题解：第二十届浙大宁波理工学院程序设计大赛

难度预期

A < B < C < I < J < D < E < F < G < H
其中，难度最低的 5 题是纯语法题，不涉及算法；后 5 题则是各种算法题，依次是：D.树上二分前缀和/离线处理树上前缀和。E.预处理状压DP，BFS/DFS转移状态。F.不存储的线段树。G.虚点建图/改初始化状态的最短路。H. $O(1)$ 求解数列前 n 项和。

A Storious的小杨辉三角

输出杨辉三角形前五行中的某一行。
如果能观察到前五行都是 11^{n-1} 则可以轻松首A。
正常写 5 个if就可以轻松AC。
不想写if的话，开个二维数组for一遍小DP一下，直接输出也能轻松AC。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
	std::ios::sync_with_stdio(false);
	cin.tie(nullptr);
	cout.tie(nullptr);
	Main();
	return 0;
}

void Main(void)
{
	ll n;
	cin >> n;
	cout << (ll)std::pow(11, n - 1) << '\n';
	return;
}
```

B Gray与坤变偶不变

把字符串中奇数位置的字母改变大小写后输出。
for一遍字符串改一下即可。
大部分语言都有写好的函数可以直接检测大小写和修改大小写，直接调用可以节省很多时间。
自己写下if也不会多花太多时间，也可以轻松AC。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
        std::ios::sync_with_stdio(false);
        cin.tie(nullptr);
        cout.tie(nullptr);
        Main();
        return 0;
}

void Main(void)
{
        ll n;
        std::string s;
        cin >> n;
        cin >> s;
        for (ll i = 0; i < n; i += 2)
        {
                if (islower(s[i]))
                        s[i] = toupper(s[i]);
                else if (isupper(s[i]))
                        s[i] = tolower(s[i]);
        }
        cout << s << '\n';
        return;
}
```

C Gray与你好谢谢小笼包再见

给当前时间和闹钟时间，问下一次闹钟响起是多少小时多少分钟后。(注意特判当前时间和闹钟时间相同的情况。)

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
        std::ios::sync_with_stdio(false);
        cin.tie(nullptr);
        cout.tie(nullptr);
        Main();
        return 0;
}
```

```
void Run(void)
{
	std::pair<ll, ll> now, bell;
	cin >> now.first >> now.second >> bell.first >> bell.second;
	if (bell == now)
	{
		cout << 24 << ' ' << 0 << '\n';
		return;
	}
	if (bell < now)
		bell.first += 24;
	std::pair<ll, ll> ans;
	ans.first = bell.first - now.first;
	ans.second = bell.second - now.second;
	if (ans.second < 0)
		--ans.first, ans.second += 60;
	cout << ans.first << ' ' << ans.second << '\n';
	return;
}
void Main(void)
{
	ll t;
	cin >> t;
	while (t--)
		Run();
	return;
}
```

D Gray与派对80

容易发现，参赛者之间的关系图是一个以 1 号为根的树，每个节点的深度意味着这个节点编号的参赛者是第几届参赛者。如果选择邀请一个节点，则他的祖先节点全部要被邀请，同时和他同深度的节点也要全部邀请。这意味着，选择的节点一定是某一届及之前的所有参赛者，才能满足条件。深度可以挑选心仪的遍历算法来得到，然后可以用前缀和来处理某届及以前共有多少人。随着届数的增加，选择的人数递增，具有单调性，可以二分处理每次查询，每次查询查前缀和数组中，小于等于 b_i 的数中的最大值。一共进行 Q 轮查询，总复杂度 $O(Q \times log(N))$ 。
另一种做法是离线排序查询，然后滑动指针实现。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
	std::ios::sync_with_stdio(false);
	cin.tie(nullptr);
	cout.tie(nullptr);
	Main();
	return 0;
```

```
}

constexpr ll SZ = 1e5 + 5;
ll n;
ll a[SZ];
ll q;
ll b[SZ];
std::vector<ll> E[SZ];
ll mx;
ll c[SZ], s[SZ];

void DFS(ll x, ll dep)
{
        mx = std::max(mx, dep);
        ++c[dep];
        for (const auto &son : E[x])
                DFS(son, dep + 1);
        return;
}
void Main(void)
{
        cin >> n;
        for (ll i = 1; i <= n; ++i)
                cin >> a[i];
        cin >> q;
        for (ll i = 1; i <= q; ++i)
                cin >> b[i];
        for (ll i = 2; i <= n; ++i)
                E[a[i]].push_back(i);
        mx = 1;
        DFS(1, 1);
        for (ll i = 1; i <= n; ++i)
                s[i] = s[i - 1] + c[i];
        for (ll i = 1; i <= q; ++i)
                cout << s[std::upper_bound(s + 1, s + mx + 1, b[i]) - s - 1] << '\n';
        return;
}
```

