# Problem A. 成分鉴定

可以使用 C++11 String Literal Delimiter 语法输出多行字符。

```
#include <bits/stdc++.h>
signed main()
{
    printf(R"(Genshin Genshin Genshin Genshin Genshin Genshin Genshin
Genshin Genshin Genshin Genshin Genshin Ganyu Genshin
Genshin Genshin Genshin Genshin Genshin Genshin Genshin
Genshin Genshin Genshin Genshin Genshin Genshin Genshin)");
    return 0;
}
```

# Problem B. 虚假营销

注意要浮点数除法。

```
#include <bits/stdc++.h>
using namespace std;
signed main()
{
    int m;
    cin >> m;
    printf("%.6f", m * 1e9 / 1024 / 1024 / 1024);
    return 0;
}
```

# Problem C. 排序算法

一种避免精度误差的办法，对判断 $\sqrt{p} \geq \frac{nk}{2}$，两边同时平方得 $p \geq \frac{n^2k^2}{4} \Rightarrow 4p \geq n^2k^2$。

```
#include <bits/stdc++.h>
using namespace std;
int n, k, p;
signed main()
{
    ios::sync_with_stdio(false), cin.tie(0);
    cin >> n >> k >> p;
    if (4 * p >= n * n * k * k)
        cout << "zhe ci xiang bu chong dou bu xing la";
    else
        cout << "xiang xin zhu zhu xia ye cheng bu liao duo jiu le";
    return 0;
}
```

# Problem D. 语言模型

可以定义 $m = \max(a, b, c)$，然后统计有多少个值等于 $m$，来避免排序。

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
signed main()
{
    ios::sync_with_stdio(false), cin.tie(0);
    ll n;
    for (cin >> n; n--;)
    {
        double a, b, c;
        cin >> a >> b >> c;
        double m = max({a, b, c});
        if ((a < 0.5 && b < 0.5 && c < 0.5) || ((a == m) + (b == m) + (c == m)) >= 2)
        {
            cout << "le\n";
        }
        else if (a == m)
        {
            cout << "dian\n";
        }
        else if (b == m)
        {
            cout << "xiao\n";
        }
        else
        {
            cout << "ji\n";
        }
    }
    return 0;
}
```

# Problem E. 兰道定理

小模拟题。可以用前缀和，不用也行。

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
signed main()
{
    ios::sync_with_stdio(false), cin.tie(0);
    ll n, s = 0;
    cin >> n;
    for (ll i = 1, v; i <= n; ++i)
    {
        cin >> v;
        s += v;
        if (s < i * (i - 1) / 2)
```

```
        {
            cout << "no";
            return 0;
        }
    }
    cout << (s == n * (n - 1) / 2 ? "yes" : "no");
    return 0;
}
```

## Problem F. 求分拆数

易得，五边形数的通项为

$$a'_n = 1 + (2+1+1) + (3+2+2) + \cdots + (n + (n-1) + (n-1)) = \sum_{i=1}^{n}(3i-2) = \frac{n(3n-1)}{2}$$

。

则广义五边形数为 $a = (a'_1, a'_1 + 1, a'_2, a'_2 + 2, \cdots, a'_n, a'_n + n)$。

然后直接套题给公式计算即可。

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const ll mn = 1e2 + 10;
ll a[mn], p[mn], n;
signed main()
{
    cin.tie(0)->ios::sync_with_stdio(false);
    cin >> n;
    p[0] = 1;
    for (ll i = 1; i <= n / 2 + 1; ++i)
    {
        a[2 * i - 1] = i * (3 * i - 1) / 2;
        a[2 * i] = i * (3 * i + 1) / 2;
    }
    for (ll i = 1; i <= n; ++i)
    {
        for (ll j = 1; a[j] <= i; ++j)
        {
            if (((j + 1) / 2 + 1) % 2 == 0)
            {
                p[i] += p[i - a[j]];
            }
            else
            {
                p[i] -= p[i - a[j]];
            }
        }
    }
    cout << p[n];
    return 0;
}
```

## Problem G. 磁盘寻道

找到最大的 $l$ 和最小的 $r$，满足 $a_l \le b < a_r$。维护双指针 $l, r$，设下标从 1 开始，当前未处理的调度下标为 $[1, l] \cup [r, n]$。初始时 $[1, l] \cup [r, n] = [1, n]$。然后每次只需要判断 $a_l, a_r$ 哪个离 $b$ 更近，就走哪个(同近走左)，走了之后移动一下指针。这样的复杂度是 $O(n)$ 的。

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const ll mn = 2e5 + 10;
ll n, a[mn], b, l, r; // 尚未处理:[1,l]U[r,n]
signed main()
{
    cin >> n >> b;
    r = n + 1;
    for (ll i = 1; i <= n; ++i)
    {
        cin >> a[i];
        if (a[i] <= b)
        {
            l = max(l, i);
        }
        else
        {
            r = min(r, i);
        }
    }
    auto dis = [&](ll i)
    { return abs(a[i] - b); };
    while (!(l == 0 && r > n))
    {
        if (l && dis(l) <= dis(r))
        {
            cout << dis(l) << ' ';
            b = a[l--];
        }
        else
        {
            cout << dis(r) << ' ';
            b = a[r++];
        }
    }
    return 0;
}
```

## Problem H. 地址变换

字符串小模拟。按题意实现即可。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define sc(x) scanf("%lld", &x)
typedef long long ll;
vector<string> v;
```

```
string s;
signed main()
{
    cin >> s;
    while (true)
    {
        ll idx = s.find(":");
        if (idx == -1)
        {
            break;
        }
        v.push_back(s.substr(0, idx));
        s = s.substr(idx + 1);
    }
    v.push_back(s);
    for (int i = 0; i < v.size(); ++i)
    {
        bool abbr = false;
        if (i > 0 && i + 1 < v.size() && v[i].size() == 0)
        {
            abbr = true;
        }
        while (v[i].size() < 4)
        {
            v[i] = '0' + v[i];
        }
        if (abbr)
        {
            while (v.size() < 8)
            {
                v.insert(v.begin() + i, "0000");
            }
        }
    }
    for (ll i = 0; i < 8; ++i)
    {
        cout << v[i] << " :"[i < 7];
    }
    return 0;
}
```

# Problem I. 网络容量

考点：数据结构，STL

本题难点可能在于理解题意上

题意：多个网络，每个网络里主机编号不相同，若存在编号间隔为1的主机未被相连，则必须相连在一起，求最小网络容量

简单来说，即可以将网络主机编号从小到大排序，遍历得到每个网络里的主机数量

可使用multiset（自动排序且不去重）

```
#include <bits/stdc++.h>
using namespace std;
```

```
#define int long long
const int mn = 1e5 + 10;

void solve()
{
    int n;
    cin >> n;
    multiset<int>s;
    for (int i = 1, a; i <= n; ++i)
    {
        cin >> a;
        s.insert(a);
    }
    int ans = INT_MAX;
    while (!s.empty())
    {
        int cnt = 1, v = *s.begin();
        s.erase(s.begin());
        for (; !s.empty();)
        {
            auto pr = s.find(++v);
            if (pr == s.end())
            {
                break;
            }
            s.erase(pr);
            ++cnt;
        }
        ans = min(cnt, ans);
    }
    cout << ans;
}
signed main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int tt = 1;
    //cin >> tt;
    while (tt--) solve();
    return 0;
}
```

# Problem J. 模数定理

考点：数据结构，滑窗

求在 $[l,r]$ 中任意两个数 $a_i \bmod a_j \ \ (l \le i \le r, l \le j \le r)$ 的最大值，即求 $[l,r]$ 中的严格次最大值，同时特判 $[l,r]$中全部相同的情况即可。

std：滑窗预处理出所有答案，窗口大小为 $k+1$

~~但实际上为了不那么难看，降低了数据强度，这题可以暴力骗很多分（鼓励暴力偏分qwq）~~

