/\*\*

\* <目录>

\* 0. 套路

\* 1. 数论

\* 2. 组合数学

\* 3. 数据结构

\* 4. 图论

\* 5. 杂项

\*

\* \*/

// ==========套路==========

/\* 三年走过最长的路就是出题人的套路TOT \*/

/\*\*

\* 0.1 计数/求和问题

\*

\* 将ans拆解, 计算每一个分量(顶点、边、序列元素……)对答案的贡献度

\* 抽离出具有特征的量(最大/小值、次大/小值、第k大/小值), 计算其对答案的贡献度

\* 树上计数优先考虑树DP, 从子树的ans推导出父节点的ans或者考虑联通分量合并时ans的更新

\* 尝试划分成若干不相交的集合, 利用某个量控制使其不会重复计数

\* \*/

/\*\*

\* 0.2 动态规划

\*

\* 根据题目描述, 发现某个限制, 就将其抽象表示为一维状态

\* 满足无后效性之后, 考虑去掉一些多余的状态(合并两维状态)

\* 多维dp方程同样可以用矩阵快速幂优化

\* 当转移决策太多时, 可以考虑分步转移(可能需要增加状态维度控制)

\* \*/

/\*\*

\* 0.3 区间/子树查询

\*

\* 将所需量转换成易于用线段树维护的东西(和、最值……)

\* 在线不容易做时考虑离线, 利用扫描线法, 从左扫到右, 同时修改某些位置的值

\* 当扫到查询区间R时, 求出该区间的ans(也可以从右往左扫)

\* 扫描线过程改变的信息可以用主席树维护, 从而在线查询

\* 子树查询问题可以利用dfs序转换为序列区间查询问题

\* \*/

/\*\*

\* 0.4 杂

\*

\* 求第k大几乎全要用二分

\* 最小化最大值、最大化最小值、最大化均值等问题常用二分解决

\* 线段树可以维护分治的过程

\* 要求某一关键字满足偏序关系时, 可以按那个关键字排序, 再用数据结构查询

\* ans与多个分量有关(区间l,r 最小值,次小值), 可以枚举某个分量, 再计算其他分量的个数/极值

\* 区间满足某特征时, 其r随l增大不减小的问题可以采用尺取

\* 要求区间满足某些与数量相关的条件时(大于某数的个数=(>或者<)小于某数的个数？0的个数=1的个数？)

\* 可以用前缀和维护(遇到某数+1，遇到某数-1)

\* 区间相关贪心题目通常需要将区间按R或L排序, 然后lower\_bound/upper\_bound/取一些极值

\* 使用优先队列维护的贪心题目通常会先构造不可行解，然后取min/max修正答案

\* 尝试套用经典模型(默慈金数可以用卡特兰数推出)

\* 观察要求的量是否满足单调性, 是否数量很少(logn的复杂度可能出现在这里)

\* 网络流建模时尝试拆点(解决某些限制, 比如时间)、删边(对图进行等效变换, 减小复杂度)、拆边(费用流)

\* \*/

// ==========数论==========

/\*\*

\* 1.1 重要的同余式

\*

\* 费马小定理: x^(p-1)%p=1, p为素数且gcd(x,p)=1

\* 欧拉定理: x^phi(p)%p=1, gcd(x,p)=1

\* 欧拉定理推论: x^n%p=x^(n%phi(p)+phi(p))%p, n>phi(p)

\* \*/

/\*\*

\* 1.2 重要的公式

\*

\* 利用欧拉函数: F(n)=sigma(f(gcd(i,n)))(i=1 to n)=sigma(f(d)\*phi(n/d))(n%d==0)

\* 莫比乌斯反演: f(n)=sigma(u(d)F(n/d))(n%d==0)

\* \*/

/\*\*

\* 1.3 欧拉函数phi

\*

\* 求单个phi的值, 时间复杂度O(sqrt(n))

\* \*/

int phi(int n)

{

int ans = n;

for (int i = 2; i \* i <= n; i++)

{

if (n % i == 0)

{

while (n % i == 0)

n /= i;

ans = ans / i \* (i - 1);

}

}

if (n > 1)

ans = ans / n \* (n - 1);

return ans;

}

/\*\*

\* 1.4 lucas定理

\*

\* 适用于模数为小质数且n,m较大的组合数取模问题

\* \*/

const int mod = 10007;

int fact[10010];

int quick\_pow(int x, int n)

{

int ans = 1;

while (n)

{

if (n & 1)

ans = ans \* x % mod;

x = x \* x % mod;

n >>= 1;

}

return ans;

}

int init\_fact()

{

fact[0] = 1;

for (int i = 1; i < 10010; i++)

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

}

int c(int n, int m)

