**lmm**

**算法竞赛**

**自用模板**

目录（Index）

1. **二分**查找（binary search）
2. **Fenwick**树（树状数组 / 二叉索引树）
3. 主席树（可持久化权值线段树）维护区间中值（不带修）
4. \_\_builtin\_popcount() 的完备手动实现
5. lowbit(x) = (x) & (-x) 【仅此】
6. bitmask（位运算 / 位掩码）若干常用性质
7. 自定义cmp及其lambda表达式写法（及iota/stable\_sort）
8. std::ceil的完备手动实现（ 速查：(a+b-1)/b == ceil(a/b) ）
9. 998244353 ：是一个大素数，通常用作模数【仅此】
10. stdcode.cpp ------ 自用标准模板
11. 快速判定一个整数是不是2的幂次 / 3的幂次
12. 更相减损结论的多元形式
13. \_\_int128 I/O模板（待补）
14. 数位和的若干结论（待补）
15. **I.计算组合数的板子**  
    **II.逆元快速幂**
16. 拓扑排序
17. 并查集
18. 快读（fread待补）
19. 高精度（待补）
20. **DFS**和**BFS**的基本形式
21. 计算几何基础：向量积判定凸性、共线判定

22.分解素因子

23.最大子段和（最大子数组和）

24.向负无穷取整 与 欧几里得取模

25.循环分解定理

26.**Dijkstra**算法（非负权图单源最短路）

27.**Nim**博弈

28.**函数图**（亦称**后继图**或**基环内向森林**）

29.筛法（待补）

30.加权随机抽样导出的一个概率恒等式

1. 二分查找

模板 1：查找满足条件的最小值

while (l < r) {

    ll mid = l + (r - l) / 2;

    if (check(mid)) {

        r = mid; // 尝试更小的值

    } else {

        l = mid + 1; // 尝试更大的值

    }

}

最终结果是   r  （或者l ------ 注意到结束时 l == r）

模板 2：查找满足条件的最大值

while (l < r) {

    ll mid = l + (r - l + 1) / 2; // 注意这里的 +1

    if (check(mid)) {

        l = mid; // 尝试更大的值

    } else {

        r = mid - 1; // 尝试更小的值

    }

}

最终结果是l或r（同理）

搜索范围是l~r，闭区间

***寻小下取整，右定左自增***

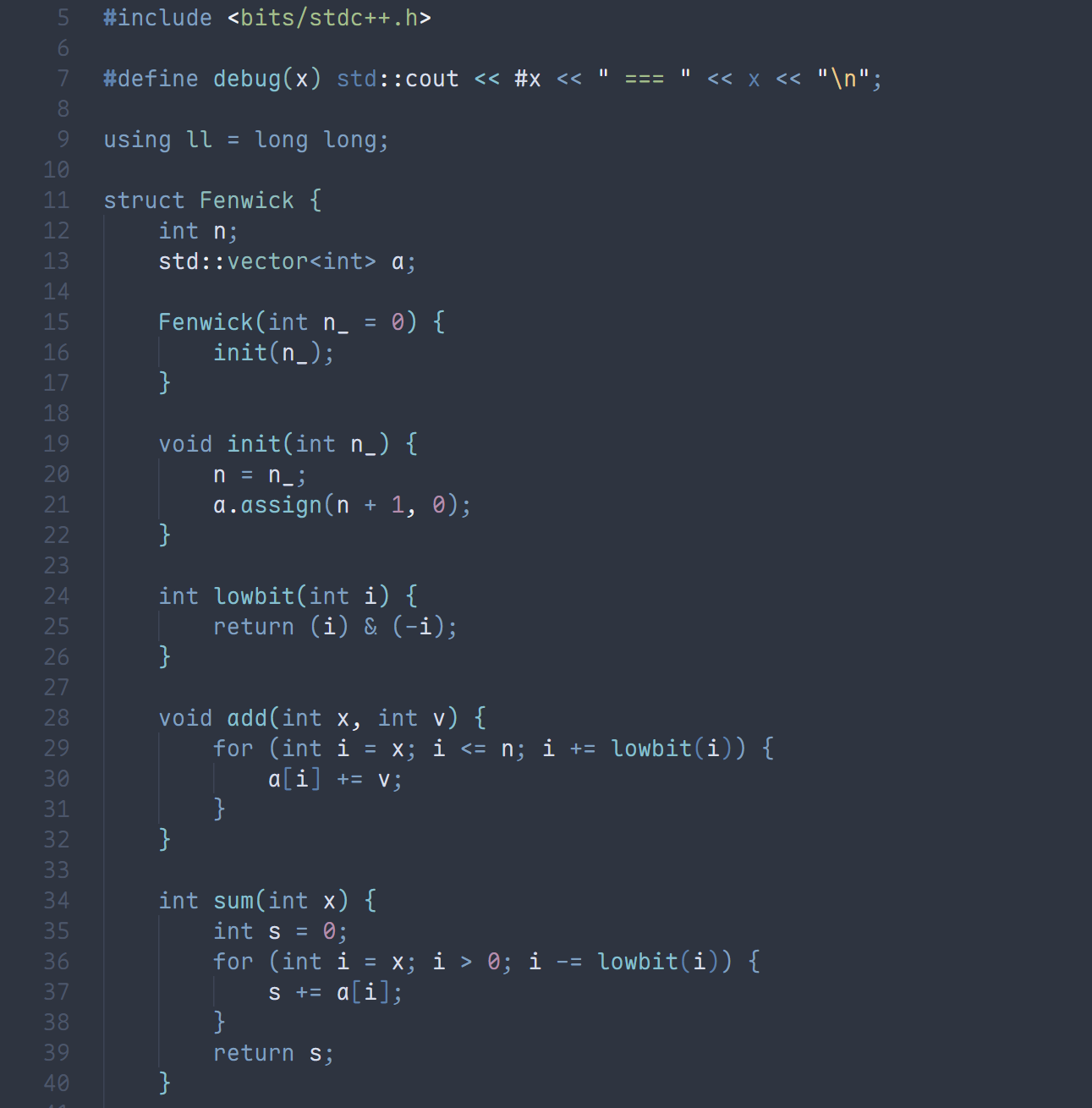
***寻大上取整，左定右自减***

1. Fenwick（树状数组）

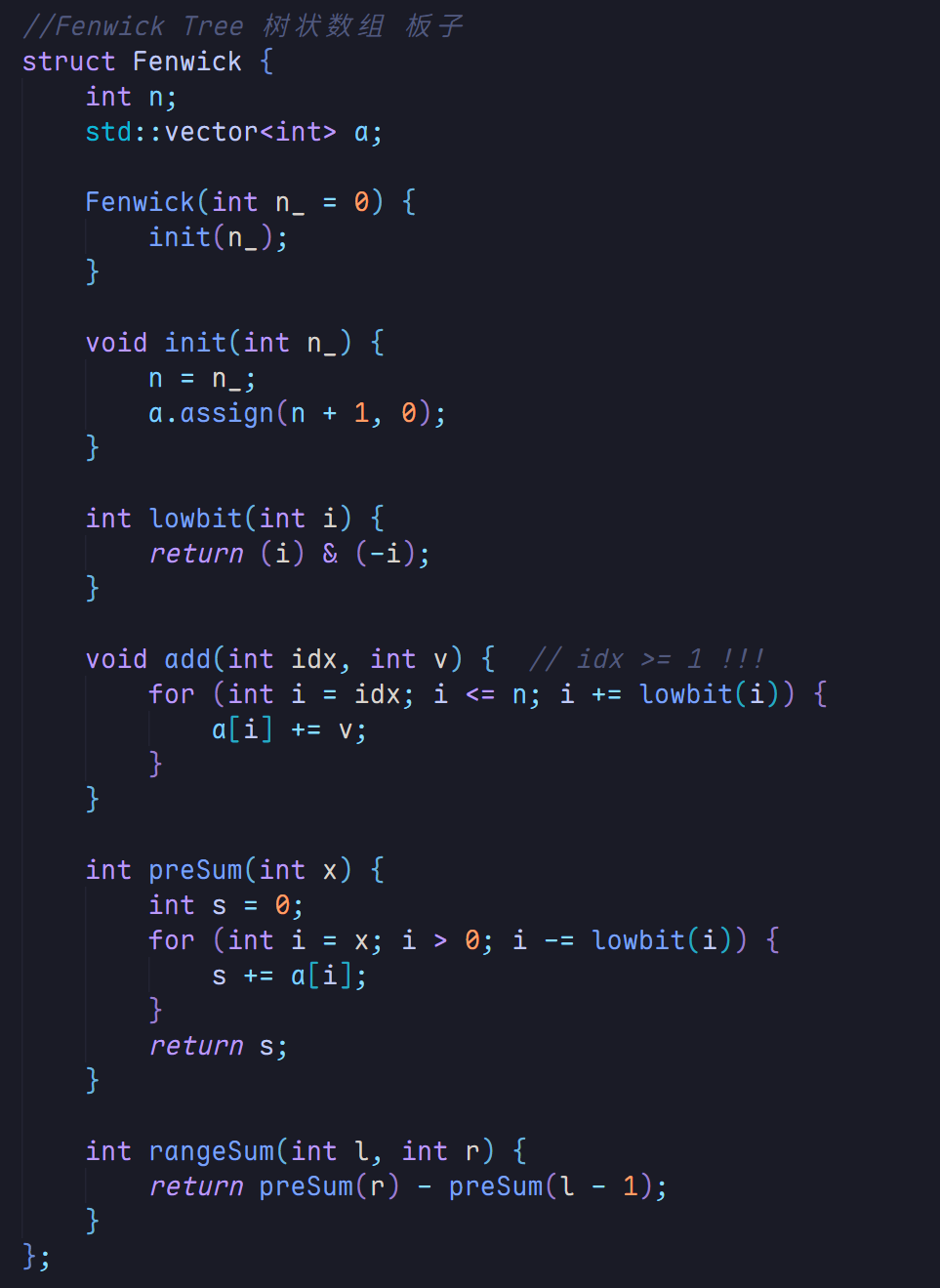
//清晰起见，我们将这个板子置为一基的，左闭右闭的

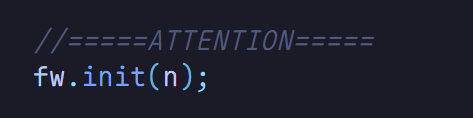
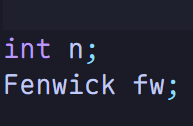
//（而不是零基的，左闭右开的）