```
#include<bits/stdc++.h>
using namespace std;
```

```
#define int long long
const int mn = 1e6 + 10;
int a[mn];

void solve()
{
    int n, k;
    cin >> n >> k;

    for (int i = 1; i <= n; i++)
    {
        cin >> a[i];
    }
    map<int, int>mp, ans;
    for (int l = 1, r = 1; r <= n; r++)
    {
        mp[-a[r]]++;
        if (r - l == k)
        {
            if (mp.begin()->second == k + 1) ans[l] = 0;
            else
            {
                int x, cnt = 0;
                for (auto& p : mp)
                {
                    cnt++;
                    x = -p.first;
                    if (cnt == 2) break;
                }
                ans[l] = x;
            }
            mp[-a[l]]--;
            if (!mp[-a[l]]) mp.erase(-a[l]);
            l++;
        }
    }
    int q;
    cin >> q;
    while (q--)
    {
        int l, r;
        cin >> l >> r;
        cout << ans[l] << "\n";
    }

}

signed main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int tt = 1;
    //cin >> tt;
    while (tt--)solve();
}
```

# Problem K. 陌上花开

洛谷cdq分治模板题 -> 陌上花开

然而此题与cdq分治无关，~~总不能这么原吧~~

首先，题目对于序列 $b_1b_2b_3$ 需要满足 $\forall i \in [1,n], \exists j \in [1,3]$ ， $b_j \geq a_{ij}$ ，然后求出 $b_1+b_2+b_3$ 的最小值。

对于三个维度上的限制，似乎有点无从下手，所以我们先考虑两个维度：对于序列 $b_2b_3$ ，需要满足 $\forall i \in [1,n], \exists j \in [2,3]$ ，求 $b_2+b_3$ 的最小值。这样是不是简单多了？

例如样例二简化成两个维度后：

```
3
3 1
4 10
10 7
```

对于这样一个无序的二维序列，很显然第一件事是排序，我们可以根据 $a_{i2}$ 从大到小排序：

```
3
10 7
4 10
3 1
```

然后我们根据 $b_2$ 从大到小遍历，并看 $b_3$ 最小能取多少 (遍历到第 $i$ 行时，$b_3$ 最小取值为 $max(a_{j3})$ ，其中 $j \leq i-1$)，然后每次遍历更新 $b_2+b_3$ 的最小值，时间复杂度为 $O(n*log_2n)$。

当 $b_2=10$ ，显然 $b_3$ 最小取 $0$ 。

当 $b_2=4$ ， $b_3$ 最小取 $7$ 。

...

最后还有 当 $b_2=0$ 时，$b_3=max(a_{i3})=10$。(这里容易漏掉)

好了，这是只有两个维度时的做法，我们衍生到三个维度：

我们可以根据第一个维度 $a_{i1}$ 进行排序，然后 $b_1$ 从大到小遍历取值 。当遍历第 $i$ 场比赛时，$b_1=a_{i1}$ ，场次大于等于 $i$ 的比赛我们都可以不用考虑，因为第一个 rating 值一定小于等于 $b_1$；我们只需要考虑场次小于 $i$ 的比赛的后两个 rating 值，也就是：$\forall j \in [1,i-1]$ ，满足 $b_2 \geq a_{j2}$ 或者$b_3 \geq a_{j3}$ 其中至少一个，那这个子问题不就是刚刚两个维度时的问题吗？

所以，我们根据第一个维度 $a_{i1}$ 进行排序，然后 $b_1$ 从大到小遍历取值 ，把 $b_1$ 遍历过的的场次的后两个 rating值都放入一个二维序列， $b_1$ 每遍历一个值就对这个二维序列进行一次排序，然后再遍历这个二维序列 (参考两个维度时的做法)，这样的做法时间复杂度为：$O(n^2*log_2n)$ ，这里的 $n$ 的数量级只有 $1e3$ ，所以已经可以过题。

$O(n^2*log_2n)$ ac代码：

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long
using pii = pair<ll, ll>;
```

```
void solve()
{
    ll n;
    cin >> n;
    vector<vector<ll>> a(n + 1, vector<ll>(3, 0));
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < 3; j++) {
            cin >> a[i][j];
        }
    }
    vector<int> p(n + 1);
    iota(p.begin(), p.end(), 0);
    sort(p.begin(), p.end(), [&](int i, int j) {return a[i][0] > a[j][0]; });
    ll ans = LLONG_MAX;
    for (int i = 0; i <= n; i++) {
        ll res1 = a[p[i]][0];
        vector<pii> b(i);
        for (int j = 0; j < i; j++) {
            b[j] = { a[p[j]][1],a[p[j]][2] };//未被res1覆盖
        }
        vector<int> pp(i);
        iota(pp.begin(), pp.end(), 0);
        sort(pp.begin(), pp.end(), [&](int i, int j) {return b[i].first >
b[j].first; });
        ll res2 = 0, res3 = 0;
        for (int j = 0; j <i; j++) {
             res2 = b[pp[j]].first;
             ans = min(ans, res1 + res2 + res3);
             res3 = max(res3, b[pp[j]].second);
        }
        ans = min(ans, res1 + res3);
    }
    cout << ans << '\n';
}


signed main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int tt = 1;

    //cin >> tt;
    while (tt--)solve();
}
```

然鹅，这还不是最优复杂度。

我们在遍历第一个维度时，都对另外两个维度组成的二维序列进行了排序，这样做太暴力了。

排序的目的是维护这个二维序列的有序性，那么直接用个set不就好了，考虑到数据可能重复，所以我们用multiset。

这样就不用每次都排序了，时间复杂度直接降为 $O(n^2)$：

```cpp
//陌上开花 n^2 解法
#include<bits/stdc++.h>
using namespace std;
using ll = long long;
using pii = pair<ll, ll>;


void solve()
{
    ll n;
    cin >> n;
    vector<vector<ll>> a(n + 1, vector<ll>(3));
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < 3; j++) {
            cin >> a[i][j];
        }
    }

    multiset<pii,greater<pii>> s;
    vector<int> p(n + 1);
    iota(p.begin(), p.end(), 0);

    sort(p.begin() + 1, p.end(), [&](int i, int j) {
        return a[i][0] > a[j][0];
        }
    );

    ll ans = LLONG_MAX;
    for (int i = 1; i <= n; i++) {
        //ll tmp = LLONG_MAX;
        ll res = a[p[i]][0];
        ll res2 = 0, res3 = 0;
        for (pii j : s) {
            res2 = j.first;
            ans = min(ans, res + res2+res3);
            res3 = max(res3, j.second);
        }
        //只包含1 3
        ans = min(ans, res + res3);
        s.insert({ a[p[i]][1],a[p[i]][2] });
    }
   //只包含2 3
    ll res2 = 0, res3 = 0;
    for (pii i : s) {
        res2 = i.first;
        ans = min(ans, res2 + res3);
        res3 = max(res3, i.second);
    }
    ans = min(ans, res3);
    cout << ans << '\n';

}


signed main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
```

```
    cout.tie(0);
    int tt = 1;

    //cin >> tt;
    while (tt--)solve();
}
```