E Gray与XX启动

给当前状态和目标状态，问最少需要多少次操作，才能从当前状态到达目标状态。
状态可以压成长度为5的二进制，总共只有 2^5 种状态。
可以把每种状态作为初始状态BFS一次，就可以预处理出所有可能的询问的答案。
预处理复杂度 $O(2^5 \times 2^5)$ ，单次询问复杂度 $O(1)$ ，总复杂度 $O(2^{10} + T)$ 。
用DFS同样可以实现。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
```

```
int main(void)
{
        std::ios::sync_with_stdio(false);
        cin.tie(nullptr);
        cout.tie(nullptr);
        Main();
        return 0;
}

constexpr ll SZ = (1 << 5);
ll f[SZ][SZ];

void BFS(ll s)
{
        std::queue<ll> q;
        f[s][s] = 0;
        q.push(s);
        while (not q.empty())
        {
                ll now = q.front();
                q.pop();
                for (ll i = 0; i <= 4; ++i)
                {
                        ll bit = (1 << i);
                        if ((now & bit) != 0)
                                continue;
                        ll nxt = now + bit;
                        bit = (1 << ((i + 2) % 5));
                        if ((now & bit) != 0)
                                nxt -= bit;
                        else
                                nxt += bit;
                        bit = (1 << ((i + 3) % 5));
                        if ((now & bit) != 0)
                                nxt -= bit;
                        else
                                nxt += bit;

                        if (f[s][nxt] != -1 )
                                continue;
                        f[s][nxt] = f[s][now] + 1;
                        q.push(nxt);
                }
        }
        return;
}
void Init(void)
{
        memset(f, -1, sizeof(f));
        for (ll s = 0; s < SZ; ++s)
                BFS(s);
        return;
}
```

```
ll Read(void)
{
	ll res = 0;
	for (ll i = 0; i <= 4; ++i)
	{
		ll x;
		cin >> x;
		if (x == 1)
			res += (1 << i);
	}
	return res;
}
void Run(void)
{
	ll s = Read();
	ll t = Read();
	cout << f[s][t] << '\n';
	return;
}
void Main(void)
{
	Init();
	ll t;
	cin >> t;
	while (t--)
		Run();
	return;
}
```

F Gray与XX铁道

线段树结构的图上查询多源最短路。
从最长的那种路开始往短的路考虑，看此次查询的最短路是否经过这条路。
如果经过这条路，说明删除这条路后，查询的两个点一个和这条路的左端点连通，一个和这条路的右端点连通。那么查询结果就是：查询的一个点到这条路左端点的最短路距离+这条路的长度+查询的另一个点到这条路的右端点的最短路距离。可以像线段树一样用DFS轻松实现。复杂度 $O(Q \times log(2^N)) = O(Q \times N)$ 。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
	std::ios::sync_with_stdio(false);
	cin.tie(nullptr);
	cout.tie(nullptr);
	Main();
	return 0;
}
```

```
ll DFS(ll l, ll r, ll a, ll b)
{
        if (a == b)
                return 0;
        ll md = (l + r) >> 1;
        if (a <= md and b <= md)
                return DFS(l, md, a, b);
        if (md < a and md < b)
                return DFS(md + 1, r, a, b);
        return DFS(l, md, l, a) + DFS(md + 1, r, b, r) + r - l;
}
void Main(void)
{
        ll n, q;
        cin >> n >> q;
        ll tot = (1 << n);
        for (ll i = 1; i <= q; ++i)
        {
                ll a, b;
                cin >> a >> b;
                cout << DFS(1, tot, a, b) << '\n';
        }
        return;
}
```

