# MODEL

## 字符串

### KMP

inline void Getnext(LL next[], char t[])  
{  
 LL p1 = 0;  
 LL p2 = next[0] = -1;  
 LL strlen\_t = strlen(t);  
 while (p1 < strlen\_t)  
 {  
 if (p2 == -1 || t[p1] == t[p2])  
 next[++p1] = ++p2;  
 else  
 p2 = next[p2];  
 }  
}  
  
inline void KMP(char string[], char pattern[], LL next[])  
{  
 LL p1 = 0;  
 LL p2 = 0;  
 LL strlen\_string = strlen(string);  
 LL strlen\_pattern = strlen(pattern);  
 while (p1 < strlen\_string)  
 {  
 if (p2 == -1 || string[p1] == pattern[p2])  
 p1++, p2++;  
 else  
 p2 = next[p2];  
 if (p2 == strlen\_pattern)  
 printf("%lld\n", p1 - strlen\_pattern + 1), p2 = next[p2];  
 }  
}

### EXKMP

string pattern;  
string s;  
LL nxt[EXKMPM];  
LL extend[EXKMPM];  
  
void getNEXT(string &pattern, LL next[])  
{  
 LL pLen = pattern.length();  
 LL a = 0, k = 0;  
  
 next[0] = pLen;  
 for (auto i = 1; i < pLen; i++)  
 {  
 if (i >= k || i + next[i - a] >= k)  
 {  
 if (i >= k)  
 k = i;  
 while (k < pLen && pattern[k] == pattern[k - i])  
 k++;  
 next[i] = k - i;  
 a = i;  
 }  
 else  
 {  
 next[i] = next[i - a];  
 }  
 }  
}  
  
void EXKMP(string &s, string &pattern, LL extend[], LL next[]) // string类得配O2不然过不了  
{  
 LL pLen = pattern.length();  
 LL sLen = s.length();  
 LL a = 0, k = 0;  
  
 getNEXT(pattern, next);  
  
 for (auto i = 0; i < sLen; i++)  
 {  
 if (i >= k || i + next[i - a] >= k)  
 {  
 if (i >= k)  
 k = i;  
 while (k < sLen && k - i < pLen && s[k] == pattern[k - i])  
 k++;  
 extend[i] = k - i;  
 a = i;  
 }  
 else  
 {  
 extend[i] = next[i - a];  
 }  
 }  
}

### AC机

#define Aho\_CorasickAutomaton 2000010  
#define CharacterCount 26  
struct TrieNode  
{  
 TrieNode \*son[CharacterCount], \*fail;  
 // LL word\_count;  
 LL logs;  
  
} T[Aho\_CorasickAutomaton];  
vector<TrieNode \*> FailEdge[Aho\_CorasickAutomaton];  
LL AC\_counter = 0;  
  
vector<TrieNode \*> trieIndex;  
  
TrieNode \*insertWords(string &s)  
{  
 auto root = &T[0];  
 for (auto i : s)  
 {  
 auto nxt = i - 'a';  
 if (root->son[nxt] == NULL)  
 root->son[nxt] = &T[++AC\_counter];  
 root = root->son[nxt];  
 }  
 // word\_count[root]++;  
  
 return root; // 返回含单词的节点号  
} // 用例：trieIndex.push\_back(insertWords(s));  
  
TrieNode \*insertWords(char \*s, LL &sLen)  
{  
 auto root = &T[0];  
 for (auto i = 0; i < sLen; i++)  
 {  
 auto nxt = s[i] - 'a';  
 if (root->son[nxt] == NULL)  
 root->son[nxt] = &T[++AC\_counter];  
 root = root->son[nxt];  
 }  
 // word\_count[root]++;  
  
 return root; // 返回含单词的节点号  
}  
  
void getFail()  
{  
 queue<TrieNode \*> Q; // bfs用  
 for (auto i = 0; i < CharacterCount; i++)  
 {  
 if (T[0].son[i] != NULL)  
 {  
 T[0].son[i]->fail = &T[0];  
 Q.push(T[0].son[i]);  
 }  
 }  
 while (!Q.empty())  
 {  
 auto now = Q.front();  
 Q.pop();  
 now->fail = now->fail == NULL ? &T[0] : now->fail;  
 for (auto i = 0; i < CharacterCount; i++)  
 {  
 if (now->son[i] != NULL)  
 {  
 now->son[i]->fail = now->fail->son[i];  
 Q.push(now->son[i]);  
 }  
 else  
 {  
 now->son[i] = now->fail->son[i];  
 }  
 }  
 }  
} // 先设T[0].fail=0;所有单词插完以后调用一次  
  
LL query(string &s)  
{  
 auto now = &T[0];  
 auto ans = 0;  
 for (auto i : s)  
 {  
 now = now->son[i - 'a'];  
 now = now == NULL ? &T[0] : now;  
 now->logs++;  
 // for (auto j = now; j /\*&& ~word\_count[j]\*/; j = fail[j])  
 // {  
 // // ans += word\_count[j];  
 // // cout << "j:" << j << endl;  
 // // if (word\_count[j])  
 // logs[j]++;  
 // // for (auto k : word\_position[j])  
 // // pattern\_count[k]++;  
 // // word\_count[j] = -1; // 标记已经遍历的节点  
 // }  
 }  
  
 for (auto i = 1; i <= AC\_counter; i++)  
 {  
 FailEdge[T[i].fail - T].push\_back(&T[i]);  
 }  
  
 return ans;  
} // 查询母串，getFail后使用一次  
  
LL query(char \*s, LL &sLen)  
{  
 auto now = &T[0];  
 auto ans = 0;  
 for (auto i = 0; i < sLen; i++)  
 {  
 now = now->son[s[i] - 'a'];  
 now = now == NULL ? &T[0] : now;  
 now->logs++;  
 // for (auto j = now; j /\*&& ~word\_count[j]\*/; j = fail[j])  
 // {  
 // ans += word\_count[j];  
 // cout << "j:" << j << endl;  
 // if (word\_count[j])  
  
 // for (auto k : word\_position[j])  
 // pattern\_count[k]++;  
 // word\_count[j] = -1; // 标记已经遍历的节点  
 // }  
 }  
  
 for (auto i = 1; i <= AC\_counter; i++)  
 {  
 FailEdge[T[i].fail - T].push\_back(&T[i]);  
 }  
  
 return ans;  
}  
  
void AC\_dfs(TrieNode \*u)  
{  
  
 for (auto i : FailEdge[u - T])  
 {  
 AC\_dfs(i);  
 u->logs += i->logs;  
 }  
} // query完后使用，一般搜0号点  
  
// 输出答案使用for(auto i:trieIndex)cout<<i.logs<<endl;这样

### 回文树

const LL M = 3e5 + 10;  
  
struct PalindromicTreeNode  
{  
 LL son[26];  
 LL suffix;  
 LL curlen;  
 LL cnt;  
 char c;  
} PTN[M];  
// char orginalString[M];  
LL PTNSIZE = 1; // SIZE - 1 actually  
LL last = 0;  
  
void \_\_init\_\_()  
{  
 PTN[0].curlen = 0;  
 PTN[0].suffix = 1;  
 PTN[0].c = '^';  
 PTN[1].c = '#';  
 PTN[1].curlen = -1;  
}  
  
LL \_\_find\_\_(LL pattern)  
{  
 while (PTN[PTNSIZE - PTN[pattern].curlen - 1].c != PTN[PTNSIZE].c)  
 pattern = PTN[pattern].suffix;  
 return pattern;  
}  
  
void \_\_add\_\_(char element)  
{  
 PTNSIZE++;  
 PTN[PTNSIZE].c = element;  
 LL offset = element - 97;  
 LL cur = \_\_find\_\_(last); // 可以加回文的点  
 if (PTN[cur].son[offset] == 0) // 没前向边这条边  
 {  
 PTN[PTNSIZE].suffix = PTN[\_\_find\_\_(PTN[cur].suffix)].son[offset]; // 正在插入的字母的后缀边不可能是cur，所以要用chk往下找合法的  
 PTN[cur].son[offset] = PTNSIZE; // 这才是加前向边  
 PTN[PTNSIZE].curlen = PTN[cur].curlen + 2;  
 }  
 last = PTN[cur].son[offset]; // 加过边以后last就是PTNSIZE  
 PTN[last].cnt++;  
}  
  
LL \_\_count\_\_()  
{  
 LL re = 0;  
 for (LL i = PTNSIZE; i >= 0; i--)  
 {  
 PTN[PTN[i].suffix].cnt += PTN[i].cnt; // 后缀边连接的节点走过次数要加上前面更高级的回文串节点走过次数  
 re = max(re, PTN[i].curlen); // 统计最长回文串长度  
 }  
 return re;  
}  
  
int main()  
{  
 ios::sync\_with\_stdio(false);  
 cin.tie(0);  
 cout.tie(0);  
 \_\_init\_\_();  
 string ss;  
 cin >> ss;  
 for (auto s : ss)  
 \_\_add\_\_(s);  
 \_\_count\_\_();  
 LL ans = 0;  
 for (auto i = 2; i <= PTNSIZE; i++)  
 {  
 ans = max(ans, PTN[i].cnt \* PTN[i].curlen); // 最长回文子串  
 }  
 cout << ans << '\n';  
 return 0;  
}

### 附：快速IO

// char buf[1<<23],\*p1=buf,\*p2=buf,obuf[1<<23],\*O=obuf; // 或者用fread更难调的快读  
// #define getchar() (p1==p2&&(p2=(p1=buf)+fread(buf,1,1<<21,stdin),p1==p2)?EOF:\*p1++)  
  
template <class T>  
void print(T x)  
{  
 if (x < 0)  
 {  
 x = -x;  
 putchar('-');  
 // \*O++ = '-';  
 }  
 if (x > 9)  
 print(x / 10);  
 putchar(x % 10 + '0');  
 // \*O++ = x%10 + '0'  
}  
// fwrite(obuf,O-obuf,1,stdout);  
  
template <class T>  
inline void qr(T &n)  
{  
 n = 0;  
 int c = getchar();  
 bool sgn = 0;  
  
 while (!isdigit(c))  
 {  
 if (c == '-')  
 sgn ^= 1;  
 c = getchar();  
 }  
  
 while (isdigit(c))  
 {  
 n = (n \* 10) + (c ^ 0x30);  
 c = getchar();  
 }  
  
 if (sgn)  
 n = -n;  
}  
  
inline char qrc()  
{  
 register char c = getchar();  
 while (c < 'a' || c > 'z')  
 c = getchar();  
 return c;  
}

### 附：没用的优化

#pragma GCC optimize(3)  
#pragma GCC target("avx")

## 图论

### Floyd

for (k = 1; k <= n; k++) {  
 for (i = 1; i <= n; i++) {  
 for (j = 1; j <= n; j++) {  
 f[i][j] = min(f[i][j], f[i][k] + f[k][j]);  
 }  
 }  
}

### Tarjan全家桶

struct Tarjan  
{  
 std::vector<int> DFN, LOW;  
 std::vector<int> belongs;  
 std::vector<int> DFS\_from; // 记父亲节点  
 std::vector<std::vector<int>> &E; // 根据实际情况选择set还是vector  
 std::vector<char> in\_stack;  
 std::stack<int> stk;  
 std::vector<int> changed; // 被缩点的点  
 int ts;  
 // std::set<int> cut; // 割点  
 int N;  
 int remaining\_point\_ctr;  
  
 /\* 构造函数确定边引用 \*/  
 Tarjan(int \_siz, std::vector<std::vector<int>> &\_E) : E(\_E), N(\_siz + 1) {}  
  
 /\* 类并查集路径压缩寻找SCC代表节点 \*/  
 inline int chk\_belongs(int x)  
 {  
 if (belongs[x] == x)  
 return x;  
 else  
 return belongs[x] = chk\_belongs(belongs[x]);  
 }  
 /\* 为多次运行准备的初始化函数 \*/  
 void init()  
 {  
 DFN.assign(N, 0);  
 LOW.assign(N, 0);  
 DFS\_from.assign(N, 0);  
 belongs.assign(N, 0);  
 in\_stack.assign(N, 0);  
 ts = 0;  
 }  
  
 /\* 入口 \*/  
 void run()  
 {  
 init();  
 for (auto i : range(1, DFN.size()))  
 if (!DFN[i])  
 tarjan(i, i);  
 remaining\_point\_ctr = N - 1 - changed.size();  
 }  
 /\* 内部用，x==f时表示本节点为根节点 \*/  
 inline void tarjan(int x, int f) // 本意是处理无向图  
 {  
 DFS\_from[x] = f;  
 DFN[x] = LOW[x] = ++ts;  
 in\_stack[x] = 1;  
 stk.push(x);  
 if (x == f) // 本节点为根  
 {  
 // set<int> realson;  
 for (auto &i : E[x])  
 {  
 if (!DFN[i])  
 {  
 tarjan(i, x);  
 LOW[x] = min(LOW[x], LOW[i]);  
 // if (realson.size() < 2)  
 // realson.insert(LOW[i]);  
 }  
 else if (in\_stack[i])  
 LOW[x] = min(LOW[x], DFN[i]);  
 }  
 // if (realson.size() >= 2)  
 // cut.insert(x);  
 }  
 else  
 {  
 for (auto &i : E[x])  
 {  
 // if (i != f) // 无向图这么写  
 if (1)  
 {  
 if (!DFN[i])  
 {  
 tarjan(i, x);  
 LOW[x] = min(LOW[x], LOW[i]);  
 // if (LOW[i] >= DFN[x])  
 // cut.insert(x);  
 }  
 else if (in\_stack[i])  
 LOW[x] = min(LOW[x], DFN[i]);  
 }  
 }  
 }  
 if (DFN[x] == LOW[x])  
 {  
 while (stk.size())  
 {  
 int tp = stk.top();  
 in\_stack[tp] = 0;  
 stk.pop();  
 belongs[tp] = x;  
 if (x != tp)  
 changed.push\_back(tp);  
 if (tp == x)  
 break;  
 }  
 }  
 }  
 /\* 注意这步还没有完全更新边，遍历边时必须使用auto i: E[x], belongs[i]，或使用下面的合并边 \*/  
 void do\_merge()  
 {  
 for (auto i : changed)  
 {  
 int fi = chk\_belongs(i);  
 for (auto j : E[i])  
 {  
 int fj = chk\_belongs(j);  
 if (fi != fj)  
 E[fi].emplace\_back(fj);  
 }  
 E[i].clear(); // 清掉已经被缩点的点上的边  
 }  
 changed.clear();  
 }  
  
 /\* 主动将合并后的边整理并去重，多次使用可能TLE \*/  
 void handle\_merged\_edge()  
 {  
 for (auto i : range(1, N))  
 {  
 if (E[i].size())  
 {  
 update\_point(i);  
 }  
 }  
 }  
  
 inline void update\_point(int x)  
 {  
 std::unordered\_set<int> tmpe;  
 for (auto j : E[x])  
 {  
 int fj = chk\_belongs(j);  
 if (fj != x)  
 tmpe.emplace(fj);  
 }  
 E[x].clear();  
 for (auto j : tmpe)  
 E[x].emplace\_back(j);  
 // swap(E[x], tmpe);  
 }  
  
 /\* 仅加一条边的缩点，在已经跑过上面的缩点之后使用，为了保证复杂度实际上只维护了一个并查集 \*/  
 void single\_edge\_SCC(int u, int v)  
 {  
 u = chk\_belongs(u);  
 v = chk\_belongs(v);  
 int father;  
 while (u != v)  
 {  
 if (DFN[u] < DFN[v])  
 swap(u, v);  
 changed.push\_back(u);  
 u = chk\_belongs(DFS\_from[u]);  
 }  
 for (auto i : changed)  
 {  
 belongs[i] = u;  
 // remaining\_points.erase(i);  
 }  
 remaining\_point\_ctr -= changed.size();  
 // do\_merge();  
 changed.clear();  
 }  
};

### 网络流

#### 最大流-HLPP+黑魔法优化

/\* 除非卡时不然别用的预流推进桶排序优化黑魔法，用例如下  
signed main()  
{  
 qr(HLPP::n);  
 qr(HLPP::m);  
 qr(HLPP::src);  
 qr(HLPP::dst);  
 while (HLPP::m--)  
 {  
 LL t1, t2, t3;  
 qr(t1);  
 qr(t2);  
 qr(t3);  
 HLPP::add(t1, t2, t3);  
 }  
 cout << HLPP::hlpp(HLPP::n + 1, HLPP::src, HLPP::dst) << endl;  
 return 0;  
}  
\*/  
namespace HLPP  
{  
 const LL INF = 0x3f3f3f3f3f3f;  
 const LL MXn = 1203;  
 const LL maxm = 520010;  
  
 vector<LL> gap;  
 LL n, m, src, dst, now\_height, src\_height;  
  
 struct NODEINFO  
 {  
 LL height = MXn, traffic;  
 LL getIndex();  
 NODEINFO(LL h = 0) : height(h) {}  
 bool operator<(const NODEINFO &a) const { return height < a.height; }  
 } node[MXn];  
  
 LL NODEINFO::getIndex() { return this - node; }  
  
 struct EDGEINFO  
 {  
 LL to;  
 LL flow;  
 LL opposite;  
 EDGEINFO(LL a, LL b, LL c) : to(a), flow(b), opposite(c) {}  
 };  
 std::list<NODEINFO \*> dlist[MXn];  
 vector<std::list<NODEINFO \*>::iterator> iter;  
 vector<NODEINFO \*> list[MXn];  
 vector<EDGEINFO> edge[MXn];  
  
 inline void add(LL u, LL v, LL w = 0)  
 {  
 edge[u].push\_back(EDGEINFO(v, w, (LL)edge[v].size()));  
 edge[v].push\_back(EDGEINFO(u, 0, (LL)edge[u].size() - 1));  
 }  
  
 priority\_queue<NODEINFO> PQ;  
 inline bool prework\_bfs(NODEINFO &src, NODEINFO &dst, LL &n)  
 {  
 gap.assign(n, 0);  
 for (auto i = 0; i <= n; i++)  
 node[i].height = n;  
 dst.height = 0;  
 queue<NODEINFO \*> q;  
 q.push(&dst);  
 while (!q.empty())  
 {  
 NODEINFO &top = \*(q.front());  
 for (auto i : edge[&top - node])  
 {  
 if (node[i.to].height == n and edge[i.to][i.opposite].flow > 0)  
 {  
 gap[node[i.to].height = top.height + 1]++;  
 q.push(&node[i.to]);  
 }  
 }  
 q.pop();  
 }  
  
 return src.height == n;  
 }  
  
 inline void relabel(NODEINFO &src, NODEINFO &dst, LL &n)  
 {  
 prework\_bfs(src, dst, n);  
 for (auto i = 0; i <= n; i++)  
 list[i].clear(), dlist[i].clear();  
  
 for (auto i = 0; i <= n; i++)  
 {  
 NODEINFO &u = node[i];  
 if (u.height < n)  
 {  
 iter[i] = dlist[u.height].insert(dlist[u.height].begin(), &u);  
 if (u.traffic > 0)  
 list[u.height].push\_back(&u);  
 }  
 }  
 now\_height = src\_height = src.height;  
 }  
  
 inline bool push(NODEINFO &u, EDGEINFO &dst) // 从x到y尽可能推流，p是边的编号  
 {  
 NODEINFO &v = node[dst.to];  
 LL w = min(u.traffic, dst.flow);  
 dst.flow -= w;  
 edge[dst.to][dst.opposite].flow += w;  
 u.traffic -= w;  
 v.traffic += w;  
 if (v.traffic > 0 and v.traffic <= w)  
 list[v.height].push\_back(&v);  
 return u.traffic;  
 }  
  
 inline void push(LL n, LL ui)  
 {  
 auto new\_height = n;  
 NODEINFO &u = node[ui];  
 for (auto &i : edge[ui])  
 {  
 if (i.flow)  
 {  
 if (u.height == node[i.to].height + 1)  
 {  
 if (!push(u, i))  
 return;  
 }  
 else  
 new\_height = min(new\_height, node[i.to].height + 1); // 抬到正好流入下一个点  
 }  
 }  
 auto height = u.height;  
 if (gap[height] == 1)  
 {  
 for (auto i = height; i <= src\_height; i++)  
 {  
 for (auto it : dlist[i])  
 {  
 gap[(\*it).height]--;  
 (\*it).height = n;  
 }  
 dlist[i].clear();  
 }  
 src\_height = height - 1;  
 }  
 else  
 {  
 gap[height]--;  
 iter[ui] = dlist[height].erase(iter[ui]);  
 u.height = new\_height;  
 if (new\_height == n)  
 return;  
 gap[new\_height]++;  
 iter[ui] = dlist[new\_height].insert(dlist[new\_height].begin(), &u);  
 src\_height = max(src\_height, now\_height = new\_height);  
 list[new\_height].push\_back(&u);  
 }  
 }  
  
 inline LL hlpp(LL n, LL s, LL t)  
 {  
 if (s == t)  
 return 0;  
 now\_height = src\_height = 0;  
 NODEINFO &src = node[s];  
 NODEINFO &dst = node[t];  
 iter.resize(n);  
 for (auto i = 0; i < n; i++)  
 if (i != s)  
 iter[i] = dlist[node[i].height].insert(dlist[node[i].height].begin(), &node[i]);  
 gap.assign(n, 0);  
 gap[0] = n - 1;  
 src.traffic = INF;  
 dst.traffic = -INF; // 上负是为了防止来自汇点的推流  
 for (auto &i : edge[s])  
 push(src, i);  
 src.traffic = 0;  
 relabel(src, dst, n);  
 for (LL ui; now\_height >= 0;)  
 {  
 if (list[now\_height].empty())  
 {  
 now\_height--;  
 continue;  
 }  
 NODEINFO &u = \*(list[now\_height].back());  
 list[now\_height].pop\_back();  
 push(n, &u - node);  
 }  
 return dst.traffic+INF;  
 }  
}

#### 最小费用最大流（MCMF）

zkw 暴力spfa（板题更快）

#include "Headers.cpp"  
  
/\* 2021.7.23 完全使用vector版本，SPFA使用SLF优化 \*/  
template <typename T>  
struct MCMF // 费用流(Dinic)zkw板子  
{ // Based on Dinic (zkw)  
  
 typedef long long LL;  
 T INF;  
 int N = 1e5 + 5; // 最大点meta参数，要按需改  
#define \_N 10006  
 std::bitset<\_N> vis; // 要一起改  
 std::vector<T> Dis;  
 int s, t; // 源点，汇点需要外部写入  
 std::vector<int> Cur; // 当前弧优化用  
 T maxflow, mincost; // 放最终答案  
  
 struct EdgeContent  
 {  
 int to;  
 T flow;  
 T cost;  
 int dualEdge;  
 EdgeContent(int a, T b, T c, int d) : to(a), flow(b), cost(c), dualEdge(d) {}  
 };  
  
 std::vector<std::vector<EdgeContent>> E;  
  