# Problem L. 里氏代换

对有向图，设 $f_{i,j}$ 表示 $i$ 是否可达 $j$，如果可达，则可以用 $i$ 里氏代换 $j$。则对每个询问 $u, v$，输出 $f_{v,u}$ 即可。

初始设 $f_{i,i} = 1$。按照拓扑序进行 DP，对点 $u$ 及其所有前驱 $v$，有 $f_{u,k} = f_{u,k}|f_{v_1,k}|f_{v_2,k}|\cdots|f_{v_p,k}$。即所有子节点求交，用按位或即可。

这样的空间复杂度为 $O(n^2)$，时间复杂度为 $O(mn)$，因为共有 $m$ 条边，每条边对 $f_{u,k}$ 的所有 $k$ 进行并。

考虑进行优化。由于每个 $f$ 只占用一个位，考虑位域，使用 bitset，可将空间复杂度降为 $O(\frac{n^2}{C})$，时间复杂度为 $O(\frac{nm}{C})$，其中 $C$ 是操作系统位数，一般为 $64$。

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const ll mn = 1e4 + 5;
signed main()
{
    ios::sync_with_stdio(false), cin.tie(0);
    ll n, m, d[mn] = {}; // 入度
    vector<ll> g[mn];
    cin >> n >> m;
    for (ll u, v; m--;)
    {
        cin >> u >> v;
        g[u].push_back(v);
        ++d[v];
    }
    bitset<mn> f[mn]; // f[i][j]:i是否是j的父类
    queue<ll> q;
    for (ll i = 1; i <= n; ++i)
    {
        f[i][i] = true;
        if (!d[i])
        {
            q.push(i);
        }
    }
    while (!q.empty())
    {
        ll u = q.front();
        q.pop();
        for (auto v : g[u])
        {
            f[v] |= f[u];
```

```
                if (--d[v] == 0)
                {
                    q.push(v);
                }
            }
        }
        ll Q;
        cin >> Q;
        for (ll u, v; Q--;)
        {
            cin >> u >> v;
            cout << (f[v][u] ? "Yes\n" : "No\n");
        }
        return 0;
}
```

## Problem M. 归零行动

**题意：**

对于 $b_i > b_j$，只有在 $a_i \geq a_j$ 的情况下才能连接，且需要花费 $|b_i - b_j|$。

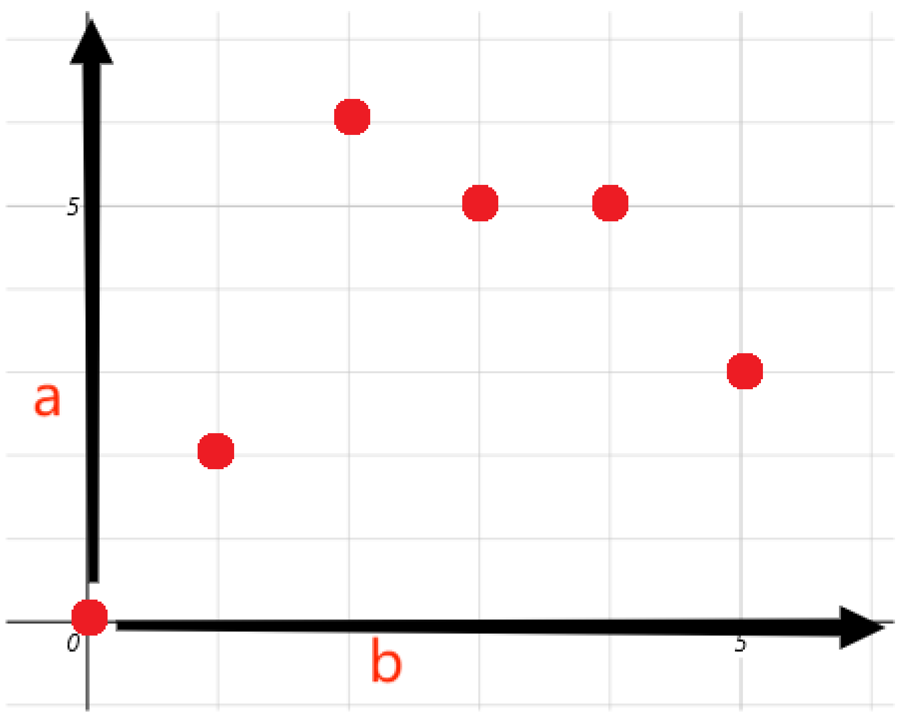
求使所有点和 (0,0) 联通，最小花费是多少。

**思路：**

先对 $n+1$ 个点根据 $b_i$ 进行排序。然后我们可以把这 $n+1$ 个点抽象在平面直角坐标系上（归零星(0,0)也算一个点）：

拿样例2举例:

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



为了花费最小，显然贪心策略就是：对于第 $i$ 个点，连接**左边最靠近自己且不高于自己的点。**

那么暴力算法的思路就出来了：对于第 $i$ 个点往左遍历，找到**第一个**不高于自己的点并连接。

但这种算法的最差时间复杂度是 $O(n^2)$ ，你可以尝试一下，但是只能拿10分。

所以，我们考虑优化。

我们从左到右遍历点，当我们遍历完一个点时：右边所有不低于这个点的点都可以选择连接它，但是连接它不一定是最优策略，因为可能还有**更靠右的点可以连接**。所以发现了吗，这其实是一个最大值查询问题：遍历到第 $i$ 个点时，查询低于它最靠右的点，并更新右边高于第 $i$ 个点可连接的最靠右点。

那么我们就可以用个树状数组/线段树/st表 来做，时间复杂度降为 $O(nlog_2n)$。

$O(nlog_2n)$ 代码：

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long
#define pii pair<int,int>
#define lowbit(x) (x&(-x))
ll n,tree[200010];
void add(int x, ll num) {
    while (x <= 200000) {
        tree[x] = max(tree[x], num);
        x += lowbit(x);
    }
}
ll query(int x) {
    ll res = 0;
    while (x) {
        res = max(tree[x], res);
        x -= lowbit(x);
    }
    return res;
}
void solve()
{

    cin >> n;
    vector<ll> 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<int> p(n + 1);
    iota(p.begin(), p.end(),0);
    sort(p.begin(), p.end(), [&](int i, int j) {return b[i] < b[j]; });
    ll ans = 0;
    for (int i = 1; i <= n; i++) {
        ll res = query(a[p[i]]);
        ans += b[p[i]] - res;
        add(a[p[i]], b[p[i]]);
    }
    cout << ans << '\n';
}


signed main()
{
    ios::sync_with_stdio(0);
```

```
    cin.tie(0);
    cout.tie(0);
    int tt = 1;
    //cin >> tt;
    while (tt--)solve();
}
```