//同时，我们不采用模板（T）写法



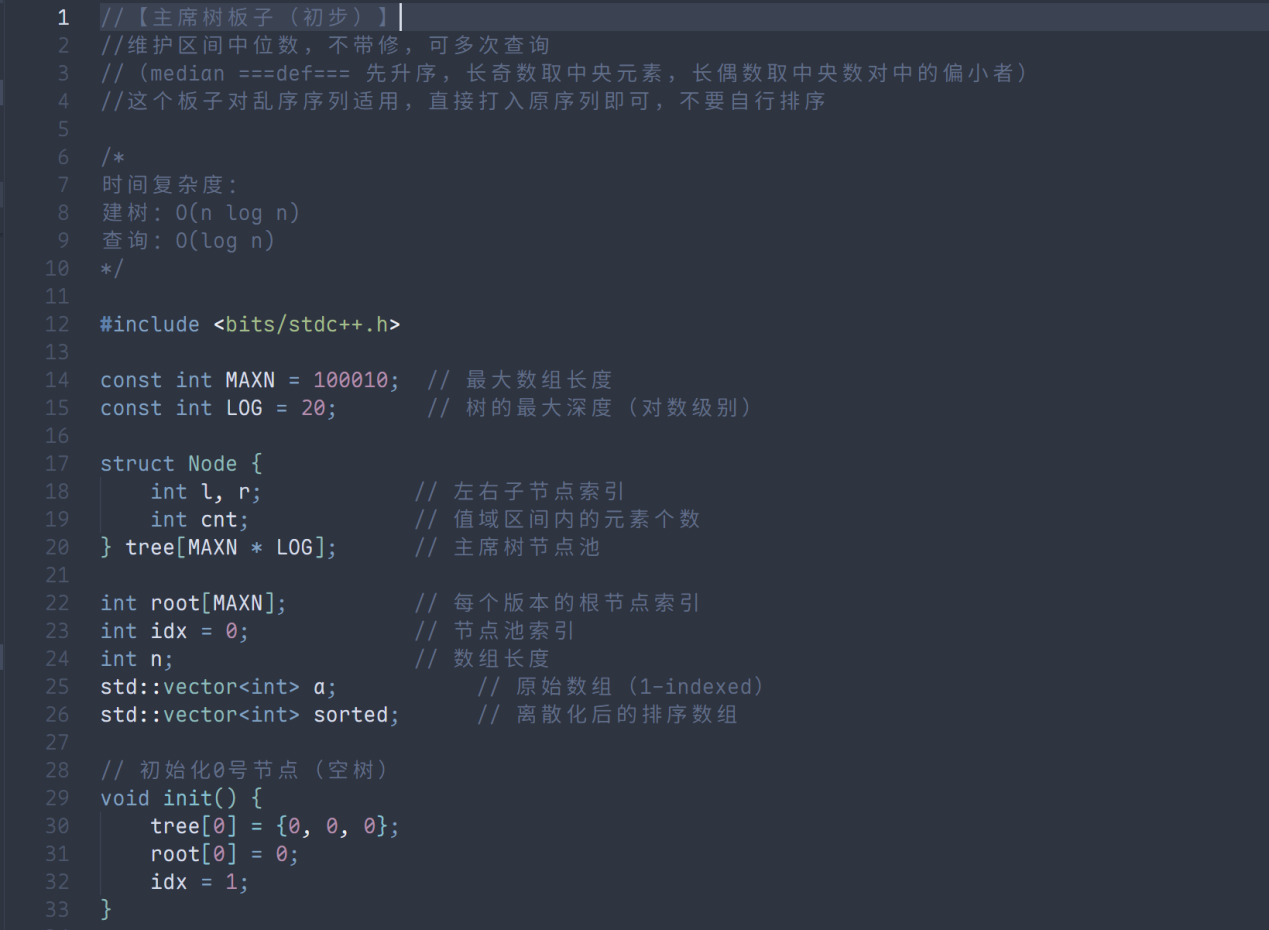


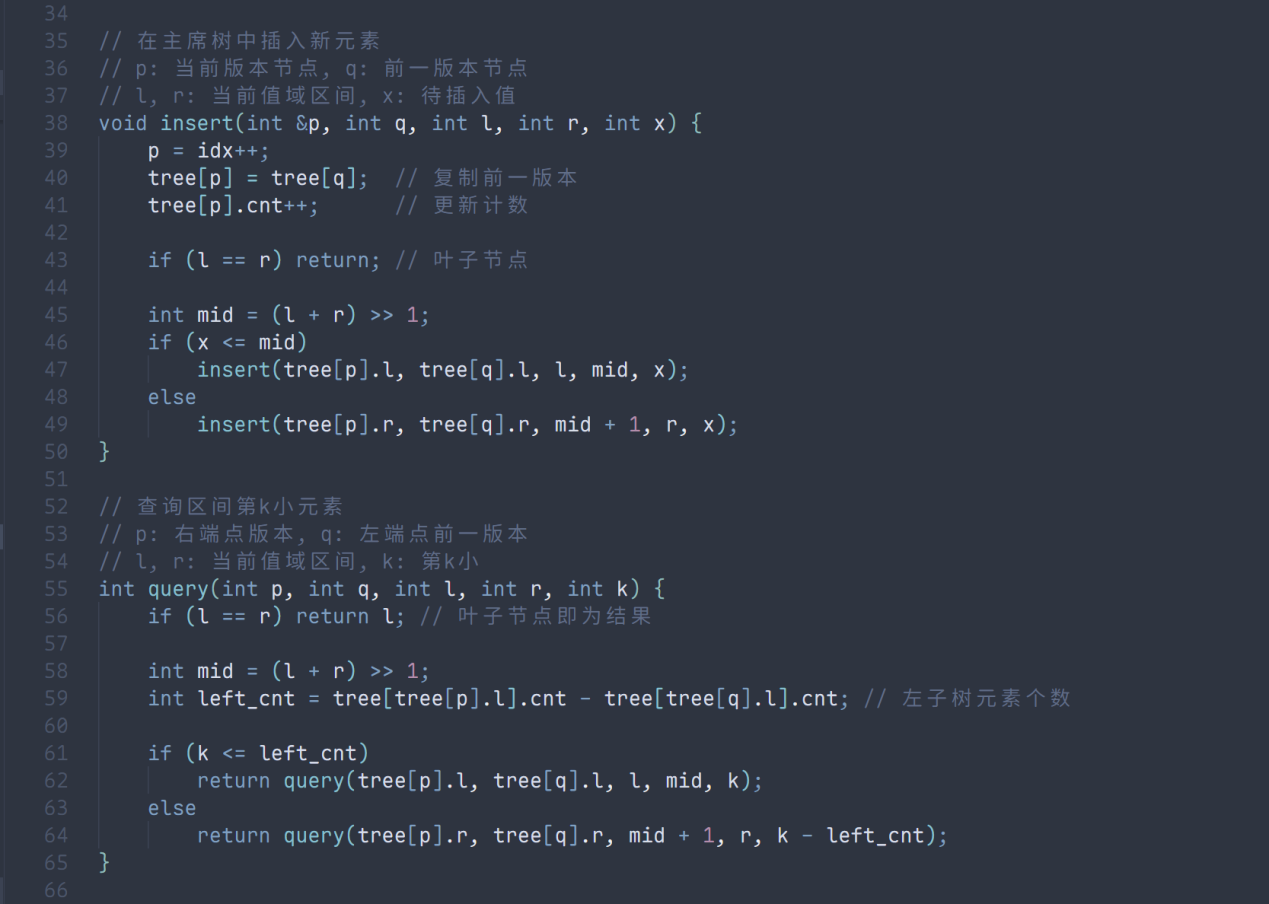




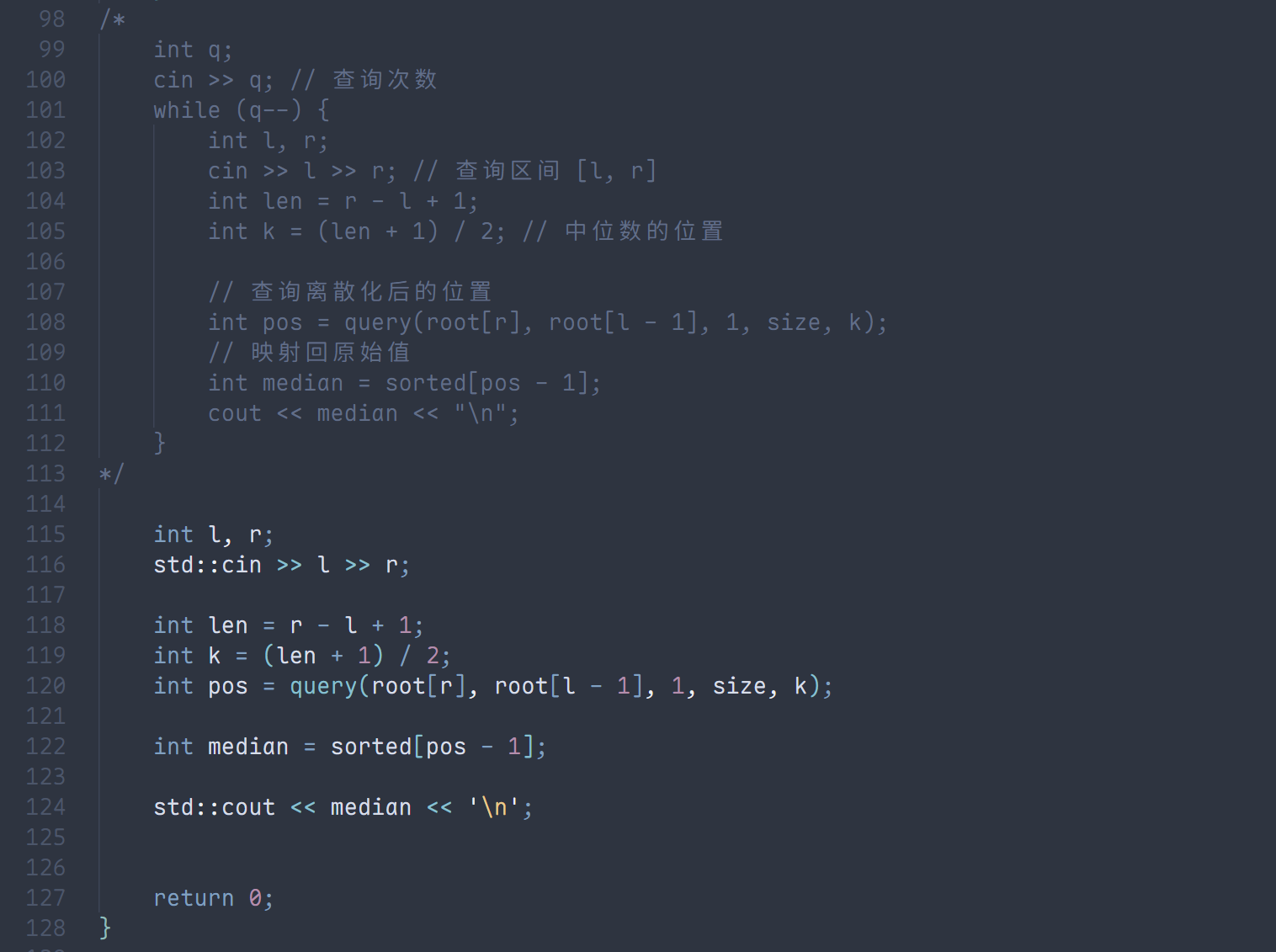
\*Fenwick上二分

1. 主席树维护区间中值









4.popcount手动实现（统计正整数二进制表达中**1**的个数）

        ll cnt1 = 0LL;

        while (x) {

            if (x & 1) {

                cnt1++;

            }

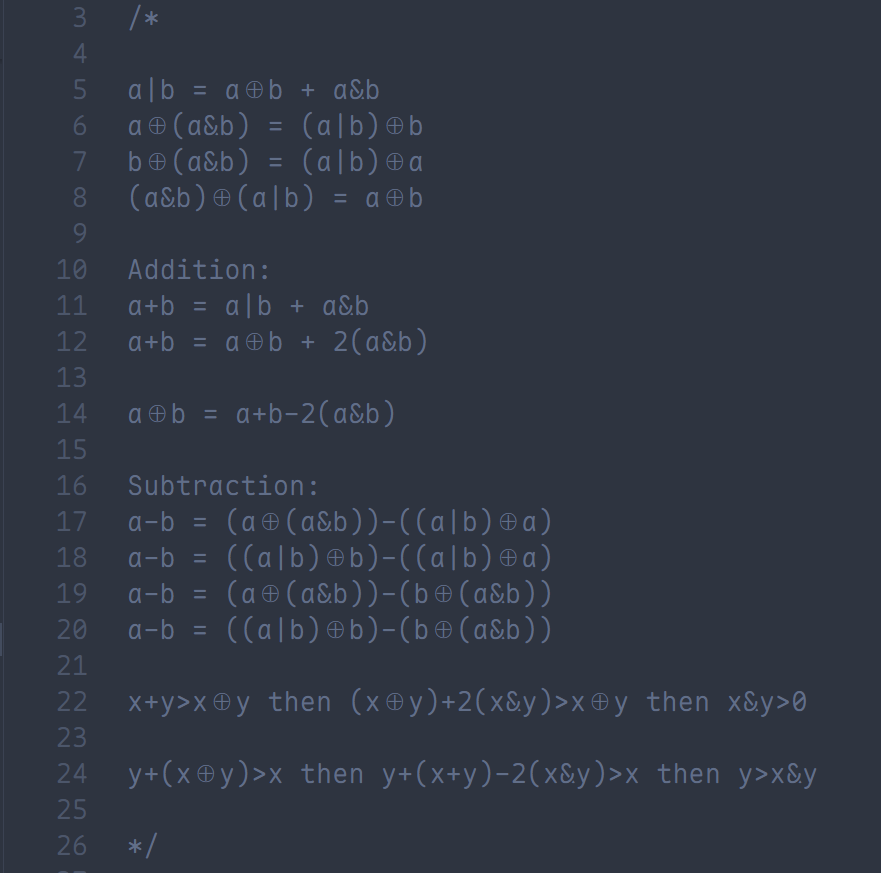
            x >>= 1; // or /=2

        }

//注意不是x >> 1;

5.lowbit，见目录

6.bitmask（位运算 / 位掩码）常用性质



另外：

奇数异或1 相当于减1，因为二进制末尾为1

偶数异或1相当于加1，因为二进制末尾为0

另外，关于异或 / ⊕ / ^ / XOR，一般地：

异或：异才为1

亦即 1^1 = 0^0 = 0

1^0 = 0^1 = 1

交换律：a⊕b = b⊕a

结合律：（a^b）^c = a^(b^c)