 /\* 构造函数，分配内存 \*/  
 MCMF(int n)  
 {  
 N = n;  
 E.assign(n + 1, std::vector<EdgeContent>());  
 Dis.assign(n + 1, 0);  
 Cur.assign(n + 1, 0);  
 maxflow = mincost = 0;  
 memset(&INF, 0x3f, sizeof(INF));  
 }  
  
 void add(int u, int v, T f, T w) // 加一条u到v流为f单位费为w的边  
 {  
 E[u].emplace\_back(v, f, w, E[v].size());  
 E[v].emplace\_back(u, 0, -w, E[u].size() - 1);  
 }  
  
 bool SPFA()  
 {  
 std::deque<int> Q;  
 Q.emplace\_back(s);  
 // memset(Dis, INF, sizeof(T) \* (N + 1));  
 Dis.assign(N + 1, INF);  
 Dis[s] = 0;  
 int k;  
 while (!Q.empty())  
 {  
 k = Q.front();  
 Q.pop\_front();  
 vis.reset(k);  
 // for (auto [to, f, w, rev] : E[k])s  
 for (auto &i : E[k])  
 {  
 auto &to = i.to;  
 auto &f = i.flow;  
 auto &w = i.cost;  
 auto &rev = i.dualEdge;  
 if (f and Dis[k] + w < Dis[to])  
 {  
 Dis[to] = Dis[k] + w;  
 if (!vis.test(to))  
 {  
 if (Q.size() and Dis[Q.front()] > Dis[to])  
 {  
 Q.emplace\_front(to);  
 }  
 else  
 Q.emplace\_back(to);  
 vis.set(to);  
 }  
 }  
 }  
 }  
 return Dis[t] != INF;  
 }  
 T DFS(int k, T flow)  
 {  
 if (k == t)  
 {  
 maxflow += flow;  
 return flow;  
 }  
 T sum = 0;  
 vis.set(k);  
 for (auto i = Cur[k]; i < E[k].size(); i++)  
 {  
 auto &to = E[k][i].to;  
 auto &f = E[k][i].flow;  
 auto &w = E[k][i].cost;  
 auto &rev = E[k][i].dualEdge;  
 // auto &[to, f, w, rev] = E[k][i];  
 if (!vis.test(to) and f and Dis[to] == Dis[k] + w)  
 {  
 Cur[k] = i;  
 T p = DFS(to, std::min(flow - sum, f));  
 sum += p;  
 f -= p;  
 E[to][rev].flow += p;  
 mincost += p \* w;  
 if (sum == flow)  
 break;  
 }  
 }  
 vis.reset(k);  
 return sum;  
 }  
  
 void Dinic() // 入口  
 {  
 while (SPFA())  
 {  
 // memset(Cur, 0, sizeof(int) \* (N + 1));  
 Cur.assign(N + 1, 0);  
 DFS(s, INF);  
 }  
 }  
};

#### MCMF Type2

重贴标号，常数大，除第一次外其它全在正权边上，建议SLF优化SPFA。

实测板题跑dj不如跑SLF SPFA

/\*   
 2021.10.26 原始对偶版本，除了第一次最短路外之后的最短路都运行在非负图上，可以使用dij   
 但板题表现是始终使用SPFA+SLF最优   
\*/  
template <typename Cap, typename Cost = Cap>  
struct MCMFDUAL // 费用流(Dinic)zkw原始对偶板子  
{ // dij开关：first\_spfa  
 // 非堆开关: #define not\_use\_heap  
 // SLF优化：自己改=\_=\\\  
  
 typedef long long LL;  
 Cap INF;  
 Cost CINF;  
 int N; // 最大点meta参数，要按需改  
 // #define \_N 10006  
 std::vector<char> vis; // 要一起改  
 std::vector<Cost> Dis;  
 int s, t; // 源点，汇点需要外部写入  
 std::vector<int> Cur; // 当前弧优化用  
 Cap maxflow;  
 Cost mincost; // 放最终答案  
 Cost D;   
 bool first\_spfa;  
  
 struct EdgeContent  
 {  
 int to;  
 Cap flow;  
 Cost cost;  
 int dualEdge;  
 EdgeContent(int a, Cap b, Cost c, int d) : to(a), flow(b), cost(c), dualEdge(d) {}  
 };  
  
 std::vector<std::vector<EdgeContent>> E; // 边数组  
  
 /\* 构造函数，分配内存 \*/  
 MCMFDUAL(int n)  
 {  
 N = n;  
 D = 0;  
 E.assign(n + 1, std::vector<EdgeContent>());  
 Dis.assign(n + 1, 0);  
 vis.assign(n + 1, 0);  
 Cur.assign(n + 1, 0);  
 maxflow = mincost = 0;  
 memset(&INF, 0x3f, sizeof(INF));  
 memset(&CINF, 0x3f, sizeof(CINF));  
 first\_spfa = true;  
 }  
  
 void add(int u, int v, Cap f, Cost w) // 加一条u到v流为f单位费为w的边  
 {  
 E[u].emplace\_back(v, f, w, E[v].size());  
 E[v].emplace\_back(u, 0, -w, E[u].size() - 1);  
 }  
  
 bool SPFA()  
 {  
 // memset(Dis, INF, sizeof(T) \* (N + 1));  
 Dis.assign(N + 1, CINF);  
 Dis[s] = 0;  
 int k;  
  
 // if(first\_spfa)  
 if (first\_spfa)  
 {  
 std::vector<char> inqueue(N + 1, 0);  
 // first\_spfa = false;  
 // std::queue<int> Q;  
 // Q.emplace(s);  
 // std::deque<int> Q;  
 // vector实现循环队列可以快0.5%  
 int qsiz = N + 1;  
 std::vector<int> Q(qsiz);  
 int lptr = 0;  
 int rptr = 0;  
  
 // Q.emplace\_back(s);  
 Q[rptr++] = s;  
  
 inqueue[s] = 1;  
 while (lptr != rptr)  
 // while (Q.size())  
 {  
 k = Q[lptr++];  
 if (lptr >= (qsiz))  
 lptr = 0;  
 // k = Q.front();  
 // Q.pop();  
 // Q.pop\_front();  
 for (auto &i : E[k])  
 {  
 auto &to = i.to;  
 auto &f = i.flow;  
 auto &w = i.cost;  
 // auto &rev = i.dualEdge;  
 if (f and Dis[k] + w < Dis[to])  
 {  
 Dis[to] = Dis[k] + w;  
 if (!inqueue[to])  
 {  
 // if (Q.size() and Dis[Q.front()] >= Dis[to])  
 if (lptr != rptr and Dis[Q[lptr]] >= Dis[to])  
 {  
 // Q.emplace\_front(to);  
 if (--lptr < 0)  
 lptr += qsiz;  
 Q[lptr] = to;  
 }  
 else  
 {  
 // Q.emplace\_back(to);  
 Q[rptr++] = to;  
 if (rptr >= (qsiz))  
 rptr = 0;  
 }  
 // Q.emplace(to);  
 inqueue[to] = 1;  
 }  
 }  
 }  
 inqueue[k] = 0;  
 }  
 }  
 else  
 {  
 std::vector<char> dvis(N + 1, 0);  
#ifndef not\_use\_heap  
 struct elem  
 {  
 int x;  
 Cost k;  
 bool operator<(const elem &b) const { return k > b.k; };  
 elem(int px, Cost key) : x(px), k(key) {}  
 };  
 std::priority\_queue<elem> Q;  
 Q.emplace(s, Dis[s]);  
 while (Q.size())  
 {  
 k = Q.top().x;  
 dvis[k] = 1;  
 Q.pop();  
 for (auto &i : E[k])  
 {  
 auto &to = i.to;  
 auto &f = i.flow;  
 auto &w = i.cost;  
 // auto &rev = i.dualEdge;  
 if (f and Dis[k] + w < Dis[to])  
 {  
 Dis[to] = Dis[k] + w;  
 if (!dvis[to])  
 Q.emplace(to, Dis[to]);  
 }  
 }  
 }  
#else  
 // 非堆  
  
 int ato = N + 1;  
 while (ato--)  
 {  
 // auto kpos = max\_element(ato.begin(), ato.end(), [&](const int &a, const int &b) -> bool  
 // { return Dis[a] > Dis[b]; });  
 // int k = \*kpos;  
 // ato.erase(kpos);  
 int k = -1;  
 for (int i = 0; i <= N; ++i)  
 if (!dvis[i] and (k == -1 or Dis[i] < Dis[k]))  
 k = i;  
 dvis[k] = 1;  
 for (auto &i : E[k])  
 {  
 auto &to = i.to;  
 auto &f = i.flow;  
 auto &w = i.cost;  
 if (f and Dis[k] + w < Dis[to])  
 Dis[to] = Dis[k] + w;  
 }  
 }  
#endif  
 }  
 for (int i = 0; i <= N; ++i)  
 for (auto &j : E[i])  
 j.cost -= Dis[j.to] - Dis[i];  
 D += Dis[t];  
 return Dis[t] != CINF;  
 }  
 Cap DFS(int k, Cap flow)  
 {  
 if (k == t)  
 {  
 maxflow += flow;  
 mincost += D \* flow;  
 return flow;  
 }  
 Cap sum = 0;  
 vis[k] = 1;  
 for (auto &i = Cur[k]; i < E[k].size(); ++i)  
 {  
 auto &to = E[k][i].to;  
 auto &f = E[k][i].flow;  
 auto &w = E[k][i].cost;  
 auto &rev = E[k][i].dualEdge;  
 // auto &[to, f, w, rev] = E[k][i];  
 if (!vis[to] and f and !w)  
 {  
 Cap p = DFS(to, std::min(flow - sum, f));  
 sum += p;  
 f -= p;  
 E[to][rev].flow += p;  
 if (sum == flow)  
 break;  
 }  
 }  
 return sum;  
 }  
  
 void Dinic() // 入口  
 {  
 while (SPFA())  
 {  
 do  
 {  
 vis.assign(N + 1, 0);  
 Cur.assign(N + 1, 0);  
 } while (DFS(s, INF));  
 }  
 }  
};

## 数据结构

### 主席树

namespace Persistent\_seg  
{  
/\* 指定宏use\_ptr使用指针定位左右儿子，指针可能会被搬家表传统艺能影响导致找不到地址 \*/  
#ifdef use\_ptr  
// using P = Node<T> \*;  
#define P Node<T> \*  
 P NIL = nullptr;  
#else  
 using P = int;  
 P NIL = -1;  
#endif  
 template <class T>  
 struct Node  
 {  
 T v, alz, mlz;  
 P l = NIL;  
 P r = NIL;  
 Node() : v(0), alz(0), mlz(1) {}  
 Node(T \_v) : v(\_v), alz(0), mlz(1) {}  
 };  
 inline int mid(int l, int r) { return l + r >> 1; }  
 /\* 用法:构造后用auto\_reserve分配空间,然后build初始化,此时初始版本被填入H[0]中 \*/  
 template <class T>  
 struct PST\_trad  
 {  
 int QL, QR;  
 int LB, RB;  
 using ND = Node<T>;  
 std::vector<ND> D;  
 std::vector<P> H;  
 T \*refarr;  
 T TMP;  
 bool new\_version;  
 ND &resolve(P x)  
 {  
#ifdef use\_ptr  
 return \*x;  
#else  
 return D[x];  
#endif  
 }  
 P getref(ND &x)  
 {  
#ifdef use\_ptr  
 return &x;  
#else  
 return &x - &D.front();  
#endif  
 }  
 PST\_trad() {}  
 PST\_trad(int n, int m) { auto\_reserve(n, m); }  
 void auto\_reserve(int n, int m)  
 {  
 D.reserve((1 + ceil(log2(n))) \* m + 2 \* n);  
 H.reserve(m);  
 }  
  
 void maintain(ND &x)  
 {  
 ND &lson = resolve(x.l);  
 ND &rson = resolve(x.r);  
 x.v = lson.v + rson.v;  
 }  
  
 void pushdown(ND &x, int l, int r)  
 {  
 ND &lson = resolve(x.l);  
 ND &rson = resolve(x.r);  
 if (x.mlz != 1)  
 {  
 lson.v \*= x.alz;  
 lson.alz \*= x.mlz;  
 lson.mlz \*= x.mlz;  
 rson.v \*= x.alz;  
 rson.alz \*= x.mlz;  
 rson.mlz \*= x.mlz;  
 x.mlz = 1;  
 }  
 if (x.alz != 0)  
 {  
 int m = mid(l, r);  
 lson.v += x.alz \* (m - l + 1);  
 lson.alz += x.alz;  
 rson.v += x.alz \* (r - m);  
 rson.alz += x.alz;  
 x.alz = 0;  
 }  
 }  
  
 P \_build(int l, int r)  
 {  
 if (l == r)  
 {  
 if (refarr == nullptr)  
 D.emplace\_back();  
 else  
 D.emplace\_back(\*(refarr + l));  
 return getref(D.back());  
 }  
 D.emplace\_back();  
 // ND &C = ;  
 P rr = getref(D.back());  
 int m = mid(l, r);  
 resolve(rr).l = \_build(l, m);  
 resolve(rr).r = \_build(m + 1, r);  
 // cerr << "REF c:" << rr << endl;  
 return rr;  
 }  
 /\* 建默认空树可以给rf填nullptr \*/  
 void build(T \*rf, int l, int r)  
 {  
 refarr = rf;  
 LB = l;  
 RB = r;  
 H.emplace\_back(\_build(l, r));  
 }  
 P \_updatem(int l, int r, P o)  
 {  
 ND &old = resolve(o);  
 if (new\_version)  
 D.emplace\_back(old);  
 ND &C = new\_version ? D.back() : old;  
 P rr = getref(C);  
 if (QL <= l and r <= QR)  
 {  
 C.alz \*= TMP;  
 C.v \*= TMP;  
 C.mlz \*= TMP;  
 return rr;  
 }  
 pushdown(C, l, r);  
 int m = mid(l, r);  
 if (QL <= m)  
 resolve(rr).l = \_updatem(l, m, C.l);  
 if (m + 1 <= QR)  
 resolve(rr).r = \_updatem(m + 1, r, C.r);  
 maintain(C);  
 return rr;  
 }  
 /\* 区间乘法，head写时间，如果是最近一次则填H.back() \*/  
 void updatem(int l, int r, T val, P head, bool new\_ver = true)  
 {  
 TMP = val;  
 QL = l;  
 QR = r;  
 new\_version = new\_ver;  
 if (not new\_ver)  
 \_updatem(LB, RB, head);  
 else  
 H.emplace\_back(\_updatem(LB, RB, head));  
 }  
 P \_updatea(int l, int r, P o)  
 {  
 ND &old = resolve(o);  
 if (new\_version)  
 D.emplace\_back(old);  
 ND &C = new\_version ? D.back() : old;  
 P rr = getref(C);  
 if (QL <= l and r <= QR)  
 {  
 int len = r - l + 1;  
 C.alz += TMP;  
 // T tp = TMP;  
 // tp \*= len;  
 C.v += TMP \* len;  
 return rr;  
 }  
 pushdown(C, l, r);  
 int m = mid(l, r);  
 if (QL <= m)  
 C.l = \_updatea(l, m, C.l);  
 if (m + 1 <= QR)  
 C.r = \_updatea(m + 1, r, C.r);  
 maintain(C);  
 return rr;  
 }  
 /\* 区间加法，head写时间，如果是最近一次则填H.back() \*/  
 void updatea(int l, int r, T val, P head, bool new\_ver = true)  
 {  
 TMP = val;  
 QL = l;  
 QR = r;  
 new\_version = new\_ver;  
 if (not new\_ver)  
 \_updatea(LB, RB, head);  
 else  
 H.emplace\_back(\_updatea(LB, RB, head));  
 }  
 T \_query(int l, int r, P p)  
 {  
 ND &C = resolve(p);  
 if (QL <= l and r <= QR)  
 return C.v;  
 pushdown(C, l, r);  
 T res = 0;  
 int m = mid(l, r);  
 if (QL <= m)  
 res += \_query(l, m, C.l);  
 if (QR >= m + 1)  
 res += \_query(m + 1, r, C.r);  
 return res;  
 }  
  
 T query(int l, int r, P head)  
 {  
 QL = l;  
 QR = r;  
 return \_query(LB, RB, head);  
 }  
  
 /\* 从0开始的区间第k大，左开右闭，填H数组的对应位置 \*/  
 int kth(T k, P l, P r)  
 {  
 QL = LB;  
 QR = RB;  
 while (QL < QR)  
 {  
 ND &u = resolve(l);  
 ND &v = resolve(r);  
 T elem = resolve(v.l).v - resolve(u.l).v;  
 int m = mid(QL, QR);  
 if (elem > k)  
 {  
 QR = m;  
 l = u.l;  
 r = v.l;  
 }  
 else  
 {  
 QL = m + 1;  
 k -= elem;  
 l = u.r;  
 r = v.r;  
 }  
 }  
 return QL;  
 }  
 };  
};

### 线段树

双标区间平方和线段树

// 2021.10.31 线段树支持使用矩阵  
namespace Tree  
{  
#define Add0 0  
#define Mul1 1  
  
 // #define Add0 Geometry::Matrix<m998>(1, 3)  
 // #define Mul1 Geometry::SquareMatrix<m998>::eye(3)  
 template <typename T, typename Tadd = T, typename Tmul = T>  
 struct \_iNode  
 {  
 Tadd lazy\_add;  
 T sum\_content;  
 Tmul lazy\_mul;  
 // T max\_content;  
 T min\_content;  
 T sqrt\_content;  
 \_iNode() : lazy\_add(Add0), sum\_content(Add0), lazy\_mul(Mul1), min\_content(Add0), sqrt\_content(Add0) {}  
 };  
  
 template <typename T, typename Tadd = T, typename Tmul = T>  
 struct SegmentTree  
 {  
 using \_Node = \_iNode<T, Tadd, Tmul>;  
 int len; // 线段树实际节点数  
 int valid\_len; // 原有效数据长度  
 int QL, QR; // 暂存询问避免递归下传  
 Tmul MTMP;  
 Tadd ATMP;  
 std::vector<\_Node> \_D;  
 // template <typename AllocationPlaceType = void>  
 SegmentTree(int length, void \*arr = nullptr) // 构造函数只分配内存  
 {  
 valid\_len = length;  
 len = 1 << 1 + (int)ceil(log2(length));  
 \_D.resize(len);  
 }  
  
 void show()  
 {  
 std::cout << '[';  
 for (\_Node \*i = \_D.begin(); i != \_D.end(); ++i)  
 std::cout << i->sum\_content << ",]"[i == \_D.end() - 1] << " \n"[i == \_D.end() - 1];  
 }  
  
 static int mid(int l, int r) { return l + r >> 1; }  
  
 void update\_mul(int node\_l, int node\_r, int x)  
 {  
 if (QL <= node\_l and node\_r <= QR)  
 {  
 \_D[x].lazy\_add \*= MTMP;  
 \_D[x].sum\_content \*= MTMP;  
 \_D[x].lazy\_mul \*= MTMP;  
 \_D[x].min\_content \*= MTMP;  
  
 \_D[x].sqrt\_content = \_D[x].sqrt\_content \* MTMP \* MTMP;  
 }  
 else  
 {  
 push\_down(x, node\_l, node\_r);  
 int mi = mid(node\_l, node\_r);  
 if (QL <= mi)  
 update\_mul(node\_l, mi, x << 1);  
 if (QR > mi)  
 update\_mul(mi + 1, node\_r, x << 1 | 1);  
 maintain(x);  
 }  
 }  
  
 void update\_add(int node\_l, int node\_r, int x)  
 {  
 if (QL <= node\_l and node\_r <= QR)  
 {  
 int my\_length = node\_r - node\_l + 1;  
 \_D[x].lazy\_add += ATMP;  
  
 \_D[x].sqrt\_content = \_D[x].sqrt\_content + 2 \* ATMP \* \_D[x].sum\_content + (ATMP \* ATMP \* my\_length);  
  
 \_D[x].sum\_content += ATMP \* my\_length;  
 \_D[x].min\_content += ATMP;  
 }  
 else  
 {  
 push\_down(x, node\_l, node\_r);  
 int mi = mid(node\_l, node\_r);  
 if (QL <= mi)  
 update\_add(node\_l, mi, x << 1);  
 if (QR > mi)  
 update\_add(mi + 1, node\_r, x << 1 | 1);  
 maintain(x);  
 }  
 }  
  
 void range\_mul(int l, int r, const Tmul &v)  
 {  
 QL = l;  
 QR = r;  
 MTMP = v;  
 update\_mul(1, valid\_len, 1);  
 }  
  
 void range\_add(int l, int r, const Tadd &v)  
 {  
 QL = l;  
 QR = r;  
 ATMP = v;  
 update\_add(1, valid\_len, 1);  
 }  
  
 inline void maintain(int i)  
 {  
 int l = i << 1;  
 int r = l | 1;  
 \_D[i].sum\_content = (\_D[l].sum\_content + \_D[r].sum\_content);  
 \_D[i].min\_content = min(\_D[l].min\_content, \_D[r].min\_content);  
 \_D[i].sqrt\_content = (\_D[l].sqrt\_content + \_D[r].sqrt\_content);  
 }  
  
 inline void push\_down(int ind, int my\_left\_bound, int my\_right\_bound)  
 {  
 int l = ind << 1;  
 int r = l | 1;  
 int mi = mid(my\_left\_bound, my\_right\_bound);  
 int lson\_length = (mi - my\_left\_bound + 1);  
 int rson\_length = (my\_right\_bound - mi);  
 if (\_D[ind].lazy\_mul != Mul1)  
 {  
 // 区间和  
 \_D[l].sum\_content \*= \_D[ind].lazy\_mul;  
  
 \_D[r].sum\_content \*= \_D[ind].lazy\_mul;  
  
 \_D[l].lazy\_mul \*= \_D[ind].lazy\_mul;  
 \_D[l].lazy\_add \*= \_D[ind].lazy\_mul;  
  
 \_D[r].lazy\_mul \*= \_D[ind].lazy\_mul;  
 \_D[r].lazy\_add \*= \_D[ind].lazy\_mul;  
  
 // RMQ  
 \_D[l].min\_content \*= \_D[ind].lazy\_mul;  
  
 \_D[r].min\_content \*= \_D[ind].lazy\_mul;  
  
 // 平方和，依赖区间和  
 \_D[l].sqrt\_content = \_D[l].sqrt\_content \* \_D[ind].lazy\_mul \* \_D[ind].lazy\_mul;  
  
 \_D[r].sqrt\_content = \_D[r].sqrt\_content \* \_D[ind].lazy\_mul \* \_D[ind].lazy\_mul;  
  
 \_D[ind].lazy\_mul = Mul1;  
 }  
 if (\_D[ind].lazy\_add != Add0)  
 {  
 // 平方和，先于区间和处理  
 \_D[l].sqrt\_content = \_D[l].sqrt\_content + 2 \* \_D[ind].lazy\_add \* \_D[l].sum\_content + \_D[ind].lazy\_add \* \_D[ind].lazy\_add \* lson\_length;  
  
 \_D[r].sqrt\_content = \_D[r].sqrt\_content + 2 \* \_D[ind].lazy\_add \* \_D[r].sum\_content + \_D[ind].lazy\_add \* \_D[ind].lazy\_add \* rson\_length;  
  
