----------Shimu_Guyue----------

实用技巧

DEV-C++编译指令

```
【工具】->【编译选项】->【√编译时加入以下命令】
-std=c++11
```

快读快写

关闭缓冲区

```
// 关闭输入输出缓存，使效率提升
std::ios::sync_with_stdio(false);
// 解除cin和cout的默认绑定，来降低IO的负担使效率提升
std::cin.tie(NULL); std::cout.tie(NULL);
```

O2 优化

```
#pragma GCC optimize(2)
```

模板函数

```
/* getchar() 的速度快于关闭缓冲区的 std::cin.get()*/
inline void Read(int &x)
{
    x = 0;
    bool flag(0);
    char c(getchar());
    while (!isdigit(c))
    {
        flag = c == '-';
        c = getchar();
    }
    while (isdigit(c))
    {
        x = (x << 1) + (x << 3) + (c ^ '0'); // x * 10 + c - '0'
        c = getchar();
    }
    flag ? x = -x : 0;
}

inline void Write(int x)
{
    if (x < 0)
    {
        putchar('-');
        x=-x;
    }
```

```
    if (x > 9)
        Write(x / 10);
    putchar(x % 10 + '0');
}
```

堆

C++ 库 priority_queue<>

```
// 大根堆
struct less
{
    _GLIBCXX14_CONSTEXPR // #define _GLIBCXX14_CONSTEXPR constexpr
    bool
    operator()(const _Tp &__x, const _Tp &__y) const
    {
        return __x < __y;
    }
};

std::priority_queue<Type, std::vector<Type>, Less<Type>> less_q;
```

数论

判断质数

```
bool Is_prime(int x)
{
    if (x < 2)
        return false;
    for (int i(2); i * i <= x; ++i)
    {
        if (x % i == 0)
            return false;
    }
    return true;
}
```

最大公因数和最小公倍数

```
/*最大公因数*/
int Gcd(int a, int b)
{
    if (a % b == 0)
        return b;
    return Gcd(b, a % b);
}

/*最小公倍数*/
int Lcm(int a, int b)
{
    return a / Gcd(a, b) * b;
}
```

快速幂

```
const int mod;

int Q_pow(int base, int n)
{
    int ans(1);
    while (n)
    {
        if (n & 1) // n % 2 == 1
            ans = ans * base % mod;
        n >>= 1; // n /= 2 ;
        base = base * base % mod;
    }
    return ans;
}
```

排列组合

组合数

```
/*不需要取模时*/
int C(int n, int m)
{
    int ans(1);
    for(int i(1);i <= m; ++i)
    {
        // 注意一定要先乘再除
        ans *= n - m + i;
        ans /= i;
    }
    return ans;
}

/*需要取模时*/
// 需要 int Q_pow(int base, int n, int mod)

const int mod;

int C(int n, int m)
{
    if (n < m)
        return 0;
    int ans(1);
    for (int i(1), j(n); i <= m; ++i, --j)
    {
        ans = ans * j % mod;
        ans = ans * Q_pow(i, mod - 2) % mod;
    }
    return ans;
}
```

Lucas 定理

```
// 需要 int C(int n, int m, int mod)

const int mod;

int Lucas(int n, int m)
{
    if (m == 0)
        return 1;
    return Lucas(n / mod, m / mod) * C(n % mod, m % mod) % mod;
}
```

全排列

```
#include <vector>
#include <algorithm>

std::vector<int> vi;

// 下一个字典序排列
std::next_permutation(vi.begin(), vi.end()); -> bool
// 上一个字典序排列
std::prev_permutation(vi.begin(), vi.end()); -> bool
```

Catalan 数列

```
/*
模型：求 (0, 0) 走到 (n, n) 的路径总数
    其中每次移动可以 x + 1 或 y + 1，但限制 y <= x
*/

int Catalan(int n)
{
    return C(2 * n, n) / (n + 1);
//  return C(2 * n, n) - C(2 * n, n - 1);
//  return Catalan(n - 1) * (4 * n - 2) / (n + 1);
}
```

