]$之间，有多少个孩子拿的饼干数量和第i个孩子**相同**（设为k个，即$[i-k+1, i]$区间内孩子拿的饼干数量是），此时，这部分孩子的怨气值可以按如下方程式表示：

$$\sum_{j=i-k+1}^{i} g_j \times (i-k)$$

对于数组中某一连续区间元素之和，我们可以用**前缀和**的思想$O(1)$求出，将区间$[1, x]$之和记为sum_x，我们得到新的转移方程如下：

$$\mathrm{dp}[i][j] = \min(\mathrm{dp}[i][j], \mathrm{dp}[i-k][j-k] + (sum_i - sum_{i-k}) \times (i-k))\ (1 \leq k \leq \min(i, j))$$

时间复杂度为$O(n^2 m)$，足以通过此题

Code

```
void solve(){
    int n, m; cin >> n >> m;
    vector<ll> a(n + 1), sum(n + 1), rsv;
    for (int i = 1; i <= n; i++) cin >> a[i];
```

```
    rsv = a;
    sort(a.begin() + 1, a.end(), [](int x, int y){
        return x > y;
    });
    sum = a;
    for (int i = 1; i <= n; i++) sum[i] += sum[i - 1];

    vector dp(n + 1, vector(m + 1, INF));
    vector bw(n + 1, vector(m + 1, make_pair(0, 0)));

    dp[0][0] = 0;
    for (int i = 1; i <= n; i++) for (int j = 1; j <= m; j++) {
        if (j >= i) dp[i][j] = dp[i][j - i], bw[i][j] = {i, j - i};
        for (int k = 1; k <= min(i, j); k++) {
            auto res = dp[i - k][j - k] + (sum[i] - sum[i - k]) * (i - k);
            if (res < dp[i][j]) {
                dp[i][j] = res, bw[i][j] = {i - k, j - k};
            }
        }
    }
    cout << dp[n][m] << endl;

    vector cnt(n + 1, 0);
    while (n && m) {
        auto [nn, nm] = bw[n][m];
        if (nn == n) {cnt[0]++; cnt[n]--;}
        else {cnt[nn]++; cnt[n]--;}
        n = nn; m = nm;
    }

    map<int, vector<int>> mp;
    for (int i = 0; i < cnt.size(); i++) {
        cnt[i] += i ? cnt[i - 1] : 0;
        mp[a[i + 1]].push_back(cnt[i]);
    }
    for (int i = 1; i < rsv.size(); i++) {
        auto v = rsv[i];
        cout << mp[v].back() << ' ';
        mp[v].pop_back();
    }
}
```