自反性：a ⊕ b = 0

恒等性：a ⊕ 0 = a

\*消去律：a ^ b = a ^ c -> b = c

这个结论对任何整数（或其他支持异或运算的数据类型，如布尔值、字节等）都成立

a⊕b⊕b = a

{ 0~31（位）能覆盖int，

这是bitmask方面的一个重要引理 }

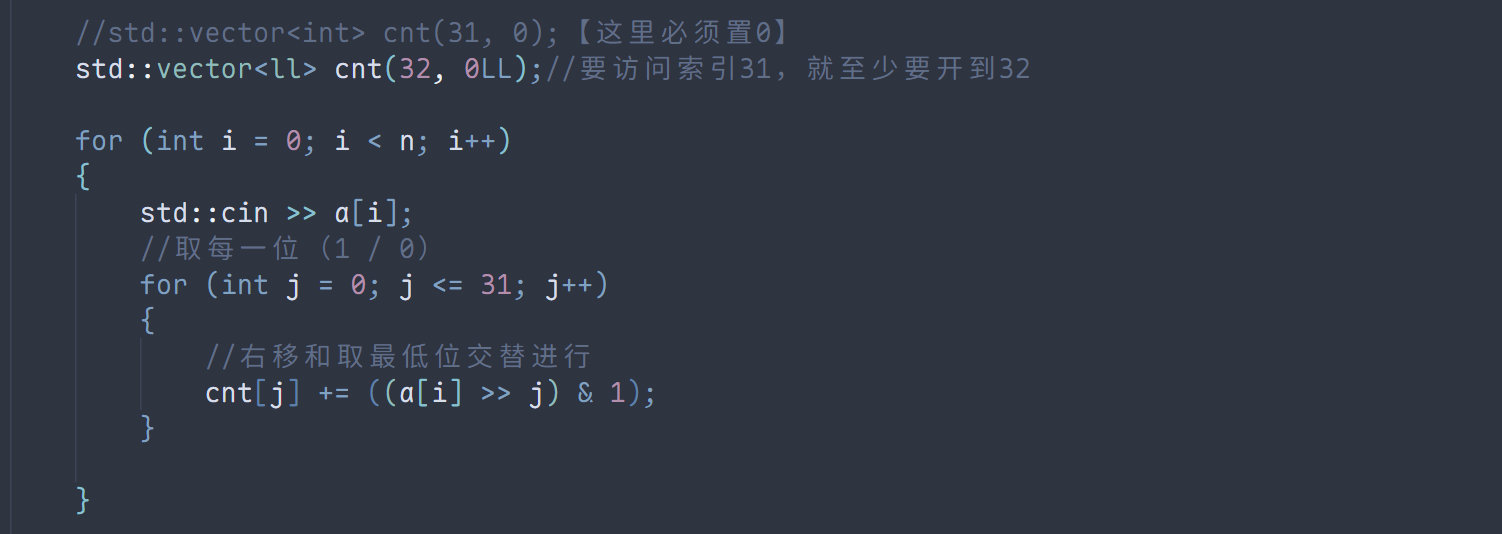
（或者说，32位（1~32））

*Lemma.*

*一个正整数 x 的二进制表示最多需要 [log\_{2} x] + 1 位*

*log2的x，下取整，加一*

一个简单的例子



这是很自然的，因为第 32 位是符号位

另外，第 64 位是 signed long long 的符号位，同理

下面介绍位与（&）的若干性质

基本地，位与：二者皆为1才得1

亦即：

1&1 = 1

1&0 = 0&1 = 0&0 = 0

于是，显然有如下的结论

a&a = a

a&0 = 0

a&1 = a\_lowest\_bit

下面引入一个定理

**Thm.**

若两数的位与等于两数的异或，则它们都是0

**Proof.**

我们知道，这两数的位与等于这两数的异或

假设该二元组中存在非零值，

则其二进制表达中存在1

则对于该位：

无论另一数的该位为1还是0

该位的位与和异或绝不相同

从而两数的位与和异或绝不相同

矛盾

证毕

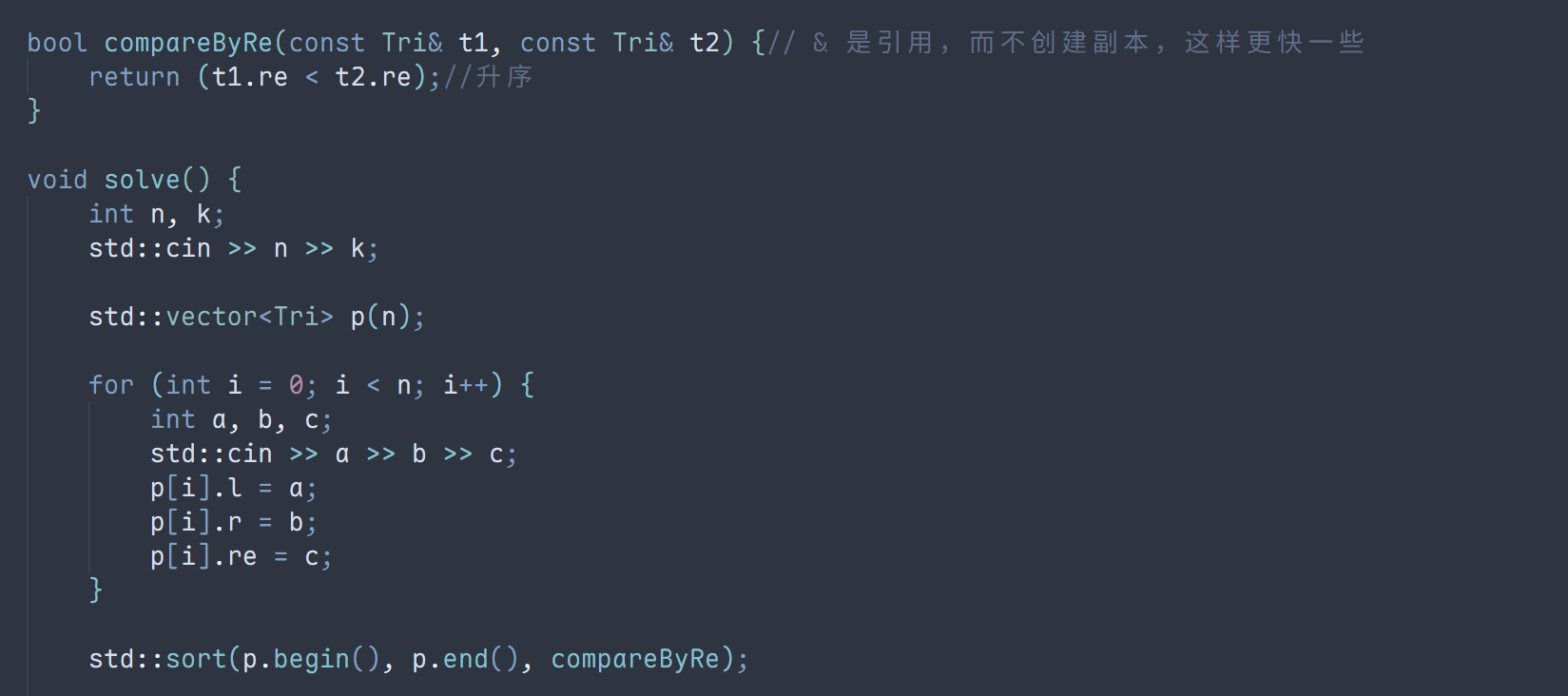
上面的证明依赖于这样一个引理：

**数的二进制表示是唯一确定的**

再者，注意移位运算的 1LL

对于ll型，(x << 1) 务必写作 (x << 1LL)

7.自定义cmp及其lambda表达式写法



【lambda】表达式 示例：

    std::stable\_sort(ord.begin(), ord.end(), [&](int i, int j) {

        return (a[i] > a[j]);  // 降序

    });

捕获列表 [&]

通过引用捕获所有外部变量（此处捕获了数组 a）

其他，有

    [&a]：显式捕获特定变量

    [=]：通过值捕获（复制变量）

    []：不捕获任何变量（这当然是最快的）

Lambda 表达式是 C++11 引入的核心特性，

提供了一种简洁的定义匿名函数对象的方式

一般地，lambda 表达式的语法如下：

[ capture-list ] ( params ) specifiers exception attr -> ret { body }

即

[ 捕获列表 ] ( 参数列表 ) 说明符\_可选 异常规范\_可选 -> ret { 函数体 }

返回类型 ->ret 也是可选的，省略时由 return 语句自动推导（auto）

再来一个例子：

std::vector<std::string> v;

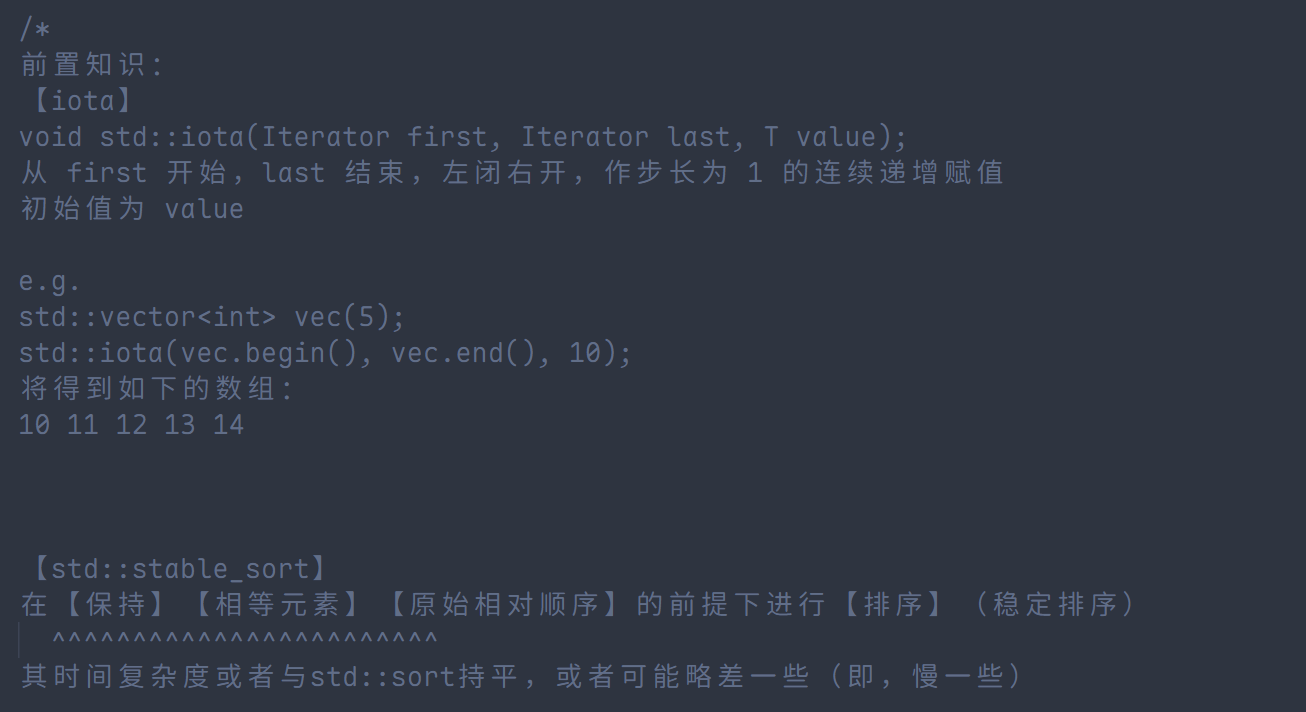
    v.push\_back("OC");

    v.push\_back("KP");

    v.push\_back("XW");

    sort(v.begin(), v.end(), [](string x, string y) {return mp[x] < mp[y];});

另注：



8.std::ceil的完备手动实现

这里，特别强调一下这个引理

理论上，

(a + b - 1) / b == ceil(1.0 \* a / b)

由初等数论中的余数表示式，以及简单的推理，我们不难证明这个公式对正整数普遍成立

RHS 对极端数据可能出错，所以我们用 LHS 代替

当然，LHS 中是整型除

类似地，

std::floor(1.0 \* a / b)对大数据也极易出错，并不可靠

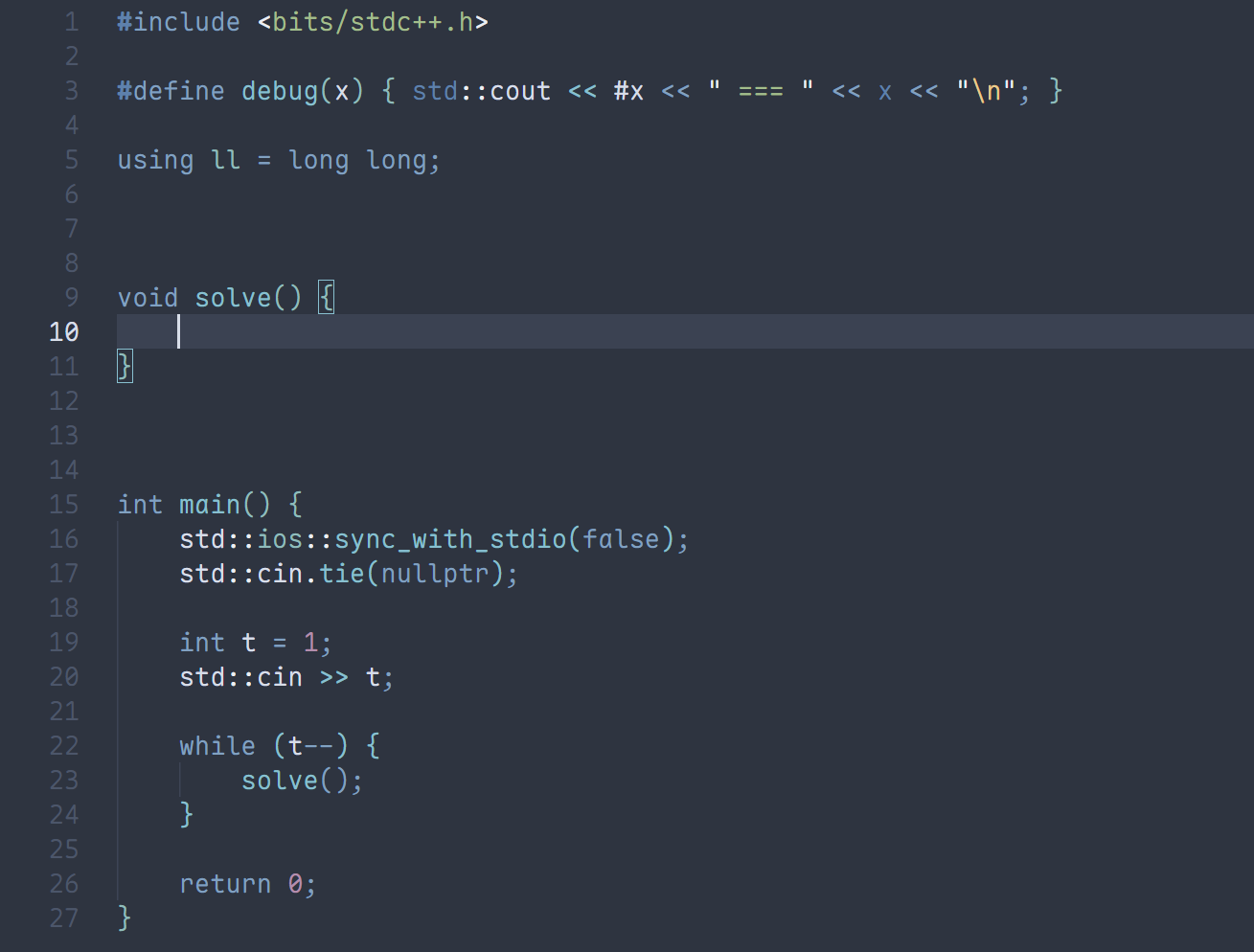
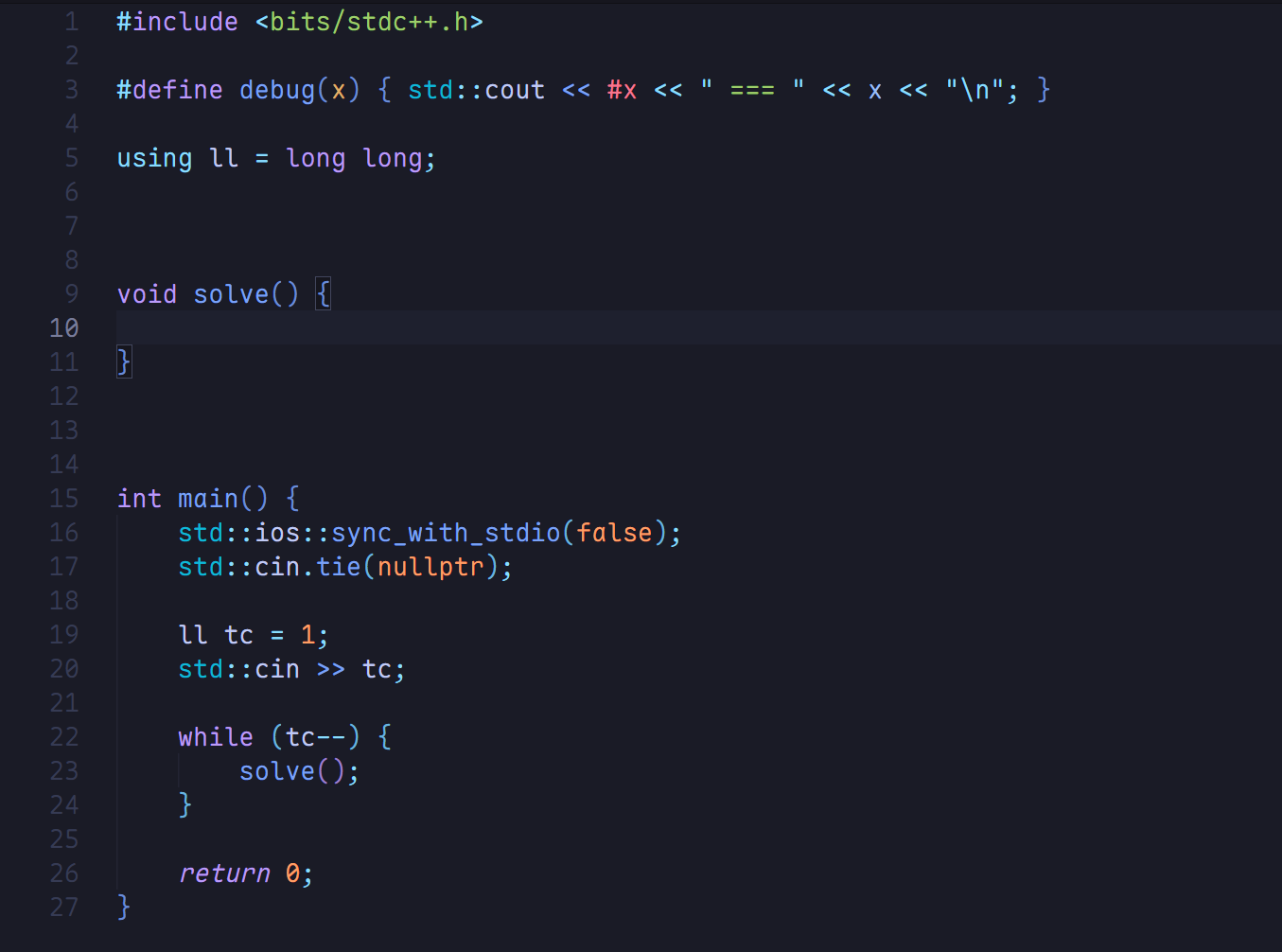
应当直接使用整型除的特性完成下取整，即