高精度运算 （逆序） （非负数）

高精度 + 高精度

```
// 以字符串形式输入
std::string a_s;
std::string b_s;
std::cin >> a_s;
std::cin >> b_s;
// 转换成数组形式倒序存储
std::vector<int> a;
std::vector<int> b;
for (int i(1); i <= a_s.size(); ++i)
{
    a.push_back(a_s[a_s.size() - i] - '0');
}
for (int i(1); i <= b_s.size(); ++i)
```

```
{
    b.push_back(b_s[b_s.size() - i] - '0');
}
// 高精度相加
std::vector<int> ans(a + b);
// 倒序输出
for (int i(1); i <= ans.size(); ++i)
{
    std::cout << ans[ans.size() - i];
}
std::cout << std::endl;

/*-------------------------------------------------*/
// 重载 + 运算符
std::vector<int> operator+(std::vector<int>& a, std::vector<int>& b)
{
    // 预设答案数组大小
    int n(std::max(a.size(), b.size()));
    std::vector<int> ans(n);
    a.resize(n);
    b.resize(n);
    // 逐位相加
    for (int i(0); i < n; ++i)
    {
        ans[i] = a[i] + b[i];
    }
    // 逐位进位
    for (int i(0); i < n; ++i)
    {
        if (ans[i] >= 10)
        {
            if (i != n - 1)
                ++ans[i + 1];
            else
                ans.push_back(1);
            ans[i] %= 10;
        }
    }
    return ans;
}
```

高精度 * 低精度

```
// 分别以字符串形式和整数形式输入
std::string a_s;
std::cin >> a_s;
int b;
std::cin >> b;
// 转换成数组形式倒序存储
std::vector<int> a;
for (int i(1); i <= a_s.size(); ++i)
{
    a.push_back(a_s[a_s.size() - i] - '0');
}
// 高精度相乘
std::vector<int> ans(a * b);
// 倒序输出
```

```cpp
for (int i(1); i <= ans.size(); ++i)
{
    std::cout << ans[ans.size() - i];
}
std::cout << std::endl;

/*------------------------------------------------*/
// 重载 * 运算符
std::vector<int> operator*(std::vector<int>& a, int b)
{
    // 预设答案数组大小
    std::vector<int> ans(a.size());
    // 逐位相乘
    for (int i(0); i < a.size(); ++i)
    {
        ans[i] += a[i] * b;
    }
    // 逐位相加
    for (int i(0); i < ans.size(); ++i)
    {
        if (ans[i] >= 10)
        {
            if (i != ans.size() - 1)
                ans[i + 1] += ans[i] / 10;
            else
                ans.push_back(ans[i] / 10);
            ans[i] %= 10;
        }
    }
    return ans;
}
```

高精度 * 高精度

```cpp
// 以字符串形式输入
std::string a_s;
std::string b_s;
std::cin >> a_s;
std::cin >> b_s;
// 转换成数组形式倒序存储
std::vector<int> a;
std::vector<int> b;
for (int i(1); i <= a_s.size(); ++i)
{
    a.push_back(a_s[a_s.size() - i] - '0');
}
for (int i(1); i <= b_s.size(); ++i)
{
    b.push_back(b_s[b_s.size() - i] - '0');
}
// 高精度相乘
std::vector<int> ans(a * b);
// 倒序输出
for (int i(1); i <= ans.size(); ++i)
{
    std::cout << ans[ans.size() - i];
}
```

```
std::cout << std::endl;

/*-------------------------------------------------*/
// 重载 * 运算符
std::vector<int> operator*(std::vector<int> &a, std::vector<int> &b)
{
    // 预设答案数组大小
    std::vector<int> ans(a.size() + b.size());
    // 错位相乘
    for (int i(0); i < a.size(); ++i)
    {
        for (int j(0); j < b.size(); ++j)
        {
            ans[i + j] += a[i] * b[j];
        }
    }
    // 逐位相加
    for (int i(0); i < ans.size(); ++i)
    {
        if (ans[i] >= 10)
        {
            if (i != ans.size() - 1)
                ans[i + 1] += ans[i] / 10;
            else
                ans.push_back(ans[i] / 10);
            ans[i] %= 10;
        }
    }
    // 去除前导0
    while (ans.back() == 0 && ans.size() > 1)
    {
        ans.pop_back();
    }
    return ans;
}
```