 \_D[l].sum\_content += \_D[ind].lazy\_add \* lson\_length;  
 \_D[l].lazy\_add += \_D[ind].lazy\_add;  
 \_D[r].sum\_content += \_D[ind].lazy\_add \* rson\_length;  
 \_D[r].lazy\_add += \_D[ind].lazy\_add;  
  
 \_D[l].min\_content += \_D[ind].lazy\_add;  
 \_D[r].min\_content += \_D[ind].lazy\_add;  
 \_D[ind].lazy\_add = Add0;  
 }  
 }  
  
 void \_query\_sum(  
 T &res,  
 int node\_l,  
 int node\_r,  
 int x)  
 {  
 if (QL <= node\_l and node\_r <= QR)  
 {  
 res += \_D[x].sum\_content;  
 }  
 else  
 {  
 push\_down(x, node\_l, node\_r);  
 int mi = mid(node\_l, node\_r);  
 if (QL <= mi)  
 \_query\_sum(res, node\_l, mi, x << 1);  
 if (QR > mi)  
 \_query\_sum(res, mi + 1, node\_r, x << 1 | 1);  
 maintain(x);  
 }  
 }  
 void \_query\_min(  
 T &res,  
 int node\_l,  
 int node\_r,  
 int x)  
 {  
 if (QL <= node\_l and node\_r <= QR)  
 {  
 res = min(res, \_D[x].min\_content);  
 }  
 else  
 {  
 push\_down(x, node\_l, node\_r);  
 int mi = mid(node\_l, node\_r);  
 if (QL <= mi)  
 \_query\_min(res, node\_l, mi, x << 1);  
 if (QR > mi)  
 \_query\_min(res, mi + 1, node\_r, x << 1 | 1);  
 maintain(x);  
 }  
 }  
  
 void \_query\_sqrt(  
 T &res,  
 int node\_l,  
 int node\_r,  
 int x)  
 {  
 if (QL <= node\_l and node\_r <= QR)  
 {  
 res += \_D[x].sqrt\_content;  
 }  
 else  
 {  
 push\_down(x, node\_l, node\_r);  
 int mi = mid(node\_l, node\_r);  
 if (QL <= mi)  
 \_query\_sqrt(res, node\_l, mi, x << 1);  
 if (QR > mi)  
 \_query\_sqrt(res, mi + 1, node\_r, x << 1 | 1);  
 maintain(x);  
 }  
 }  
  
 T query\_sum(int l, int r)  
 {  
 T res = Add0;  
 QL = l;  
 QR = r;  
 \_query\_sum(res, 1, valid\_len, 1);  
 return res;  
 }  
  
 T query\_min(int l, int r)  
 {  
 T res;  
 memset(&res, 0x3f, sizeof(res));  
 QL = l;  
 QR = r;  
 \_query\_min(res, 1, valid\_len, 1);  
 return res;  
 }  
  
 T query\_sqrt(int l, int r)  
 {  
 T res = Add0;  
 QL = l;  
 QR = r;  
 \_query\_sqrt(res, 1, valid\_len, 1);  
 return res;  
 }  
 };  
}

### 轻重链剖分

struct NodeModel  
{  
 LL size, height, dfs\_order, value;  
 NodeModel \*son, \*fa, \*top;  
 vector<NodeModel \*> E;  
 void d1(NodeModel \*father)  
 {  
 this->size = 1;  
 this->son = NULL;  
 this->fa = father;  
 this->height = father->height + 1;  
 for (auto cur : this->E)  
 {  
 // auto dst = N[E[cur].to];  
 if (cur != father)  
 {  
 cur->d1(this);  
 this->size += cur->size;  
 if (this->son == NULL || cur->size > this->son->size)  
 this->son = cur;  
 }  
 }  
 }  
 void d2(NodeModel \*father, NodeModel \*k)  
 {  
 this->top = k;  
 this->dfs\_order = CNT;  
 ARRAY[CNT++] = this->value;  
 if (this->son != NULL)  
 this->son->d2(this, k);  
 for (auto i : this->E)  
 if (i != father && i != this->son)  
 i->d2(this, i);  
 }  
  
 void add\_edge(NodeModel \*dst)  
 {  
 this->E.push\_back(dst);  
 }  
} N[M << 2];  
  
LL NODECNT = 0;  
struct SegmentTreeNode  
{  
 SegmentTreeNode \*l, \*r;  
 LL value, lazy, max\_value;  
 LL query\_sum(LL rg1, LL rg2, LL operation\_l, LL operation\_r, LL add\_val)  
 {  
 if (operation\_l > operation\_r)  
 swap(operation\_l, operation\_r);  
 if (operation\_l <= rg1 && rg2 <= operation\_r)  
 return (this->value + (LL)(rg2 - rg1 + 1) \* (add\_val % mod)) % mod;  
 else  
 {  
 LL v1 = (DMID(rg1, rg2) >= operation\_l) ? this->l->query\_sum(rg1, DMID(rg1, rg2), operation\_l, operation\_r, (add\_val + this->lazy) % mod) : 0;  
 LL v2 = (DMID(rg1, rg2) + 1 <= operation\_r) ? this->r->query\_sum(DMID(rg1, rg2) + 1, rg2, operation\_l, operation\_r, (add\_val + this->lazy) % mod) : 0;  
 return ((v1 + v2) % mod + mod) % mod;  
 }  
 }  
 LL query\_max(LL rg1, LL rg2, LL operation\_l, LL operation\_r, LL add\_val)  
 {  
 if (operation\_l > operation\_r)  
 swap(operation\_l, operation\_r);  
 if (operation\_l <= rg1 && rg2 <= operation\_r)  
 return this->max\_value /\*+ add\_val\*/;  
 else  
 {  
 LL v1 = (DMID(rg1, rg2) >= operation\_l) ? this->l->query\_max(rg1, DMID(rg1, rg2), operation\_l, operation\_r, add\_val + this->lazy) : -INF;  
 LL v2 = (DMID(rg1, rg2) + 1 <= operation\_r) ? this->r->query\_max(DMID(rg1, rg2) + 1, rg2, operation\_l, operation\_r, add\_val + this->lazy) : -INF;  
 return max(v1, v2);  
 }  
 }  
 void modify(LL rg1, LL rg2, LL operation\_l, LL operation\_r, LL x) // 单点修改  
 {  
 if (operation\_l <= rg1 && rg2 <= operation\_r)  
 {  
 this->value += x \* (rg2 - rg1 + 1);  
 this->value %= mod;  
 this->lazy += x;  
 this->lazy %= mod;  
 return;  
 }  
 if (operation\_l <= DMID(rg1, rg2))  
 this->l->modify(rg1, DMID(rg1, rg2), operation\_l, operation\_r, x);  
 if (DMID(rg1, rg2) + 1 <= operation\_r)  
 this->r->modify(DMID(rg1, rg2) + 1, rg2, operation\_l, operation\_r, x);  
 this->value = this->l->value + this->r->value + this->lazy \* (rg2 - rg1 + 1);  
 this->value %= mod;  
 // this->max\_value = max(this->l->max\_value, this->r->max\_value);  
 }  
  
} T[M];  
SegmentTreeNode \*build(LL rg1, LL rg2)  
{  
 LL tmp = NODECNT++;  
 if (rg2 > rg1)  
 {  
 T[tmp].l = build(rg1, DMID(rg1, rg2));  
 T[tmp].r = build(DMID(rg1, rg2) + 1, rg2);  
 T[tmp].value = (T[tmp].l->value + T[tmp].r->value) % mod;  
 T[tmp].max\_value = max(T[tmp].l->max\_value, T[tmp].r->max\_value);  
 }  
 else  
 {  
 T[tmp].value = T[tmp].max\_value = ARRAY[rg1];  
 }  
 T[tmp].lazy = 0;  
 return &T[tmp];  
}  
/\* 路径求和 \*/  
LL query\_SUM(NodeModel \*u, NodeModel \*v)  
{  
 LL ans = 0;  
 while (u->top != v->top) // top相同应该是在一条链上  
 {  
 if (u->top->height < v->top->height)  
 swap(u, v);  
 ans += T[0].query\_sum(0, n - 1, u->top->dfs\_order, u->dfs\_order, 0);  
 ans %= mod;  
 u = u->top->fa;  
 }  
 if (v->height > u->height)  
 swap(u, v);  
 ans += T[0].query\_sum(0, n - 1, v->dfs\_order, u->dfs\_order, 0);  
 ans %= mod;  
 return (ans + mod) % mod;  
}  
/\* 路径修改 \*/  
void MODIFY(NodeModel \*u, NodeModel \*v, LL x)  
{  
 LL ans = 0;  
 while (u->top != v->top) // top相同应该是在一条链上  
 {  
 if (u->top->height < v->top->height)  
 swap(u, v);  
 T[0].modify(0, n - 1, u->top->dfs\_order, u->dfs\_order, x);  
 u = u->top->fa;  
 }  
 if (v->height > u->height)  
 swap(u, v);  
 T[0].modify(0, n - 1, v->dfs\_order, u->dfs\_order, x);  
}  
/\* 路径求极值 \*/  
LL query\_MAX(NodeModel \*u, NodeModel \*v)  
{  
 LL ans = -INF;  
 while (u->top != v->top) // top相同应该是在一条链上  
 {  
 if (u->top->height < v->top->height)  
 swap(u, v);  
 ans = max(ans, T[0].query\_max(0, n - 1, u->top->dfs\_order, u->dfs\_order, 0));  
 u = u->top->fa;  
 }  
 if (v->height > u->height)  
 swap(u, v);  
 ans = max(ans, T[0].query\_max(0, n - 1, v->dfs\_order, u->dfs\_order, 0));  
 return ans;  
}  
signed main()  
{  
 // freopen("F:/py/Heavy-Light\_Decomposition/P3384\_2.in", "r", stdin);  
 // freopen("F:/py/Heavy-Light\_Decomposition/P3384\_2.ans", "w", stdout);  
 n = qr();  
 LL m = qr();  
 LL root = qr();  
 mod = qr();  
 CNT = NODECNT = 0;  
 mem(T, 0);  
 mem(ARRAY, 0);  
 mem(N, 0);  
  
 for (auto i = 1; i <= n; i++)  
 {  
 N[i].value = qr();  
 }  
 for (auto i = 1; i < n; i++)  
 {  
 LL src = qr();  
 LL dst = qr();  
 N[src].add\_edge(&N[dst]);  
 N[dst].add\_edge(&N[src]);  
 }  
 N[root].d1(&N[0]);  
 N[root].d2(&N[0], &N[root]);  
 build(0, n - 1);  
 LL o1, o2, o3, o4;  
 while (m--)  
 {  
 o1 = qr();  
 switch (o1)  
 {  
 case 1: // 路径修改  
 o2 = qr();  
 o3 = qr();  
 o4 = qr();  
 MODIFY(&N[o2], &N[o3], o4);  
 break;  
 case 2: // 路径求和  
 o2 = qr();  
 o3 = qr();  
 printf("%lld\n", query\_SUM(&N[o2], &N[o3]) % mod);  
 break;  
 case 3: // 子树修改  
 o2 = qr();  
 o3 = qr();  
 T[0].modify(0, n - 1, N[o2].dfs\_order, N[o2].dfs\_order + N[o2].size - 1, o3);  
 break;  
 default: // 子树查询  
 o2 = qr();  
 printf("%lld\n", T[0].query\_sum(0, n - 1, N[o2].dfs\_order, N[o2].dfs\_order + N[o2].size - 1, 0) % mod);  
 break;  
 }  
 }  
 return 0;  
}

### Splay和LCT

namespace BalancedTree   
{   
/\* 指定宏use\_ptr使用指针定位左右儿子，指针可能会被搬家表传统艺能影响导致找不到地址 \*/   
#ifdef use\_ptr   
// using P = Node<T> \*;   
#define P Node<T> \*   
#else   
 using P = int;   
#endif   
 template <class T>   
 struct Node   
 {   
 // si: 虚子树信息总和   
 T v = 0, su = 0 /\*, alz = 0, mlz = 1\*/, si = 0;   
 unsigned siz = 1;   
 bool rev = 0;   
 P f;   
 P son[2];   
 Node() {}   
 };   
 inline int mid(int l, int r) { return l + r >> 1; }   
   
 template <class T>   
 struct Splay   
 {   
 using ND = Node<T>;   
 std::vector<ND> D;   
 std::vector<int> gc; // 删除节点垃圾收集   
 int siz;   
 P root;   
 P NIL; // 0号点是保留的哨兵点   
#ifdef use\_ptr   
 ;   
 inline ND &resolve(P x) { return \*x; }   
 inline P getref(ND &x) { return &x; }   
#else   
 ;   
 inline ND &resolve(P x) { return D[x]; }   
 inline P getref(ND &x) { return &x - &D.front(); }   
#endif   
   
 inline ND &father(ND &x)   
 {   
 return resolve(x.f);   
 }   
 inline ND &lson(ND &x) { return resolve(x.son[0]); }   
 inline ND &rson(ND &x) { return resolve(x.son[1]); }   
 /\* 按中序遍历往后（reversed填true来往前）移动一个节点，对pushdown不安全 \*/   
   
 inline ND &move(ND &x, bool reversed = false)   
 {   
 // if (reversed == 0)   
 // return x;   
 bool s = reversed;   
 pushdown(x);   
 if (x.son[!s] != NIL)   
 {   
 P p = x.son[!s];   
 while (resolve(p).son[s] != NIL)   
 pushdown(resolve(p)), p = resolve(p).son[s];   
 return resolve(p);   
 }   
 else   
 {   
 P p = getref(x);   
 P y = x.f;   
 while (resolve(y).son[!s] == p)   
 p = y, y = resolve(p).f;   
 if (resolve(p).son[!s] != y)   
 p = y;   
 return resolve(p);   
 }   
 }   
 /\* 注意begin不是O1的 \*/   
 // using NDP = Node \*;   
 struct iterator   
 {   
 Splay \*self;   
 P \_;   
 iterator(ND &x, Splay \*s) : \_(resolve(x)), self(s) {}   
 iterator(P x, Splay \*s) : \_(x), self(s) {}   
 // iterator(P &&x) : \_(x) {}   
 inline operator ND &() { return self->resolve(\_); }   
 inline operator P() { return \_; }   
 inline ND &operator\*() { return self->resolve(\_); }   
 inline iterator &operator++()   
 {   
 \_ = self->getref(self->move(self->resolve(\_)));   
 return \*this;   
 }   
 inline iterator &operator--()   
 {   
 \_ = self->getref(self->move(self->resolve(\_), true));   
 return \*this;   
 }   
 inline bool operator!=(const iterator &rhs) const { return \_ != rhs.\_; }   
 };   
 inline iterator begin()   
 {   
 P p = root;   
 while (resolve(p).son[0] != NIL)   
 pushdown(resolve(p)), p = resolve(p).son[0];   
 return iterator(p, this);   
 }   
 inline iterator end() { return iterator(NIL, this); }   
   
 inline void pushup(ND &x)   
 {   
 if (getref(x) == NIL)   
 return;   
 x.siz = 1 + lson(x).siz + rson(x).siz;   
 // 下面的是LCT用的   
 // 维护树链   
 // x.su = lson(x).su + rson(x).su + x.v;   
 // 维护子树   
 x.su = lson(x).su + rson(x).su + x.v + x.si;   
 }   
 inline void pinrev(ND &x)   
 {   
 std::swap(x.son[0], x.son[1]);   
 x.rev ^= 1;   
 }   
 // inline void pinmul(ND &x, const T c)   
 // {   
 // x.su \*= c;   
 // x.v \*= c;   
 // x.mlz \*= c;   
 // x.alz \*= c;   
 // }   
 // inline void pinadd(ND &x, const T c)   
 // {   
 // x.su += c \* T(x.siz);   
 // x.v += c;   
 // x.alz += c;   
 // }   
   
 inline void pushdown(ND &x)   
 {   
 // if (x.mlz != T(1))   
 // pinmul(lson(x), x.mlz), pinmul(rson(x), x.mlz), x.mlz = 1;   
 // if (x.alz)   
 // pinadd(lson(x), x.alz), pinadd(rson(x), x.alz), x.alz = 0;   
 if (x.rev)   
 {   
 if (x.son[0] != NIL)   
 pinrev(lson(x));   
 if (x.son[1] != NIL)   
 pinrev(rson(x));   
 x.rev = 0;   
 }   
 }   
 Splay(int size)   
 {   
 D.reserve(size + 1);   
 gc.reserve(size);   
 D.emplace\_back();   
 D[0].siz = D[0].rev = 0;   
 D[0].f = D[0].son[0] = D[0].son[1] = getref(D[0]);   
 root = NIL = getref(D[0]);   
 siz = 0;   
 }   
 inline ND &allocate(T val, P father)   
 {   
 ++siz;   
 if (gc.size())   
 {   
 ND &b = D[gc.back()];   
 b.si = 0;   
 b.su = b.v = val;   
 b.siz = 1;   
 b.f = father;   
 b.rev = 0;   
 b.son[0] = b.son[1] = NIL;   
 gc.pop\_back();   
 return b;   
 }   
 else   
 {   
 D.emplace\_back();   
 ND &b = D.back();   
 b.si = 0;   
 b.su = b.v = val;   
 b.siz = 1;   
 b.f = father;   
 b.rev = 0;   
 b.son[0] = b.son[1] = NIL;   
   
 return b;   
 }   
 }   
   
 inline void rotate(ND &x)   
 {   
 ND &y = resolve(x.f);   
 ND &z = resolve(y.f);   
 bool k = getref(x) == y.son[1];   
 z.son[z.son[1] == getref(y)] = getref(x);   
 x.f = getref(z);   
 y.son[k] = x.son[!k];   
 resolve(x.son[!k]).f = getref(y);   
 x.son[!k] = getref(y);   
 y.f = getref(x);   
 pushup(y);   
 pushup(x);   
 }   
 /\*将x旋为goal的儿子 \*/   
 inline void splay(ND &x, ND &goal)   
 {   
 while (x.f != getref(goal))   
 {   
 ND &y = resolve(x.f);   
 ND &z = resolve(y.f);   
 if (getref(z) != getref(goal))   
 (z.son[1] == getref(y)) ^ (y.son[1] == getref(x)) ? rotate(x) : rotate(y);   
 rotate(x);   
 }   
 if (getref(goal) == NIL)   
 root = getref(x);   
 }   
   
 T \*arr;   
 P \_build(int l, int r, P fa)   
 {   
 if (l > r)   
 return NIL;   
 int m = mid(l, r);   
 ND &C = allocate(arr[m], fa);   
 C.son[0] = \_build(l, m - 1, getref(C));   
 C.son[1] = \_build(m + 1, r, getref(C));   
 pushup(C);   
 return getref(C);   
 }   
   
 void build(T \*\_arr, int siz, int beginwith = 0)   
 {   
 arr = \_arr;   
 // siz = \_siz;   
 root = \_build(beginwith, beginwith + siz - 1, NIL);   
 }   
   
 /\* insert在维护区间reverse以后就不能用，意义不一样 \*/   
 inline void insert(const T x)   
 {   
 P u = root;   
 P ff = NIL;   
 while (u != NIL)   
 {   
 ff = u;   
 u = resolve(u).son[resolve(u).v < x];   
 }   
 ND &U = allocate(x, ff);   
 u = getref(U);   
 if (ff != NIL)   
 {   
 resolve(ff).son[resolve(ff).v < x] = u;   
 }   
 splay(U, resolve(NIL));   
 // ++siz;   
 }   
 /\* 从0开始,与键值无关,只与左右儿子的子树siz有关,若k>=树的大小则返回最靠右的点 \*/   
 inline ND &kth(int k)   
 {   
 P u = root;   
 while (1)   
 {   
 ND &U = resolve(u);   
 pushdown(U);   
 ND &ls = lson(U);   
 if (ls.siz > k)   
 u = U.son[0];   
 else if (ls.siz == k or U.son[1] == NIL)   
 return U;   
 else   
 k -= ls.siz + 1, u = U.son[1];   
 }   
 }   
   
 inline void reverse(int l, int r)   
 {   
 if (l <= 0 and r >= siz - 1)   
 {   
 pinrev(resolve(root));   
 }   
 else if (l <= 0)   
 {   
 splay(kth(r + 1), resolve(NIL));   
 pinrev(lson(resolve(root)));   
 }   
 else if (r >= siz - 1)   
 {   
 splay(kth(l - 1), resolve(NIL));   
 pinrev(rson(resolve(root)));   
 }   
 else   
 {   
 ND &L = kth(l - 1);   
 ND &R = kth(r + 1);   
 splay(L, resolve(NIL));   
 splay(R, L);   
 pinrev(lson(rson(resolve(root))));   
 }   
 }   
   