G Gray与美了吗外卖

给一个有向图，不保证连通，可能存在重边，保证不存在自环。每个点有以下属性：颜色，本地生成费用。每个边有以下属性：通过收费站的基础费用，通过收费站需要的颜色转换费用。问给每个点一个蛋糕(本地生成或者其他点生成后运输过来)的最小费用。
额外加一个虚点，表示蛋糕生成的源头。虚点和图中每个点建个边（可以证明双向单向都正确），边权是在这个点生成蛋糕的费用。
再在图中修正边，如果一个边连接着 u 和 v ，那么，如果不能把蛋糕的颜色从 u 点颜色转成 v 颜色，u -> v 的有向边就需要删除(v -> u 同理)。如果能够转换，则边权在收费站基础费用的基础上，再加上颜色转换的费用。
之后以虚点为源点跑一次有向图的单源最短路，到每个点的最短路距离就是让这个点得到一个蛋糕的最小费用。所有点的最短路距离和就是答案。
注意，颜色之间转换时，可以把其他颜色当成中转站，所以颜色转换也需要跑一次多源最短路。
如果了解单源最短路的原理，也可以通过跑最短路前，先在队列里加入 N 个点来替代建虚点的操作。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
        std::ios::sync_with_stdio(false);
        cin.tie(nullptr);
        cout.tie(nullptr);
```

```
        Main();
        return 0;
}

#define VE std::vector
#define HEAP std::priority_queue
#define PLL std::pair<ll, ll>
#define fi first
#define se second
#define FOR(i, l, r) for (ll i = (l); i <= (r); ++i)
constexpr int inf = 0x3f3f3f3f;
constexpr ll INF = 0x3f3f3f3f3f3f3f3fLL;
template <ll V_SZ>
struct DIJ
{
public:
        static constexpr ll ERR = -1;
        ll v_sz;
        VE<PLL> E[V_SZ];
        ll dis[V_SZ];
        void Init(ll v_sz)
        {
                this->v_sz = v_sz;
                FOR(i, 0, v_sz)
                E[i].clear();
                return;
        }
        void Push(ll bg, ll ed, ll wt) { E[bg].push_back({ed, wt}); }
        void Build(ll s)
        {
                FOR(i, 0, v_sz)
                {
                        dis[i] = INF;
                        vis[i] = false;
                }
                dis[s] = 0;
                HEAP<PLL, VE<PLL>, std::greater<PLL>> he;
                he.push({0, s});
                while (not he.empty())
                {
                        ll now = he.top().se;
                        he.pop();
                        if (vis[now])
                                continue;
                        vis[now] = true;
                        for (auto &[aim, wt] : E[now])
                        {
                                if (vis[aim])
                                        continue;
                                if (dis[aim] > dis[now] + wt)
                                {
                                        dis[aim] = dis[now] + wt;
                                        he.push({dis[aim], aim});
```

```
                        }
                }
        }
        return;
    }
    ll Qry(ll p)
    {
        if (p < 1 or p > v_sz or dis[p] == INF)
            return ERR;
        return dis[p];
    }

protected:
    bool vis[V_SZ];
};
constexpr ll C_SZ = 1e2 + 5;
constexpr ll SZ = 1e5 + 5;
ll c, k;
ll f[C_SZ][C_SZ];
ll n;
ll col[SZ];
ll cost[SZ];
ll m;
DIJ<SZ> G;

void Main(void)
{
    cin >> c >> k;
    memset(f, inf, sizeof(f));
    for (ll i = 1; i <= k; ++i)
    {
        ll x, y, z;
        cin >> x >> y >> z;
        f[x][y] = std::min(f[x][y], z);
    }
    for (ll i = 1; i <= c; ++i)
        f[i][i] = 0;
    for (ll i2 = 1; i2 <= c; ++i2)
        for (ll i1 = 1; i1 <= c; ++i1)
            for (ll i3 = 1; i3 <= c; ++i3)
                f[i1][i3] = std::min(f[i1][i3], f[i1][i2] + f[i2][i3]);
    cin >> n;
    for (ll i = 1; i <= n; ++i)
        cin >> col[i];
    for (ll i = 1; i <= n; ++i)
        cin >> cost[i];
    G.Init(n + 1);
    cin >> m;
    for (ll i = 1; i <= m; ++i)
    {
        ll u, v, w;
        cin >> u >> v >> w;
        if (f[col[u]][col[v]] != INF)
```