{

if (m > n)

return 0;

return fact[n] \* quick\_pow(fact[n - m] \* fact[m] % mod, mod - 2) % mod;

}

int lucas(int n, int m)

{

if (!m)

return 1;

return c(n % mod, m % mod) \* lucas(n / mod, m / mod) % mod;

}

/\*\*

\* 1.5 狄利克雷卷积

\*

\*

\* 满足交换律, 结合律, 对加法的分配律

\* 可以利用快速幂算法进行计算

\* 单次计算复杂度可优化为O(nlog(n))

\* 积性函数的狄利克雷卷积仍为积性函数

\* \*/

// ==========组合数学==========

/\*\*

\* 2.1 卡特兰数

\*

\* 合法括号、出入栈序列、二叉树的种数

\* catalan(n)=C(2n,n)/(n+1)

\* catalan(n)=(4n-2)/(n+1)\*catalan(n-1), catalan(0)=1

\* \*/

/\*\*

\* 2.2 集合划分数

\*

\* 将一个集合划分成若干非空子集的方案数

\* f(n)=sigma(f(n-i)\*C(n-1,i-1))(i=1 to n)

\* \*/

/\*\*

\* 2.3 组合数

\*

\* 求C(n,m), 当n,m在1e6左右时, 预处理阶乘fact, 直接fact(n)/(fact(n-m)\*fact(m))

\* 当n,m很大但mod数较小时采用lucas定理求

\* 当n,m均不大时, 使用递推式C(n,m)=C(n-1,m)+C(n-1,m-1)

\* \*/

/\*\*

\* 2.4 组合数求方程整数解个数

\*

\* x1+x2+x3+...+xn=m (xi>=1)的解的个数为C(m-1,n-1)

\* 如果要求xi>=ai, 那么m的值需要增加sigma(1-ai)

\* \*/

/\*\*

\* 2.5 错排列

\*

\* D(n)=(n-1)(D(n-1)+D(n-2)), D(1)=0, D(2)=1

\* \*/

/\*\*

\* 2.6 无根带标号树的计数

\*

\* n个节点的树共有n^(n-2)种

\* n个节点的树对应于长度为n-2的prufer序列

\* 每个节点编号在prufer序列中出现的次数即为该节点的度减1

\* \*/

// ==========数据结构==========

/\*\*

\* 3.1 树状数组

\*

\* 区间值满足可减性(F(L,R)=F(1,R)-F(1,L-1))即可使用树状数组

\* 维护区间[1, n], 注意modify函数x参数为增量而不是修改为x

\* 非递归, 运行效率高

\* \*/

int tree[100010];

int lowbit(int x) { return x & -x; }

void bit\_modify(int pos, int x)

{

while (pos <= n)

{

tree[pos] += x;

pos += lowbit(pos);

}

}

int bit\_query(int pos)

{

int ans = 0;

while (pos > 0)

{

ans += tree[pos];

pos -= lowbit(pos);

}

return ans;

}

/\*\*

\* 3.2 ST表

\*

\* 区间值满足重叠区间可合并求值即可使用ST表

\* 一般情况维护区间最大/小值以及GCD

\* 维护区间[1, n], 不可修改, 查询效率高

\* \*/

int dp[100010][20];

int a[100010];

void st\_init(int n)

{

for (int i = 1; i <= n; i++) dp[i][0] = a[i];

int up = 0;

while ((1 << up + 1) <= n) up++;

for (int j = 1; j <= up; j++)

for (int i = 0; i + (1 << j) - 1 <= n; i++)

dp[i][j] = min(dp[i][j - 1], dp[i + (1 << j - 1)][j - 1]);

}

int st\_query(int l, int r)

{

int k = 0;

while ((1 << k + 1) <= r - l + 1) k++;

return min(dp[l][k], dp[r - (1 << k) + 1][k]);

}

/\*\*

\* 3.3 树蓓增

\*

\* 类似于ST表的思想, 维护节点i到其第2j父亲节点的信息

\* 利用二进制思想, 通过预处理数据, 计算节点i到其第n个父亲节点的信息

\* 以2的幂为步长, 快速寻找LCA

\* \*/

vector<int> tree[200005];

int siz[200005];

int dep[200005];

int fa[200005][20];

long long sum[200005][20];

void dfs(int u, int f)

{

siz[u] = 1;

for (auto o : tree[u])

{

if (o != f)

{

fa[o][0] = u;

dep[o] = dep[u] + 1;

dfs(o, u);

siz[u] += siz[o];

}

}

sum[u][0] = siz[u];

}

void init(int n)

{

for (int j = 1; (1 << j) <= n; j++)

{

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

{

fa[i][j] = fa[fa[i][j - 1]][j - 1];

sum[i][j] = sum[i][j - 1] + sum[fa[i][j - 1]][j - 1];