Res = a / b

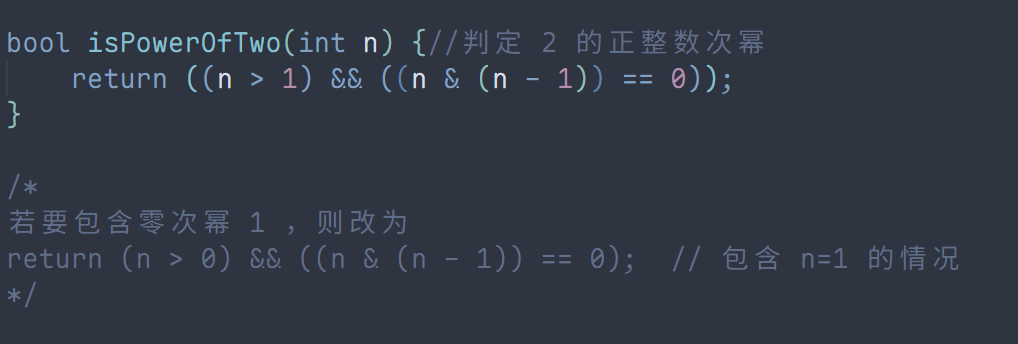
而不是使用 floor()

9. 998244353，见目录

10. stdcode.cpp

11.判定2的幂次（或3）



这里强调，不要太依赖这个结论

有时，我们需要搜索、筛出或采取一些2的幂次，并作分析处理

这时，依次遍历范围内正整数并判别 n&(n-1)==0 未必是优解，因为这是 O(n) 的

很可能还不如从某个2^p（比如1或者2）倍增遍历，这是 O(log n) 的，更加高效

判定p是不是3的幂次：  
如果范围确定，我们只须跑出这个范围的最大3^n，记为常数mx，

然后判断p是不是mx的因子

譬如

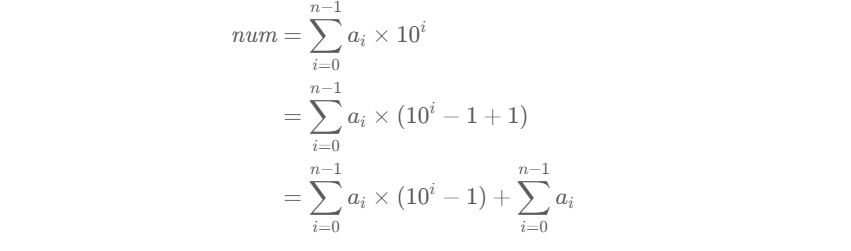
bool isPowerOfThree(int n) {

return n > 0 && 1162261467 % n == 0;

}

这当然是O(1)的

当然，也可以while%3==0就不断除以3，判断最终结果是不是1，这个方法是O(logn)的

1. 更相减损结论的多元形式  
   我们知道，gcd(x,y)=gcd(x-y,y)   
   gcd(x, y, z, ...) = gcd(x-y, y, z, ...)  
   注：  
   1.这是一个简单形式，读者可以自行推广出其他变体  
   譬如 gcd(a1, a2, ...aj) = gcd(a1, a2-a1, ..., aj-a1)  
   2.谨慎地说，我们通常建议，保证作差后的新值是正数  
   事实上，gcd(x,y)=gcd(x-y,y) 并不要求x >= y  
   另外，C++标准保证std::gcd的行为如下：  
   gcd(a, b) = gcd(|a|, |b|)  
   一方为0时返回另一方  
   特别地，双方为零时，返回0  
   所以，由数学推导，我们看到，通常情况下，我们的确有：  
   对于一切整数x和y，std::gcd(x, y) == std::gcd(x - y, y)  
   也就是说，可以应用更相减损结论
2. 000
3. 数位和  
   digit(x)是各位数字之和，求得方法很简单，while模拟即可  
   下面介绍一些结论  
     
   由上可知，num与其数位和模9同余  
   我们定义，dr为num的数根，又称数字根，表示不断执行数位和算子，最终得到的个位数  
   不难发现，数字根当然也与num模9同余，因此：  
   if num % 9 == 0:  
    if num == 0 => dr = 0  
    else => dr = 9(dr > 0)  
   else => dr = num % 9
4. 计算组合数的板子

我们给出如下计算组合数的板子：

//这是 dp 预计算组合数的板子

//基于我们高中所熟悉的递推式：

// C\_i^j = C\_{i-1}^{j} + C\_{i-1}^{j-1}

c[0][0] = 1;

for (int i = 1; i <= n; i++) {

c[i][0] = 1;

for (int j = 1; j <= i; j++) {

c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % M;

}

}

要不要 %M 取决于题设

    ll n, k;

    std::cin >> n >> k;

    std::vector<std::vector<ll>> c(n + 1, std::vector<ll> (n + 1));

    c[0][0] = 1;

    for (ll i = 1; i <= n; i++) {

        c[i][0] = 1;

        for (ll j = 1; j <= i; j++) {

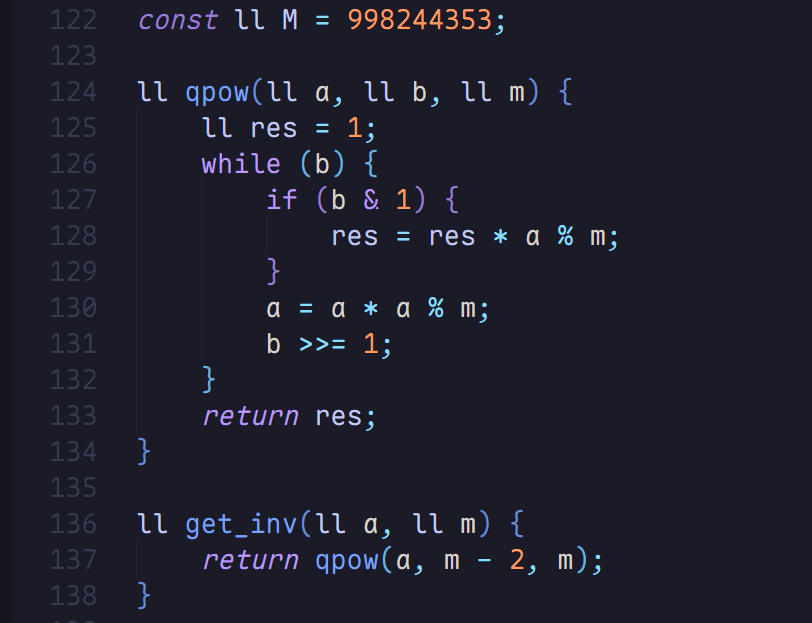
            c[i][j] = (c[i - 1][j] % M + c[i - 1][j - 1] % M) % M;

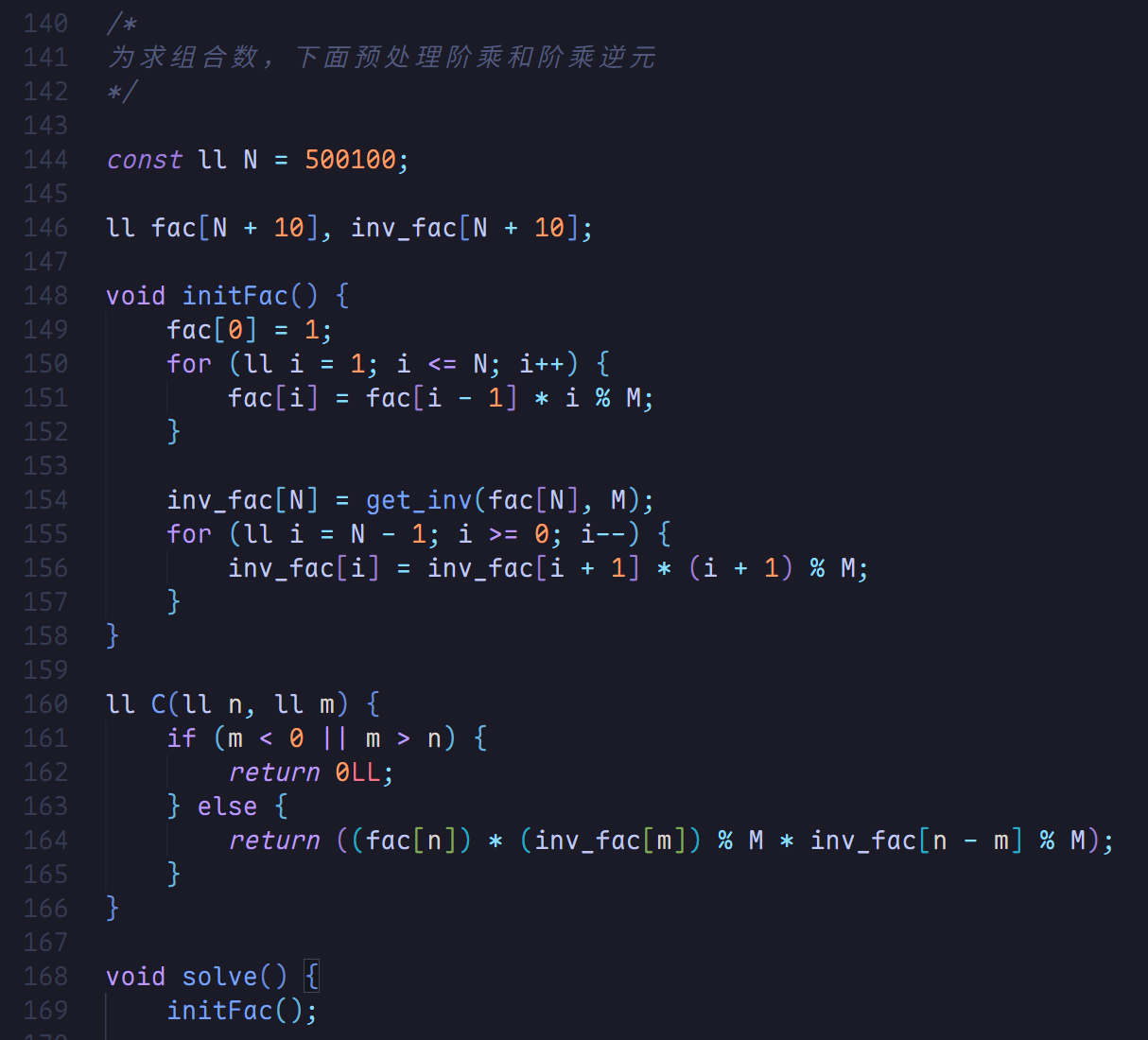
        }

    }

    std::cout << c[n][k] << '\n';

这个算法大概是平方复杂度，远远优于暴力的阶乘复杂度

利用逆元/快速幂，有：  




Init通常在main中

1. 拓扑排序

拓扑排序：

对有向无环图（DAG）的顶点进行线性排序，使得对于任意有向边(u, v)：

序列中，u必出现在v之前

这样的序列可能不唯一，我们将这样的序列称为拓扑序

实现，有多种思路：

1.

不断选择入度为0的点，添加到序列中，再删除其本身及其所有出边

如此往复即可，直至队列为空

（BFS，利用队列维护入度为0的点）

2.

预处理所有点的入度

法二与法一相似，区别在于，不是删除出边，

而是将0入度的点指向的所有顶点的入度减一（减到0则入队）

另外，我们指出，拓扑排序要在无环图中进行

若有环，过程将提前终止，即，0入度队列将在目标序列长度达到节点数前耗尽

这一点可以模拟发现

所以，拓扑排序也能用于判环：

如果队列为空，但仍有顶点未被加入拓扑排序的结果序列中，说明图中存在环

事实上，只需判断：

结果列表的节点数少于总节点数，则说明图中存在环

（一个有向图能够被拓扑排序的充分必要条件是，它是一个有向无环图）

参考：

    std::vector<int> deg(n, 0);*//入度*

    std::vector<std::vector<int>> g(n);*//入边*

std::vector<std::vector<int>> gg(n);*//出边*

*//...*

    std::queue<int> q;

    for (int i = 0; i < n; i++) {

        if (deg[i] == 0) {

            q.push(i);

        }

    }

    std::vector<int> p(n);

    for (int i = 1; i <= n; i++) {

        int u = q.front();

        p.push\_back(u);

        q.pop();

        for (*const* auto &v : gg[u]) {

            deg[v]--;

            if (deg[v] == 0) {

                q.push(v);

            }

        }

    }

其中，u是拓扑序的项

另外，在拓扑序上 dp ，我们能简单地求得 DAG 的最长路径

（暂且定义len是路径所含边数）

确切来说，初始化是0，转移方程是

dp[v] = std::max(dp[v], dp[u] + 1);

最后对dp数组取最大即可  
可以看出，拓扑排序不只是为了得到那个序列，有时也是为了依拓扑序去访问严格相连的两个节点  
  
另外，指出一点：

**该算法基于队列，不断将入度为0的顶点加入队列并处理，直到队列为空。如果图中无环，最终会输出所有n个顶点；如果有环，队列会在输出部分顶点后变空。因此，拓扑排序是执行到队列为空，而不是固定执行n次（除非已经确定是DAG）。**

1. 并查集

*/\**

*并查集*

*这套并查集板子同时采用 路径压缩 和 按秩合并 2 种优化，相当高效*

*确切来说，*

*查询操作的均摊时间复杂度能达到 反阿克曼函数 级别：O(alpha(n))*

*在算竞场景下，我们完全可以认为其近似 O(1) 级别*

*\*/*

struct DSU {

    std::vector<int> p, sz;*// parent、size*

    DSU(int n) {

        p.assign(n, 0);

        sz.assign(n, 1);

        for (int i = 0; i < n; i++) {

            p[i] = i;

        }

    }

    int find(int u) {

        if (p[u] == u) {

*return* u;

        }

        p[u] = find(p[u]);

*return* p[u];

    }

    void unite(int u, int v) {

        u = find(u);

        v = find(v);

        if (u == v) {

*return*;

        }

        if (sz[u] < sz[v]) {

            std::swap(u, v);

        }

        p[v] = u;

        sz[u] += sz[v];

    }

    bool same(int u, int v) {

*return* find(u) == find(v);

    }

    int size(int u) {

        u = find(u);

*return* sz[u];