这种解法已经可以过题，不过还是不够优雅， 于是我们想到有一种叫单调栈的东西：在 $O(n)$ 复杂度找到左边第一个小于等于自己的数。这不就是个单调栈板子吗。

$O(n)$ 代码：

```
#include<bits/stdc++.h>
using namespace std;
using ll = long long;
using pii = pair<ll, ll>;
void solve()
{
    ll n;
    cin >> n;
    vector<ll> 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<int> p(n + 1);
    iota(p.begin(), p.end(), 0);
    sort(p.begin() + 1, p.end(), [&](int i, int j) {
        return b[i] < b[j];
        });
    stack<pii> s;
    s.push({ 0,0 });
    ll ans = 0;
    for (int i = 1; i <= n; i++) {
        while (a[p[i]] < s.top().first) s.pop();
        ans += b[p[i]] - s.top().second;
        s.push({ a[p[i]],b[p[i]] });
    }
    cout << ans << '\n';
}


signed main()
{
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int tt = 1;

    //cin >> tt;
    while (tt--)solve();
}
```

## Problem N. 一条归桥

先考虑一个稍微简单的问题：如何求 $\leq k$ 的路径数 $s'_{i,j,k} = \sum_{l=1}^{k} r'_{i,j,l}$，而不是求路径和 $s_{i,j,k}$。

设邻接矩阵为 $A'_{i,j}=cnt_{i,j}$，其中 $cnt_{i,j}$ 是 $i\to j$ 有向边的总数。显然，$r'_{i,j,1}=A'_{i,j}$。

那么，根据 DP 的思想，利用加法原理和乘法原理，有：

$$r'_{i,j,p}=\sum_{k=1}^{n}r'_{i,k,p-1}\cdot r'_{k,j,1}$$

也就是枚举中间点 $k$，先从 $i$ 走 $p-1$ 步到 $k$，再从 $k$ 一步走到 $j$。

例如对 $n=3,p=3$，有 $r'_{1,2,3}=r'_{1,1,2}r'_{1,2,1}+r'_{1,2,2}r'_{2,2,1}+r'_{1,3,2}r'_{3,2,1}$。容易知道该式子的组合数学意义。

不难发现，上式恰好就是矩阵乘法的表达式。也就是说，有 $r'_{i,j,k}=A'^{k}_{i,j}$，其中 $A'^{k}_{i,j}=A'^{k-1}_{i,j}\times A'_{i,j}$。

现在从求 $r'_k$ 即求 $A'^k$ 问题转化为求 $s'_k$ 即 $B'^k=A'+A'^2+A'^3+\cdots+A'^k$ 问题。

> 由于矩阵不满足等比数列求 $k$ 项和公式，**不能**直接像实数一样对 $x+x^2+x^3+\cdots+x^k$ 求 $\dfrac{x(x^k-1)}{x-1}$。这是因为，不能保证 $A-I$(其中 $I$ 是同型单位阵)总是可逆的。矩阵 $M$ 可逆的条件是行列式 $|M|$ 不为 $0$。考虑一个每点只有一个自环的图，显然 $A=I,|A-I|=0$。所以无法直接套用实数等比数列公式。

所以考虑分治求解。问题转化为，已知 $n$ 阶方阵 $A$，求 $A+A^2+A^3+\cdots+A^k$。

令 $B_i=A+A^2+\cdots+A^i$，$n$ 阶单位阵为 $E$。若 $k$ 为偶数，上式可以转化为：

$$B_k=(A+A^2+\cdots+A^{\frac{k}{2}})+A^{\frac{k}{2}}(A+A^2+\cdots+A^{\frac{k}{2}})=B_{\frac{k}{2}}+A^{\frac{k}{2}}B_{\frac{k}{2}}=(E+A^{\frac{k}{2}})B_{\frac{k}{2}}$$

若 $k$ 为奇数，可以直接用上一个偶数，即 $B_k=B_{k-1}+A^k$。

其中，$A^i$ 可由矩阵快速幂计算，复杂度为 $O(n^3\log i)$。而 $B_k$ 的递推需要 $\log k$ 次，故总复杂度为 $O(n^3\log^2 k)$。

到此为止，如果是求路径数，已经做完了。但是本题要求路径和，所以需要进一步修改。

> **不能**直接把边权和当成路径数直接套用上面的公式。考虑每个点只有一条边权为 $1$ 的自环图， 那么由于 $1\cdot 1=1$，会求出 $A'^k=A=I$，显然不符合事实。因为考虑 $k=2$，就有 $1+1=2$ 的路径，$s_{i,i,2}=3$ 才是正确的。

设初始条件是 $r_{i,j,1}=\sum w_{(i,j)}$，即所有 $i\to j$ 边权和。边数是 $r'_{i,j,k}$。正确的的组合数学 DP 公式为：

$$r_{i,j,p}=\sum_{k=1}^{n}(r_{i,k,p-1}\cdot r'_{k,j,1}+r_{k,j,1}\cdot r'_{i,k,p-1})$$

含义是：所有已经从 $i$ 走到 $k$，走 $p-1$ 步的方案里，每个方案可以分裂出 $r'_{k,j,1}$ 个子方案，每个子方案在原有的基础上都有 $r_{i,k,p-1}$ 的边权和。所有子方案里新加的权和是 $r_{k,j,1}\cdot r'_{i,k,p-1}$，具体而言这些子方案可以细分为 $r'_{i,k,p-1}$ 组，每组恰好包含 $k\to j$ 的所有边一次且仅一次，这些边的权和为 $r_{k,j,1}$。

> 举个例子，对 $n=2$ 的图，只有三条边，$w_{(1,2)}=1,w_{(1,2)}=2,w_{(2,1)}=4$。显然有：
>
> $$r_{i,j,1}=A=\begin{pmatrix}0&3\\4&0\end{pmatrix},r'_{i,j,1}=A'=\begin{pmatrix}0&2\\1&0\end{pmatrix}$$
>
> 则：

> $$\begin{aligned} r_{1,1,2} &= (r_{1,1,1} \cdot r'_{1,1,1} + r_{1,1,1} \cdot r'_{1,1,1}) + (r_{1,2,1} \cdot r'_{2,1,1} + r_{2,1,1} \cdot r'_{1,2,1}) \\ &= (0 \cdot 0 + 0 \cdot 0) + (3 \cdot 1 + 4 \cdot 2) = 11 \\ r_{1,2,2} &= (r_{1,1,1} \cdot r'_{1,2,1} + r_{1,2,1} \cdot r'_{1,1,1}) + (r_{1,2,1} \cdot r'_{2,2,1} + r_{2,2,1} \cdot r'_{1,2,1}) \\ &= (0 \cdot 2 + 3 \cdot 0) + (3 \cdot 0 + 0 \cdot 2) = 0 \\ r_{2,1,2} &= (r_{2,1,1} \cdot r'_{1,1,1} + r_{1,1,1} \cdot r'_{2,1,1}) + (r_{2,2,1} \cdot r'_{2,1,1} + r_{2,1,1} \cdot r'_{2,2,1}) \\ &= (4 \cdot 0 + 0 \cdot 1) + (0 \cdot 1 + 4 \cdot 0) = 0 \\ r_{2,2,2} &= (r_{2,1,1} \cdot r'_{1,2,1} + r_{1,2,1} \cdot r'_{2,1,1}) + (r_{2,2,1} \cdot r'_{2,2,1} + r_{2,2,1} \cdot r'_{2,2,1}) \\ &= (4 \cdot 2 + 3 \cdot 1) + (0 \cdot 0 + 0 \cdot 0) = 11 \end{aligned}$$
>
> 即 $r_{i,j,2} = \begin{pmatrix} 11 & 0 \\ 0 & 11 \end{pmatrix}$。
>
> 任意 $A'^k$ 仍然可以通过上文方法计算，即显然有 $r'_{i,j,2} = A'^2 = A' \times A' = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$。
>
> 同理，可以计算出 $r'_{i,j,3} = A'^3 = \begin{pmatrix} 0 & 4 \\ 2 & 0 \end{pmatrix}, r_{i,j,3} = \begin{pmatrix} 0 & 28 \\ 19 & 0 \end{pmatrix}$，可以不断往下递推，求出任意 $r_{i,j,k}$。