 /\* 区间平移,将原序列第[l, r](从0开始算排名)的元素移动至除开这段序列后的第k位置的左边 \*/   
 inline void translate(int l, int r, int k)   
 {   
 P cutdown;   
 if (l <= 0 and r >= siz - 1)   
 return;   
 if (l <= 0)   
 {   
 splay(kth(r + 1), resolve(NIL));   
 cutdown = resolve(root).son[0];   
 resolve(root).son[0] = NIL;   
 }   
 else if (r >= siz - 1)   
 {   
 splay(kth(l - 1), resolve(NIL));   
 cutdown = resolve(root).son[1];   
 resolve(root).son[1] = NIL;   
 }   
 else   
 {   
 ND &L = kth(l - 1);   
 ND &R = kth(r + 1);   
 splay(L, resolve(NIL));   
 splay(R, L);   
 cutdown = R.son[0];   
 R.son[0] = NIL;   
 }   
 ND &CD = resolve(cutdown);   
 pushup(father(CD));   
 pushup(father(father(CD)));   
 if (k >= siz - CD.siz)   
 {   
 splay(kth(siz - CD.siz - 1), resolve(NIL));   
 resolve(root).son[1] = cutdown;   
 CD.f = root;   
 }   
 else if (k <= 0)   
 {   
 splay(kth(0), resolve(NIL));   
 resolve(root).son[0] = cutdown;   
 CD.f = root;   
 }   
 else   
 {   
 ND &L = kth(k - 1);   
 ND &R = kth(k);   
 splay(L, resolve(NIL));   
 splay(R, L);   
 R.son[0] = cutdown;   
 CD.f = L.son[1];   
 }   
 pushup(father(CD));   
 pushup(father(father(CD)));   
 }   
   
 std::function<void(T)> tempf;   
 void \_foreach(ND &x)   
 {   
 pushdown(x);   
 if (x.son[0] != NIL)   
 \_foreach(resolve(x.son[0]));   
 if (getref(x) != NIL)   
 tempf(x.v);   
 if (x.son[1] != NIL)   
 \_foreach(resolve(x.son[1]));   
 }   
 void foreach (std::function<void(T)> F)   
 {   
 tempf = F;   
 \_foreach(resolve(root));   
 }   
 };   
   
 template <typename T>   
 struct LCT : public Splay<T>   
 {   
 // using Splay<T>::ND;   
 using ND = Node<T>;   
 using Splay<T>::getref;   
 using Splay<T>::resolve;   
 using Splay<T>::rson;   
 using Splay<T>::lson;   
 using Splay<T>::father;   
 using Splay<T>::pushup;   
 using Splay<T>::pinrev;   
 // using Splay<T>::pinadd;   
 // using Splay<T>::pinmul;   
 using Splay<T>::pushdown;   
 using Splay<T>::NIL;   
 LCT(int size) : Splay<T>(size) {}   
   
 inline bool isnot\_root(ND &x)   
 {   
 return getref(lson(father(x))) == getref(x) or getref(rson(father(x))) == getref(x);   
 }   
   
 inline void rotate(ND &x)   
 {   
 ND &y = father(x);   
 ND &z = father(y);   
 bool k = getref(x) == y.son[1];   
 P rw = x.son[!k];   
 if (isnot\_root(y))   
 z.son[z.son[1] == getref(y)] = getref(x);   
 x.son[!k] = getref(y);   
 y.son[k] = rw;   
 if (rw != NIL)   
 resolve(rw).f = getref(y);   
 y.f = getref(x);   
 x.f = getref(z);   
 pushup(y);   
 // pushup(x);   
 // pushup(z);   
 }   
   
 inline void splay(ND &x)   
 {   
 P ry = getref(x);   
 vector<P> stk(1, ry);   
 while (isnot\_root(resolve(ry)))   
 stk.emplace\_back(ry = resolve(ry).f);   
 // pushdown((resolve(ry)));   
 while (stk.size())   
 {   
 pushdown(resolve(stk.back()));   
 stk.pop\_back();   
 }   
 while (isnot\_root(x))   
 {   
 ry = x.f;   
 ND &y = resolve(ry);   
 ND &z = resolve(y.f);   
 if (isnot\_root(y))   
 rotate((y.son[0] == getref(x)) ^ (z.son[0] == ry) ? x : y);   
 rotate(x);   
 }   
 pushup(x);   
 }   
   
 inline void access(ND &x)   
 {   
 P rx = getref(x);   
 for (P ry = NIL; rx != NIL; rx = resolve(ry = rx).f)   
 {   
 splay(resolve(rx));   
 // resolve(rx).son[1] = ry;   
 // 维护虚子树改成下两句   
 resolve(rx).si += resolve(resolve(rx).son[1]).su;   
 resolve(rx).si -= resolve(resolve(rx).son[1] = ry).su;   
 //   
 pushup(resolve(rx));   
 }   
 }   
   
 inline void chroot(ND &x)   
 {   
 access(x);   
 splay(x);   
 pinrev(x);   
 }   
   
 inline ND &findroot(ND &x)   
 {   
 access(x);   
 splay(x);   
 P rx = getref(x);   
 while (resolve(rx).son[0] != NIL)   
 pushdown(resolve(rx)), rx = resolve(rx).son[0];   
 splay(resolve(rx));   
 return resolve(rx);   
 }   
 /\* 路径分离出来之后y上的su值即为x->y上路径的信息() \*/   
 inline void split(ND &x, ND &y)   
 {   
 chroot(x);   
 access(y);   
 splay(y);   
 //   
 pushup(x);   
 }   
 // inline void path\_add(ND &x, ND &y, const T c)   
 // {   
 // split(x, y);   
 // pinadd(y, c);   
 // }   
 // inline void path\_mul(ND &x, ND &y, const T c)   
 // {   
 // split(x, y);   
 // pinmul(y, c);   
 // }   
 inline T path\_query(ND &x, ND &y)   
 {   
 split(x, y);   
 return y.su;   
 }   
 inline bool link(ND &x, ND &y)   
 {   
 chroot(x);   
 if (getref(findroot(y)) != getref(x))   
 {   
 // x.f = getref(y);   
 // LCT子树   
 chroot(y);   
 resolve(x.f = getref(y)).si += x.su;   
 //   
 pushup(y);   
 return true;   
 }   
 return false;   
 }   
 inline bool cut(ND &x, ND &y)   
 {   
 chroot(x);   
 if (getref(findroot(y)) == getref(x) and y.f == getref(x) and y.son[0] == NIL)   
 {   
 y.f = x.son[1] = NIL;   
 pushup(x);   
 return true;   
 }   
 return false;   
 }   
 };   
   
};

### LCA

/\* 从1到n都可用，0是保留字 5b4026638a0f469f91d26a4ff0dee4bf \*/  
struct LCA  
{  
 std::vector<std::vector<int>> fa;  
 std::vector<int> dep, siz;  
 std::vector<std::vector<int>> &E;  
  
 /\* 构造函数分配内存，传入边数组 \*/  
 LCA(int \_siz, std::vector<std::vector<int>> &\_E) : E(\_E)  
 {  
 \_siz++;  
 fa.assign(\_siz, vector<int>(log2int(\_siz) + 1, 0));  
 dep.assign(\_siz, 0);  
 siz.assign(\_siz, 0);  
 }  
  
 void dfs(int x, int from)  
 {  
 fa[x][0] = from;  
 dep[x] = dep[from] + 1;  
 siz[x] = 1;  
 for (auto i : range(1, log2int(dep[x]) + 1))  
 fa[x][i] = fa[fa[x][i - 1]][i - 1];  
 for (auto &i : E[x])  
 if (i != from)  
 {  
 dfs(i, x);  
 siz[x] += siz[i];  
 }  
 }  
  
 /\* 传入边 \*/  
 void prework(int root)  
 {  
 // dep[root] = 1;  
 dfs(root, 0);  
 siz[0] = siz[root];  
 // for (auto &i : E[root])  
 // dfs(i, root);  
 }  
  
 /\* LCA查找 \*/  
 int lca(int x, int y)  
 {  
 if (dep[x] < dep[y])  
 swap(x, y);  
 while (dep[x] > dep[y])  
 x = fa[x][log2int(dep[x] - dep[y])];  
 if (x == y)  
 return x;  
 for (auto k : range(log2int(dep[x]), -1, -1))  
 if (fa[x][k] != fa[y][k])  
 x = fa[x][k], y = fa[y][k];  
 return fa[x][0];  
 }  
  
 /\* 拿x所在father的子树的节点数 \*/  
 int subtree\_size(int x, int father)  
 {  
 if (x == father)  
 return 0;  
 for (auto i : range(fa[x].size() - 1, -1, -1))  
 x = (dep[fa[x][i]] > dep[father] ? fa[x][i] : x);  
 return siz[x];  
 }  
  
 /\* 判断tobechk是否在from -> to的路径上 \*/  
 bool on\_the\_way(int from, int to, int tobechk)  
 {  
 int k = lca(from, to);  
 return ((lca(from, tobechk) == tobechk) or (lca(tobechk, to) == tobechk)) and lca(tobechk, k) == k;  
 }  
};

### ST表

template <typename INTEGER>  
struct STMax  
{  
 // 从0开始  
 std::vector<std::vector<INTEGER>> data;  
 STMax(int siz)  
 {  
 int upper\_pow = clz(siz) + 1;  
 data.resize(upper\_pow);  
 data.assign(upper\_pow, vector<INTEGER>());  
 data[0].assign(siz, 0);  
 }  
 INTEGER &operator[](int where)  
 {  
 return data[0][where];  
 }  
 void generate\_max()  
 {  
 for (auto j : range(1, data.size()))  
 {  
 data[j].assign(data[0].size(), 0);  
 for (long long i = 0; i + (1LL << j) - 1 < data[0].size(); i++)  
 {  
 data[j][i] = std::max(data[j - 1][i], data[j - 1][i + (1 << (j - 1))]);  
 }  
 }  
 }  
 /\*闭区间[l, r]，注意有效位从0开始\*/  
 INTEGER query\_max(int l, int r)  
 {  
 int k = 31 - \_\_builtin\_clz(r - l + 1);  
 return std::max(data[k][l], data[k][r - (1 << k) + 1]);  
 }  
};

## 计算几何

### 必要的头

namespace Geometry  
{  
 using FLOAT\_ = double;  
  
 constexpr const FLOAT\_ Infinity = INFINITY;  
 const FLOAT\_ decimal\_round = 1e-8; // 精度参数  
  
 const FLOAT\_ DEC = 1.0 / decimal\_round;  
  
 int intereps(FLOAT\_ x)  
 {  
 if (x < -decimal\_round)  
 return -1;  
 else if (x > decimal\_round)  
 return 1;  
 return 0;  
 }  
  
 const FLOAT\_ PI = acos(-1);  
 bool round\_compare(FLOAT\_ a, FLOAT\_ b) { return round(DEC \* a) == round(DEC \* b); }  
 FLOAT\_ Round(FLOAT\_ a) { return round(DEC \* a) / DEC; }  
   
 /\* 解一元二次方程，传出的x1为+delta，x2为-delta，如果无解返回两个nan \*/  
 std::pair<FLOAT\_, FLOAT\_> solveQuadraticEquation(FLOAT\_ a, FLOAT\_ b, FLOAT\_ c)  
 {  
 FLOAT\_ delta = pow(b, 2) - 4 \* a \* c;  
 if (delta < 0)  
 return std::make\_pair(nan(""), nan(""));  
 else  
 {  
 delta = sqrt(delta);  
 FLOAT\_ x1 = (-b + delta) / (2 \* a);  
 FLOAT\_ x2 = (-b - delta) / (2 \* a);  
 return std::make\_pair(x1, x2);  
 }  
 }  
}

### 分数类

template <typename PrecisionType = long long>  
struct Fraction  
{  
 PrecisionType upper, lower;  
  
 Fraction(PrecisionType u = 0, PrecisionType l = 1)  
 {  
 upper = u;  
 lower = l;  
 }  
 void normalize()  
 {  
 if (upper)  
 {  
 PrecisionType g = abs(std::\_\_gcd(upper, lower));  
 upper /= g;  
 lower /= g;  
 }  
 else  
 lower = 1;  
 if (lower < 0)  
 {  
 lower = -lower;  
 upper = -upper;  
 }  
 }  
 long double ToFloat() { return (long double)upper / (long double)lower; }  
 bool operator==(Fraction b) { return upper \* b.lower == lower \* b.upper; }  
 bool operator>(Fraction b) { return upper \* b.lower > lower \* b.upper; }  
 bool operator<(Fraction b) { return upper \* b.lower < lower \* b.upper; }  
 bool operator<=(Fraction b) { return !(\*this > b); }  
 bool operator>=(Fraction b) { return !(\*this < b); }  
 bool operator!=(Fraction b) { return !(\*this == b); }  
 Fraction operator-() { return Fraction(-upper, lower); }  
 Fraction operator+(Fraction b) { return Fraction(upper \* b.lower + b.upper \* lower, lower \* b.lower); }  
 Fraction operator-(Fraction b) { return (\*this) + (-b); }  
 Fraction operator\*(Fraction b) { return Fraction(upper \* b.upper, lower \* b.lower); }  
 Fraction operator/(Fraction b) { return Fraction(upper \* b.lower, lower \* b.upper); }  
 Fraction &operator+=(Fraction b)  
 {  
 \*this = \*this + b;  
 this->normalize();  
 return \*this;  
 }  
 Fraction &operator-=(Fraction b)  
 {  
 \*this = \*this - b;  
 this->normalize();  
 return \*this;  
 }  
 Fraction &operator\*=(Fraction b)  
 {  
 \*this = \*this \* b;  
 this->normalize();  
 return \*this;  
 }  
 Fraction &operator/=(Fraction b)  
 {  
 \*this = \*this / b;  
 this->normalize();  
 return \*this;  
 }  
 friend Fraction fabs(Fraction a) { return Fraction(abs(a.upper), abs(a.lower)); }  
 std::string to\_string() { return lower == 1 ? std::to\_string(upper) : std::to\_string(upper) + '/' + std::to\_string(lower); }  
 friend std::ostream &operator<<(std::ostream &o, Fraction a)  
 {  
 return o << "Fraction(" << std::to\_string(a.upper) << ", " << std::to\_string(a.lower) << ")";  
 }  
 friend std::istream &operator>>(std::istream &i, Fraction &a)  
 {  
 char slash;  
 return i >> a.upper >> slash >> a.lower;  
 }  
 friend isfinite(Fraction a) { return a.lower != 0; }  
 void set\_value(PrecisionType u, PrecisionType d = 1) { upper = u, lower = d; }  
};

### 二维向量

struct Vector2  
{  
 FLOAT\_ x, y;  
 Vector2(FLOAT\_ \_x, FLOAT\_ \_y) : x(\_x), y(\_y) {}  
 Vector2(FLOAT\_ n) : x(n), y(n) {}  
 // Vector2(const glm::vec2& v) : x(v.x), y(v.y) {}  
 // inline glm::vec2 toglm(){return {x, y};}  
 Vector2() : x(0.0), y(0.0) {}  
 inline Vector2 &operator=(const Vector2 &b)  
 {  
 this->x = b.x;  
 this->y = b.y;  
 return \*this;  
 }  
 inline bool operator==(const Vector2 &b) const { return round\_compare(this->x, b.x) and round\_compare(this->y, b.y); }  
 inline bool operator!=(const Vector2 &b) const { return not((\*this) == b); }  
  
 /\* 转为字符串 \*/  
 inline std::string ToString() const  
 {  
 std::ostringstream ostr;  
 ostr << "Vector2(" << this->x << ", " << this->y << ")";  
 return ostr.str();  
 }  
  
 // 省略各种运算符重载  
  
 /\* 绕原点逆时针旋转多少度 \*/  
 inline void rotate(FLOAT\_ theta, bool use\_degree = false)  
 {  
 FLOAT\_ ox = x;  
 FLOAT\_ oy = y;  
 theta = (use\_degree ? theta / 180 \* PI : theta);  
 FLOAT\_ costheta = cos(theta);  
 FLOAT\_ sintheta = sin(theta);  
 this->x = ox \* costheta - oy \* sintheta;  
 this->y = oy \* costheta + ox \* sintheta;  
 }  
  
 inline bool operator<(const Vector2 &b) const { return this->x < b.x or this->x == b.x and this->y < b.y; }  
  
 /\* 向量的平方模 \*/  
 inline FLOAT\_ sqrMagnitude() const { return pow(this->x, 2) + pow(this->y, 2); }  
 /\* 向量的模 \*/  
 inline FLOAT\_ magnitude() const { return sqrt(this->sqrMagnitude()); }  
 /\*判等\*/  
 inline bool equals(const Vector2 &b) { return (\*this) == b; }  
  
 /\* 用极坐标换算笛卡尔坐标 \*/  
 inline static Vector2 fromPolarCoordinate(const Vector2 &v, bool use\_degree = 1) { return v.toCartesianCoordinate(use\_degree); }  
  
 /\* 转为笛卡尔坐标 \*/  
 inline Vector2 toCartesianCoordinate(bool use\_degree = 1) const  
 {  
 return Vector2(  
 x \* cos(y \* (use\_degree ? PI / 180.0 : 1)),  
 x \* sin(y \* (use\_degree ? PI / 180.0 : 1)));  
 }  
 /\* 转为极坐标 \*/  
 inline Vector2 toPolarCoordinate(bool use\_degree = 1) const  
 {  
 return Vector2(  
 magnitude(),  
 toPolarAngle(use\_degree));  
 }  
  
 /\* 获取极角 \*/  
 inline FLOAT\_ toPolarAngle(bool use\_degree = 1) const { return atan2(y, x) \* (use\_degree ? 180.0 / PI : 1); }  
  
 /\* 转为极坐标 \*/  
 inline static Vector2 ToPolarCoordinate(const Vector2 &coordinate, bool use\_degree = 1) { return coordinate.toPolarCoordinate(use\_degree); }  
  
 /\* 向量单位化 \*/  
 inline void Normalize()  
 {  
 FLOAT\_ \_m = this->magnitude();  
 this->x /= \_m;  
 this->y /= \_m;  
 }  
  
 /\* 返回与该向量方向同向的单位向量 \*/  
 inline Vector2 normalized() const  
 {  
 FLOAT\_ \_m = this->magnitude();  
 return Vector2(this->x / \_m, this->y / \_m);  
 }  
 /\* 距离 \*/  
 inline static FLOAT\_ Distance(const Vector2 &a, const Vector2 &b) { return (a - b).magnitude(); }  
  
 /\* 向量线性插值 \*/  
 inline static Vector2 LerpUnclamped(const Vector2 &a, const Vector2 &b, const FLOAT\_ &t) { return a + (b - a) \* t; }  
  
 /\* 向量圆形插值 \*/  
 inline static Vector2 SlerpUnclamped(Vector2 a, Vector2 b, const FLOAT\_ &t)  
 {  
 // Vector2 c = b - a;  
 a = a.toPolarCoordinate();  
 b = b.toPolarCoordinate();  
 return LerpUnclamped(a, b, t).toCartesianCoordinate();  
 }  
  
 /\* 拿它的垂直向量（逆时针旋转90°） \*/  
 inline static Vector2 Perpendicular(const Vector2 &inDirection) { return Vector2(-inDirection.y, inDirection.x); }  
 /\* 根据inNormal法向反射inDirection向量，参考光的平面镜反射，入射光为inDirection，平面镜的法线为inNormal \*/  
 inline static Vector2 Reflect(const Vector2 &inDirection, const Vector2 &inNormal) { return inDirection - 2 \* Vector2::Dot(inDirection, inNormal) \* inNormal; }  
 /\* 点积 \*/  
 inline static FLOAT\_ Dot(const Vector2 &lhs, const Vector2 &rhs) { return lhs.x \* rhs.x + lhs.y \* rhs.y; }  
 /\* 叉积 \*/  
 inline static FLOAT\_ Cross(const Vector2 &lhs, const Vector2 &rhs) { return lhs.x \* rhs.y - lhs.y \* rhs.x; }  
 /\* 有符号弧度夹角 \*/  
 inline static FLOAT\_ SignedRad(const Vector2 &from, const Vector2 &to) { return atan2(Vector2::Cross(from, to), Vector2::Dot(from, to)); }  
 /\* 无符号弧度夹角 \*/  
 inline static FLOAT\_ Rad(const Vector2 &from, const Vector2 &to) { return abs(Vector2::SignedRad(from, to)); }  
 /\* 有符号角度夹角 \*/  
 inline static FLOAT\_ SignedAngle(const Vector2 &from, const Vector2 &to) { return Vector2::SignedRad(from, to) \* 180.0 / PI; }  
 /\* 无符号角度夹角 \*/  
 inline static FLOAT\_ Angle(const Vector2 &from, const Vector2 &to) { return abs(Vector2::SignedAngle(from, to)); }  
  
 /\* 返回俩向量中x的最大值和y的最大值构造而成的向量 \*/  
 inline static Vector2 Max(const Vector2 &lhs, const Vector2 &rhs) { return Vector2(std::max(lhs.x, rhs.x), std::max(lhs.y, rhs.y)); }  
  
 /\* 返回俩向量中x的最小值和y的最小值构造而成的向量 \*/  
 inline static Vector2 Min(const Vector2 &lhs, const Vector2 &rhs) { return Vector2(std::min(lhs.x, rhs.x), std::min(lhs.y, rhs.y)); }  
  
 /\* 获得vector在onNormal方向的投影，onNormal需要单位化 \*/  
 inline static Vector2 Project(const Vector2 &vector, const Vector2 &onNormal) { return cos(Rad(vector, onNormal)) \* vector.magnitude() \* onNormal; }  
  
 inline static FLOAT\_ ProjectLength(const Vector2 &vector, const Vector2 &onNormal) { return cos(Rad(vector, onNormal)) \* vector.magnitude(); }  
};  
struct PolarSortCmp  
{  
 inline bool operator()(const Vector2 &a, const Vector2 &b) const { return a.toPolarAngle(0) < b.toPolarAngle(0); }  
};  
/\* 相等的向量可能不会贴着放，不能保证排完之后遍历一圈是旋转360°，慎用 \*/  
struct CrossSortCmp  
{  
 inline bool operator()(const Vector2 &a, const Vector2 &b) const { return Vector2::Cross(a, b) > 0; }  
};

### 三维向量

struct Vector3 // 三维向量  
{  
 FLOAT\_ x, y, z;  
 Vector3(FLOAT\_ \_x, FLOAT\_ \_y, FLOAT\_ \_z) : x(\_x), y(\_y), z(\_z) {}  
 Vector3(FLOAT\_ n) : x(n), y(n), z(n) {}  
 Vector3(const Vector2 &b) : x(b.x), y(b.y), z(0.0) {}  
 Vector3() : x(0.0), y(0.0), z(0.0) {}  
  
 inline bool operator==(const Vector3 &b) const { return round\_compare(this->x, b.x) and round\_compare(this->y, b.y) and round\_compare(this->z, b.z); }  
 inline bool operator!=(const Vector3 &b) const { return not((\*this) == b); }  
 // 省略运算符重载  
  
 /\* 向量的平方模 \*/  
 inline FLOAT\_ sqrMagnitude() const { return pow(this->x, 2) + pow(this->y, 2) + pow(this->z, 2); }  
 /\* 向量的模 \*/  
 inline FLOAT\_ magnitude() const { return sqrt(this->sqrMagnitude()); }  
 /\* 判等 \*/  
 inline bool equals(const Vector3 &b) const { return (\*this) == b; }  
 /\* 向量单位化 \*/  
 inline void Normalize()  
 {  
 FLOAT\_ \_m = this->magnitude();  
 this->x /= \_m;  
 this->y /= \_m;  
 this->z /= \_m;  
 }  
  
 /\* 转为字符串 \*/  
 inline std::string ToString() const  
 {  
 std::ostringstream ostr;  
 ostr << "Vector3(" << this->x << ", " << this->y << ", " << this->z << ")";  
 return ostr.str();  
 }  
  
 /\* 返回与该向量方向同向的单位向量 \*/  
 inline Vector3 normalized() const  
 {  
 FLOAT\_ \_m = this->magnitude();  
 return Vector3(this->x / \_m, this->y / \_m, this->z / \_m);  
 }  
 /\* 距离 \*/  
 inline static FLOAT\_ Distance(const Vector3 &a, const Vector3 &b) { return (a - b).magnitude(); }  
  
 /\* 向量线性插值 \*/  
 inline static Vector3 LerpUnclamped(const Vector3 &a, const Vector3 &b, const FLOAT\_ &t) { return a + (b - a) \* t; }  
  
 /\* 拿它的垂直向量（逆时针旋转90°） \*/  
 inline static Vector3 Perpendicular(const Vector3 &inDirection) { return Vector3(-inDirection.y, inDirection.x, 0); }  
 /\*根据inNormal法向反射inDirection向量，参考光的平面镜反射，入射光为inDirection，平面镜的法线为inNormal\*/  
 inline static Vector3 Reflect(const Vector3 &inDirection, const Vector3 &inNormal) { return inDirection - 2 \* Vector3::Dot(inDirection, inNormal) \* inNormal; }  
  
 /\* 点积 \*/  
 inline static FLOAT\_ Dot(const Vector3 &lhs, const Vector3 &rhs) { return lhs.x \* rhs.x + lhs.y \* rhs.y + lhs.z \* rhs.z; }  
 /\* 叉积 \*/  
 inline static Vector3 Cross(const Vector3 &lhs, const Vector3 &rhs) { return Vector3(lhs.y \* rhs.z - lhs.z \* rhs.y, lhs.z \* rhs.x - lhs.x \* rhs.z, lhs.x \* rhs.y - lhs.y \* rhs.x); }  
  