```
                    G.Push(u, v, w + f[col[u]][col[v]]);
            if (f[col[v]][col[u]] != INF)
                    G.Push(v, u, w + f[col[v]][col[u]]);
    }
    for (ll i = 1; i <= n; ++i)
            G.Push(n + 1, i, cost[i]);
    G.Build(n + 1);
    ll ans = 0;
    for (ll i = 1; i <= n; ++i)
            ans += G.Qry(i);
    cout << ans << '\n';
    return;
}
```

H Gray与原元源矩阵

给一个类似菱形的范围，$O(1)$查询范围内的和。

首先观察发现，查询的范围是一个类似菱形的范围。对于任意一个与选中的 (X,Y) 点数字相同的点，以它为中心 选中同样的范围，得到的答案相同，所以可以先把 X 和 Y 先分别对 N 和 M 取模，让实际查询的范围尽可能靠近原点。每一个点上的数字是 $M\times x+y$ ，因此最终答案就是范围内的 $\sum M\times x+y=M\times\sum x+\sum y$ 。可以分开计算 $\sum x$ 和 $\sum y$ 。

以 $\sum y$ 为例，可以发现，每一列上的数字的 y 部分贡献是相同的，但每一列的数字出现的个数不同。先考虑 R 足够大，远大于 M 的情况。将需要计算的菱形范围分为左 $R+1$ 和右 R 两个三角形区域，分别计算贡献。以左 $R+1$ 为例，最左列的数字为 $Y-R(modM)$ ，只出现一次，我们称之为 a_1 。则可以发现，左 $R+1$ 每一列上的数字，从左到右，是对 M 在取模意义下以 a_1 为首项，公差为 1 的等差数列。但是答案计算并不是在取模意义下的，所以分开考虑 a_1 到 $M-1$ 的部分和 0 到 $a1-1$ 的部分(如果 $a_1=0$ ，则忽略这部分)。以 a_1 到 $M-1$ 的部分为例，先考虑最多端，完整的，长度为 $M-a_1$ 的连续几列。定义 $a_0=a_1-0$ ， $A_i=a_0+i$, $b_0=-1$ $Bi=b_0+2\times i$, $C_i=A_i\times B_i$ 。A_i 代表第 i 列上的数字，B_i 代表第 i 列的数字重复的次数， C_i 代表第 i 列的贡献。则第一个完整的部分的贡献是 $\sum_{i=1}^{M-a_1}C_i$ ，可以发现这部分是个二阶等差数列求和，可以 $O(1)$ 计算这一整个部分的贡献。如果不了解二阶等差数列求和的计算，也可以通过普通的运算律加上预处理来 $O(1)$ 处理这个部分:

$\sum_{i=1}^{M-a_1}C_i=\sum_{i=1}^{M-a_1}A_i\times B_i=\sum_{i=1}^{M-a_1}(a_0+i)\times(b_0+2\times i)$

$=\sum_{i=1}^{M-a_1}a_0\times b_0+(2\times a_0+b_0)\times i+2\times i^2$

$=(M-a_1)\times(a_0\times b_0)+(2\times a_0+b_0)\times\sum_{i=1}^{M-a_1}i+2\times\sum_{i=1}^{M-a_1}i^2$

其中 $\sum_{i=1}^{M-a_1}i$ 和 $\sum_{i=1}^{M-a_1}i^2$ 可以直接用公式计算，也可以提前预处理出来在之后的查询中使用。显然 $M-a_1$ 不会超过 10^3 复杂度足够，也不会爆long long范围。

这样完整的，长度为 $M-a_1$ 的部分会稳定出现 $\lfloor\frac{R+1}{M}\rfloor$个，每个都可以视作二阶等差数列求和来 $O(1)$ 计算。对于最后可能出现的不完整部分，其长度为 $min(R+1-\frac{R+1}{M}\times M,M-a_1)$ 。同样也是二阶等差数列求和，可 $O(1)$ 计算。当 R 较小，以至于没有完整部分时，直接 $O(1)$ 计算不完整部分即可。

0 到 $a1-1$ 部分同理，也是二阶等差数列，可以 $O(1)$ 计算。这样左 $R+1$ 的部分的贡献就可以在 $O(\frac{R+1}{M})$ 的复杂度内计算出来。但在最坏情况下，这样的总复杂度 $O(T\times(\frac{R}{M}+\frac{R}{N}))$ 会到达 10^9 ，复杂度还是不够，考虑进一步优化。

再次观察发现，对于完整的部分，其二阶等差数列只有 B_i 部分的首项 b_1 会发生变化。而因为中间距离相同，所以 $b1$ 的变化呈现为一阶等差数列，每隔一个区间， b_1 会增加 M 。再观察二阶等差数列 C_i ，可以把除了 $b1$ 以外的都看是确定的常量。定义 $k=(M-a_1)\times a_0+\sum_{i=1}^{M-a_1}i$ ， $c=2\times a_0\times\sum_{i=1}^{M-a_1}i+2\times\sum_{i=1}^{M-a_1}i^2$ ，则 $\sum_{i=1}^{M-a_1}C_i=k\times b_0+c$ 。所以 y 部分的左 $R+1$ 部分的完整部分的 a_1 到 $M-1$ 部分的贡献就是 $\sum_{i=1}^{\lfloor\frac{R+1}{M}\rfloor}k\times D_i+c$，其中 D_i 表示的是以 b_0 为首项， M 为公差的等差数列的第 i 项。显然这是个一阶等差数

列求和，可以直接 $O(1)$ 计算，既可以用公式直接计算，也可以像前面一样用运算律化出 $\sum i$ 和 $\sum i^2$ ，然后用预处理的结果代入 $O(1)$ 完成等差数列求和的计算。对于不完整的部分，最多只计算一次二阶等差数列求和。
其余 0 到 $a1-1$ 的部分同理，右 R 部分同理， x 部分同理，最终就可以 $O(1)$ 得到所有的贡献。
很多部分的计算过程是非常接近的，可以用函数实现，从而减少码量。
综上，即使完全不了解等差数列，也可以利用预处理和基础的运算律 $O(1)$ 实现每次查询。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
        std::ios::sync_with_stdio(false);
        cin.tie(nullptr);
        cout.tie(nullptr);
        Main();
        return 0;
}

// x对mod取模