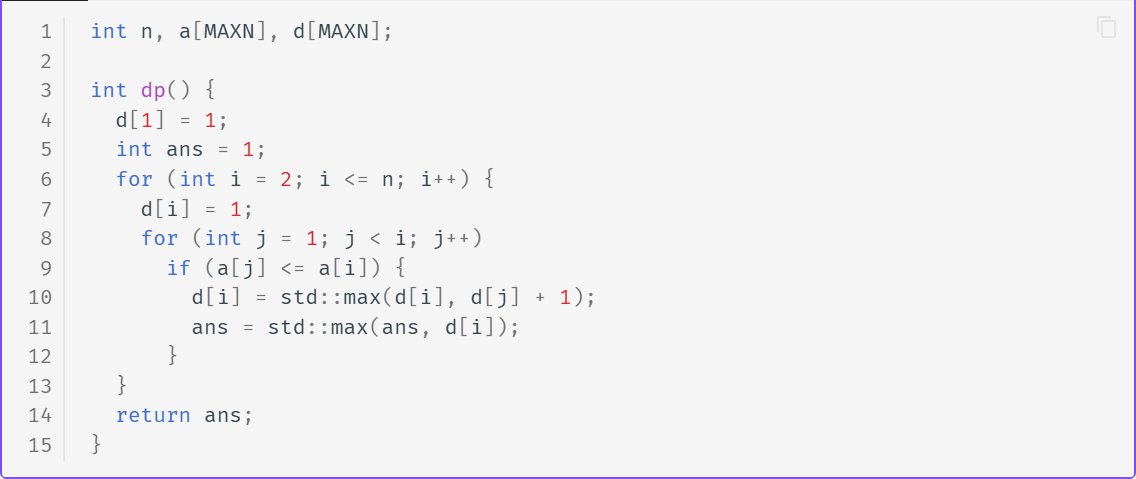
    }

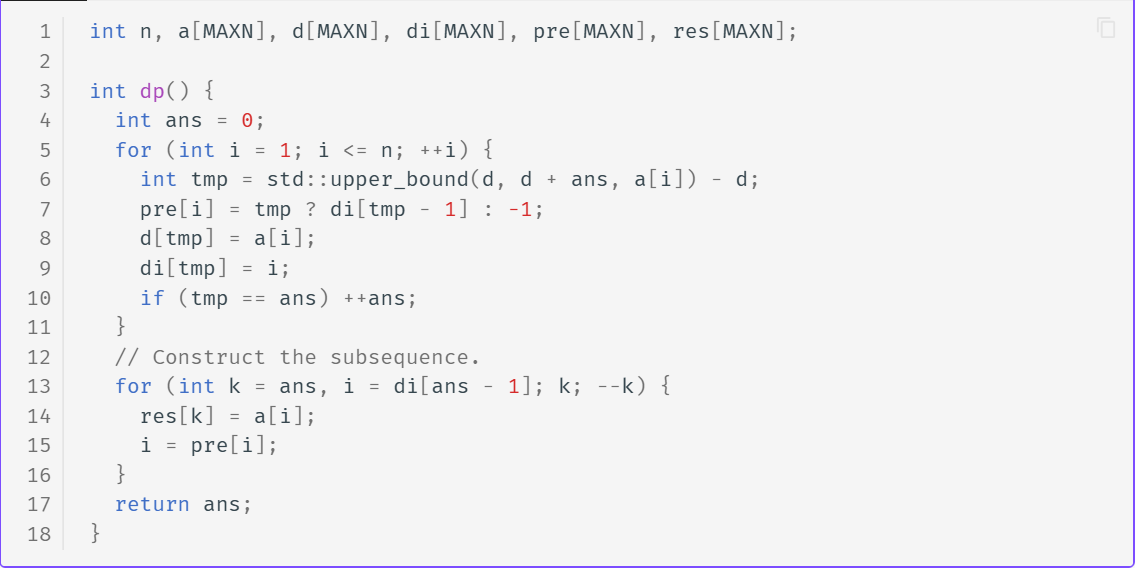
};

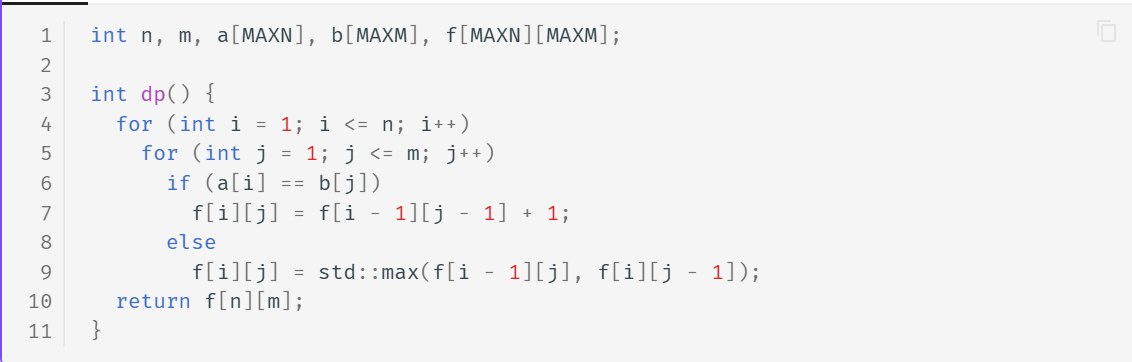
【下面是一些临时拉来的未经整理验证的板子，可供参考】

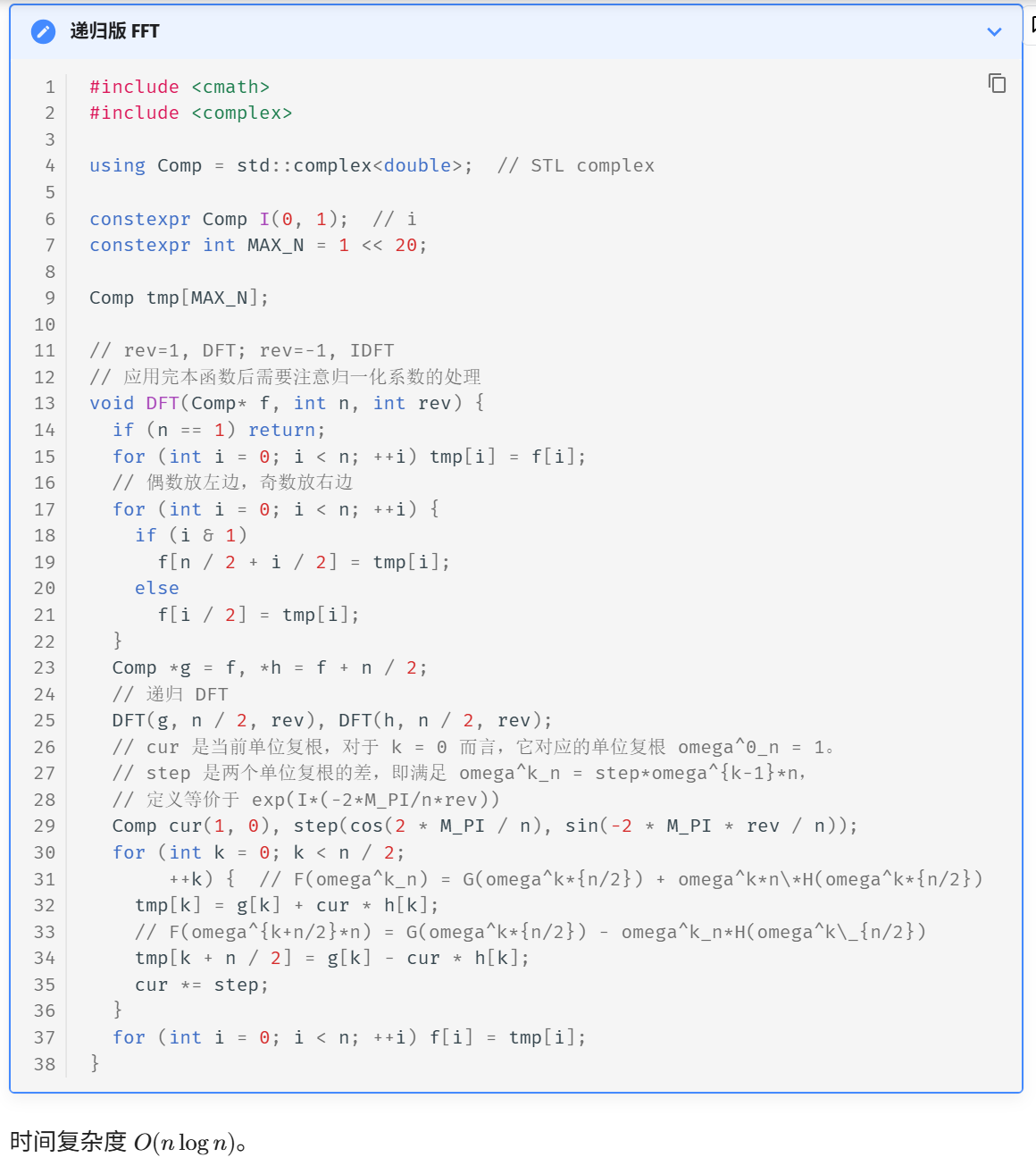
【1】最长不降子序列、最长公共子序列

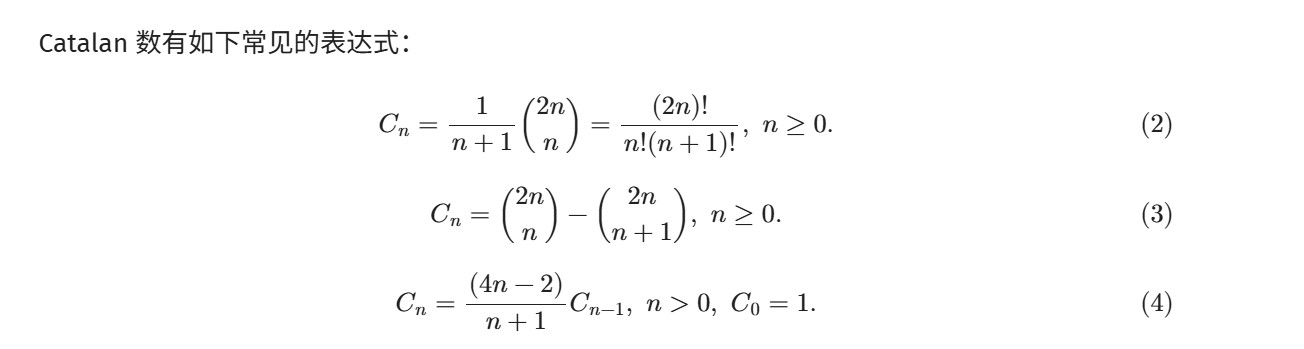
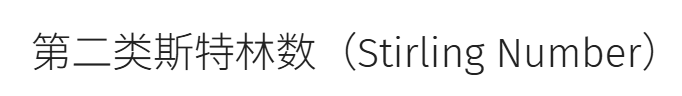
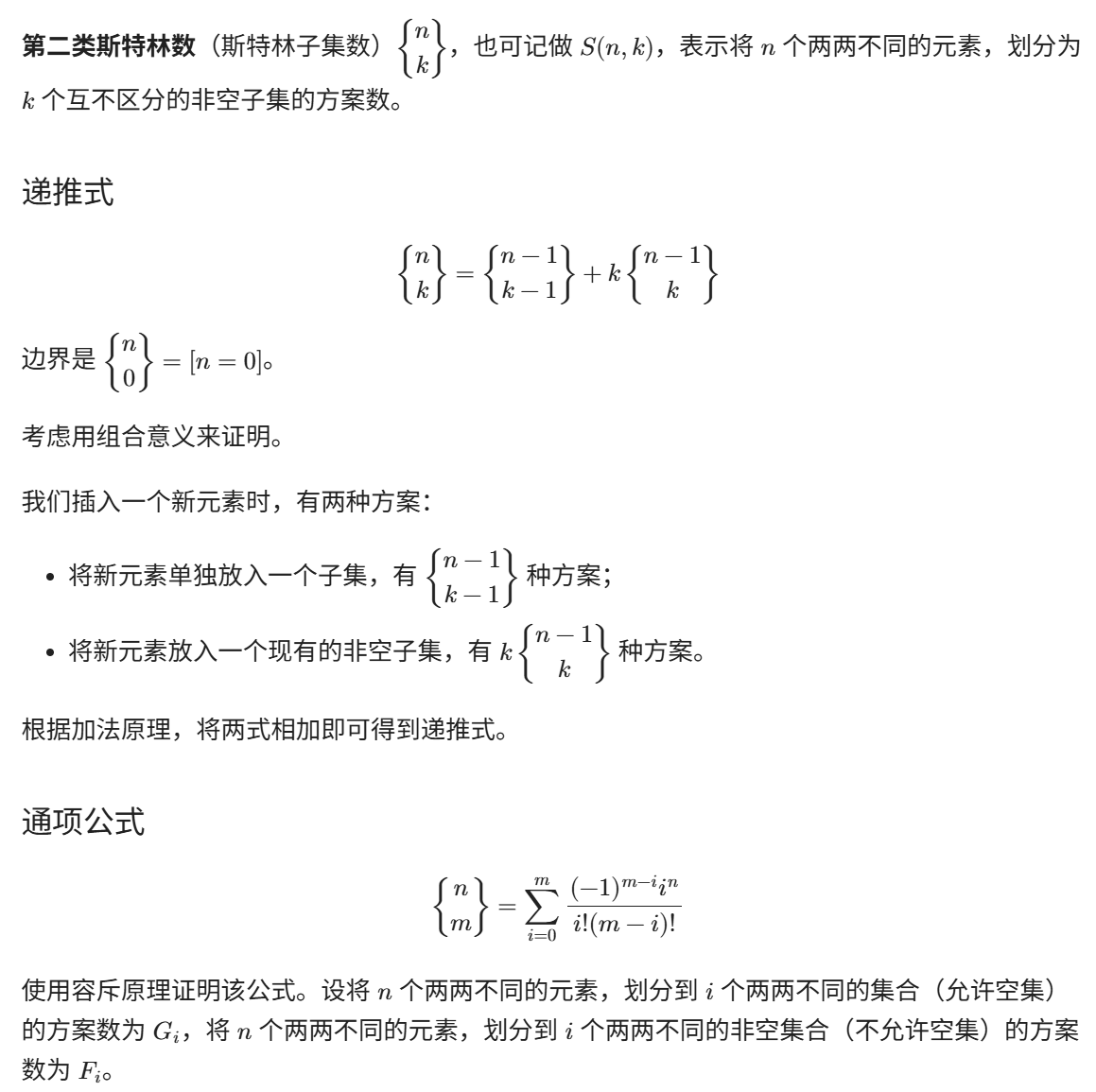
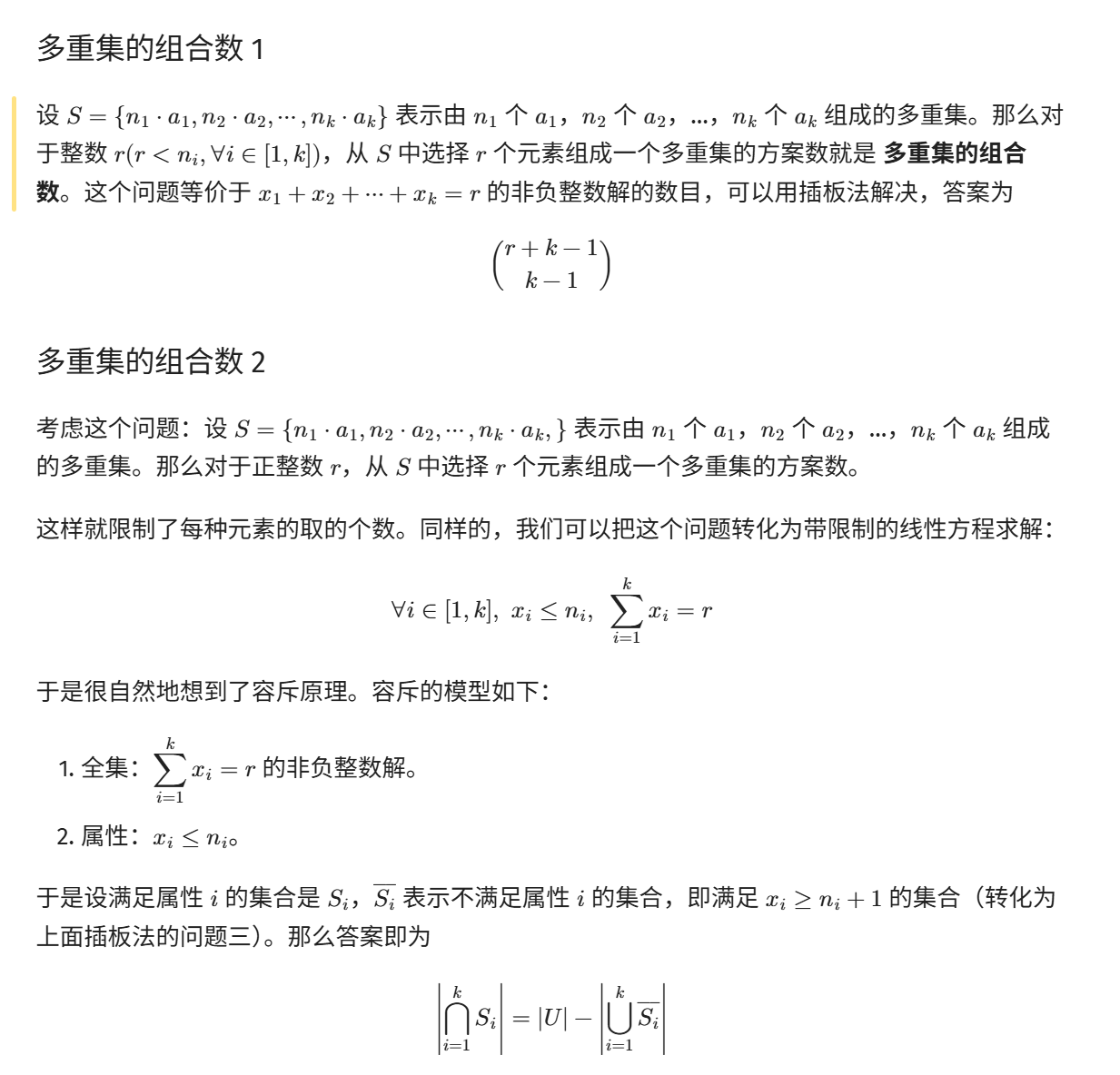
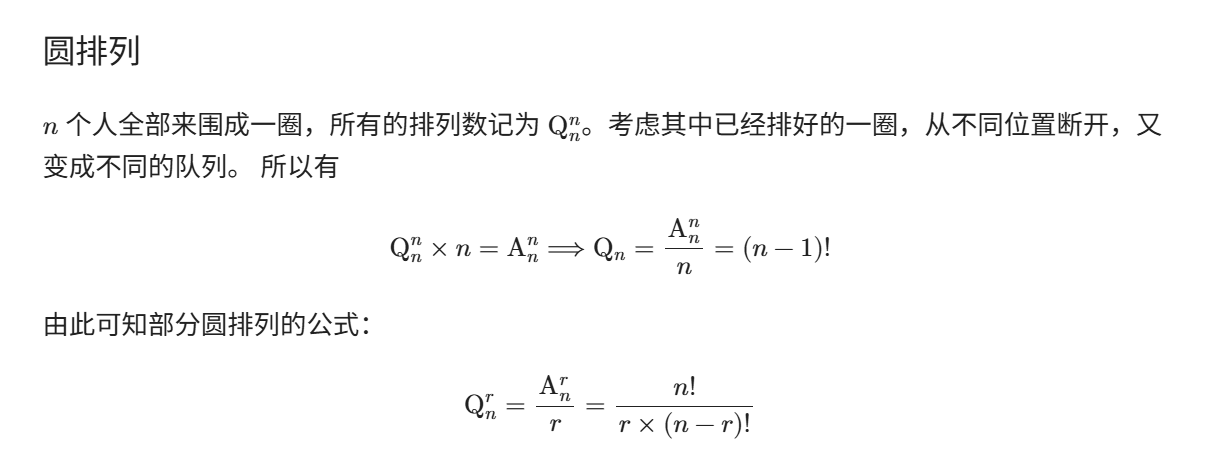
//LIS