 /\* 无符号弧度夹角 \*/  
 inline static FLOAT\_ Rad(const Vector3 &from, const Vector3 &to) { return acos(Dot(from, to) / (from.magnitude() \* to.magnitude())); }  
  
 /\* 无符号角度夹角 \*/  
 inline static FLOAT\_ Angle(const Vector3 &from, const Vector3 &to) { return Rad(from, to) \* 180 / PI; }  
  
 /\* 返回该方向上最大不超过maxLength长度的向量 \*/  
 inline static Vector3 ClampMagnitude(const Vector3 &vector, const FLOAT\_ &maxLength)  
 {  
 if (vector.magnitude() <= maxLength)  
 return vector;  
 else  
 return vector.normalized() \* maxLength;  
 }  
 /\* 返回俩向量中x的最大值和y的最大值构造而成的向量 \*/  
 inline static Vector3 Max(const Vector3 &lhs, const Vector3 &rhs) { return Vector3(max(lhs.x, rhs.x), max(lhs.y, rhs.y), max(lhs.z, rhs.z)); }  
  
 /\* 返回俩向量中x的最小值和y的最小值构造而成的向量 \*/  
 inline static Vector3 Min(const Vector3 &lhs, const Vector3 &rhs) { return Vector3(min(lhs.x, rhs.x), min(lhs.y, rhs.y), min(lhs.z, rhs.z)); }  
  
 /\* 获得vector在onNormal(请自己单位化)方向的投影 \*/  
 inline static Vector3 Project(const Vector3 &vector, const Vector3 &onNormal) { return vector.magnitude() \* cos(Rad(vector, onNormal)) \* onNormal; }  
  
 /\* 正交化：将两个向量单位化，并调整切线位置使之垂直于法向 \*/  
 inline static void OrthoNormalize(Vector3 &normal, Vector3 &tangent)  
 {  
 normal.Normalize();  
 tangent = tangent - Project(tangent, normal);  
 tangent.Normalize();  
 }  
  
 /\* 正交化：将三个向量单位化，并调整使之两两垂直 \*/  
 inline static void OrthoNormalize(Vector3 &normal, Vector3 &tangent, Vector3 &binormal)  
 {  
 normal.Normalize();  
 tangent = tangent - Project(tangent, normal);  
 tangent.Normalize();  
 binormal -= Project(binormal, normal);  
 binormal -= Project(binormal, tangent);  
 binormal.Normalize();  
 }  
  
 /\* 获得vector在以planeNormal为法向量的平面的投影 \*/  
 inline static Vector3 ProjectOnPlane(Vector3 vector, Vector3 planeNormal)  
 {  
 FLOAT\_ mag = vector.magnitude();  
 FLOAT\_ s = Rad(vector, planeNormal);  
 OrthoNormalize(planeNormal, vector);  
 return mag \* sin(s) \* vector;  
 }  
  
 /\* 罗德里格旋转公式，获得current绕轴normal(请自己单位化)旋转degree度（默认角度）的向量，右手螺旋意义 \*/  
 inline static Vector3 Rotate(const Vector3 &current, const Vector3 &normal, const FLOAT\_ &degree, bool use\_degree = 1)  
 {  
 FLOAT\_ r = use\_degree ? degree / 180 \* PI : degree;  
 FLOAT\_ c = cos(r);  
 return c \* current + (1.0 - c) \* Dot(normal, current) \* normal + Cross(sin(r) \* normal, current);  
 }  
  
 /\* 将current向target转向degree度，如果大于夹角则返回target方向长度为current的向量 \*/  
 inline static Vector3 RotateTo(const Vector3 &current, const Vector3 &target, const FLOAT\_ &degree, bool use\_degree = 1)  
 {  
 FLOAT\_ r = use\_degree ? degree / 180 \* PI : degree;  
 if (r >= Rad(current, target))  
 return current.magnitude() / target.magnitude() \* target;  
 else  
 {  
 // FLOAT\_ mag = current.magnitude();  
 Vector3 nm = Cross(current, target).normalized();  
 return Rotate(current, nm, r);  
 }  
 }  
  
 /\* 球面插值 \*/  
 inline static Vector3 SlerpUnclamped(const Vector3 &a, const Vector3 &b, const FLOAT\_ &t)  
 {  
 Vector3 rot = RotateTo(a, b, Rad(a, b) \* t, false);  
 FLOAT\_ l = b.magnitude() \* t + a.magnitude() \* (1 - t);  
 return rot.normalized() \* l;  
 }  
  
 /\* 根据经纬，拿一个单位化的三维向量，以北纬和东经为正 \*/  
 inline static Vector3 FromLongitudeAndLatitude(const FLOAT\_ &longitude, const FLOAT\_ &latitude)  
 {  
 Vector3 lat = Rotate(Vector3(1, 0, 0), Vector3(0, -1, 0), latitude);  
 return Rotate(lat, Vector3(0, 0, 1), longitude);  
 }  
  
 /\* 球坐标转换为xyz型三维向量 \*/  
 inline static Vector3 FromSphericalCoordinate(const Vector3 &spherical, bool use\_degree = 1) { return FromSphericalCoordinate(spherical.x, spherical.y, spherical.z, use\_degree); }  
 /\* 球坐标转换为xyz型三维向量，半径r，theta倾斜角（纬度），phi方位角（经度），默认输出角度 \*/  
 inline static Vector3 FromSphericalCoordinate(const FLOAT\_ &r, FLOAT\_ theta, FLOAT\_ phi, bool use\_degree = 1)  
 {  
 theta = use\_degree ? theta / 180 \* PI : theta;  
 phi = use\_degree ? phi / 180 \* PI : phi;  
 return Vector3(  
 r \* sin(theta) \* cos(phi),  
 r \* sin(theta) \* sin(phi),  
 r \* cos(theta));  
 }  
 /\* 直角坐标转换为球坐标，默认输出角度 \*/  
 inline static Vector3 ToSphericalCoordinate(const Vector3 &coordinate, bool use\_degree = 1)  
 {  
 FLOAT\_ r = coordinate.magnitude();  
 return Vector3(  
 r,  
 acos(coordinate.z / r) \* (use\_degree ? 180.0 / PI : 1),  
 atan2(coordinate.y, coordinate.x) \* (use\_degree ? 180.0 / PI : 1));  
 }  
 /\* 直角坐标转换为球坐标，默认输出角度 \*/  
 inline Vector3 toSphericalCoordinate(bool use\_degree = 1) { return ToSphericalCoordinate(\*this, use\_degree); }  
};

### 矩阵

#### 静态矩阵

template <size\_t R, size\_t C, typename T = int>  
struct StaticMatrix : std::array<std::array<T, C>, R>  
{  
 std::string ToString() const  
 {  
 std::ostringstream ostr;  
 ostr << "StaticMatrix" << R << "x" << C << "[\n";  
 for (auto &i : \*this)  
 {  
 for (auto &j : i)  
 ostr << '\t' << j;  
 ostr << "\n";  
 }  
 ostr << "]";  
 return ostr.str();  
 }  
  
 friend std::ostream &operator<<(std::ostream &o, StaticMatrix &m) { return o << m.ToString(); }  
 friend std::ostream &operator<<(std::ostream &o, StaticMatrix &&m) { return o << m.ToString(); }  
  
 inline static StaticMatrix eye()  
 {  
 static\_assert(R == C);  
 StaticMatrix ret;  
 for (int i = 0; i < R; ++i)  
 ret[i][i] = 1;  
 return ret;  
 }  
 /\*交换两行\*/  
 inline void swap\_rows(const int from, const int to) { std::swap((\*this)[from], (\*this)[to]); }  
  
 /\*化为上三角矩阵\*/  
 inline void triangularify(bool unitriangularify = false)  
 {  
 int mx;  
 int done\_rows = 0;  
 for (int j = 0; j < C; j++) // 化为上三角  
 {  
 mx = done\_rows;  
 for (int i = done\_rows + 1; i < R; i++)  
 {  
 if (fabs((\*this)[i][j]) > fabs((\*this)[mx][j]))  
 mx = i;  
 }  
 if ((\*this)[mx][j] == 0)  
 continue;  
 if (mx != done\_rows)  
 swap\_rows(mx, done\_rows);  
  
 for (int i = done\_rows + 1; i < R; i++)  
 {  
 T tmp = (\*this)[i][j] / (\*this)[done\_rows][j];  
 if (tmp != 0)  
 for (int k = 0; k < C; ++k)  
 (\*this)[i][k] -= (\*this)[done\_rows][k] \* tmp;  
 }  
 if (unitriangularify)  
 {  
 auto tmp = (\*this)[done\_rows][j];  
 for (int k = 0; k < C; ++k)  
 (\*this)[done\_rows][k] /= tmp; // 因为用了引用，这里得拷贝暂存  
 }  
 done\_rows++;  
 if (done\_rows == R)  
 break;  
 }  
 }  
  
 /\*化为行最简型\*/  
 inline void row\_echelonify()  
 {  
 triangularify(true);  
 int valid\_pos = 1;  
 for (int i = 1; i < R; i++)  
 {  
 while (valid\_pos < C and (\*this)[i][valid\_pos] == 0)  
 valid\_pos++;  
 if (valid\_pos == C)  
 break;  
 for (int ii = i - 1; ii >= 0; ii--)  
 {  
 for (int jj = 0; jj < C; ++jj)  
 (\*this)[ii][jj] -= (\*this)[i][jj] \* (\*this)[ii][valid\_pos];  
 }  
 }  
 }  
  
 /\*返回一个自身化为上三角矩阵的拷贝\*/  
 inline StaticMatrix triangular(bool unitriangularify = false) const  
 {  
 StaticMatrix ret(\*this);  
 ret.triangularify(unitriangularify);  
 return ret;  
 }  
  
 /\*求秩，得先上三角化\*/  
 inline int \_rank() const  
 {  
 int res = 0;  
 for (auto &i : (\*this))  
 res += (i.back() != 0);  
 return res;  
 }  
  
 /\*求秩\*/  
 inline int rank() const { return triangular().\_rank(); }  
  
 /\*高斯消元解方程组\*/  
 inline bool solve()  
 {  
 triangularify();  
 if (!(\*this).back().back())  
 return false;  
 for (int i = R - 1; i >= 0; i--)  
 {  
 for (int j = i + 1; j < R; j++)  
 (\*this)[i][C - 1] -= (\*this)[i][j] \* (\*this)[j][C - 1];  
 if ((\*this)[i][i] == 0)  
 return false;  
 (\*this)[i][C - 1] /= (\*this)[i][i];  
 }  
 return true;  
 }  
  
 /\*矩阵乘法\*/  
 template <size\_t \_C>  
 inline StaticMatrix<R, \_C, T> dot(const StaticMatrix<C, \_C, T> &rhs) const  
 {  
 StaticMatrix<R, \_C, T> ret;  
 for (int i = 0; i < R; ++i)  
 for (int k = 0; k < C; ++k)  
 {  
 const T &s = (\*this)[i][k];  
 for (int j = 0; j < \_C; ++j)  
 ret[i][j] += s \* rhs[k][j];  
 }  
 return ret;  
 }  
 inline bool operator!=(const StaticMatrix &rhs) const  
 {  
 for (int i = 0; i < R; ++i)  
 for (int j = 0; j < C; ++j)  
 if ((\*this)[i][j] != rhs[i][j])  
 return true;  
 return false;  
 }  
 inline bool operator==(const StaticMatrix &rhs) const { return !(\*this == rhs); }  
 template <size\_t \_C>  
 inline StaticMatrix<R, \_C, T> operator\*(const StaticMatrix<C, \_C, T> &rhs) const { return dot(rhs); }  
 template <size\_t \_C>  
 inline StaticMatrix<R, \_C, T> &operator\*=(const StaticMatrix<C, \_C, T> &rhs) { return (\*this) = dot(rhs); }  
 inline StaticMatrix &operator+=(const StaticMatrix &rhs)  
 {  
 for (int i = 0; i < R; ++i)  
 for (int j = 0; j < C; ++j)  
 (\*this)[i][j] += rhs[i][j];  
 return \*this;  
 }  
 inline StaticMatrix &operator+=(const T &rhs)  
 {  
 for (int i = 0; i < R; ++i)  
 for (int j = 0; j < C; ++j)  
 (\*this)[i][j] += rhs;  
 return \*this;  
 }  
 inline StaticMatrix operator+(const StaticMatrix &rhs) const { return StaticMatrix(\*this) += rhs; }  
 inline friend StaticMatrix operator+(const T &rhs, StaticMatrix mat) { return mat + rhs; }  
 inline StaticMatrix &operator\*=(const T &rhs)  
 {  
 for (auto &i : (\*this))  
 for (auto &j : i)  
 j \*= rhs;  
 return (\*this);  
 }  
 inline StaticMatrix operator\*(const T &rhs) const { return StaticMatrix(\*this) \*= rhs; }  
 inline friend StaticMatrix operator\*(const T &rhs, StaticMatrix mat) { return mat \* rhs; }  
};

### 方阵（求逆矩阵）

template <typename VALUETYPE = FLOAT\_>  
struct SquareMatrix : Matrix<VALUETYPE>  
{  
  
 static SquareMatrix eye(int siz)  
 {  
 SquareMatrix ret(siz);  
 for (siz--; siz >= 0; siz--)  
 ret[siz][siz] = 1;  
 return ret;  
 }  
  
 SquareMatrix quick\_power(long long p, long long mod = 0)  
 {  
 SquareMatrix ans = eye(this->ROW);  
 SquareMatrix rhs(\*this);  
 while (p)  
 {  
 if (p & 1)  
 {  
 ans = ans.dot(rhs, mod);  
 }  
 rhs = rhs.dot(rhs, mod);  
 p >>= 1;  
 }  
 return ans;  
 }  
  
 SquareMatrix inv(long long mod = 0)  
 {  
 Matrix<VALUETYPE> ret(\*this);  
 ret.rconcat(eye(this->ROW));  
 ret.row\_echelonify(mod); // 行最简形  
 // cerr << ret << endl;  
 for (int i = 0; i < this->ROW; i++)  
 {  
 if (ret[i][i] != 1)  
 throw "Error at matrix inverse: cannot identify extended matrix";  
 }  
 ret.lerase(this->ROW);  
 return ret;  
 }  
};

### 二维直线

struct Line2  
{  
 FLOAT\_ A, B, C;  
 Line2(Vector2 u, Vector2 v) : A(u.y - v.y), B(v.x - u.x), C(u.y \* (u.x - v.x) - u.x \* (u.y - v.y))  
 {  
 if (u == v)  
 {  
 if (u.x)  
 {  
 A = 1;  
 B = 0;  
 C = -u.x;  
 }  
 else if (u.y)  
 {  
 A = 0;  
 B = 1;  
 C = -u.y;  
 }  
 else  
 {  
 A = 1;  
 B = -1;  
 C = 0;  
 }  
 }  
 }  
 Line2(FLOAT\_ a, FLOAT\_ b, FLOAT\_ c) : A(a), B(b), C(c) {}  
 std::string ToString()  
 {  
 std::ostringstream ostr;  
 ostr << "Line2(" << this->A << ", " << this->B << ", " << this->C << ")";  
 return ostr.str();  
 }  
 friend std::ostream &operator<<(std::ostream &o, Line2 v)  
 {  
 o << v.ToString();  
 return o;  
 }  
 FLOAT\_ k() { return -A / B; }  
 FLOAT\_ b() { return -C / B; }  
 FLOAT\_ x(FLOAT\_ y) { return -(B \* y + C) / A; }  
 FLOAT\_ y(FLOAT\_ x) { return -(A \* x + C) / B; }  
 /\*点到直线的距离\*/  
 FLOAT\_ distToPoint(Vector2 p) { return abs(A \* p.x + B \* p.y + C / sqrt(A \* A + B \* B)); }  
 /\*直线距离公式，使用前先判平行\*/  
 static FLOAT\_ Distance(Line2 a, Line2 b) { return abs(a.C - b.C) / sqrt(a.A \* a.A + a.B \* a.B); }  
 /\*判断平行\*/  
 static bool IsParallel(Line2 u, Line2 v)  
 {  
 bool f1 = round\_compare(u.B, 0.0);  
 bool f2 = round\_compare(v.B, 0.0);  
 if (f1 != f2)  
 return false;  
 return f1 or round\_compare(u.k(), v.k());  
 }  
  
 /\*单位化（？）\*/  
 void normalize()  
 {  
 FLOAT\_ su = sqrt(A \* A + B \* B + C \* C);  
 if (A < 0)  
 su = -su;  
 else if (A == 0 and B < 0)  
 su = -su;  
 A /= su;  
 B /= su;  
 C /= su;  
 }  
 /\*返回单位化后的直线\*/  
 Line2 normalized()  
 {  
 Line2 t(\*this);  
 t.normalize();  
 return t;  
 }  
  
 bool operator==(Line2 v) { return round\_compare(A, v.A) and round\_compare(B, v.B) and round\_compare(C, v.C); }  
 bool operator!=(Line2 v) { return !(\*this == v); }  
  
 static bool IsSame(Line2 u, Line2 v)  
 {  
 return Line2::IsParallel(u, v) and round\_compare(Distance(u.normalized(), v.normalized()), 0.0);  
 }  
  
 /\*计算交点\*/  
 static Vector2 Intersect(Line2 u, Line2 v)  
 {  
 FLOAT\_ tx = (u.B \* v.C - v.B \* u.C) / (v.B \* u.A - u.B \* v.A);  
 FLOAT\_ ty = (u.B != 0.0 ? -u.A \* tx / u.B - u.C / u.B : -v.A \* tx / v.B - v.C / v.B);  
 return Vector2(tx, ty);  
 }  
  
 /\*判断三个点是否共线\*/  
 static bool Collinear(Vector2 a, Vector2 b, Vector2 c)  
 {  
 Line2 l(a, b);  
 return round\_compare((l.A \* c.x + l.B \* c.y + l.C), 0.0);  
 }  
};

### 二维有向线段

struct Segment2 : Line2 // 二维有向线段  
{  
 Vector2 from, to;  
 Segment2(Vector2 a, Vector2 b) : Line2(a, b), from(a), to(b) {}  
 Segment2(FLOAT\_ x, FLOAT\_ y, FLOAT\_ X, FLOAT\_ Y) : Line2(Vector2(x, y), Vector2(X, Y)), from(Vector2(x, y)), to(Vector2(X, Y)) {}  
 bool is\_online(Vector2 poi)  
 {  
 return round\_compare((Vector2::Distance(poi, to) + Vector2::Distance(poi, from)), Vector2::Distance(from, to));  
 }  
 Vector2 &operator[](int i)  
 {  
 switch (i)  
 {  
 case 0:  
 return from;  
 break;  
 case 1:  
 return to;  
 break;  
 default:  
 throw "数组越界";  
 break;  
 }  
 }  
};

### 二维多边形

struct Polygon2  
{  
 std::vector<Vector2> points;  
  
private:  
 Vector2 accordance;  
  
public:  
 inline Polygon2 ConvexHull()  
 {  
 Polygon2 ret;  
 std::sort(points.begin(), points.end());  
 std::vector<Vector2> &stk = ret.points;  
  
 std::vector<char> used(points.size(), 0);  
 std::vector<int> uid;  
 for (auto &i : points)  
 {  
 while (stk.size() >= 2 and Vector2::Cross(stk.back() - stk[stk.size() - 2], i - stk.back()) <= 0)  
 {  
 used[uid.back()] = 0;  
 uid.pop\_back();  
 stk.pop\_back();  
 }  
  
 used[&i - &points.front()] = 1;  
 uid.emplace\_back(&i - &points.front());  
 stk.emplace\_back(i);  
 }  
 used[0] = 0;  
 int ts = stk.size();  
 for (auto ii = ++points.rbegin(); ii != points.rend(); ii++)  
 {  
 Vector2 &i = \*ii;  
 if (!used[&i - &points.front()])  
 {  
 while (stk.size() > ts and Vector2::Cross(stk.back() - stk[stk.size() - 2], i - stk.back()) <= 0)  
 {  
 used[uid.back()] = 0;  
 uid.pop\_back();  
 stk.pop\_back();  
 }  
 used[&i - &points.front()] = 1;  
 uid.emplace\_back(&i - &points.front());  
 stk.emplace\_back(i);  
 }  
 }  
 stk.pop\_back();  
 return ret;  
 }  
  
 /\* 凸多边形用逆时针排序 \*/  
 inline void autoanticlockwiselize()  
 {  
 accordance = average();  
 anticlockwiselize();  
 }  
  
 inline void anticlockwiselize()  
 {  
 auto anticlock\_comparator = [&](Vector2 &a, Vector2 &b) -> bool  
 {  
 return (a - accordance).toPolarCoordinate(false).y < (b - accordance).toPolarCoordinate(false).y;  
 };  
 std::sort(points.begin(), points.end(), anticlock\_comparator);  
 }  
  
 inline Vector2 average() const  
 {  
 Vector2 avg(0, 0);  
 for (auto &i : points)  
 {  
 avg += i;  
 }  
 return avg / points.size();  
 }  
  
 /\* 求周长 \*/  
 inline FLOAT\_ perimeter() const  
 {  
 FLOAT\_ ret = Vector2::Distance(points.front(), points.back());  
 for (int i = 1; i < points.size(); i++)  
 ret += Vector2::Distance(points[i], points[i - 1]);  
 return ret;  
 }  
 /\* 面积 \*/  
 inline FLOAT\_ area() const  
 {  
 FLOAT\_ ret = Vector2::Cross(points.back(), points.front());  
 for (int i = 1; i < points.size(); i++)  
 ret = ret + Vector2::Cross(points[i - 1], points[i]);  
 return ret / 2;  
 }  
 /\* 求几何中心（形心、重心） \*/  
 inline Vector2 center() const  
 {  
 Vector2 ret = (points.back() + points.front()) \* Vector2::Cross(points.back(), points.front());  
 for (int i = 1; i < points.size(); i++)  
 ret = ret + (points[i - 1] + points[i]) \* Vector2::Cross(points[i - 1], points[i]);  
 return ret / area() / 6;  
 }  
 /\* 求边界整点数 \*/  
 inline long long boundary\_points() const  
 {  
 long long b = 0;  
 for (int i = 0; i < points.size() - 1; i++)  
 {  
 b += std::\_\_gcd((long long)abs(points[i + 1].x - points[i].x), (long long)abs(points[i + 1].y - points[i].y));  
 }  
 return b;  
 }  
 /\* Pick定理：多边形面积=内部整点数+边界上的整点数/2-1；求内部整点数 \*/  
 inline long long interior\_points(FLOAT\_ A = -1, long long b = -1) const  
 {  
 if (A < 0)  
 A = area();  
 if (b < 0)  
 b = boundary\_points();  
 return (long long)A + 1 - (b / 2);  
 }  
  
 inline bool is\_inner(const Vector2 &p) const  
 {  
 bool res = false;  
 Vector2 j = points.back();  
 for (auto &i : points)  
 {  
 if ((i.y < p.y and j.y >= p.y or j.y < p.y and i.y >= p.y) and (i.x <= p.x or j.x <= p.x))  
 res ^= (i.x + (p.y - i.y) / (j.y - i.y) \* (j.x - i.x) < p.x);  
 j = i;  
 }  
 return res;  
 }  
};