高精度 / 低精度

```
// 分别以字符串形式和整数形式输入
std::string a_s;
std::cin >> a_s;
int b;
std::cin >> b;
// 转换成数组形式倒序存储
std::vector<int> a;
for (int i(1); i <= a_s.size(); ++i)
{
    a.push_back(a_s[a_s.size() - i] - '0');
}
// 高精度相除
std::vector<int> ans(a / b);
// 倒序输出
for (int i(1); i <= ans.size(); ++i)
{
    std::cout << ans[ans.size() - i];
}
std::cout << std::endl;
```

```
/*--------------------------------------------------*/
// 重载 / 运算符
std::vector<int> operator/(std::vector<int>& a, int b)
{
    // 答案数组无需预设大小
    std::vector<int> ans;
    // 逐位相除
    bool ok(false); // 答案不包含前导 0
    int temp(0);
    for (int i(a.size() - 1); i >= 0; --i)
    {
        temp = temp * 10 + a[i];
        if (temp >= b)
            ok = true;
        if (ok)
        {
            ans.push_back(temp / b);
            temp %= b;
        }
    }
    if (ans.empty())
        ans.push_back(0);
    return std::vector<int>(ans.rbegin(), ans.rend());
}
```

高精度比较大小

```
// 重载 > 运算符
bool operator>(std::vector<int>& a, std::vector<int>& b)
{
    if (a.size() != b.size())
        return a.size() > b.size();
    int n(a.size());
    for (int i(n - 1); i >= 0; --i)
    {
        if (a[i] > b[i])
            return true;
        else if (a[i] < b[i])
            return false;
    }
    return false;
}
```

搜索

二分搜索（升序）

[l, mid - 1] + [mid, r]

```cpp
// 向右搜索
int Binary_Search(int l, int r, int k)
{
    while (l < r)
    {
        int mid((l + r + 1) / 2);
        if (Check(mid, k))
            l = mid;
        else
            r = mid - 1;
    }
    return l;
}
```

[l, mid] + [mid + 1, r]

```cpp
// 向左搜索
int Binary_Search(int l, int r, int k)
{
    while (l < r)
    {
        int mid((l + r) / 2);
        if (Check(mid, k))
            r = mid;
        else
            l = mid + 1;
    }
    return r;
}
```

C++ 库 upper_bound() 和 lower_bound()

```cpp
#include <algorithm>

/* 升序 */
// 第一个大于 cmp_ele 的元素的位置
std::upper_bound(v.begin(), v.end(), cmp_ele) // -> iterator
// 第一个大于等于 cmp_ele 的元素的位置
std::lower_bound(v.begin(), v.end(), cmp_ele) // -> iterator

/* 有序 */
// 第一个符合 cmp 比较规则的元素 pos_ele 的位置
bool cmp(cmp_ele, pos_ele);
std::upper_bound(v.begin(), v.end(), cmp_ele, cmp) // -> iterator
```

三分

```cpp
/* 凸函数的极大值 */
int Ternary_Search(int l, int r)
{
    while (l < r)
    {
        int l1((l * 2 + r * 1) / 3);
        int r1((l * 1 + r * 2) / 3);
        if (F(l1) > F(r1))
```

```
            r = r1 - 1;
        else if (F(l1 < F(r1)))
            l = l1 + 1;
        else
            l = l1 + 1, r = r1 - 1;
    }
    return std::max(F(l), F(r));
}

/* 凹函数的极小值 */
int Ternary_Search(int l, int r)
{
    while (l < r)
    {
        int l1((l * 2 + r * 1) / 3);
        int r1((l * 1 + r * 2) / 3);
        if (F(l1) < F(r1))
            r = r1 - 1;
        else if (F(l1 > F(r1)))
            l = l1 + 1;
        else
            l = l1 + 1, r = r1 - 1;
    }
    return std::min(F(l), F(r));
}
```