// O(n^2)  


// O(nlogn)  


//LCS  


1. 快速傅里叶变换FFT与数论变换NTT（略）  
   【FFT】  
   

  
  
(1 / (n+1)) \* C\_{2n}^{n}  
  
  
  
  
  
组合数（快速）  
#include <vector>

#include <iostream>

using namespace std;

class Combinatorics {

private:

int max\_n;

long long mod;

vector<long long> fac, inv\_fac, inv;

vector<vector<long long>> dp;

long long power(long long a, long long b) {

long long res = 1;

while (b) {

if (b & 1) res = res \* a % mod;

a = a \* a % mod;

b >>= 1;

}

return res;

}

public:

Combinatorics(int n, long long m = 1e9 + 7) : max\_n(n), mod(m) {

// 预处理阶乘和逆元

fac.resize(n + 1);

inv\_fac.resize(n + 1);

inv.resize(n + 1);

fac[0] = fac[1] = 1;

inv[1] = 1;

for (int i = 2; i <= n; i++) {

fac[i] = fac[i - 1] \* i % mod;

inv[i] = (mod - mod / i) \* inv[mod % i] % mod;

}

inv\_fac[n] = power(fac[n], mod - 2);

for (int i = n - 1; i >= 0; i--) {

inv\_fac[i] = inv\_fac[i + 1] \* (i + 1) % mod;

}

// 预处理杨辉三角

dp.resize(n + 1, vector<long long>(n + 1, 0));

for (int i = 0; i <= n; i++) {

dp[i][0] = dp[i][i] = 1;

for (int j = 1; j < i; j++) {

dp[i][j] = (dp[i - 1][j - 1] + dp[i - 1][j]) % mod;

}

}

}

// 组合数 C(n, m)

long long C(int n, int m) {

if (m < 0 || m > n) return 0;

if (n <= max\_n) return dp[n][m];

return fac[n] \* inv\_fac[m] % mod \* inv\_fac[n - m] % mod;

}

// 排列数 A(n, m)

long long A(int n, int m) {

if (m < 0 || m > n) return 0;

return fac[n] \* inv\_fac[n - m] % mod;

}

// 阶乘

long long factorial(int n) {

return fac[n];

}

// 逆元

long long inverse(int n) {

return inv[n];

}

// 快速幂

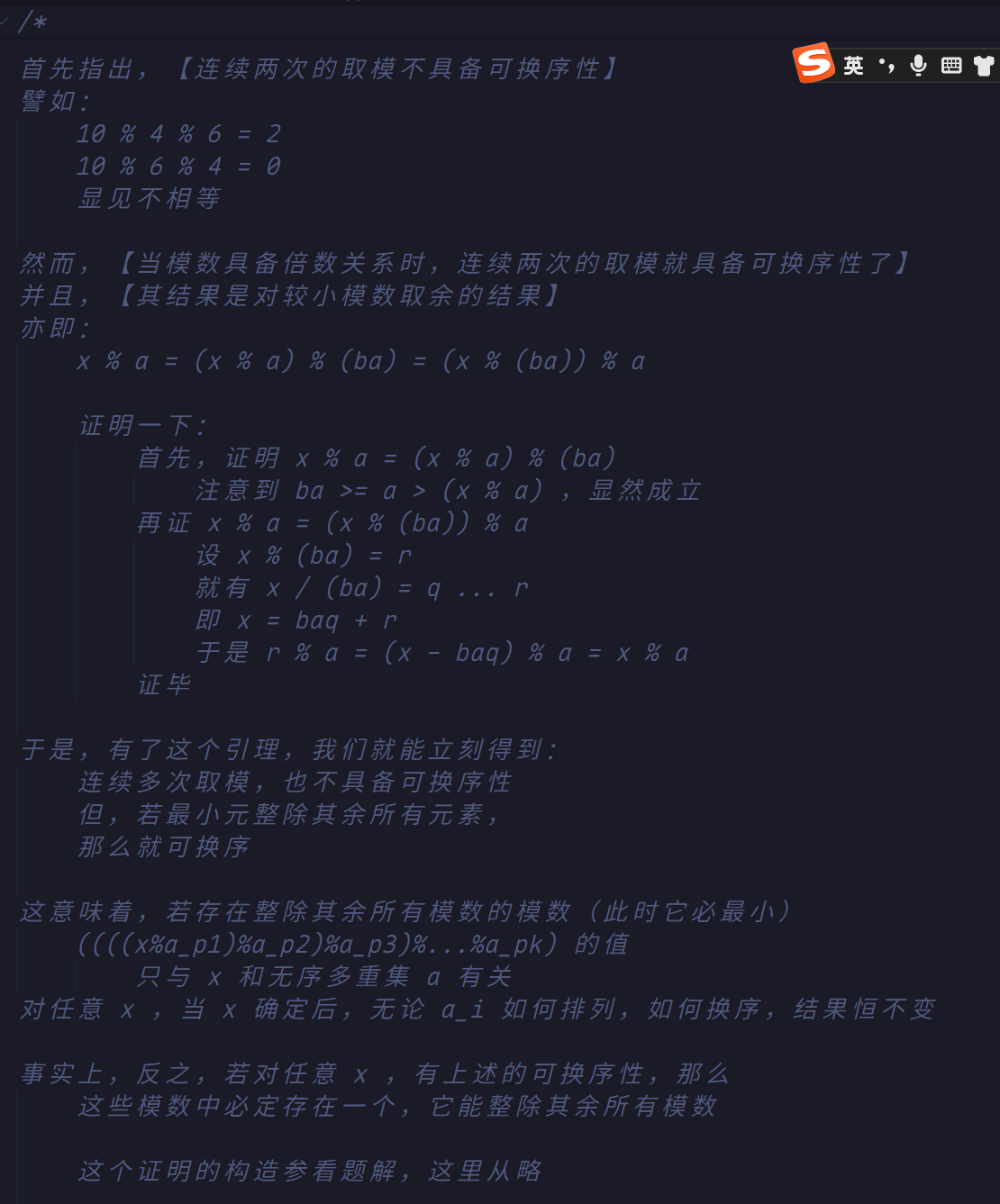
long long pow(long long a, long long b) {

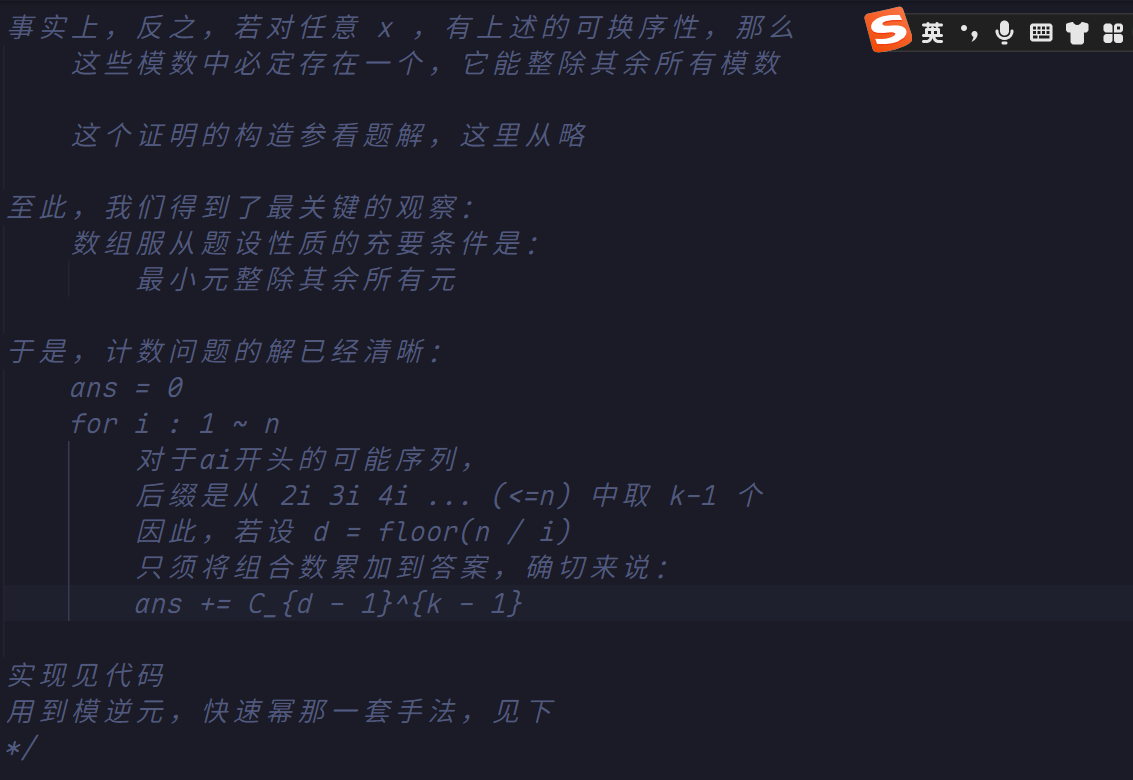
return power(a, b);