### 三维面

struct Face3 : std::array<Vector3, 3>  
{  
 Face3(const Vector3 &v0, const Vector3 &v1, const Vector3 &v2) : std::array<Vector3, 3>({v0, v1, v2}) {}  
 inline static Vector3 normal(const Vector3 &v0, const Vector3 &v1, const Vector3 &v2) { return Vector3::Cross(v1 - v0, v2 - v0); }  
 inline static FLOAT\_ area(const Vector3 &v0, const Vector3 &v1, const Vector3 &v2) { return normal(v0, v1, v2).magnitude() / FLOAT\_(2); }  
 inline static bool visible(const Vector3 &v0, const Vector3 &v1, const Vector3 &v2, const Vector3 &\_v) { return Vector3::Dot(\_v - v0, normal(v0, v1, v2)) > 0; }  
 inline Vector3 normal() { return Vector3::Cross((\*this)[1] - (\*this)[0], (\*this)[2] - (\*this)[0]); }  
 inline FLOAT\_ area() { return normal().magnitude() / FLOAT\_(2); }  
 inline bool visible(const Vector3 &\_v) { return Vector3::Dot(\_v - (\*this)[0], normal()) > 0; }  
};

### 三维多边形（三维凸包）

struct Polygon3  
{  
 std::vector<Vector3> points;  
  
 inline Vector3 average()  
 {  
 Vector3 avg(0);  
 for (auto i : points)  
 avg += i;  
 return avg / points.size();  
 }  
 /\* n^2增量法三维凸包，返回面列表(下标顶点引用) \*/  
 inline std::vector<std::array<int, 3>> ConvexHull()  
 {  
 for (auto &i : points)  
 {  
 i.x += randreal(-decimal\_round, decimal\_round);  
 i.y += randreal(-decimal\_round, decimal\_round);  
 i.z += randreal(-decimal\_round, decimal\_round);  
 }  
 std::vector<std::array<int, 3>> rf, rC;  
 std::vector<std::vector<char>> vis(points.size(), std::vector<char>(points.size()));  
 rf.emplace\_back(std::array<int, 3>({0, 1, 2}));  
 rf.emplace\_back(std::array<int, 3>({2, 1, 0}));  
 int cnt = 2;  
 for (int i = 3, cc = 0; i < points.size(); ++i)  
 {  
 bool vi;  
 int cct = 0;  
 for (auto &j : rf)  
 {  
 if (!(vi = Face3::visible(points[j[0]], points[j[1]], points[j[2]], points[i])))  
 rC.emplace\_back(rf[cct]);  
 for (int k = 0; k < 3; ++k)  
 vis[j[k]][j[(k + 1) % 3]] = vi;  
 ++cct;  
 }  
 for (auto &j : rf)  
 for (int k = 0; k < 3; ++k)  
 {  
 int x = j[k];  
 int y = j[(k + 1) % 3];  
 if (vis[x][y] and not vis[y][x])  
 rC.emplace\_back(std::array<int, 3>({x, y, i}));  
 }  
 swap(rf, rC);  
 rC.clear();  
 }  
 return rf;  
 }  
};

### 圆

namespace Geometry  
{  
 /\* https://www.luogu.com.cn/record/51674409 模板题需要用long double \*/  
 struct Circle  
 {  
 Vector2 center;  
 FLOAT\_ radius;  
 Circle(Vector2 c, FLOAT\_ r) : center(c), radius(r) {}  
 Circle(FLOAT\_ x, FLOAT\_ y, FLOAT\_ r) : center(x, y), radius(r) {}  
 Circle(Vector2 a, Vector2 b, Vector2 c)  
 {  
 Vector2 p1 = Vector2::LerpUnclamped(a, b, 0.5);  
 Vector2 v1 = b - a;  
 swap(v1.x, v1.y);  
 v1.x = -v1.x;  
 Vector2 p2 = Vector2::LerpUnclamped(b, c, 0.5);  
 Vector2 v2 = c - b;  
 swap(v2.x, v2.y);  
 v2.x = -v2.x;  
  
 center = Line2::Intersect(Line2(p1, v1, false), Line2(p2, v2, false));  
  
 radius = (center - a).magnitude();  
 }  
 Vector2 fromRad(FLOAT\_ A)  
 {  
 return Vector2(center.x + radius \* cos(A), center.y + radius \* sin(A));  
 }  
 std::pair<Vector2, Vector2> intersect\_points(Line2 l)  
 {  
 FLOAT\_ k = l.k();  
 // 特判  
 if (isnan(k))  
 {  
 FLOAT\_ x = -l.C / l.A;  
 FLOAT\_ rhs = pow(radius, 2) - pow(x - center.x, 2);  
 if (rhs < 0)  
 return make\_pair(Vector2(nan(""), nan("")), Vector2(nan(""), nan("")));  
 else  
 {  
 rhs = sqrt(rhs);  
 return make\_pair(Vector2(x, rhs + radius), Vector2(x, -rhs + radius));  
 }  
 }  
 FLOAT\_ lb = l.b();  
 FLOAT\_ a = k \* k + 1;  
 FLOAT\_ b = 2 \* k \* (lb - center.y) - 2 \* center.x;  
 FLOAT\_ c = pow(lb - center.y, 2) + pow(center.x, 2) - pow(radius, 2);  
 FLOAT\_ x1, x2;  
 std::tie(x1, x2) = solveQuadraticEquation(a, b, c);  
 if (isnan(x1))  
 {  
 return make\_pair(Vector2(nan(""), nan("")), Vector2(nan(""), nan("")));  
 }  
 else  
 {  
 return make\_pair(Vector2(x1, l.y(x1)), Vector2(x2, l.y(x2)));  
 }  
 }  
 /\* 使用极角和余弦定理算交点，更稳，但没添加处理相离和相包含的情况 \*/  
 std::pair<Vector2, Vector2> intersect\_points(Circle cir)  
 {  
 Vector2 distV = (cir.center - center);  
 FLOAT\_ dist = distV.magnitude();  
 FLOAT\_ ang = distV.toPolarAngle(false);  
 FLOAT\_ dang = acos((pow(radius, 2) + pow(dist, 2) - pow(cir.radius, 2)) / (2 \* radius \* dist)); //余弦定理  
 return make\_pair(fromRad(ang + dang), fromRad(ang - dang));  
 }  
  
 FLOAT\_ area() { return PI \* radius \* radius; }  
  
 bool is\_outside(Vector2 p)  
 {  
 return (p - center).magnitude() > radius;  
 }  
 bool is\_inside(Vector2 p)  
 {  
 return intereps((p - center).magnitude() - radius) < 0;  
 }  
 static intersect\_area(Circle A, Circle B)  
 {  
 Vector2 dis = A.center - B.center;  
 FLOAT\_ sqrdis = dis.sqrMagnitude();  
 FLOAT\_ cdis = sqrt(sqrdis);  
 if (sqrdis >= pow(A.radius + B.radius, 2))  
 return FLOAT\_(0);  
 if (A.radius >= B.radius)  
 std::swap(A, B);  
 if (cdis + A.radius <= B.radius)  
 return PI \* A.radius \* A.radius;  
 if (sqrdis >= B.radius \* B.radius)  
 {  
 FLOAT\_ area = 0.0;  
 FLOAT\_ ed = sqrdis;  
 FLOAT\_ jiao = ((FLOAT\_)B.radius \* B.radius + ed - A.radius \* A.radius) / (2.0 \* B.radius \* sqrt((FLOAT\_)ed));  
 jiao = acos(jiao);  
 jiao \*= 2.0;  
 area += B.radius \* B.radius \* jiao / 2;  
 jiao = sin(jiao);  
 area -= B.radius \* B.radius \* jiao / 2;  
 jiao = ((FLOAT\_)A.radius \* A.radius + ed - B.radius \* B.radius) / (2.0 \* A.radius \* sqrt((FLOAT\_)ed));  
 jiao = acos(jiao);  
 jiao \*= 2;  
 area += A.radius \* A.radius \* jiao / 2;  
 jiao = sin(jiao);  
 area -= A.radius \* A.radius \* jiao / 2;  
 return area;  
 }  
 FLOAT\_ area = 0.0;  
 FLOAT\_ ed = sqrdis;  
 FLOAT\_ jiao = ((FLOAT\_)A.radius \* A.radius + ed - B.radius \* B.radius) / (2.0 \* A.radius \* sqrt(ed));  
 jiao = acos(jiao);  
 area += A.radius \* A.radius \* jiao;  
 jiao = ((FLOAT\_)B.radius \* B.radius + ed - A.radius \* A.radius) / (2.0 \* B.radius \* sqrt(ed));  
 jiao = acos(jiao);  
 area += B.radius \* B.radius \* jiao - B.radius \* sqrt(ed) \* sin(jiao);  
 return area;  
 }  
 };  
}

### 球

struct Sphere  
{  
 FLOAT\_ radius;  
 Vector3 center;  
 Sphere(Vector3 c, FLOAT\_ r) : center(c), radius(r) {}  
 Sphere(FLOAT\_ x, FLOAT\_ y, FLOAT\_ z, FLOAT\_ r) : center(x, y, z), radius(r) {}  
 FLOAT\_ volumn() { return 4.0 \* PI \* pow(radius, 3) / 3.0; }  
 FLOAT\_ intersectVolumn(Sphere o)  
 {  
 Vector3 dist = o.center - center;  
 FLOAT\_ distval = dist.magnitude();  
 if (distval > o.radius + radius)  
 return 0;  
 if (distval < abs(o.radius - radius))  
 {  
 return o.radius > radius ? volumn() : o.volumn();  
 }  
 FLOAT\_ &d = distval;  
 //球心距  
 FLOAT\_ t = (d \* d + o.radius \* o.radius - radius \* radius) / (2.0 \* d);  
 //h1=h2，球冠的高  
 FLOAT\_ h = sqrt((o.radius \* o.radius) - (t \* t)) \* 2;  
 FLOAT\_ angle\_a = 2 \* acos((o.radius \* o.radius + d \* d - radius \* radius) / (2.0 \* o.radius \* d)); //余弦公式计算r1对应圆心角，弧度  
 FLOAT\_ angle\_b = 2 \* acos((radius \* radius + d \* d - o.radius \* o.radius) / (2.0 \* radius \* d)); //余弦公式计算r2对应圆心角，弧度  
 FLOAT\_ l1 = ((o.radius \* o.radius - radius \* radius) / d + d) / 2;  
 FLOAT\_ l2 = d - l1;  
 FLOAT\_ x1 = o.radius - l1, x2 = radius - l2; //分别为两个球缺的高度  
 FLOAT\_ v1 = PI \* x1 \* x1 \* (o.radius - x1 / 3); //相交部分r1圆所对应的球缺部分体积  
 FLOAT\_ v2 = PI \* x2 \* x2 \* (radius - x2 / 3); //相交部分r2圆所对应的球缺部分体积  
 //相交部分体积  
 return v1 + v2;  
 }  
 FLOAT\_ joinVolumn(Sphere o)  
 {  
 return volumn() + o.volumn() - intersectVolumn(o);  
 }  
};

### 退火

#include "Headers.cpp"  
  
using FT = long double;  
  
FT fun(FT angle) // 根据需要改 评估函数  
{  
 FT res = 0;  
 for (auto &[TT, SS, AA] : V)  
 {  
 FT deg = abs(angle - AA);  
 res += max(FT(0.0), TT - SS \* (deg >= pi ? oneround - deg : deg));  
 }  
  
 return res;  
}  
  
FT randreal(FT begin = -pi, FT end = pi)  
{  
 static std::default\_random\_engine eng(time(0));  
 std::uniform\_real\_distribution<FT> skip\_rate(begin, end);  
 return skip\_rate(eng);  
}  
  
template <typename IT>  
IT randint(IT begin, IT end)  
{  
 static std::default\_random\_engine eng(time(0));  
 std::uniform\_int\_distribution<IT> skip\_rate(begin, end);  
 return skip\_rate(eng);  
}  
  
void sa(FT temperature = 300, FT cooldown = 1e-14, FT cool = 0.986)  
{  
 FT cangle = randreal(0, oneround);  
 FT jbj = fun(cangle); // 局部解  
 MX = max(MX, jbj); // 全局解  
  
 while (temperature > cooldown)   
 {  
 FT curangle = fmod(cangle + randreal(-1, 1) \* temperature, oneround);  
 while (curangle < 0)  
 curangle += oneround;  
  
 FT energy = fun(curangle);  
 FT de = jbj - energy;  
 MX = max(jbj, MX);  
 if (de < 0)  
 {  
 cangle = curangle;  
 jbj = energy;  
 }  
 else if (exp(-de / (temperature)) > randreal(0, 1))  
 {  
 cangle = curangle;  
 jbj = energy;  
 }  
 temperature \*= cool;  
 }  
}

## 数学

### exgcd全解

/\* 解同余方程ax + by = c \*/  
void exgcd\_solve()  
{  
 qr(a);  
 qr(b);  
 qr(c);  
  
 LL GCD = exgcd(a, b, x, y);  
 if (c % GCD != 0) // 无解  
 {  
 puts("-1");  
 return;  
 }  
  
 LL xishu = c / GCD;  
  
 LL x1 = x \* xishu;  
 LL y1 = y \* xishu;  
 // 为了满足 a \* (x1 + db) + b \* (y1 - da) = c的形式  
 // x1, y1 是特解，通过枚举【实数】d可以得到通解  
 LL dx = b / GCD; // 构造 x = x1 + s \* dx ，即a的系数  
 LL dy = a / GCD; // 构造 y = y1 - s \* dy ，即b的系数  
 // 这步的s就可以是整数了  
 // 限制 x>0 => x1 + s \* dx > 0 => s > - x1 / dx (实数)  
 // 限制 y>0 => y1 - s \* dy > 0 => s < y1 / dy (实数)  
  
 LL xlower = ceil(double(-x1 + 1) / dx); // s可能的最小值  
 LL yupper = floor(double(y1 - 1) / dy); // s可能的最大值  
 if (xlower > yupper)  
 {  
 LL xMin = x1 + xlower \* dx; // x的最小正整数解  
 LL yMin = y1 - yupper \* dy; // y的最小正整数解  
 printf("%lld %lld\n", xMin, yMin);  
 }  
 else  
 {  
 LL s\_range = yupper - xlower + 1; // 正整数解个数  
 LL xMax = x1 + yupper \* dx; // x的最大正整数解  
 LL xMin = x1 + xlower \* dx; // x的最小正整数解  
 LL yMax = y1 - xlower \* dy; // y的最大正整数解  
 LL yMin = y1 - yupper \* dy; // y的最小正整数解  
 printf("%lld %lld %lld %lld %lld\n", s\_range, xMin, yMin, xMax, yMax);  
 }  
}

### 数论和杂项

#### 模数类

/\* 静态模数类，只能用有符号类型做T和EXT参数 \*/  
template <int mod, class T = int, class EXT = long long>  
struct mint  
{  
 T x;  
 template <class TT>  
 mint(TT \_x)  
 {  
 x = EXT(\_x) % mod;  
 if (x < 0)  
 x += mod;  
 }  
 mint() : x(0) {}  
 mint &operator++()  
 {  
 ++x;  
 if (x == mod)  
 x = 0;  
 return \*this;  
 }  
 mint &operator--()  
 {  
 x = (x == 0 ? mod - 1 : x - 1);  
 return \*this;  
 }  
 mint operator++(int)  
 {  
 mint tmp = \*this;  
 ++\*this;  
 return tmp;  
 }  
 mint operator--(int)  
 {  
 mint tmp = \*this;  
 --\*this;  
 return tmp;  
 }  
 mint &operator+=(const mint &rhs)  
 {  
 x += rhs.x;  
 if (x >= mod)  
 x -= mod;  
 return \*this;  
 }  
 mint &operator-=(const mint &rhs)  
 {  
 x -= rhs.x;  
 if (x < 0)  
 x += mod;  
 return \*this;  
 }  
 mint &operator\*=(const mint &rhs)  
 {  
 x = EXT(x) \* rhs.x % mod;  
 return \*this;  
 }  
 mint &operator/=(const mint &rhs)  
 {  
 x = EXT(x) \* inv(rhs.x, mod) % mod;  
 return \*this;  
 }  
 mint operator+() const { return \*this; }  
 mint operator-() const { return mod - \*this; }  
 friend mint operator+(const mint &lhs, const mint &rhs) { return mint(lhs) += rhs; }  
 friend mint operator-(const mint &lhs, const mint &rhs) { return mint(lhs) -= rhs; }  
 friend mint operator\*(const mint &lhs, const mint &rhs) { return mint(lhs) \*= rhs; }  
 friend mint operator/(const mint &lhs, const mint &rhs) { return mint(lhs) /= rhs; }  
 friend bool operator==(const mint &lhs, const mint &rhs) { return lhs.x == rhs.x; }  
 friend bool operator!=(const mint &lhs, const mint &rhs) { return lhs.x != rhs.x; }  
 friend std::ostream &operator<<(std::ostream &o, const mint &m) { return o << m.x; }  
 friend std::istream &operator>>(std::istream &i, const mint &m)  
 {  
 i >> m.x;  
 m.x %= mod;  
 if (m.x < 0)  
 m.x += mod;  
 return i;  
 }  
};  
using m998 = mint<998244353>;  
using m1e9\_7 = mint<1000000007>;  
using m1e9\_9 = mint<1000000009>;

#### Cipolla求奇质数的二次剩余

/\* def94200d616892a0187be01c94ea9c1 使用Cipolla计算二次剩余 \*/  
template <typename T>  
struct Cipolla  
{  
 T re\_al, im\_ag;  
 /\* 定义I = a^2 - n，实际上是单位负根的平方 \*/  
 inline static T mod, I; // 17特性，不行就提全局  
  
 inline static Cipolla power(Cipolla x, LL p)  
 {  
 Cipolla res(1);  
 while (p)  
 {  
 if (p & 1)  
 res = res \* x;  
 x = x \* x;  
 p >>= 1;  
 }  
 return res;  
 }  
 /\* 检查x是不是二次剩余 \*/  
 inline static bool check\_if\_residue(T x)  
 {  
 return power(x, mod - 1 >> 1) == 1;  
 }  
  
 /\* 算法入口，要求p是奇素数 \*/  
 static void solve(T n, T p, T &x0, T &x1)  
 {  
 n %= p;  
 mod = p;  
 if (n == 0)  
 {  
 x0 = x1 = 0;  
 return;  
 }  
 if (!check\_if\_residue(n))  
 {  
 x0 = x1 = -1; // 无解  
 return;  
 }  
 T a;  
 do  
 {  
 a = randint(T(1), mod - 1);  
 } while (check\_if\_residue((a \* a + mod - n) % mod));  
 I = (a \* a - n + mod) % mod;  
 x0 = T(power(Cipolla(a, 1), mod + 1 >> 1).real());  
 x1 = mod - x0;  
 }  
 /\* 实际上是个模意义复数类 \*/  
 Cipolla(T \_real = 0, T \_imag = 0) : re\_al(\_real), im\_ag(\_imag) {}  
 inline T &real() { return re\_al; }  
 inline T &imag() { return im\_ag; }  
 inline bool operator==(const Cipolla &y) const  
 {  
 return re\_al == y.re\_al and im\_ag == y.im\_ag;  
 }  
 inline Cipolla operator\*(const Cipolla &y) const  
 {  
 return Cipolla((re\_al \* y.re\_al + I \* im\_ag % mod \* y.im\_ag) % mod,  
 (im\_ag \* y.re\_al + re\_al \* y.im\_ag) % mod);  
 }  
};

#### 类欧模意义不等式

/\* 取 l <= dx%m <= r 的最小非负x \*/  
LL modinq(LL m, LL d, LL l, LL r)  
{  
 // 0 <= l <= r < m, d < m, minimal non-negative solution  
 if (r < l)  
 return -1;  
 if (l == 0)  
 return 0;  
 if (d == 0)  
 return -1;  
 if ((r / d) \* d >= l)  
 return (l - 1) / d + 1;  
 LL res = modinq(d, m % d, (d - r % d) % d, (d - l % d) % d);  
 return res == -1 ? -1 : (m \* res + l - 1) / d + 1;  
}

#### 欧拉筛

typedef long long LL  
// #define ORAFM 2333  
int prime[ORAFM + 5], prime\_number = 0, prv[ORAFM + 5];  
// 莫比乌斯函数  
int mobius[ORAFM + 5];  
// 欧拉函数  
LL phi[ORAFM + 5];  
  
bool marked[ORAFM + 5];  
  
void ORAfliter(LL MX)  
{  
 mobius[1] = phi[1] = 1;  
 for (unsigned int i = 2; i <= MX; i++)  
 {  
 if (!marked[i])  
 {  
 prime[++prime\_number] = i;  
 prv[i] = i;  
 phi[i] = i - 1;  
 mobius[i] = -1;  
 }  
 for (unsigned int j = 1; j <= prime\_number && i \* prime[j] <= MX; j++)  
 {  
 marked[i \* prime[j]] = true;  
 prv[i \* prime[j]] = prime[j];  
 if (i % prime[j] == 0)  
 {  
 phi[i \* prime[j]] = prime[j] \* phi[i];  
 break;  
 }  
 phi[i \* prime[j]] = phi[prime[j]] \* phi[i];  
 mobius[i \* prime[j]] = -mobius[i]; // 平方因数不会被处理到，默认是0  
 }  
 }  
 // 这句话是做莫比乌斯函数和欧拉函数的前缀和  
 for (unsigned int i = 2; i <= MX; ++i)  
 {  
 mobius[i] += mobius[i - 1];  
 phi[i] += phi[i - 1];  
 }  
}