动态规划 DP

01背包

```
int Dp01(int n, int v, std::vector<int>& costs, std::vector<int>& values)
{
    // n 个物品，v 个空间
    std::vector<std::vector<int>> dp(n + 1, std::vector<int>(v + 1));
    for (int i(1); i <= n; ++i)
    {
        for (int j(1); j <= v; ++j)
        {
            if (j >= costs[i])
                dp[i][j] = std::max(dp[i - 1][j], dp[i - 1][j - costs[i]] +
values[i]);
            else
                dp[i][j] = dp[i - 1][j];
        }
    }
    return dp[n][v];
}

// 空间优化到一维
int Dp01(int n, int v, std::vector<int>& costs, std::vector<int>& values)
{
    std::vector<int> dp(v + 1);
    for (int i(1); i <= n; ++i)
    {
        for (int j(v); j >= costs[i]; --j)
        {
```

```
            dp[j] = std::max(dp[j], dp[j - costs[i]] + values[i]);
        }
    }
    return dp[v];
}
```

矩阵

矩阵旋转

```
/*顺时针旋转*/
for (int i(0); i < n; ++i)
{
    for (int j(0); j < n; ++j)
    {
        newGRid[j][n - 1 - i] = oldGrid[i][j];
    }
}

/*逆时针旋转*/
for (int i(0); i < n; ++i)
{
    for (int j(0); j < n; ++j)
    {
        newGRid[n - 1 - j][i] = oldGrid[i][j];
    }
}
```

求连通块数量

```
/*求 (l1, l1) ~ (l2, r2) 有多少连通块*/
/* ' ' 表示连通，'x'表示不连通*/

struct point
{
    int x;
    int y;
};

void Dfs(std::vector<std::vector<char>>& grid, int x, int y, int l1, int l2, int
r1, int r2, std::vector<Point>& points, bool& add)
{
    for (Point point : points)
    {
        if (point.x == x && point.y == y)
            return;
    }

    add = true;
    points.push_back({ x, y });

    if (x - 1 >= l1 && grid[x - 1][y] == ' ')
        Dfs(grid, x - 1, y, l1, l2, r1, r2, points, add);
    if (x + 1 <= r1 && grid[x + 1][y] == ' ')
        Dfs(grid, x + 1, y, l1, l2, r1, r2, points, add);
```

```
    if (y - 1 >= l2 && grid[x][y - 1] == ' ')
        Dfs(grid, x, y - 1, l1, l2, r1, r2, points, add);
    if (y + 1 <= r2 && grid[x][y + 1] == ' ')
        Dfs(grid, x, y + 1, l1, l2, r1, r2, points, add);
}

int Conut_ConnectedBlock(std::vector<std::vector<char>>& grid, int l1, int l2,
int r1, int r2)
{
    int ans(0);
    std::vector<Point> points;
    for (int i(l1); i<= r1; ++i)
    {
        for (int j(l2); j <= r2; ++j)
        {
            if (grid[i][j] == 'x')
                continue;

            bool add(false);
            Dfs(grid, i, j, l1, l2, points, add);
            if (add)
                ++ans;
        }
    }
    return ans;
}
```

图论

最短路

Floyd 算法

```
void Floyd(std::vector<std::vector<int>>& grid)
{
    int n(grid.size());
    for (int k(0); k < n; ++k)
    {
        for (int i(0); i < n; ++i)
        {
            for (int j(0); j < n; ++j)
            {
                a[i][j] = std::min(a[i][j], a[i][k] + a[k][j]);
            }
        }
    }
}
```