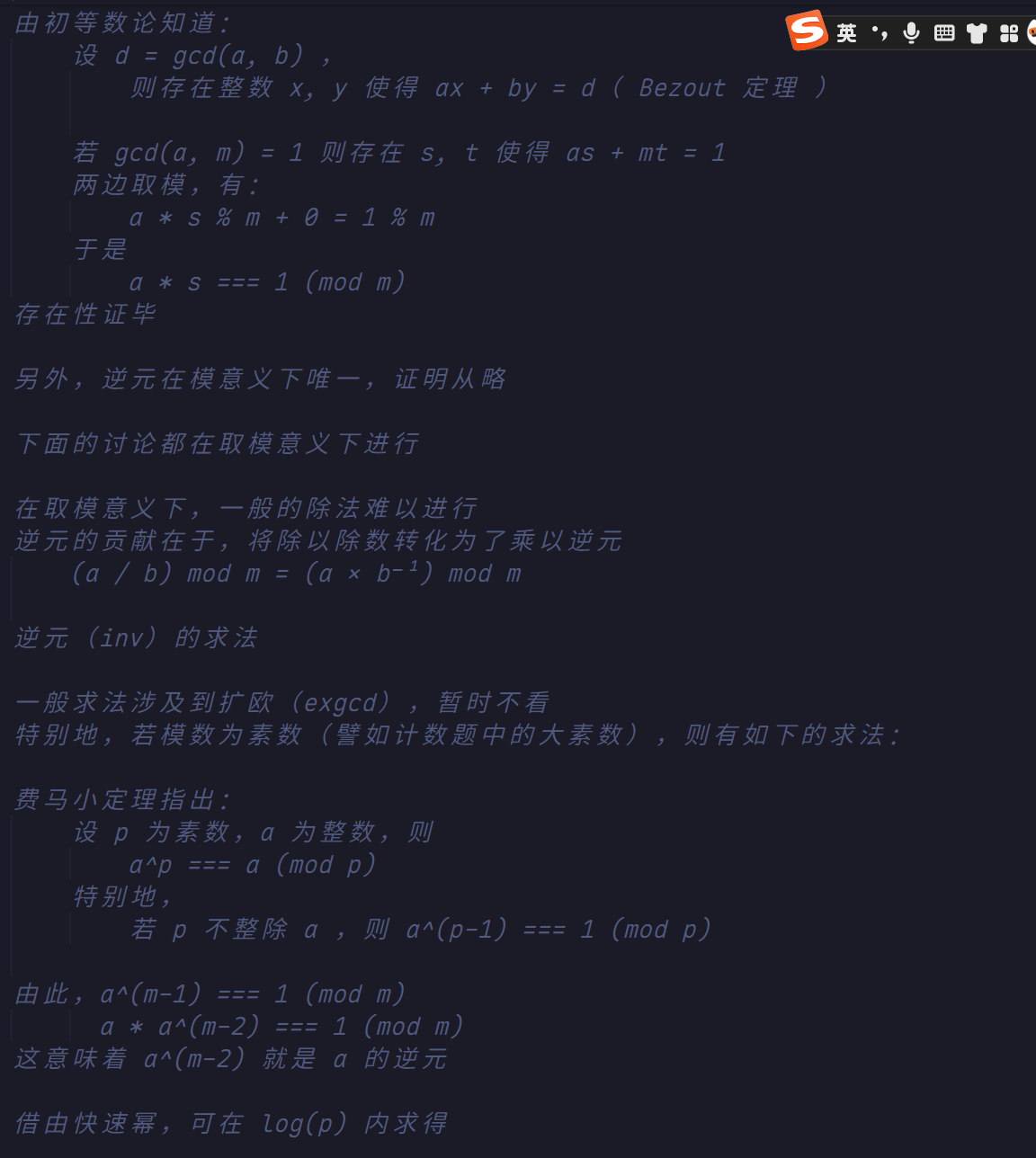
}

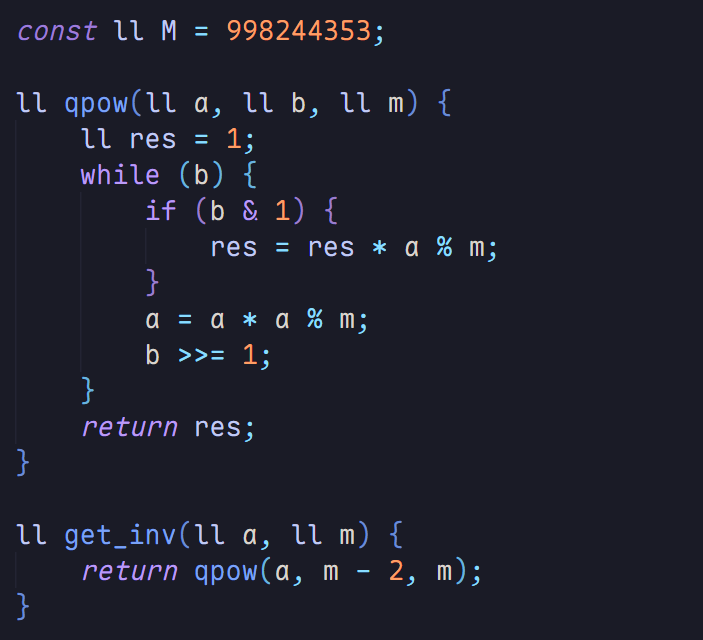
};

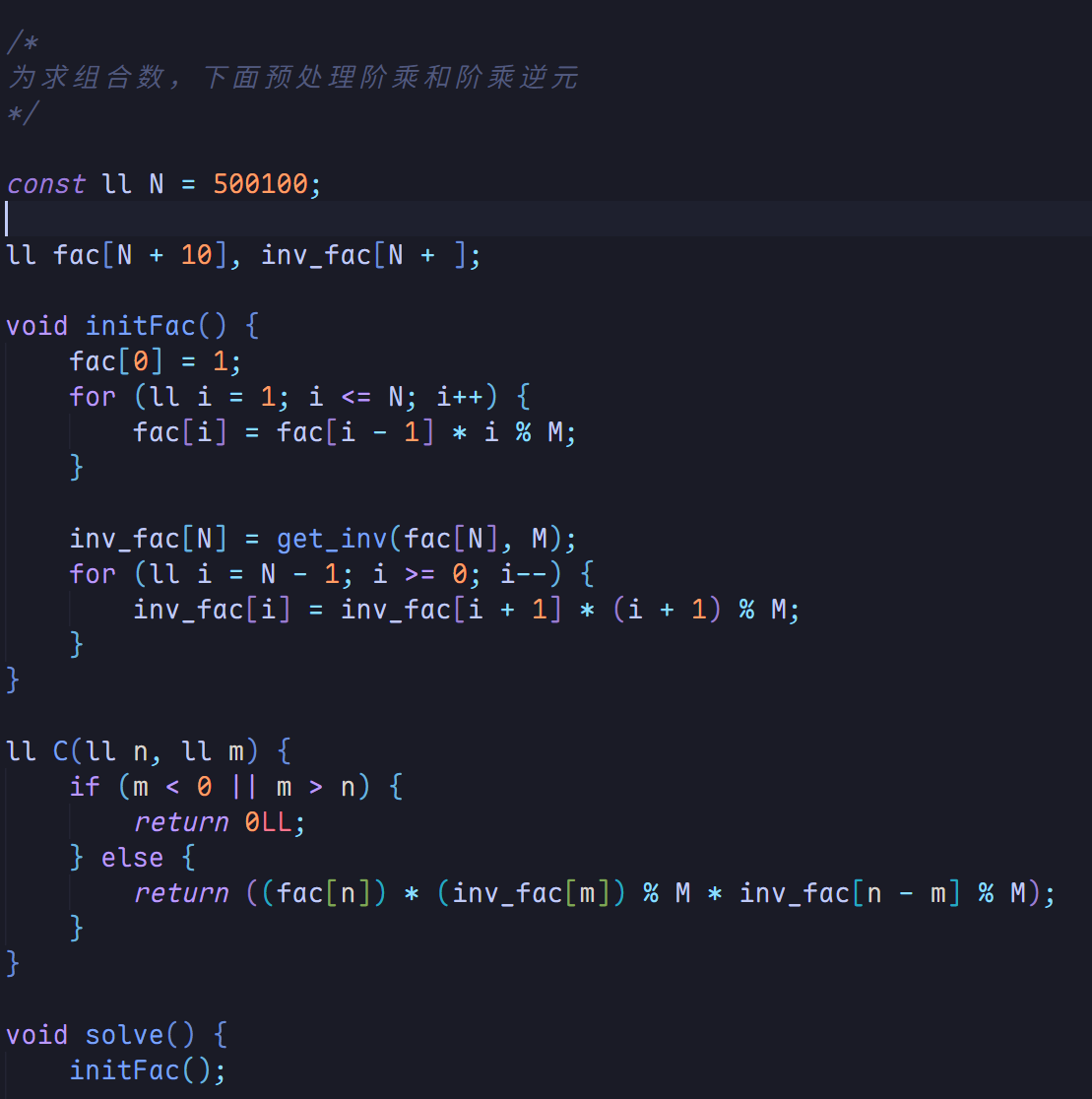
给一个规范的例子：











20. DFS / BFS基本形式

void dfs(int u, int p) {

【探索子树前】

for (int v : g[u]) {

if (v == p) {

continue;

}

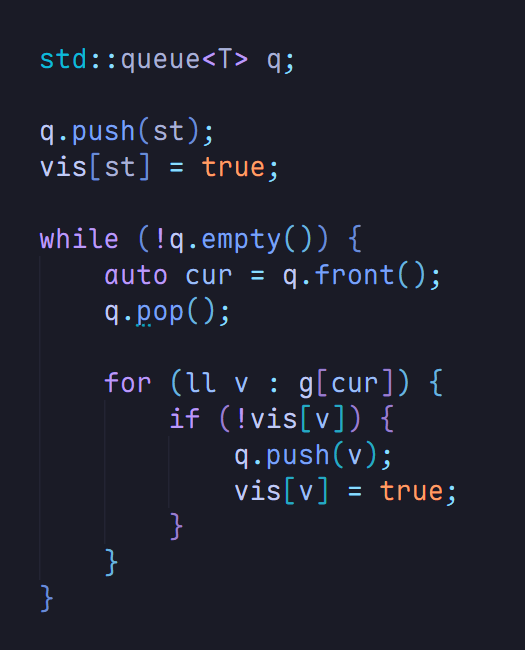
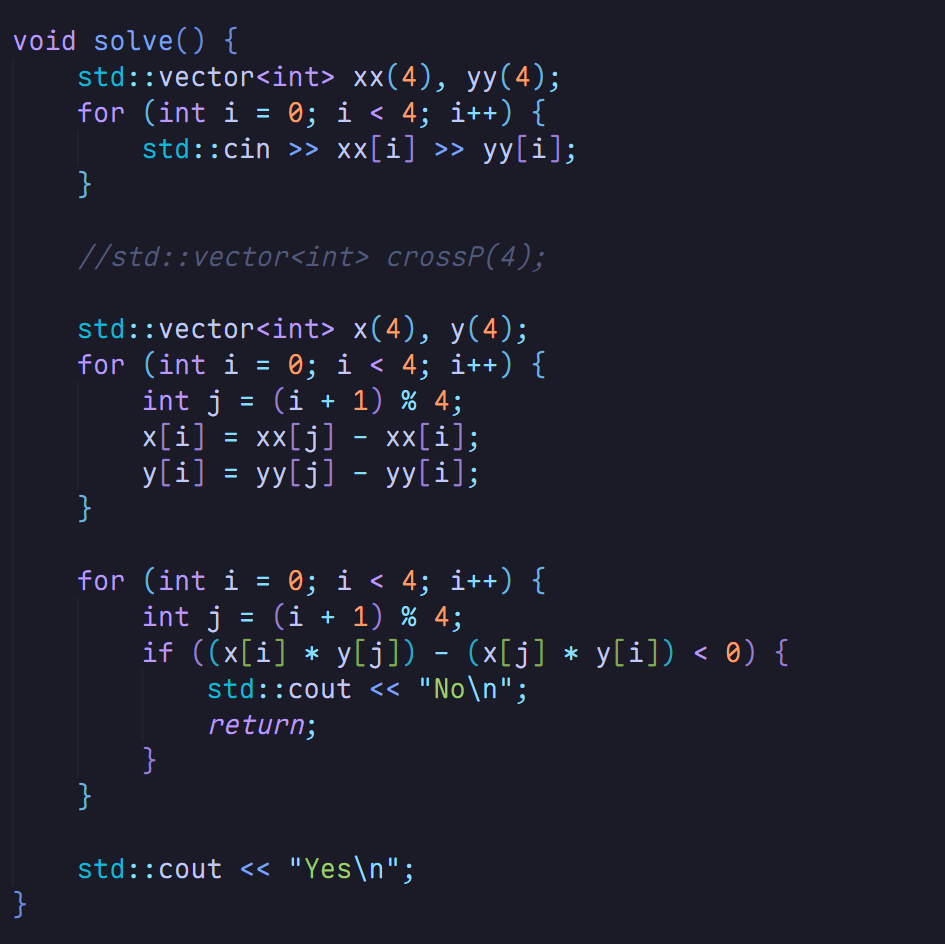
dfs(v, u);【探索子树】