通过上述办法，虽然可以正确求出答案，但是每次递推的复杂度为 $O(n^3)$，要求出 $s_{i,j,k} = \sum_{l=1}^{k} r_{i,j,l}$ 需要 $O(n^3k)$，显然是不可接受的。下面考虑优化该递推。

上面的表达式跟矩阵乘法十分相似，唯一不同是 $a \cdot b$ 变成了 $a \cdot b + c \cdot d$。

设 $M^{p-1} = \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix}, M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$，有:

$$M^p = M^{p-1} \times M = \begin{pmatrix} a'a + b'c & a'b + b'd \\ c'a + d'c & c'b + d'd \end{pmatrix}$$

不妨设二阶方阵 $F_{i,j} = \begin{pmatrix} r'_{i,j,1} & r_{i,j,1} \\ 0 & r'_{i,j,1} \end{pmatrix}$，$F_{i,j}^{p-1} = \begin{pmatrix} r'_{i,j,p-1} & r_{i,j,p-1} \\ 0 & r'_{i,j,p-1} \end{pmatrix}$，则有:

$$F_{i,j}^p = F_{i,j}^{p-1} \times F_{i,j} = \begin{pmatrix} r'_{i,j,p-1} r'_{i,j,1} & r_{i,j,p-1} r'_{i,j,1} + r_{i,j,1} r'_{i,j,p-1} \\ 0 & r'_{i,j,p-1} r'_{i,j,1} \end{pmatrix}$$

> 如何想出要“设”这样的方阵的思路见下文。

该方阵的主对角线刚好满足上文 $r'_{i,j,p}$ 的计算公式，且第一行第二个元素刚好满足上文 $r_{i,j,p}$ 的计算公式。

我们用二阶方阵代替矩阵里的每一个元素，设 $F_{i,j} = \begin{pmatrix} r'_{i,j,1} & r_{i,j,1} \\ 0 & r'_{i,j,1} \end{pmatrix}$，构造分块(/嵌套)矩阵:

$$A = \begin{pmatrix} F_{1,1} & F_{1,2} & \cdots & F_{1,n} \\ F_{2,1} & F_{2,2} & \cdots & F_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ F_{n,1} & F_{n,2} & \cdots & F_{n,n} \end{pmatrix}$$

则该矩阵的幂公式为 $A^p = A^{p-1} \times A^p$，逐步递推，根据数学归纳法，可得:

$$
\begin{aligned}
A_{i,j}^{p} &= \sum_{k=1}^{n} A_{i,k}^{p-1} \times F_{k,j} \\
&= \sum_{k=1}^{n} \begin{pmatrix} r'_{i,k,p-1} r'_{k,j,1} & r_{i,k,p-1} r'_{k,j,1} + r_{k,j,1} r'_{i,k,p-1} \\ 0 & r'_{i,k,1} r'_{k,j,p-1} \end{pmatrix} \\
&= \begin{pmatrix} \sum_{k=1}^{n} r'_{i,k,p-1} r'_{k,j,1} & \sum_{k=1}^{n} r_{i,k,p-1} r'_{k,j,1} + r_{k,j,1} r'_{i,k,p-1} \\ 0 & \sum_{k=1}^{n} r'_{i,k,1} r'_{k,j,p-1} \end{pmatrix} \\
&= \begin{pmatrix} r'_{i,j,p} & r_{i,j,p} \\ 0 & r'_{i,j,p} \end{pmatrix}
\end{aligned}
$$

根据线性代数知识，分块矩阵的计算可以视为一般矩阵的计算，即可将 $A$ 视大小为 $2n \times 2n$ 的矩阵：

$$
A^p = \begin{pmatrix}
r'_{1,1,p} & r_{1,1,p} & r'_{1,2,p} & r_{1,2,p} & \cdots & r'_{1,n,p} & r_{1,n,p} \\
0 & r'_{1,1,p} & 0 & r'_{1,2,p} & \cdots & 0 & r'_{1,n,p} \\
r'_{2,1,p} & r_{2,1,p} & r'_{2,2,p} & r_{2,2,p} & \cdots & r'_{2,n,p} & r_{2,n,p} \\
0 & r'_{2,1,p} & 0 & r'_{2,2,p} & \cdots & 0 & r'_{2,n,p} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
r'_{n,1,p} & r_{n,1,p} & r'_{n,2,p} & r_{n,2,p} & \cdots & r'_{n,n,p} & r_{n,n,p} \\
0 & r'_{n,1,p} & 0 & r'_{n,2,p} & \cdots & 0 & r'_{n,n,p}
\end{pmatrix}
$$

则 $r_{i,j,k} = A^k_{2i,2j-1}, s_{i,j,k} = B^k_{2i-1,2j} = A_{2i-1,2j} + A^2_{2i-1,2j} + \cdots + A^k_{2i-1,2j}$。故只需要求出 $B^k$，然后输出所有 $1 \le i,j \le n, B^k_{2i-1,2j}$ 即可。

由于新方阵的大小为 $2n$，故本题总时间复杂度为 $O((2n)^3 \log^2 k) = O(8n^3 \log^2 k) \approx 9 \times 10^8$。空间复杂度为 $O((2n)^2) = O(4n^2)$。

> 附：构造出二阶方阵 $F$ 的思路简述：令向量 $f^T = (r_1, r'_1)$ 维护二维信息(单点自环的个数 $r'$，自环边权和 $r$)，且 $f^{kT}$ 维护长为 $k$ 的自环路径数 $r_k$ 与自环边权和 $r'_k$。类比矩阵快速幂求斐波那契，考虑构造递推：
>
> $$f^k = \begin{pmatrix} r_k \\ r'_k \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} r_{k-1} \\ r'_{k-1} \end{pmatrix} = F f^{k-1}$$
>
> 即：
>
> $$\begin{cases} r_k = a_{11} r_{k-1} + a_{12} r'_{k-1} \\ r'_k = a_{21} r_{k-1} + a_{22} r'_{k-1} \end{cases}$$
>
> 根据上文公式(那两个 DP)，可以得出，$r_k = r'_1 r_{k-1} + r_1 r'_{k-1}, r'_k = r'_1 r'_{k-1}$。代入系数，得：
>
> $$a_{11} = r'_1, a_{12} = r_1, a_{21} = 0, a_{22} = r'_1$$
>
> 即构造出 $F$ 为 $\begin{pmatrix} r' & r \\ 0 & r' \end{pmatrix}$。

参考代码：

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const ll mod = 998244353;
using arr = vector<vector<ll>>;
struct matrix
{
    ll n;
```