ll Mod(ll x, ll mod) { return (x % mod + mod) % mod; }
// 首项为a1，公差为d的等差数列第i项的值
ll GetA(ll a1, ll d, ll i) { return a1 + (i - 1) * d; }
// 首项为a1，公差为d的等差数列前n项和
ll GetS1(ll a1, ll d, ll n)
{
        ll res = 0;
        res += n * a1;
        res += n * (n - 1) / 2 * d;
        return res;
}
// 首项为a1，公差为1的等差数列 乘上 首项为b1，公差为d的等差数列 得到的新的二阶等差数列 的前n项和
ll GetS2(ll a1, ll b1, ll d, ll n)
{
        ll res = 0;
        res += n * (n + 1) * (2 * n + 1) / 6 * d;
        res += n * (n + 1) / 2 * (b1 + a1 * d - 2 * d);
        res += n * (a1 * b1 + d - b1 - a1 * d);
        return res;
}
// 二阶等差数列求和公式转一阶等差求和数列时的系数计算
void GetKC(ll a1, ll d, ll n, ll &k, ll &c)
{
        k = n * (n + 1) / 2 + n * a1 - n;
        c = 0;
        c += n * (n + 1) * (2 * n + 1) / 6 * d;
        c += n * (n + 1) / 2 * (a1 * d - 2 * d);
        c += n * d - n * a1 * d;
        return;
}
// 以s为中心点，p为当前维度模数，共有cnt个此模式构成的整体，在第一个此模式的贡献次数的等差数列的首项是b1，
```

```
公差是d，模式长度为len的整体部分贡献
ll GetBLK(ll s, ll p, ll cnt, ll b1, ll d, ll len)
{
        ll k, c;
        GetKC(s, d, len, k, c);
        ll a1 = k * b1 + c;
        ll step = k * d * p;
        return GetS1(a1, step, cnt);
}
// 以s点为中心点，p为当前维度模数，询问范围为r的区域，此维度的贡献
ll Qry(ll s, ll p, ll r)
{
        ll res = 0;
        ll a1, b1, cnt, len1, len2, tot, n;

        // 前r+1部分的整块部分的贡献
        a1 = Mod(s - r, p);
        cnt = (r + 1) / p;

        // 模式1 整块部分的贡献
        len1 = p - a1;
        b1 = GetA(1, 2, 1);
        res += GetBLK(a1, p, cnt, b1, 2, len1);

        // 模式2 整块部分的贡献
        len2 = p - len1;
        b1 = GetA(1, 2, len1 + 1);
        res += GetBLK(0, p, cnt, b1, 2, len2);

        // 前r+1部分的多余部分的贡献
        tot = cnt * p;
        n = std::min(p - a1, r + 1 - tot);

        // 模式1 多余部分的贡献
        b1 = GetA(1, 2, tot + 1);
        res += GetS2(a1, b1, 2, n);

        // 模式2 多余部分的贡献
        b1 = GetA(1, 2, tot + n + 1);
        res += GetS2(0, b1, 2, r + 1 - (tot + n));

        // 后r部分的整块部分的贡献
        a1 = Mod(s + 1, p);
        cnt = r / p;

        // 模式3 整块部分的贡献
        len1 = p - a1;
        b1 = GetA(2 * r - 1, -2, 1);
        res += GetBLK(a1, p, cnt, b1, -2, len1);

        // 模式4 整块部分的贡献
        len2 = p - len1;
        b1 = GetA(2 * r - 1, -2, len1 + 1);
```

```
        res += GetBLK(0, p, cnt, b1, -2, len2);

        // 后r部分的多余部分的贡献
        tot = cnt * p;
        n = std::min(p - a1, r - tot);

        // 模式3 多余部分的贡献
        b1 = GetA(2 * r - 1, -2, tot + 1);
        res += GetS2(a1, b1, -2, n);

        // 模式4 多余部分的贡献
        b1 = GetA(2 * r - 1, -2, tot + n + 1);
        res += GetS2(0, b1, -2, r - (tot + n));

        return res;
}
void Run(void)
{
        ll n, m, x, y, r;
        cin >> n >> m >> x >> y >> r;
        x = Mod(x, n);
        y = Mod(y, m);
        ll ans = m * Qry(x, n, r) + Qry(y, m, r);
        cout << ans << '\n';
        return;
}
void Main(void)
{
        ll t;
        cin >> t;
        while (t--)
                Run();
        return;
}
```