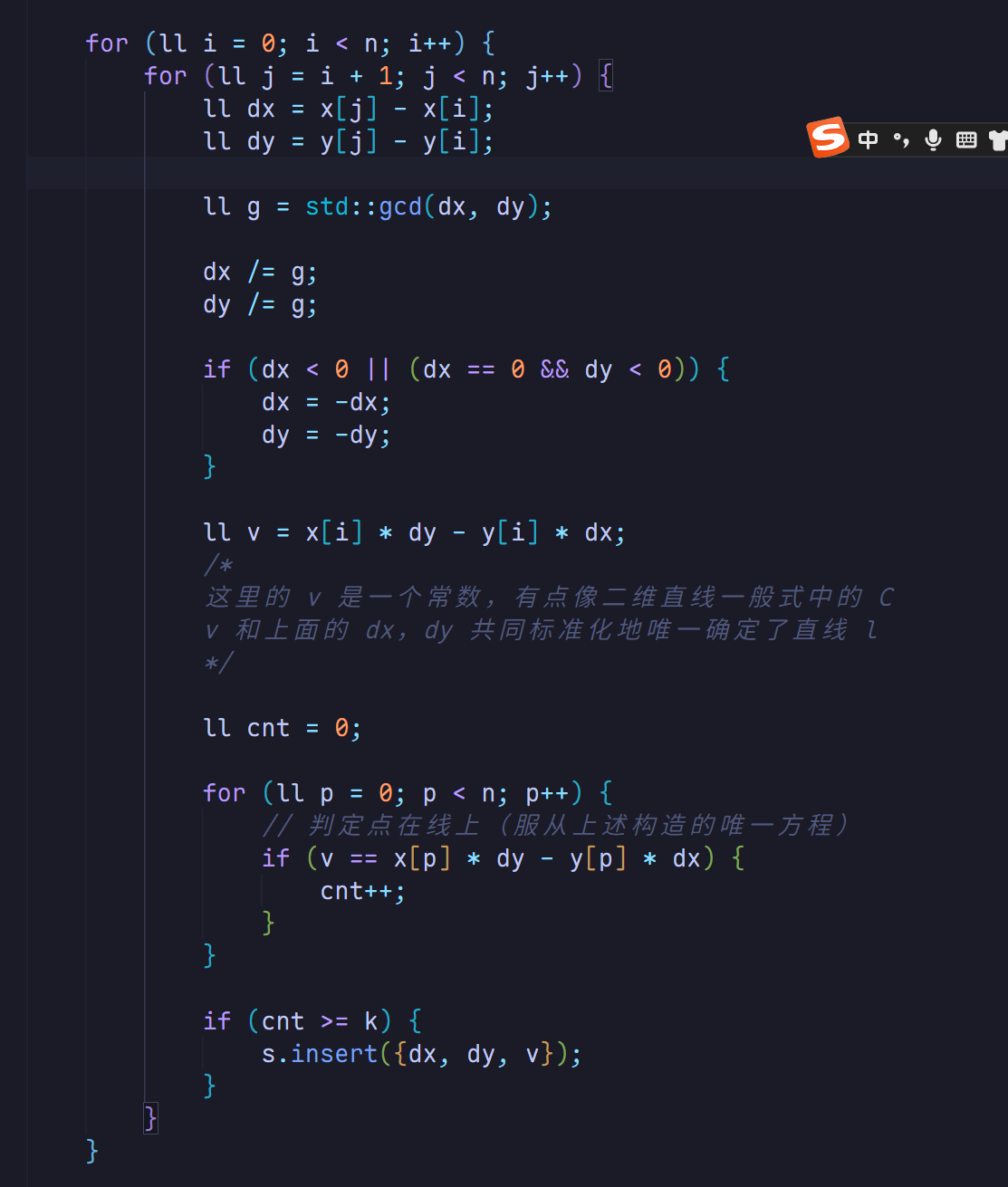
【回溯后】

}

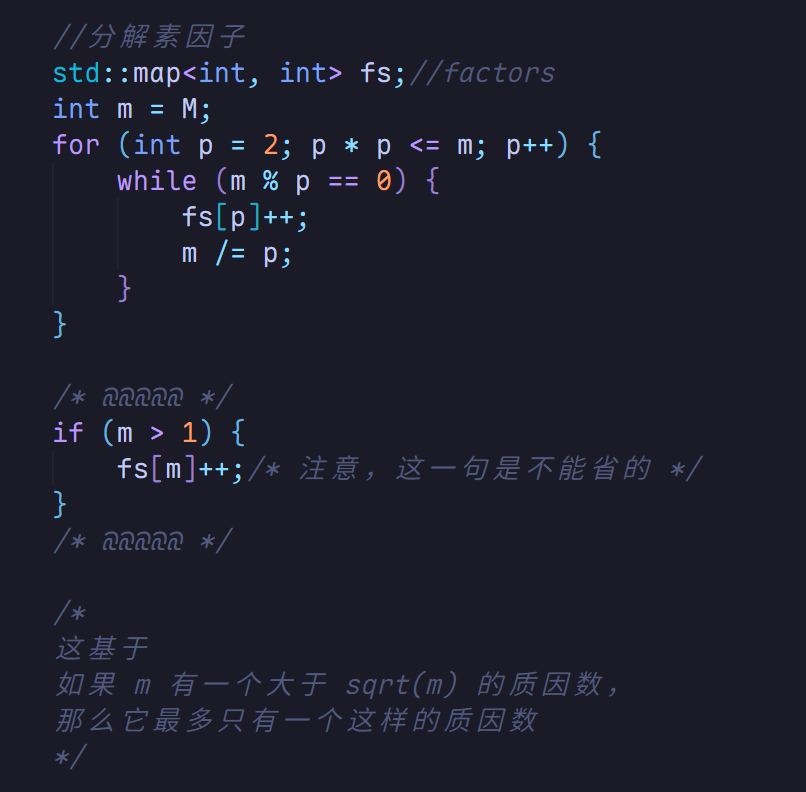
}

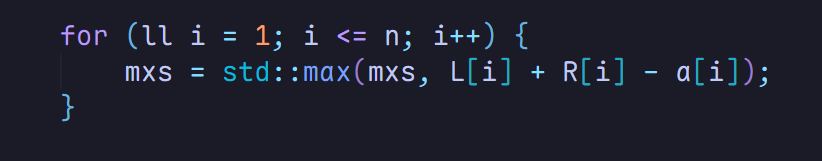
下面是bfs：

  
  
21.计算几何基础：向量积判定凸性、共线判定  
凸性判定示例：  


直线：  


1. 分解素因子





mxs的初值取决于能不能取空数组，能就是0，反之是 -infty

1. 最大子段和（最大子数组和）  
   

最后扫一遍，考虑  
L(i)+R(i)-self  
即L(i)+R(i)-a[i]  
24.向负无穷取整 与 欧几里得取模



欧几里得取模的定义是：

Reminder:

the remainder of a number x modulo p is the smallest non-negative y such that

there exists an integer q and x=p\*q+y.

25.循环分解定理

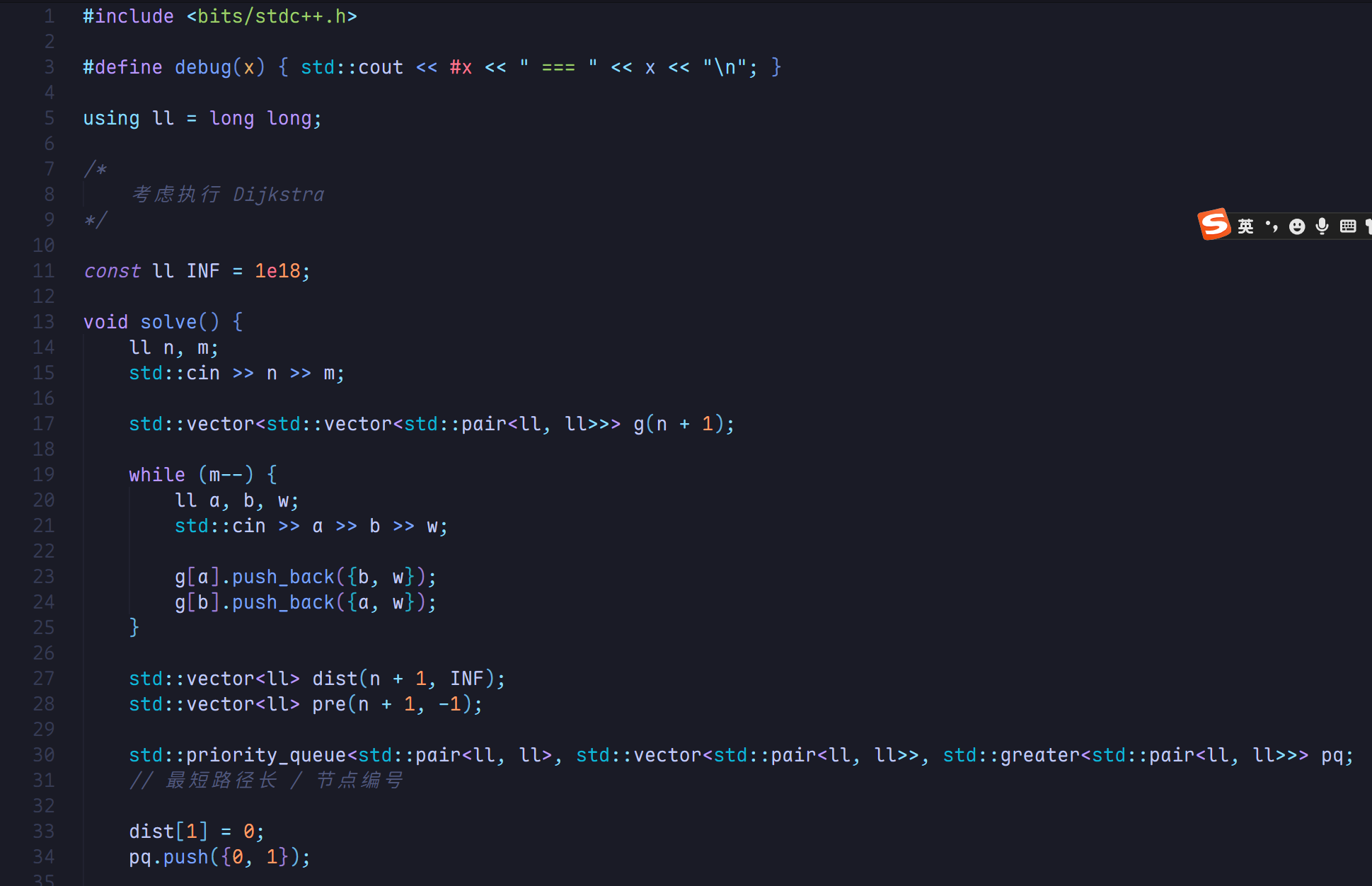
**Thm. 置换的【循环分解定理】**

**若考虑 1~n 的排列 p，**

**在所有的 i -> p[i] 间存在有向边 i -> p[i]，**

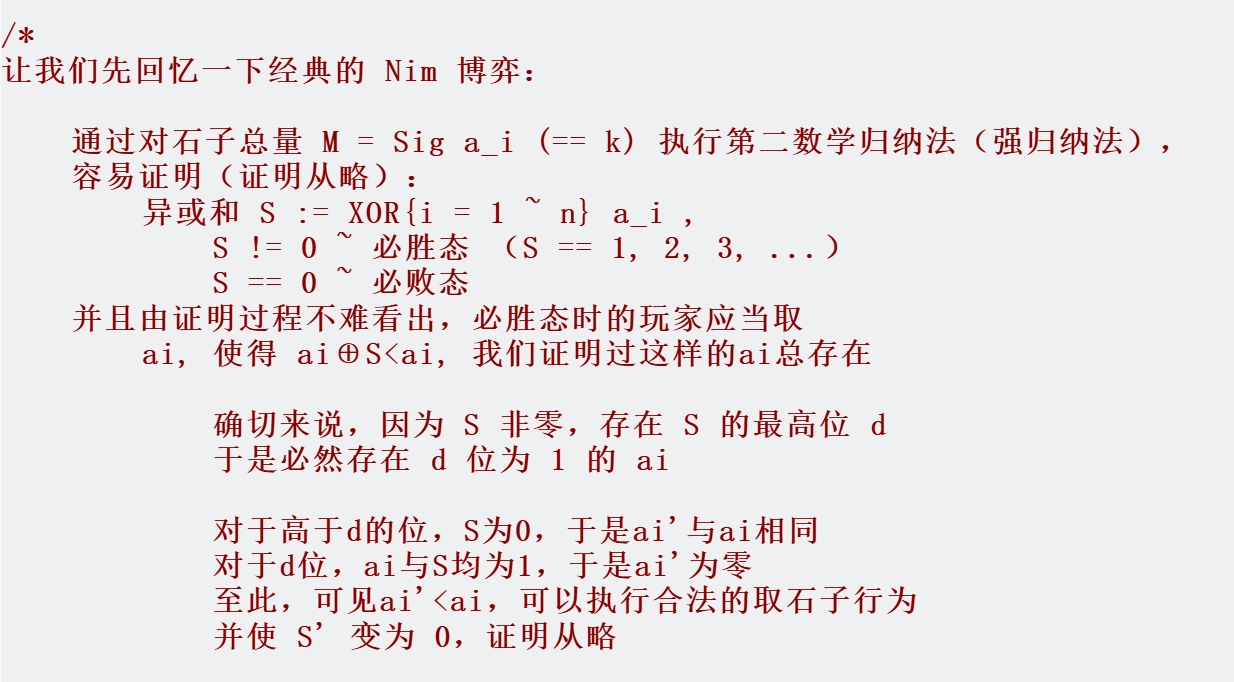
**则这个图由若干个不交的环构成**

**26.Dijkstra算法（非负权图单源最短路，迪杰斯特拉）**





注：

1. 这里给出的是经过小根堆优化的版本，复杂度不是近似n^2而是近似nlogn
2. 这里也展示了路径的复原过程
3. 其中的if (d > dist[u]) {continue;}是一种惰性删除 / 懒删除，对效率有益
4. 若用emplace\_back和emplace写法分别代替push\_back和push写法，会有微弱的加速（别忘了emplace系列不用花括号{}）  
     
     
   27.Nim博弈  
   

  
28.**函数图**（亦称**后继图**或**基环内向森林**）

若图的每个顶点都**有且仅有**一个后继，那么：

这样的图称为**函数图**（functional graphs）或后继图（successor graphs）

我们不加证明地指出：

这样的图可以划分为 k (k >= 1) 个分量，

其中每个分量必包含一个环

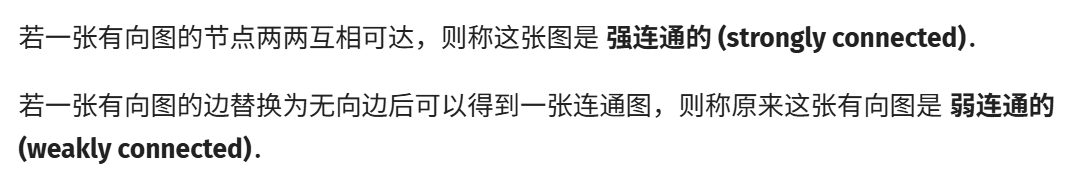
并且，每个节点，

要么处在环上，

要么处在通往环上的路径上

我们指出，函数图就是**基环内向森林**

下面补充一些图论概念：

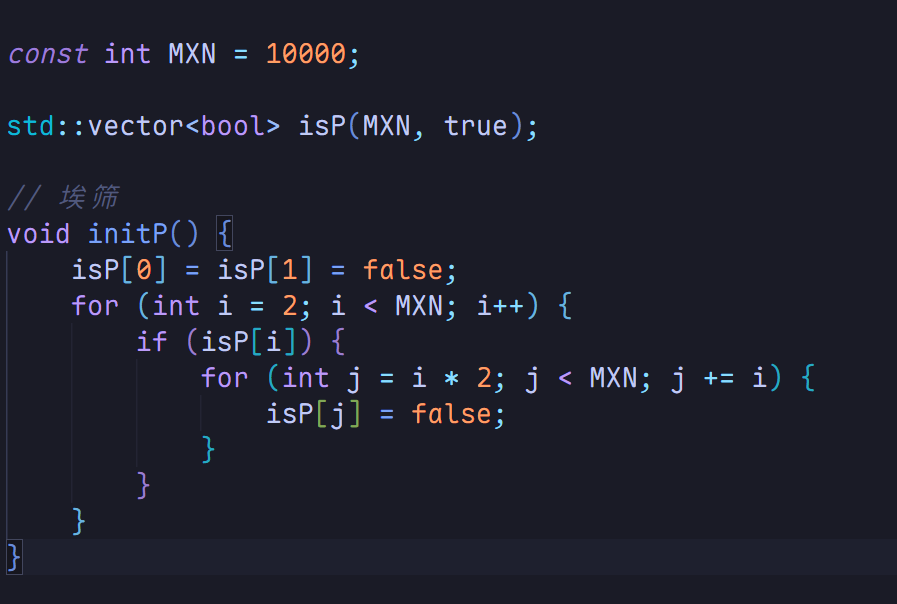




29.筛法

（1）埃筛

即 Eratosthenes 筛法，简称埃氏筛或埃筛，用于求范围内所有的素数



若将循环部分改为

for (ll i = 2; i \* i < MXN; i++) {

if (isP[i]) {

for (ll j = i \* i; j < MXN; j += i) {

isP[j] = false;

}

}

}

可能会有微小的优化（常数级别）

30.加权随机抽样导出的一个概率恒等式



上面的证明，是设定每次抽取到某球的概率与权值成正比，容易看出这是不妨的