```
    arr a;
    matrix(ll nn) : n(nn)
    {
        a = arr(n, vector<ll>(n));
    }
    matrix(arr aa) : n(aa.size()), a(aa) {}
    friend matrix operator+(const matrix &x, const matrix &y)
    {
        matrix r = matrix(x.n);
        for (ll i = 0; i < x.n; ++i)
        {
            for (ll j = 0; j < x.n; ++j)
            {
                r.a[i][j] = (x.a[i][j] + y.a[i][j]) % mod;
            }
        }
        return r;
    }
    friend matrix operator*(const matrix &x, const matrix &y)
    {
        matrix r = matrix(x.n);
        for (ll i = 0; i < x.n; ++i)
        {
            for (ll j = 0; j < x.n; ++j)
            {
                for (ll k = 0; k < x.n; ++k)
                {
                    r.a[i][j] = (r.a[i][j] + x.a[i][k] * y.a[k][j]) % mod;
                }
            }
        }
        return r;
    }
    matrix static unit(ll k)
    { // 单位阵
        matrix r = matrix(k);
        for (ll i = 0; i < k; ++i)
        {
            r.a[i][i] = 1;
        }
        return r;
    }
    matrix pow(ll k)
    {
        matrix c = *this, r = unit(n);
        for (; k; k >>= 1)
        {
            if (k & 1)
            {
                r = r * c;
            }
            c = c * c;
        }
        return r;
    }
    matrix psum(ll k)
    {
        if (k == 0)
```