I Gevin的RGB区间和

二维for一下计算出R，B的区间和，用总区间和减去R，B的区间和即可得到G区间和。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
        std::ios::sync_with_stdio(false);
        cin.tie(nullptr);
        cout.tie(nullptr);
        Main();
        return 0;
```

```
}

constexpr ll SZ = 1e3 + 3;
ll n;
ll a[SZ][SZ];

void Main(void)
{
        cin >> n;
        for (ll i = 1; i <= n; ++i)
                for (ll j = 1; j <= n; ++j)
                        cin >> a[i][j];
        ll sum = 0;
        for (ll i = 1; i <= n; ++i)
                for (ll j = 1; j <= n; ++j)
                        sum += a[i][j];
        ll r = 0, g = 0, b = 0;
        for (ll i = 1; i <= n / 2; ++i)
                for (ll j = 1; j <= n / 2 + 1 - i; ++j)
                {
                        r += a[i][j];
                        b += a[n + 1 - i][n + 1 - j];
                }
        g = sum - r - b;
        cout << r << ' ' << g << ' ' << b << '\n';
        return;
}
```

J Gevin的打印服饰

每个长度为 n 的线单独打印，可以重叠。一共12条线。都是水平/竖直/45度。

```
#include <bits/stdc++.h>
using std::cin;
using std::cout;
using ll = long long;
void Main(void);
int main(void)
{
        std::ios::sync_with_stdio(false);
        cin.tie(nullptr);
        cout.tie(nullptr);
        Main();
        return 0;
}

constexpr ll SZ = 1e3 + 3;
ll n;
char a[SZ][SZ];

void Main(void)
```

```
{
        cin >> n;
        for (ll i = 1; i <= n * 2 - 1; ++i)
                for (ll j = 1; j <= n * 3 - 2; ++j)
                        a[i][j] = ' ';
        for (ll i = 1; i <= n; ++i)
        {
                a[1][n + i - 1] = '*';
                a[i][n + i - 1] = '*';
                a[n + 1 - i][n + i - 1] = '*';
                a[n + 1 - i][i] = '*';
                a[i][n * 2 - 2 + i] = '*';
                a[n][i] = '*';
                a[n][n + n - 2 + i] = '*';
                a[n + 1][n - 1 + i] = '*';
                a[n * 2 - 1][n - 1 + i] = '*';
                a[n - 1 + i][n] = '*';
                a[n - 1 + i][n * 2 - 1] = '*';
                a[n * 2 - i][n - 1 + i] = '*';
        }
        for (ll i = 1; i <= n * 2 - 1; ++i)
        {
                for (ll j = 1; j <= n * 3 - 2; ++j)
                        cout << a[i][j];
                cout << '\n';
        }
        return;
}
```