#### min\_25筛框架

inline void prework(LL n)  
{  
 int tot = 0;  
 for (LL l = 1, r; l <= n; l = r + 1)  
 {  
 r = n / (n / l); // 数论分块？  
 w[++tot] = n / l;  
 // g1[tot] = w[tot] % mo;  
 // g2[tot] = (g1[tot] \* (g1[tot] + 1) >> 1) % mo \* ((g1[tot] << 1) + 1) % mo \* inv3 % mo;  
 // g2[tot]--;  
 // g1[tot] = (g1[tot] \* (g1[tot] + 1) >> 1) % mo - 1;  
 valposition(n / l, n) = tot;  
 g1[tot] = n / l - 1;  
 // g2[tot] = n / l - 1;  
 }  
 for (int i = 1; i <= prime\_number; i++)  
 {  
 for (int j = 1; j <= tot and (LL) prime[i] \* prime[i] <= w[j]; j++)  
 {  
 LL n\_div\_m\_val = w[j] / prime[i];  
 if (n\_div\_m\_val)  
 {  
 int n\_div\_m = valposition(n\_div\_m\_val, n); // m: prime[i]  
 g1[j] -= g1[n\_div\_m] - (i - 1); // 枚举第i个质数，所以可以直接减去i-1，这里无需记录sp  
 }  
 // g1[j] -= (LL)prime[i] \* (g1[k] - sp1[i - 1] + mo) % mo;  
 // g2[j] -= (LL)prime[i] \* prime[i] % mo \* (g2[k] - sp2[i - 1] + mo) % mo;  
 // g1[j] %= mo,  
 // g2[j] %= mo;  
 // if (g1[j] < 0)  
 // g1[j] += mo;  
 // if (g2[j] < 0)  
 // g2[j] += mo;  
 }  
 }  
}  
// 1~x中最小质因子大于y的函数值  
inline LL S\_(LL x, int y)  
{  
 if (prime[y] >= x)  
 return 0;  
 int k = valposition(x, n);  
 // 此处g1、g2代表1、2次项  
 LL ans = (g2[k] - g1[k] + mo - (sp2[y] - sp1[y]) + mo) % mo;  
 // ans = (ans + mo) % mo;  
 for (int i = y + 1; i <= prime\_number and prime[i] \* prime[i] <= x; ++i)  
 {  
 LL pe = prime[i];  
 for (int e = 1; pe <= x; e++, pe \*= prime[i])  
 {  
 LL xx = pe % mo;  
 // 大概这里改ans？原题求p^k\*(p^k-1)  
 ans = (ans + xx \* (xx - 1) % mo \* (S\_(x / pe, i) + (e != 1))) % mo;  
 }  
 }  
 return ans % mo;  
}  
  
  
  
// 递归，分段缓存版本  
unordered\_map<ULL, LL> UM;  
unordered\_map<unsigned, unsigned> IM;  
LL ans[100010];  
vector<vector<pair<LL, LL>>> QUERY(17, vector<pair<LL, LL>>());  
  
unsigned gfi(unsigned n, int j)  
{  
 unsigned mpk = unsigned(j) \* 1000000001 + n;  
 if (IM.count(mpk))  
 return IM[mpk];  
 else  
 {  
 LL ret;  
 if (n < 2)  
 ret = 0;  
 else if (n == 2)  
 ret = 1;  
 else if (j < 1)  
 ret = n - 1;  
 else if (prime[j] \* prime[j] > n)  
 ret = gfi(n, j - 1);  
 else  
 ret = gfi(n, j - 1) - (gfi(n / prime[j], j - 1) - (j - 1));  
 // if (n < 1e9)  
 // return UM[mpk] = ret;  
 return ret;  
 }  
}  
  
LL gf(LL n, LL j)  
{  
 if (n < 1e9)  
 return gfi(unsigned(n), j);  
 ULL mpk = ULL(j) \* 1000000000000000001 + n;  
 if (UM.count(mpk))  
 return UM[mpk];  
 else  
 {  
 LL ret;  
 if (n < 2)  
 ret = 0;  
 else if (n == 2)  
 ret = 1;  
 else if (j < 1)  
 ret = n - 1;  
 else if (prime[j] \* prime[j] > n)  
 ret = gf(n, j - 1);  
 else  
 {  
 // ret = gf(n, j - 1) - (gf(n / prime[j], j - 1)) - (j - 1);  
 ret = gf(n, j - 1);  
 LL dv = n / prime[j];  
 LL ret2 = (n < 1e9 ? gfi(dv, j - 1) : gf(dv, j - 1)) - (j - 1);  
 ret -= ret2;  
 }  
 // if (n < 1e9)  
 // return UM[mpk] = ret;  
 return UM[mpk] = ret;  
 }  
}

#### 卢卡斯定理

LL fact[LUCASM];  
  
inline void get\_fact(LL fact[], LL length, LL mo) // 预处理阶乘  
{  
 fact[0] = 1;  
 fact[1] = 1;  
 for (auto i = 2; i < length; i++)  
 fact[i] = fact[i - 1] \* i % mo;  
}  
// 需要先预处理出fact[]，即阶乘  
inline LL C(LL m, LL n, LL p)  
{  
 return m < n ? 0 : fact[m] \* inv(fact[n], p) % p \* inv(fact[m - n], p) % p;  
}  
inline LL lucas(LL m, LL n, LL p) // 求解大数组合数C(m,n) % p,传入依次是下面那个m和上面那个n和模数p（得是质数  
{  
 return n == 0 ? 1 % p : lucas(m / p, n / p, p) \* C(m % p, n % p, p) % p;  
}

#### EXCRT

inline LL EXCRT(LL factors[], LL remains[], LL length) // 传入除数表，剩余表和两表的长度，若没有解，返回-1，否则返回合适的最小解  
{  
 bool valid = true;  
 for (auto i = 1; i < length; i++)  
 {  
 LL GCD = gcd(factors[i], factors[i - 1]);  
 LL M1 = factors[i];  
 LL M2 = factors[i - 1];  
 LL C1 = remains[i];  
 LL C2 = remains[i - 1];  
 LL LCM = M1 \* M2 / GCD;  
 if ((C1 - C2) % GCD != 0)  
 {  
 valid = false;  
 break;  
 }  
 factors[i] = LCM;  
 remains[i] = (inv(M2 / GCD, M1 / GCD) \* (C1 - C2) / GCD) % (M1 / GCD) \* M2 + C2; // 对应合并公式  
 remains[i] = (remains[i] % factors[i] + factors[i]) % factors[i]; // 转正  
 }  
 return valid ? remains[length - 1] : -1;  
}

#### 扩欧求逆元

inline void exgcd(LL a, LL b, LL &x, LL &y)  
{  
 if (!b)  
 {  
 x = 1;  
 y = 0;  
 return;  
 }  
 exgcd(b, a % b, y, x);  
 y -= a / b \* x;  
}  
inline LL inv(LL a, LL mo)  
{  
 LL x, y;  
 exgcd(a, mo, x, y);  
 return x >= 0 ? x : x + mo;  
}

#### 递推求逆元

//递推求法  
std::vector<LL> getInvRecursion(LL upp, LL mod)  
{  
 std::vector<LL> vinv(1, 0);  
 vinv.emplace\_back(1);  
 for (LL i = 2; i <= upp; i++)  
 vinv.emplace\_back((mod - mod / i) \* vinv[mod % i] % mod);  
 return vinv;  
}

#### 多项式

/\*  
g 是mod(r\*2^k+1)的原根  
素数 r k g  
3 1 1 2  
5 1 2 2  
17 1 4 3  
97 3 5 5  
193 3 6 5  
257 1 8 3  
7681 15 9 17  
12289 3 12 11  
40961 5 13 3  
65537 1 16 3  
786433 3 18 10  
5767169 11 19 3  
7340033 7 20 3  
23068673 11 21 3  
104857601 25 22 3  
167772161 5 25 3  
469762049 7 26 3  
1004535809 479 21 3  
2013265921 15 27 31  
2281701377 17 27 3  
3221225473 3 30 5  
75161927681 35 31 3  
77309411329 9 33 7  
206158430209 3 36 22  
2061584302081 15 37 7  
2748779069441 5 39 3  
6597069766657 3 41 5  
39582418599937 9 42 5  
79164837199873 9 43 5  
263882790666241 15 44 7  
1231453023109121 35 45 3  
1337006139375617 19 46 3  
3799912185593857 27 47 5  
4222124650659841 15 48 19  
7881299347898369 7 50 6  
31525197391593473 7 52 3  
180143985094819841 5 55 6  
1945555039024054273 27 56 5  
4179340454199820289 29 57 3  
\*/  
  
/\* 多项式 2021.10.31 \*/  
template <typename T>  
struct Polynomial  
{  
 std::vector<T> cof; // 各项系数 coefficient 低次在前高次在后  
 LL mod = 998244353; // 模数  
 LL G = 3; // 原根  
 LL Gi = 332748118; // 原根的逆元  
 using pointval = std::pair<T, T>;  
 std::vector<pointval> points; // x在前y在后  
  
 inline LL modadd(LL &x, LL y) { return (x += y) >= mod ? x -= mod : x; }  
 inline LL modsub(LL &x, LL y) { return (x -= y) < 0 ? x += mod : x; }  
 inline LL madd(LL x, LL y) { return (x += y) >= mod ? x - mod : x; }  
 inline LL msub(LL x, LL y) { return (x -= y) < 0 ? x + mod : x; }  
  
 Polynomial() {}  
 Polynomial(int siz) : cof(siz) {}  
  
 /\* n^2 拉格朗日插值 \*/  
 void interpolation()  
 {  
 cof.assign(points.size(), 0);  
 std::vector<T> num(cof.size() + 1, 0);  
 std::vector<T> tmp(cof.size() + 1, 0);  
 std::vector<T> invs(cof.size(), 0);  
 num[0] = 1;  
 for (int i = 1; i <= cof.size(); swap(num, tmp), ++i)  
 {  
 tmp[0] = 0;  
 invs[i - 1] = inv(mod - points[i - 1].first, mod);  
 for (int j = 1; j <= i; ++j)  
 tmp[j] = num[j - 1];  
 for (int j = 0; j <= i; ++j)  
 modadd(tmp[j], num[j] \* (mod - points[i - 1].first) % mod);  
 }  
 for (int i = 1; i <= cof.size(); ++i)  
 {  
 T den = 1, lst = 0;  
 for (int j = 1; j <= cof.size(); ++j)  
 if (i != j)  
 den = den \* (points[i - 1].first - points[j - 1].first + mod) % mod;  
 den = points[i - 1].second \* inv(den) % mod;  
 for (int j = 0; j < cof.size(); ++j)  
 {  
 tmp[j] = (num[j] - lst + mod) \* invs[i - 1] % mod;  
 modadd(cof[j], den \* tmp[j] % mod), lst = tmp[j];  
 }  
 }  
 }  
 /\* 给f(0)~f(n)，求f(m) \*/  
 T interpolation\_continuity\_single(T m, T beg = 0) const  
 {  
 T n = cof.size();  
 if (m >= beg and m <= beg + n - 1)  
 return cof[m - beg];  
 vector<T> fac(beg + n + 1, 1);  
 vector<T> facinv(beg + n + 1, 1);  
  
 for (int i = 2; i <= fac.size() - 1; ++i)  
 fac[i] = ((LL)fac[i - 1] \* i % mod);  
 facinv[n] = inv(fac[n], mod);  
 for (int i = facinv.size() - 1; i > 1; --i)  
 facinv[i - 1] = (LL)facinv[i] \* (i) % mod;  
 vector<T> krr(1, m - beg);  
 for (int i = 1; i < n; ++i)  
 krr.emplace\_back(((m - beg - i) % mod + mod) % mod);  
 vector<T> pre(1, 1);  
 vector<T> suf(1, 1);  
 for (auto i : krr)  
 pre.emplace\_back((LL)pre.back() \* i % mod);  
 for (auto i = krr.rbegin(); i != krr.rend(); ++i)  
 suf.emplace\_back((LL)suf.back() \* (\*i) % mod);  
 reverse(suf.begin(), suf.end());  
 T ret = 0;  
 for (int i = 0; i < n; ++i)  
 {  
 T cur = (LL)cof[i] \* pre[i] % mod \* suf[i + 1] % mod \* facinv[i] % mod \* (facinv[n - i - 1]) % mod;  
 if (n - i - 1 & 1)  
 cur = msub(mod, cur);  
 ret = madd(ret, cur);  
 }  
 return ret;  
 }  
  
 /\* P5667 给f(0)~f(n)，算f(m)~f(m+n)，nlogn，int安全，1.6e5下710ms \*/  
 Polynomial interpolation\_continuity(T m) const  
 {  
 Polynomial B;  
  
 T n = cof.size() - 1;  
 T nbound = n << 1 | 1;  
 B.cof.resize(nbound);  
 vector<T> fac(nbound + 1); // 阶乘  
 vector<T> facinv(nbound + 1); // 阶乘的逆元  
 vector<T> Bfac(nbound + 1); // B数组的分母前缀积  
 vector<T> Bfacinv(nbound + 1); // B数组的分母前缀积的逆元  
 // vector<T> Binv(nbound + 1); // B数组的分母的逆元，即B真正存的东西  
 fac[0] = Bfac[0] = 1;  
 for (int i = 1; i <= nbound; ++i)  
 {  
 fac[i] = (LL)fac[i - 1] \* i % mod;  
 Bfac[i] = (LL)Bfac[i - 1] \* (m - n + i - 1) % mod;  
 }  
  
 Bfacinv.back() = inv(Bfac.back(), mod);  
 facinv.back() = inv(fac.back(), mod);  
  
 for (int i = nbound; i; --i)  
 {  
 facinv[i - 1] = (LL)facinv[i] \* i % mod;  
 Bfacinv[i - 1] = (LL)Bfacinv[i] \* (m - n + i - 1) % mod;  
 // Binv[i]  
 B.cof[i - 1] = (LL)Bfacinv[i] \* Bfac[i - 1] % mod;  
 }  
 for (int i = 0; i <= n; ++i)  
 {  
 cof[i] = (LL)cof[i] \* facinv[i] % mod \* facinv[n - i] % mod;  
 if (n - i & 1)  
 cof[i] = mod - cof[i];  
 }  
 Polynomial C = NTTMul(\*this, B);  
 for (int i = n; i < nbound; ++i)  
 {  
 B.cof[i - n] = (LL)Bfac[i + 1] \* Bfacinv[i - n] % mod \* C.cof[i] % mod;  
 }  
 B.cof.resize(nbound - n);  
 return B;  
 }  
  
 /\* 计算多项式在x这点的值 \*/  
 T eval(T x) const  
 {  
 T ret = 0, px = 1;  
 for (auto i : cof)  
 {  
 modadd(ret, i \* px % mod);  
 px = px \* x % mod;  
 }  
 return ret;  
 }  
  
 /\* rev是蝴蝶操作数组，lim为填充到2的幂的值，mode为0正变换，1逆变换，逆变换后系数需要除以lim才是答案 \*/  
 void NTT(const std::vector<int> &rev, int lim, bool mode = 0)  
 {  
 int l;  
 for (int i = 0; i < lim; ++i)  
 if (i < rev[i])  
 swap(cof[i], cof[rev[i]]);  
 for (int mid = 1; mid < lim; mid = l)  
 {  
 l = mid << 1;  
 T Wn = power(mode ? Gi : G, (mod - 1) / (mid << 1), mod);  
 for (int j = 0; j < lim; j += l)  
 {  
 T w = 1;  
 for (int k = 0; k < mid; ++k, w = ((LL)w \* Wn) % mod)  
 {  
 T x = cof[j | k], y = (LL)w \* cof[j | k | mid] % mod;  
 cof[j | k] = madd(x, y); // 已经不得不用这个优化了  
 cof[j | k | mid] = msub(x, y);  
 }  
 }  
 }  
 if (mode)  
 {  
 T iv = inv(lim, mod);  
 for (auto &i : cof)  
 i = ((LL)i \* iv) % mod;  
 }  
 }  
  
 /\* FWT or变换，mode=1为逆变换 \*/  
 void FWTor(int limpow, bool mode = 0)  
 {  
 T m = (mode ? -1 : 1);  
 int i, j, k;  
 for (i = 1; i <= limpow; ++i)  
 for (j = 0; j < (1 << limpow); j += 1 << i)  
 for (k = 0; k < (1 << i - 1); ++k)  
 cof[j | (1 << i - 1) | k] += cof[j | k] \* m;  
 }  
 /\* FWT and变换，mode=1为逆变换 \*/  
 void FWTand(int limpow, bool mode = 0)  
 {  
 T m = (mode ? -1 : 1);  
 int i, j, k;  
 for (i = 1; i <= limpow; ++i)  
 for (j = 0; j < (1 << limpow); j += 1 << i)  
 for (k = 0; k < (1 << i - 1); ++k)  
 cof[j | k] += cof[j | (1 << i - 1) | k] \* m;  
 }  
 /\* FWT xor变换，mode=1为逆变换 \*/  
 void FWTxor(int limpow, bool mode = 0)  
 {  
 T m = (mode ? T(1) / T(2) : 1);  
 int i, j, k;  
 T x, y;  
 for (i = 1; i <= limpow; ++i)  
 for (j = 0; j < (1 << limpow); j += 1 << i)  
 for (k = 0; k < (1 << i - 1); ++k)  
 x = (cof[j | k] + cof[j | (1 << i - 1) | k]) \* m,  
 y = (cof[j | k] - cof[j | (1 << i - 1) | k]) \* m,  
 cof[j | k] = x,  
 cof[j | (1 << i - 1) | k] = y;  
 }  
  
 Polynomial operator|(const Polynomial &b) const  
 {  
 int lim, limpow, retsiz;  
 Polynomial A(\*this);  
 Polynomial B(b);  
 Resize(A, B, lim, limpow, retsiz);  
 A.FWTor(limpow);  
 B.FWTor(limpow);  
 for (int i = 0; i < lim; ++i)  
 A.cof[i] \*= B.cof[i];  
 A.FWTor(limpow, 1);  
 A.cof.resize(retsiz);  
 return A;  
 }  
 Polynomial operator&(const Polynomial &b) const  
 {  
 int lim, limpow, retsiz;  
 Polynomial A(\*this);  
 Polynomial B(b);  
 Resize(A, B, lim, limpow, retsiz);  
 A.FWTand(limpow);  
 B.FWTand(limpow);  
 for (int i = 0; i < lim; ++i)  
 A.cof[i] \*= B.cof[i];  
 A.FWTand(limpow, 1);  
 A.cof.resize(retsiz);  
 return A;  
 }  
 Polynomial operator^(const Polynomial &b) const  
 {  
 int lim, limpow, retsiz;  
 Polynomial A(\*this);  
 Polynomial B(b);  
 Resize(A, B, lim, limpow, retsiz);  
 A.FWTxor(limpow);  
 B.FWTxor(limpow);  
 for (int i = 0; i < lim; ++i)  
 A.cof[i] \*= B.cof[i];  
 A.FWTxor(limpow, 1);  
 A.cof.resize(retsiz);  
 return A;  
 }  
  
 /\* 精度更高的写法 \*/  
 void FFT(const std::vector<int> &rev, int n, bool mode, const std::vector<T> &Wn)  
 {  
 if (mode)  
 for (int i = 1; i < n; i++)  
 if (i < (n - i))  
 std::swap(cof[i], cof[n - i]);  
 for (int i = 0; i < n; i++)  
 if (i < rev[i])  
 std::swap(cof[i], cof[rev[i]]);  
  
 for (int m = 1, l = 0; m < n; m <<= 1, l++)  
 {  
 for (int i = 0; i < n; i += m << 1)  
 {  
 for (int k = i; k < i + m; k++)  
 {  
 T W = Wn[1ll \* (k - i) \* n / m];  
 T a0 = cof[k], a1 = cof[k + m] \* W;  
 cof[k] = a0 + a1;  
 cof[k + m] = a0 - a1;  
 }  
 }  
 }  
 if (mode)  
 for (auto &i : cof)  
 i /= n;  
 }  
  
 /\* 多项式求逆，建议模数满足原根时使用，1e5 O2 331ms，无O2 612ms, 写成循环只优化了空间 \*/  
 void N\_inv(int siz, Polynomial &B) const  
 {  
 B.cof.emplace\_back(inv(cof[0], mod));  
 int bas = 2, lim = 4, limpow = 2;  
 Polynomial A;  
 while (bas < (siz << 1))  
 {  
 B.cof.resize(lim, 0);  
 if (bas <= cof.size())  
 A.cof.assign(cof.begin(), cof.begin() + bas);  
 else  
 A.cof = cof;  
 A.cof.resize(lim, 0);  
 std::vector<int> rev(generateRev(lim, limpow));  
 A.NTT(rev, lim, 0);  
 B.NTT(rev, lim, 0);  
 for (int i = 0; i < lim; ++i)  
 B.cof[i] = (LL)B.cof[i] \* (2 + mod - (LL)B.cof[i] \* A.cof[i] % mod) % mod;  
 B.NTT(rev, lim, 1);  
 std::fill(B.cof.begin() + bas, B.cof.end(), 0);  
 bas <<= 1;  
 lim <<= 1;  
 ++limpow;  
 }  
 B.cof.resize(siz);  
 }  
 /\* 两次MTT的任意模数多项式求逆，1e5 O2 550ms，无O2 2.11s \*/  
 void F\_inv(int siz, Polynomial &B) const  
 {  
 if (siz == 1)  
 {  
 B.cof.emplace\_back(inv(LL(round(cof[0].real())), mod));  
 return;  
 }  
 F\_inv((siz + 1) >> 1, B);  
 Polynomial C;  
 C.cof.assign(cof.begin(), cof.begin() + siz);  
 Polynomial BC(MTT\_FFT(B, C));  
 for (auto &i : BC.cof)  
 i = LL(round(i.real())) % mod;  
 Polynomial BBC(MTT\_FFT(BC, B));  
 B.cof.resize(siz, 0);  
 for (int i = 0; i < siz; ++i)  
 {  
 B.cof[i] = msub(  
 madd(  
 LL(round(B.cof[i].real())),  
 LL(round(B.cof[i].real()))),  
 LL(round(BBC.cof[i].real())) % mod);  
 }  
 }  
 /\* G2 = (G1^2 + A)/2G1 \*/  
 Polynomial getsqrt() const  
 {  
 Polynomial B;  
 int siz = cof.size();  
 LL s1, s2;  
 Cipolla<LL>::solve((LL)cof[0], (LL)mod, s1, s2);  
 if (s2 < s1)  
 swap(s2, s1);  
 B.cof.emplace\_back(s1);  
 LL bas = 2, lim = 4, limpow = 2;  
 Polynomial A;  
 // T inv2 = inv(2, mod);  
 while (bas < (siz << 1))  
 {  
 Polynomial Binv(B.getinv(bas));  
 B.cof.resize(lim, 0);  
 if (bas <= cof.size())  
 A.cof.assign(cof.begin(), cof.begin() + bas);  
 else  
 A.cof = cof;  
 A.cof.resize(lim, 0);  
 std::vector<int> rev(generateRev(lim, limpow));  
 Binv.cof.resize(lim);  
 A.NTT(rev, lim, 0);  
 B.NTT(rev, lim, 0);  
 Binv.NTT(rev, lim, 0);  
  
 for (int i = 0; i < lim; ++i)  
 {  
 B.cof[i] = (LL)Binv.cof[i] \* (A.cof[i] + (LL)B.cof[i] \* B.cof[i] % mod) % mod;  
 B.cof[i] = (B.cof[i] & 1) ? (B.cof[i] + mod >> 1) : B.cof[i] >> 1;  
 }  
  
 B.NTT(rev, lim, 1);  
 std::fill(B.cof.begin() + bas, B.cof.end(), 0);  
 bas <<= 1;  
 lim <<= 1;  
 ++limpow;  
 }  
 B.cof.resize(siz);  
 return B;  
 }  
  
 /\* siz为需要求的多项式逆的次数，为0时默认取自己次数的 \*/  
 Polynomial getinv(int siz = 0) const  
 {  
 if (!siz)  
 siz = cof.size();  
 int orisiz = cof.size();  
 cof.resize(siz);  
 Polynomial B;  
 // LL lim, limpow, retsize;  
 // Resize(\*this, \*this, lim, limpow, retsize);  
 N\_inv(siz, B); // N\_inv为使用NTT，F\_inv为使用MTT  
 B.cof.resize(siz);  
 cof.resize(orisiz);  
 return B;  
 }  
  