```
        {
            return matrix(n);
        }
        if (k == 1)
        {
            return *this;
        }
        if (k % 2 == 1)
        {
            return psum(k - 1) + pow(k);
        }
        return (unit(n) + this->pow(k / 2)) * psum(k / 2);
    }
};
using mat = matrix;
ll n, m, k;
signed main()
{
    ios::sync_with_stdio(false), cin.tie(0);
    cin >> n >> m >> k;
    mat e = mat(n * 2);
    for (ll u, v, w; m--;)
    {
        cin >> u >> v >> w;
        --u, --v;
        mat f = mat({{1, w}, {0, 1}});
        (e.a[u * 2][v * 2] += f.a[0][0]) %= mod;
        (e.a[u * 2][v * 2 + 1] += f.a[0][1]) %= mod;
        (e.a[u * 2 + 1][v * 2] += f.a[1][0]) %= mod;
        (e.a[u * 2 + 1][v * 2 + 1] += f.a[1][1]) %= mod;
    }
    e = e.psum(k);
    for (ll i = 0; i < n; ++i)
    {
        for (ll j = 0; j < n; ++j)
        {
            cout << e.a[2 * i][2 * j + 1] << ' ';
        }
        cout << '\n';
    }
    return 0;
}
```

## Problem O. 黄金买卖

若只有至多一次买卖，等价于求 $\max_{1\leq i<j\leq n} p_j - p_i$。

对每个 $j$，其 $a_i$ 必定在最小值处取得。即维护前缀 min $m_i = \min_{1\leq k\leq i} p_i$，等价于求 $\max_{1\leq i\leq n} p_i - m_i$，可以 $O(n)$ 实现。有可能不进行买卖(一直降价)，所以最终结果与 0 取 max。复杂度 $O(n)$。

现在扩展到做两笔交易，即求 $\max p_j - p_i + p_l - p_k, i < j < k < l$。

跟一笔交易一样，先处理 $i, j$，然后对 $(j, n]$ 部分，反过来，即逆序枚举 $k$，寻找最大的 $p_l$。根据两次枚举 $j, k$ 可以分别求出前缀 max 答案 $lans$ 和后缀答案 $rans$，故答案为：
$ans = \max_{i=1}^{n-1} lans_i + rans_{i+1}$。复杂度 $O(n)$。

现在再扩展到做 $k$ 笔交易，设 $g_{i,j}$ 是对前 $j$ 天，已经买卖了前 $i-1$ 笔交易且买入了第 $i$ 笔交易下最低的总额，即 $\min a_{buy_k} + \sum_{k=1}^{i-1} a_{sell_k} - a_{buy_k}, buy_k, sell_k \le j$

设 $f_{i,j}$ 是对前 $j$ 天，已经买卖了前 $i$ 笔的最高利润。即 $\max \sum_{k=1}^{i} a_{sell_k} - a_{buy_k}, , buy_k, sell_k \le j$

递推方程为：

$$f_{i,j} = \max(f_{i,j-1}, p_j - g_{i,j-1})$$
$$g_{i,j} = \min(g_{i,j-1}, p_j - f_{i-1,j-1})$$

因为要差价最大化，所以每次卖出前，让买入最小。对于新的一笔交易，将之前的交易得到的所有差价抽象等价为负的价格，即假设已经赚到的最大差价是 $c$，那么新一次买黄金的价格设为 $p - c$，即抽象成了这次买入额外降价了 $c$。

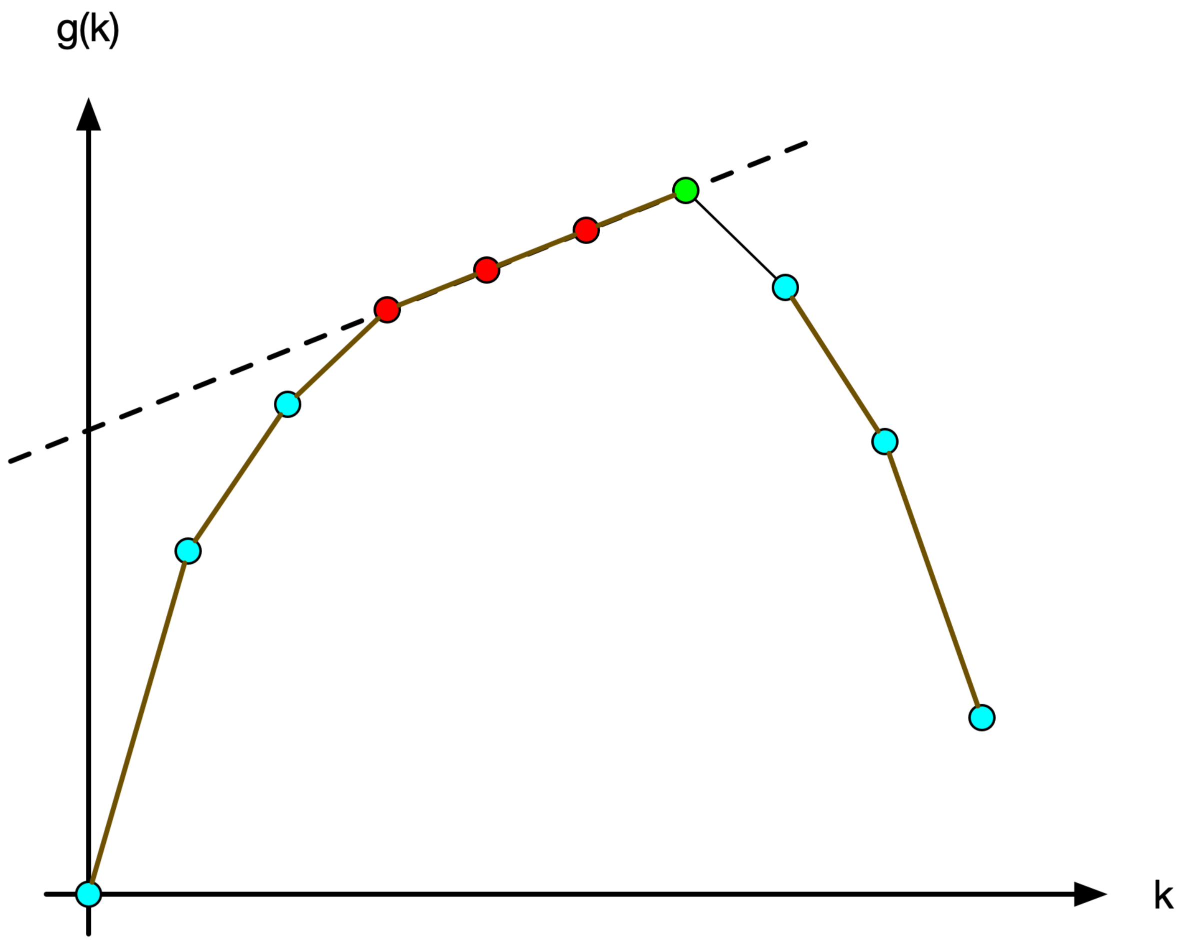
将 $i$ 状态压缩，可以优化空间复杂度从 $O(nk)$ 到 $O(n)$。时间复杂度为 $O(nk)$。但是仍然无法过题。

考虑进一步优化。

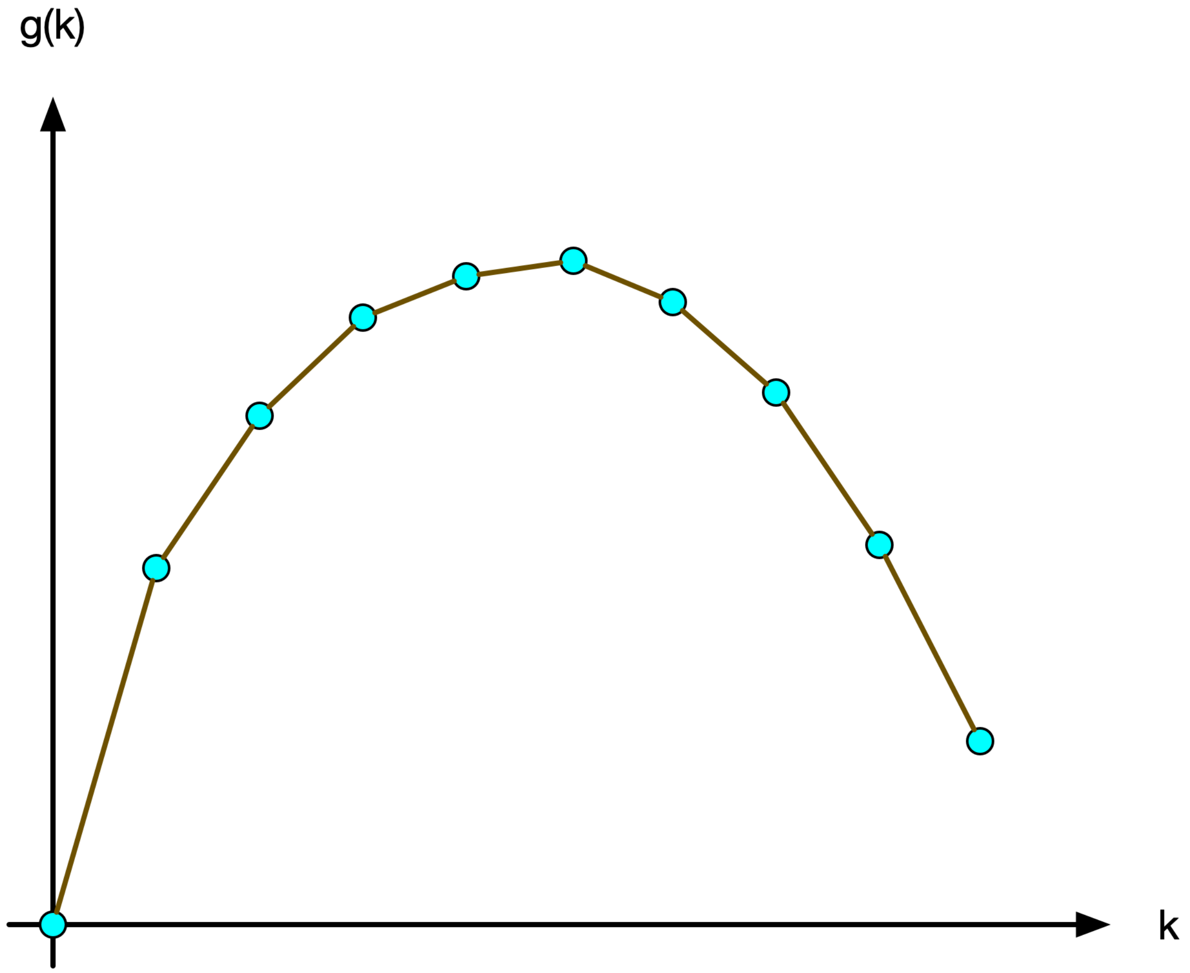
我们使用一种称为 wqs二分(王钦石在2012国家集训队论文提出)的技巧，国外又称为 Alien Trick(由 [IOI2016Aliens题目]得名)。

设恰好完成 $k$ 次买卖时，最大收益是 $g_k$。设增量(/导数)为 $g'_k = g_k - g_{k-1}$那么可以得到一个结论，增量 $g'_k$ 一定单调不降(不严格单调递增)。根据反证法已知，如果存在递减的方案，可以交换递减时的两次交易，使其变为不递减的。

当 $k$ 不断增大到最大 $n$($n$ 次交易即都当天买当天卖等价于 0 次交易)时，$g_k$ 会降到 $0$，即 $g_0 = g_n = 0$，一阶导 $g'_k$ 单调递减，不难判断 $g$ 呈倒 U 型，如图所示：



我们的任务是求出 $g$ 数列(/离散函数)的最大值 $\max g$。用几何思路，考虑作直线与 $g$ 图像相切，设枚举了斜率为 $c(c>0)$ 的直线，过所有点 $g_i(1 \le i \le n)$ 作 $n$ 条直线，如图所示：(只画出了前五条)




由于 $c>0$，所以 $y$ 截距最大的直线一定是相切的。设有斜率为 $c$ 过点 $(k, g_k)$ 的直线，不难计算得截距是 $b = g_k - kc$。回到要求的问题上，可以将截距抽象成进行了 $k$ 次买卖，净利润为 $g_k$，但每次卖出需要付手续费 $c$。