}

}

}

long long getSum(int u, int n)

{

long long ans = 0;

int i = 0;

while (n)

{

if (n & 1)

{

ans += sum[u][i];

u = fa[u][i];

}

i++;

n >>= 1;

}

return ans;

}

int getlca(int a, int b)

{

if (dep[a] > dep[b])

swap(a, b);

for (int i = 19; i >= 0; i--)

if (dep[a] < dep[b] && dep[fa[b][i]] >= dep[a])

b = fa[b][i];

for (int i = 19; i >= 0; i--)

if (fa[a][i] != fa[b][i])

a = fa[a][i], b = fa[b][i];

if (a == b)

return a;

return fa[a][0];

}

/\*\*

\* 3.4 莫队算法

\*

\* 解决区间查询问题的万能算法, 但前提是不能有修改操作

\* 莫队的核心是分块, 即对查询进行分块

\* 如果知道[l, r]的答案, 必须在O(1)或O(log(n))复杂度内计算出

\* [l+1, r], [l-1, r], [l, r-1]和[l, r+1]的答案

\* \*/

struct Query

{

int l, r, id, ans;

};

Query Q[50005];

int block[50005];

int n, q;

bool cmp\_lr(const Query& a, const Query& b)

{

if (block[a.l] != block[b.l]) return block[a.l] < block[b.l];

return a.r < b.r;

}

bool cmp\_id(const Query& a, const Query& b)

{

return a.id < b.id;

}

{

for (int i = 0; i != n; i++)

block[i] = i / 225;

sort(Q, Q + q, cmp\_lr);

int l = 0, r = -1, ans = 0;

for (int i = 0; i != q; i++)

{

if (r < Q[i].r)

{

for (r = r + 1; r < Q[i].r; r++)

modify(r, ans, 1);

modify(r, ans, 1);

}

else if (r > Q[i].r)

{

for (; r > Q[i].r; r--)

modify(r, ans, -1);

}

if (l > Q[i].l)

{

for (l = l - 1; l > Q[i].l; l--)

modify(l, ans, 1);

modify(l, ans, 1);

}

else if (l < Q[i].l)

{

for (; l < Q[i].l; l++)

modify(l, ans, -1);

}

Q[i].ans = ans;

}

sort(Q, Q + q, cmp\_id);

}

/\*\*

\* 3.5 线段树

\*

\* 可维护各种区间值, 支持区间修改及查询

\* 维护区间[1, n], 注意lazy标记的设置及维护

\* 查询及修改时, 下传lazy标记并修改相应节点的值

\* 回溯时根据子节点值重新计算根的值

\* \*/

int data[100010 << 2];

int lazy[100010 << 2];

int a[100010];

void seq\_build(int o, int l, int r)

{

if (l == r)

data[o] = a[l];

else

{

int mid = (l + r) >> 1;

seq\_build(o << 1, l, mid);

seq\_build(o << 1 | 1, mid + 1, r);

data[o] = f(data[o << 1], data[o << 1 | 1]);

}

}

void pushdown(int o, int l, int r)

{

if (lazy[o] != -1)

{

lazy[o << 1] = lazy[o];

lazy[o << 1 | 1] = lazy[o];

int mid = (l + r) >> 1;

data[o << 1] = g(data[o << 1], l, mid, lazy[o]);

data[o << 1 | 1] = g(data[o << 1 | 1], mid + 1, r, lazy[o]);

lazy[o] = -1;

}

}

int seq\_query(int o, int l, int r, int ql, int qr)

{

if (ql <= l && r <= qr)

return data[o];

if (r < ql || qr < l)

return 0;

int mid = (l + r) >> 1;

pushdown(o, l, r);

return f(seq\_query(o << 1, l, mid, ql, qr), seq\_query(o << 1 | 1, mid + 1, r, ql, qr));

}

void seq\_modify(int o, int l, int r, int ql, int qr, int v)

{

if (ql <= l && r <= qr)

{

lazy[o] = v;

data[o] = g(data[o], l, r, lazy[o]);

return;

}

if (r < ql || qr < l)

return;

int mid = (l + r) >> 1;

pushdown(o, l, r);

seq\_modify(o << 1, l, mid, ql, qr, v);

seq\_modify(o << 1 | 1, mid + 1, r, ql, qr, v);

data[o] = f(data[o << 1], data[o << 1 | 1]);

}

/\*\*

\* 3.6 树链剖分

\*

\* 用于处理树上路径查询及修改问题, 将树剖分成轻重链, 用线段树进行维护

\* 注意top[x]为x所在重链的顶端节点编号, 如果x是轻节点则top[x]=x

\* 最后u!=v的情况发生在u,v逼近lca时有一条链为重链

\* 因此w[u]+1表示重链上在lca下的第一个节点在线段树中的编号

\* \*/

vector<pair<int, int> > tree[maxn]; // 树的边上有权值

int siz[maxn], dep[maxn], son[maxn], fa[maxn], top[maxn], w[maxn];

int tid;

void tc\_dfs(int u)