 Polynomial operator\*(const Polynomial &rhs) const  
 {  
 return NTTMul(\*this, rhs);  
 }  
 /\* 获取F(x) = G(x) \* Q(x) + R(x)的Q(x) \*/  
 Polynomial operator/(const Polynomial &G) const  
 {  
 Polynomial F(\*this);  
 int beforen = F.cof.size();  
 std::reverse(F.cof.begin(), F.cof.end());  
 std::reverse(G.cof.begin(), G.cof.end());  
 int beforem = G.cof.size();  
 F.cof.resize(beforen - beforem + 1);  
 Polynomial tmp(F \* G.getinv(beforen));  
 // G.cof.resize(beforem);  
 tmp.cof.resize(beforen - beforem + 1);  
 std::reverse(tmp.cof.begin(), tmp.cof.end());  
 // std::reverse(cof.begin(), cof.end());  
 return tmp;  
 }  
  
 /\* 获取F(x) = G(x) \* Q(x) + R(x)的R(x) \*/  
 static Polynomial getremain(const Polynomial &F, const Polynomial &G, const Polynomial &Q)  
 {  
 Polynomial C(G \* Q);  
 C.cof.resize(G.cof.size() - 1);  
 for (int i = 0; i < G.cof.size() - 1; ++i)  
 C.cof[i] = F.msub(F.cof[i], C.cof[i]);  
 return C;  
 }  
  
 static std::vector<int> generateRev(int lim, int limpow)  
 {  
 std::vector<int> rev(lim, 0);  
 for (int i = 0; i < lim; ++i)  
 rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (limpow - 1));  
 return rev;  
 }  
  
 static std::vector<T> generateWn(int lim)  
 {  
 std::vector<T> Wn;  
 for (int i = 0; i < lim; i++)  
 Wn.emplace\_back(cos(M\_PI / lim \* i), sin(M\_PI / lim \* i));  
 return Wn;  
 }  
  
 /\* NTT卷积 板题4.72s \*/  
 static Polynomial NTTMul(Polynomial A, Polynomial B)  
 {  
 int lim, limpow, retsiz;  
 Resize(A, B, lim, limpow, retsiz);  
  
 std::vector<int> rev(generateRev(lim, limpow));  
 A.NTT(rev, lim, 0);  
 B.NTT(rev, lim, 0);  
 for (int i = 0; i < lim; i++)  
 A.cof[i] = ((LL)A.cof[i] \* B.cof[i] % A.mod);  
 A.NTT(rev, lim, 1);  
  
 A.cof.resize(retsiz - 1);  
  
 return A;  
 }  
 /\* FFT卷积 板题1.98s 使用手写复数 -> 1.33s\*/  
 static Polynomial FFTMul(Polynomial A, Polynomial B)  
 {  
 int lim, limpow, retsiz;  
 Resize(A, B, lim, limpow, retsiz);  
  
 std::vector<int> rev(generateRev(lim, limpow));  
 std::vector<T> Wn(generateWn(lim));  
 A.FFT(rev, lim, 0, Wn);  
 B.FFT(rev, lim, 0, Wn);  
 for (int i = 0; i < lim; ++i)  
 A.cof[i] \*= B.cof[i];  
 A.FFT(rev, lim, 1, Wn);  
  
 A.cof.resize(retsiz - 1);  
 return A;  
 }  
  
 inline static void Resize(Polynomial &A, Polynomial &B, int &lim, int &limpow, int &retsiz)  
 {  
 lim = 1;  
 limpow = 0;  
 retsiz = A.cof.size() + B.cof.size();  
 while (lim <= retsiz)  
 lim <<= 1, ++limpow;  
 A.cof.resize(lim, 0);  
 B.cof.resize(lim, 0);  
 }  
  
 static Polynomial MTT\_FFT(const Polynomial &A, const Polynomial &B)  
 {  
 int lim, limpow, retsiz;  
 Polynomial A0, B0;  
 LL thr = sqrt(A.mod) + 1; // 拆系数阈值  
 for (auto i : A.cof)  
 {  
 LL tmp = i.real();  
 A0.cof.emplace\_back(tmp / thr, tmp % thr);  
 }  
 for (auto i : B.cof)  
 {  
 LL tmp = i.real();  
 B0.cof.emplace\_back(tmp / thr, tmp % thr);  
 }  
 Resize(A0, B0, lim, limpow, retsiz);  
  
 std::vector<int> rev(generateRev(lim, limpow));  
 std::vector<T> Wn(generateWn(lim));  
  
 A0.FFT(rev, lim, 0, Wn);  
 B0.FFT(rev, lim, 0, Wn);  
 std::vector<T> Acp(A0.cof);  
 std::vector<T> Bcp(B0.cof);  
 const T IV(0, 1);  
 const T half(0.5);  
 for (int ii = 0; ii < lim; ++ii)  
 {  
 T i = A0.cof[ii];  
 T j = (Acp[ii ? lim - ii : 0]).conj();  
 T a0 = (j + i) \* half;  
 T a1 = (j - i) \* half \* IV;  
 i = B0.cof[ii];  
 j = (Bcp[ii ? lim - ii : 0]).conj();  
 T b0 = (j + i) \* half;  
 T b1 = (j - i) \* half \* IV;  
 A0.cof[ii] = a0 \* b0 + IV \* a1 \* b0;  
 B0.cof[ii] = a0 \* b1 + IV \* a1 \* b1;  
 }  
 A0.FFT(rev, lim, 1, Wn);  
 B0.FFT(rev, lim, 1, Wn);  
  
 for (int i = 0; i < retsiz - 1; ++i)  
 {  
 T &ac = A0.cof[i];  
 T &bc = B0.cof[i];  
 A0.cof[i] = thr \* thr \* (\_\_int128)round(ac.real()) % A.mod +  
 thr \* (\_\_int128)round(ac.imag() + bc.real()) % A.mod +  
 (\_\_int128)round(bc.imag()) % A.mod;  
 }  
 A0.cof.resize(retsiz - 1);  
 return A0;  
 }  
};  
/\* 使用手写的以后 2.00s -> 1.33s\*/  
template <typename T>  
struct Complex  
{  
 T re\_al, im\_ag;  
 inline T &real() { return re\_al; }  
 inline T &imag() { return im\_ag; }  
 Complex() { re\_al = im\_ag = 0; }  
 Complex(T x) : re\_al(x), im\_ag(0) {}  
 Complex(T x, T y) : re\_al(x), im\_ag(y) {}  
 inline Complex conj() { return Complex(re\_al, -im\_ag); }  
 inline Complex operator+(Complex rhs) const { return Complex(re\_al + rhs.re\_al, im\_ag + rhs.im\_ag); }  
 inline Complex operator-(Complex rhs) const { return Complex(re\_al - rhs.re\_al, im\_ag - rhs.im\_ag); }  
 inline Complex operator\*(Complex rhs) const { return Complex(re\_al \* rhs.re\_al - im\_ag \* rhs.im\_ag,  
 im\_ag \* rhs.re\_al + re\_al \* rhs.im\_ag); }  
 inline Complex operator\*=(Complex rhs) { return (\*this) = (\*this) \* rhs; }  
 //(a+bi)(c+di) = (ac-bd) + (bc+ad)i  
 friend inline Complex operator\*(T x, Complex cp) { return Complex(x \* cp.re\_al, x \* cp.im\_ag); }  
 inline Complex operator/(T x) const { return Complex(re\_al / x, im\_ag / x); }  
 inline Complex operator/=(T x) { return (\*this) = (\*this) / x; }  
 friend inline Complex operator/(T x, Complex cp) { return x \* cp.conj() / (cp.re\_al \* cp.re\_al - cp.im\_ag \* cp.im\_ag); }  
 inline Complex operator/(Complex rhs) const  
 {  
 return (\*this) \* rhs.conj() / (rhs.re\_al \* rhs.re\_al - rhs.im\_ag \* rhs.im\_ag);  
 }  
 inline Complex operator/=(Complex rhs) { return (\*this) = (\*this) / rhs; }  
 inline Complex operator=(T x)  
 {  
 this->im\_ag = 0;  
 this->re\_al = x;  
 return \*this;  
 }  
 inline T length() { return sqrt(re\_al \* re\_al + im\_ag \* im\_ag); }  
};  
using \_MTT = Complex<double>;  
using \_NTT = long long;

### 公式

卡特兰数 K(x) = C(2\*x, x) / (x + 1)

### 自然数幂和表

MP = {  
 0:"1 1 0",  
 1:"2 1 1 0",  
 2:"6 2 3 1 0",  
 3:"4 1 2 1 0 0",  
 4:"30 6 15 10 0 -1 0",  
 5:"12 2 6 5 0 -1 0 0",  
 6:"42 6 21 21 0 -7 0 1 0",  
 7:"24 3 12 14 0 -7 0 2 0 0",  
 8:"90 10 45 60 0 -42 0 20 0 -3 0",  
 9:"20 2 10 15 0 -14 0 10 0 -3 0 0",  
 10:"66 6 33 55 0 -66 0 66 0 -33 0 5 0",  
 11:"24 2 12 22 0 -33 0 44 0 -33 0 10 0 0",  
 12:"2730 210 1365 2730 0 -5005 0 8580 0 -9009 0 4550 0 -691 0",  
 13:"420 30 210 455 0 -1001 0 2145 0 -3003 0 2275 0 -691 0 0",  
 14:"90 6 45 105 0 -273 0 715 0 -1287 0 1365 0 -691 0 105 0",  
 15:"48 3 24 60 0 -182 0 572 0 -1287 0 1820 0 -1382 0 420 0 0",  
 16:"510 30 255 680 0 -2380 0 8840 0 -24310 0 44200 0 -46988 0 23800 0 -3617 0",  
 17:"180 10 90 255 0 -1020 0 4420 0 -14586 0 33150 0 -46988 0 35700 0 -10851 0 0",  
 18:"3990 210 1995 5985 0 -27132 0 135660 0 -529074 0 1469650 0 -2678316 0 2848860 0 -1443183 0 219335 0",  
 19:"840 42 420 1330 0 -6783 0 38760 0 -176358 0 587860 0 -1339158 0 1899240 0 -1443183 0 438670 0 0",  
 20:"6930 330 3465 11550 0 -65835 0 426360 0 -2238390 0 8817900 0 -24551230 0 44767800 0 -47625039 0 24126850 0 -3666831 0"  
}

### 来自bot的球盒问题

def A072233\_list(n: int, m: int, mod=0) -> list:  
 """n个无差别球塞进m个无差别盒子方案数"""  
 mod = int(mod)  
 f = [[0] \* (m + 1)] \* (n + 1)  
 f[0][0] = 1  
 for i in range(1, n+1):  
 for j in range(1, min(i+1, m+1)): # 只是求到m了话没必要打更大的  
 f[i][j] = f[i-1][j-1] + f[i-j][j]  
 if mod: f[i][j] %= mod  
 return f  
  
def A048993\_list(n: int, m: int, mod=0) -> list:  
 """第二类斯特林数"""  
 mod = int(mod)  
 f = [1] + [0] \* m  
 for i in range(1, n+1):  
 for j in range(min(m, i), 0, -1):  
 f[j] = f[j-1] + f[j] \* j  
 if mod: f[j] %= mod  
 f[0] = 0  
 return f  
  
  
def A000110\_list(m, mod=0):  
 """集合划分方案总和，或者叫贝尔数"""  
 mod = int(mod)  
 A = [0 for i in range(m)]  
 # m -= 1  
 A[0] = 1  
 # R = [1, 1]  
 for n in range(1, m):  
 A[n] = A[0]  
 for k in range(n, 0, -1):  
 A[k-1] += A[k]  
 if mod: A[k-1] %= mod  
 # R.append(A[0])  
 # return R  
 return A[0]  
  
async def 球盒(\*attrs, kwargs={}):  
 """求解把n个球放进m个盒子里面有多少种方案的问题。  
必须指定盒子和球以及允不允许为空三个属性。  
用法：  
 #球盒 <盒子相同？(0/1)><球相同？(0/1)><允许空盒子？(0/1)> n m  
用例：  
 #球盒 110 20 5  
 上述命令求的是盒子相同，球相同，不允许空盒子的情况下将20个球放入5个盒子的方案数。"""  
 # 参考https://www.cnblogs.com/sdfzsyq/p/9838857.html的算法  
 if len(attrs)!=3:  
 return '不是这么用的！请输入#h #球盒'  
 n, m = map(int, attrs[1:3])  
 if attrs[0] == '110':  
 f = A072233\_list(n, m)  
 return f[n][m]  
 elif attrs[0] == '111':  
 f = A072233\_list(n, m)  
 return sum(f[-1])  
 elif attrs[0] == '100':  
 return A048993\_list(n, m)[-1]  
 elif attrs[0] == '101':  
 return sum(A048993\_list(n, m))  
 elif attrs[0] == '010':  
 return comb(n-1, m-1)  
 elif attrs[0] == '011':  
 return comb(n+m-1, m-1)  
 elif attrs[0] == '000': # 求两个集合的满射函数的个数可以用  
 return A048993\_list(n, m)[-1] \* math.factorial(m)  
 elif attrs[0] == '001':  
 return m\*\*n

## OTHER

### 模数类

/\* 静态模数类，只能用有符号类型做T和EXT参数 \*/  
template <int mod, class T = int, class EXT = long long>  
struct mint  
{  
 T x;  
 template <class TT>  
 mint(TT \_x)  
 {  
 x = EXT(\_x) % mod;  
 if (x < 0)  
 x += mod;  
 }  
 mint() : x(0) {}  
 mint &operator++()  
 {  
 ++x;  
 if (x == mod)  
 x = 0;  
 return \*this;  
 }  
 mint &operator--()  
 {  
 x = (x == 0 ? mod - 1 : x - 1);  
 return \*this;  
 }  
 mint operator++(int)  
 {  
 mint tmp = \*this;  
 ++\*this;  
 return tmp;  
 }  
 mint operator--(int)  
 {  
 mint tmp = \*this;  
 --\*this;  
 return tmp;  
 }  
 mint &operator+=(const mint &rhs)  
 {  
 x += rhs.x;  
 if (x >= mod)  
 x -= mod;  
 return \*this;  
 }  
 mint &operator-=(const mint &rhs)  
 {  
 x -= rhs.x;  
 if (x < 0)  
 x += mod;  
 return \*this;  
 }  
 mint &operator\*=(const mint &rhs)  
 {  
 x = EXT(x) \* rhs.x % mod;  
 return \*this;  
 }  
 mint &operator/=(const mint &rhs)  
 {  
 x = EXT(x) \* inv(rhs.x, mod) % mod;  
 return \*this;  
 }  
 mint operator+() const { return \*this; }  
 mint operator-() const { return mod - \*this; }  
 friend mint operator+(const mint &lhs, const mint &rhs) { return mint(lhs) += rhs; }  
 friend mint operator-(const mint &lhs, const mint &rhs) { return mint(lhs) -= rhs; }  
 friend mint operator\*(const mint &lhs, const mint &rhs) { return mint(lhs) \*= rhs; }  
 friend mint operator/(const mint &lhs, const mint &rhs) { return mint(lhs) /= rhs; }  
 friend bool operator==(const mint &lhs, const mint &rhs) { return lhs.x == rhs.x; }  
 friend bool operator!=(const mint &lhs, const mint &rhs) { return lhs.x != rhs.x; }  
 friend std::ostream &operator<<(std::ostream &o, const mint &m) { return o << m.x; }  
 friend std::istream &operator>>(std::istream &i, const mint &m)  
 {  
 i >> m.x;  
 m.x %= mod;  
 if (m.x < 0)  
 m.x += mod;  
 return i;  
 }  
};  
using moha = mint<19260817>;  
using m998 = mint<998244353>;  
using m1e9\_7 = mint<1000000007>;  
using m1e9\_9 = mint<1000000009>;

### 常用宏及函数与快读

#include <ext/pb\_ds/tree\_policy.hpp>  
#include <ext/pb\_ds/assoc\_container.hpp>  
\_\_gnu\_pbds::tree<int, \_\_gnu\_pbds::null\_type, std::less<int>, \_\_gnu\_pbds::rb\_tree\_tag, \_\_gnu\_pbds::tree\_order\_statistics\_node\_update> TTT;  
  
// 函数不返回值可能会 RE  
// 少码大数据结构，想想复杂度更优的做法  
// 小数 二分/三分 注意break条件  
// 浮点运算 sqrt(a^2-b^2) 可用 sqrt(a+b)\*sqrt(a-b) 代替，避免精度问题  
// long double -> %Lf 别用C11 (C14/16)  
// 控制位数 cout << setprecision(10) << ans;  
// reverse vector 注意判空 不然会re  
// 分块注意维护块上标记 来更新块内数组a[]  
// vector+lower\_bound常数 < map/set/(unordered\_map)  
// map.find不会创建新元素 map[]会 注意空间  
// 别对指针用memset  
// 用位运算表示2^n注意加LL 1LL<<20  
// 注意递归爆栈  
// 注意边界  
// 注意memset 多组会T  
  
// lambda  
  
// sort(p + 1, p + 1 + n,  
// [](const point &x, const point &y) -> bool { return x.x < y.x; });  
  
// append l1 to l2 (l1 unchanged)  
  
// l2.insert(l2.end(),l1.begin(),l1.end());  
  
// append l1 to l2 (elements appended to l2 are removed from l1)  
// (general form ... TG gave form that is actually better suited  
// for your needs)  
  
// l2.splice(l2.end(),l1,l1.begin(),l1.end());  
  
//位运算函数  
//int \_\_builtin\_ffs (unsigned int x最后一位1的是从后向前第几位，1110011001000 返回4  
//int \_\_builtin\_clz (unsigned int x)前导0个数  
//int \_\_builtin\_ctz (unsigned int x)末尾0个数  
//int \_\_builtin\_popcount (unsigned int x) 1的个数  
//此外，这些函数都有相应的usigned long和usigned long long版本，只需要在函数名后面加上l或ll就可以了，比如int \_\_builtin\_clzll。  
  
//java大数  
//import java.io.\*;  
//import java.math.BigInteger;  
//import java.util.\*;  
//public class Main {  
// public static void main(String args[]) throws Exception {  
// Scanner cin=new Scanner(System.in);  
// BigInteger a;  
// BigInteger b;  
// a = cin.nextBigInteger();  
// b = cin.nextBigInteger();  
//  
// System.out.println(a.add(b));  
// }  
//}  
//  
  
double randreal(double begin, double end)  
{  
 static std::default\_random\_engine eng(time(0));  
 std::uniform\_real\_distribution<> skip\_rate(begin, end);  
 return skip\_rate(eng);  
}  
  
int randint(int begin, int end)  
{  
 static std::default\_random\_engine eng(time(0));  
 std::uniform\_int\_distribution<> skip\_rate(begin, end);  
 return skip\_rate(eng);  
}

### 石子合并 4e4

找到第一个a[i]满足a[i-1]<=a[i+1]，将他俩合并。

从第i位往前找第一个a[k]满足a[k]>刚才的合并结果。

将合并结果放在k位置之后，若无满足条件的k，放在第一个位置。若i不存在，直接合并最后两个数。

#include <bits/stdc++.h>  
using namespace std;  
#define ll long long  
const ll N=41000;  
ll n,a[N],ans,now=1,pro;  
int main()  
{  
 scanf("%lld",&n);  
 for(ll i=1;i<=n;i++) scanf("%lld",&a[i]);  
 while(now<n-1)  
 {  
 for(pro=now;pro<n-1;pro++)  
 {  
 if(a[pro+2]<a[pro]) continue;  
 a[pro+1]+=a[pro];  
 ans+=a[pro+1];ll k;  
 for(k=pro;k>now;k--) a[k]=a[k-1];   
 now++; k=pro+1;  
 while(now<k&&a[k-1]<a[k]) {a[k]^=a[k-1]^=a[k]^=a[k-1];k--;}  
 break;  
 }  
 if(pro==n-1) {a[n-1]+=a[n];ans+=a[n-1];n--;}  
 }  
 if(now==n-1) ans+=(a[n-1]+a[n]);   
 printf("%lld\n",ans);  
 return 0;  
}

### 常见博弈

### 质数表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1e2 | 1e3 | 1e4 | 1e5 | 1e6 |
| 101 | 1009 | 10007 | 100003 | 1000003 |
| 103 | 1013 | 10009 | 100019 | 1000033 |
| 107 | 1019 | 10037 | 100043 | 1000037 |
| 109 | 1021 | 10039 | 100049 | 1000039 |
| 113 | 1031 | 10061 | 100057 | 1000081 |
| 127 | 1033 | 10067 | 100069 | 1000099 |
| 131 | 1039 | 10069 | 100103 | 1000117 |
| 137 | 1049 | 10079 | 100109 | 1000121 |
| 139 | 1051 | 10091 | 100129 | 1000133 |
| 149 | 1061 | 10093 | 100151 | 1000151 |
| 151 | 1063 | 10099 | 100153 | 1000159 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1e7 | 1e8 | 1e10 | 1e11 | 1e12 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10000019 | 100000007 | 1000000007 | 10000000019 | 100000000003 |
| 10000079 | 100000037 | 1000000009 | 10000000033 | 100000000019 |
| 10000103 | 100000039 | 1000000021 | 10000000061 | 100000000057 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1e13 | 1e14 | 1e15 | 1e16 |

|  |  |  |  |
| --- | --- | --- | --- |
| 10000000000037 | 100000000000031 | 1000000000000037 | 10000000000000061 |
| 10000000000051 | 100000000000067 | 1000000000000091 | 10000000000000069 |
| 10000000000099 | 100000000000097 | 1000000000000159 | 10000000000000079 |
| 10000000000129 | 100000000000099 | 1000000000000187 | 10000000000000099 |
| 10000000000183 | 100000000000133 | 1000000000000223 | 10000000000000453 |

|  |  |
| --- | --- |
| 1e17 | 1e18 |

|  |  |
| --- | --- |
| 100000000000000003 | 1000000000000000003 |
| 100000000000000013 | 1000000000000000009 |
| 100000000000000019 | 1000000000000000031 |
| 100000000000000021 | 1000000000000000079 |
| 100000000000000049 | 1000000000000000177 |

### 计时

std::chrono::\_V2::steady\_clock::time\_point C = chrono::steady\_clock::now();  
std::chrono::duration<double> D;  
void gt(string s = "")  
{  
 cerr << s << endl;  
 cerr << setprecision(12) << fixed << '\t' << (D = chrono::steady\_clock::now() - C).count() << "s" << endl;  
 C = chrono::steady\_clock::now();  
}

### clz相关

int clz(int N){return N ? 32 - \_\_builtin\_clz(N) : -INF;}   
int clz(unsigned long long N){return N ? 64 - \_\_builtin\_clzll(N) : -INF;}   
int log2int(int x) { return 31 - \_\_builtin\_clz(x); }   
int log2ll(long long x) { return 63 - \_\_builtin\_clzll(x); }