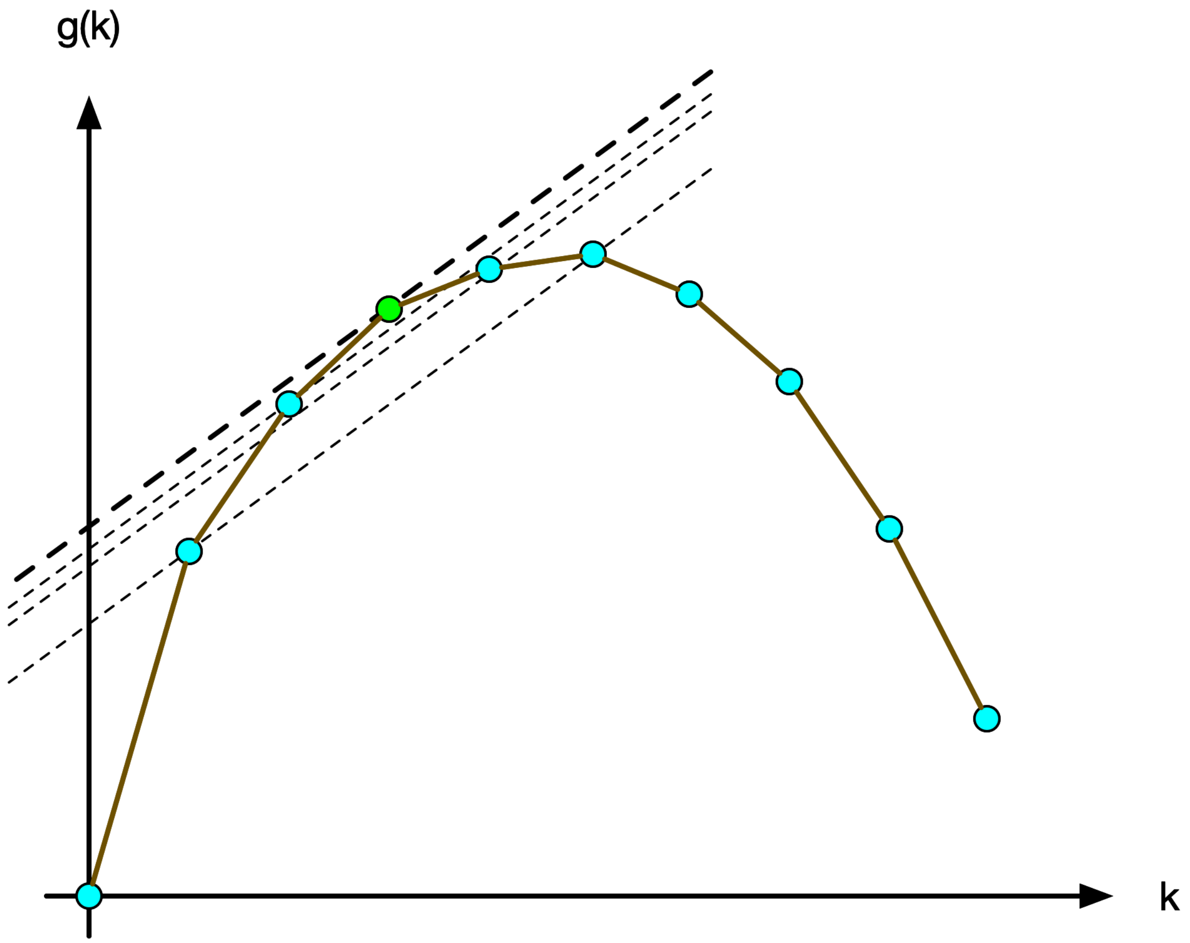
下面考虑给定 $c$ 时，如何快速求出最大的截距。抽象为已知手续费 $c$，且不限交易次数，求出最大的含手续费利润 $b$。

因为不限交易次数，考虑贪心，即只要能赚的钱不低于手续费就买卖一次。具体而言，维护最大利润 $sellp$，初始设为 $0$；持有一笔未卖出下的最大利润 $buyp$ 初始设为 $-p_1$。并维护取得这两个最值下的 $k$ 值为 $buycnt, sellcnt$。

如果发现在上一笔交易后 $sellp$ 状态下选择买入 $p_i$ 比之前维护的 $buyp$ 更优，就买入 $p_i$ 更新 $buyp$。如果发现在当前卖出 $buyp$ 得到的 $sellp$ 比之前维护的 $sellp$ 更优就卖出 $p_i$ 更新 $sellp$。(具体参见代码)

由此可以 $O(n)$ 计算出对固定的斜率 $c$ 得到的最大切线截距 $b = g_k - kc$ 及其 $k$ 值($sellcnt$)。那么设真的在 $k$ 取得 $\max g$，即 $sellp = b = g_k - kc \to g_k = sellp + kc$。

考虑到可能会有相等的情况，如图所示：




而对答案 $\max g = b + gk$ 而言，$b$ 相等时要尽可能大的 $k$，所以在上述贪心时，在利润最大化的同时，应当再最大化交易次数。即只要不亏钱就买卖，而不是赚钱才买卖。

而这道题交易次数是有限的，也就是说如果找到的 $sellcnt$ 是大于题目所要求的 $k$，是不能取的。

对斜率范围 $(0, \infty)$ 而言，不难发现随着斜率的减少，取得切线的最大 $sellcnt$ 会越来越大，超过某个界限 $c'$ 后，总是会满足 $sellcnt > k$(即不符合题意)，在这之前则总是满足 $sellcnt \le k$，满足单调性。可以用这个条件来二分斜率 $c$，在 $g$ 未下降前，找到最大的 $sellcnt \le k$。比较显然的是，只要斜率 $c > 0$，随着斜率减小切线截距一定增大，故越小的斜率答案 $g$ 值一定最大。

注意到斜率一定不会超过增量的最大值即 $\max p$，故二分上界取 $\max p$ 即可。而作图易得，取得同一点相切的斜率是一个范围，这个范围的边界一定是与增量折线斜率相等的，而增量相差 $1$ 下标，分母为 $1$，故折线斜率为整数，即二分只需要在整数范围内即可，不需要到实数范围。同时也易知，最小的增量一定是 $1$，即二分左边界是 $1$。

特别地，如果题给的 $k$ 在 $g$ 单调递减时取得(即最高点 $k' < k, g_{k'} > g_k$)，也就是说不限次数地贪心得到的 $sellcnt < k$，那么此时也可以直接不限次数贪心求出 $\max g$，只要 $p_i > p_{i-1}$ 就可以前一天买今天卖。

时间复杂度是 $O(n \log \max p)$。

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
signed main()
{
    ll k, n;
    cin >> n >> k;
    vector<ll> p(n);
    for (int i = 0; i < n; ++i)
    {
```

```
        cin >> p[i];
    }
    ll lf = 1, rf = *max_element(p.begin(), p.end());
    ll ans = -1;
    while (lf <= rf)
    {
        ll cf = (lf + rf) >> 1; // 斜率
        ll buycnt = 0, sellcnt = 0;
        ll buyp = -p[0], sellp = 0;
        for (int i = 1; i < n; ++i) // 贪心
        {
            if (sellp - p[i] >= buyp)
            {
                buyp = sellp - p[i];
                buycnt = sellcnt;
            }
            if (buyp + p[i] - cf >= sellp)
            {
                sellp = buyp + p[i] - cf;
                sellcnt = buycnt + 1;
            }
        }
        // cout << cf << ' ' << sellp << ' ' << sellcnt << '\n';
        if (sellcnt >= k)
        {
            ans = sellp + k * cf; // 注意不是+sellcnt,是+k
            lf = cf + 1;
        }
        else
        {
            rf = cf - 1;
        }
    }
    if (ans == -1)
    {
        ans = 0;
        for (int i = 1; i < n; ++i)
        {
            ans += max(p[i] - p[i - 1], 0LL);
        }
    }
    cout << ans;
    return 0;
}
/*
8 3
1 100 1 4 1 4 1 4
*/
```

> 对于上面第三个的图的情况，设 $k$ 在红点中间，二分一定能找到该斜率，这是因为最高点前取得的任意增量对应的折线斜率都能被二分出来。考虑注释给的用例。