{

siz[u] = 1;

son[u] = 0;

for (int v = 0; v < tree[u].size(); v++)

{

if (tree[u][v].first != fa[u])

{

fa[tree[u][v].first] = u;

dep[tree[u][v].first] = dep[u] + 1;

tc\_dfs(tree[u][v].first);

if (siz[tree[u][v].first] > siz[son[u]])

son[u] = tree[u][v].first;

siz[u] += siz[tree[u][v].first];

}

}

}

void tc\_build(int u, int tp)

{

w[u] = ++tid;

top[u] = tp;

if (son[u])

tc\_build(son[u], tp);

for (int v = 0; v < tree[u].size(); v++)

if (tree[u][v].first != son[u] && tree[u][v].first != fa[u])

tc\_build(tree[u][v].first, tree[u][v].first);

}

void tc\_init()

{

tid = 0;

tc\_dfs(1);

tc\_build(1, 1);

}

void tc\_modify(int u, int v, int val)

{

while (top[u] != top[v])

{

if (dep[top[u]] < dep[top[v]])

swap(u, v);

seq\_modify(1, 1, tid, w[top[u]], w[u], val);

u = fa[top[u]];

}

if (u != v)

{

if (dep[u] > dep[v])

swap(u, v);

seq\_modify(1, 1, tid, w[u] + 1, w[v], val);

}

}

int tc\_query(int u, int v)

{

int ans = 0;

while (top[u] != top[v])

{

if (dep[top[u]] < dep[top[v]])

swap(u, v);

ans += seq\_query(1, 1, tid, w[top[u]], w[u]);

u = fa[top[u]];

}

if (u != v)

{

if (dep[u] > dep[v])

swap(u, v);

ans += seq\_query(1, 1, tid, w[u] + 1, w[v]);

}

return ans;

}

/\*\*

\* 3.7 主席树

\*

\* 即可持久化线段树, 可以查询区间内在某个值域内的数的个数, 区间第k大等等

\* 对值域利用线段树进行维护, 一般情况要求值域范围不大

\* 对于值域范围很大但查询离散的情况同样可以维护, 以查询点作为分界维护值域

\* roots[x]表示[1, x]区间对应的线段树的根

\* 查询时使用query(root[r],...)-query(root[l-1],...)

\* 空间复杂度O(nlog(n))

\* \*/

int lson[3000010];

int rson[3000010];

int data[3000010];

int roots[100010];

int tot;

int build(int l, int r)

{

int cur = ++tot;

int mid = (l + r) >> 1;

if (l != r)

{

lson[cur] = build(l, mid);

rson[cur] = build(mid + 1, r);

}

return cur;

}

int append(int root, int l, int r, int x)

{

int cur = ++tot;

data[cur] = data[root] + 1;

int mid = (l + r) >> 1;

if (l != r)

{

if (x <= mid)

{

rson[cur] = rson[root];

lson[cur] = append(lson[root], l, mid, x);

}

else

{

lson[cur] = lson[root];

rson[cur] = append(rson[root], mid + 1, r, x);

}

}

return cur;

}

int query(int root, int l, int r, int ql, int qr)

{

if (qr < l || ql > r)

return 0;

if (l >= ql && r <= qr)

return data[root];

int mid = (l + r) >> 1;

return query(lson[root], l, mid, ql, qr) + query(rson[root], mid + 1, r, ql, qr);

}

// ==========图论==========

/\*\*

\* 4.1 网络流dinic算法

\*

\* 对图进行分层, 时间效率较好的网络流算法

\* 所谓玄学复杂度, 相对较大的图同样可以用dinic求网络流

\* 最大流 = 最小割

\* \*/

struct Edge

{

int to, cap, rev;

Edge(int \_to, int \_cap, int \_rev)

{

to = \_to;

cap = \_cap;

rev = \_rev;

}

};

vector<Edge> G[90000];

int level[90000];

int cur[90000];

void clearG()

{

for (int i = 0; i < 90000; i++)

G[i].clear();

}

void addEdge(int from, int to, int cap)

{

G[from].push\_back(Edge(to, cap, G[to].size()));

G[to].push\_back(Edge(from, 0, G[from].size() - 1));

}

bool dinic\_bfs(int s, int t)

{

memset(level, -1, sizeof(level));

queue<int> q;

q.push(s);

level[s] = 0;

while (!q.empty())

{

int v = q.front();

q.pop();

for (vector<Edge>::iterator it = G[v].begin(); it != G[v].end(); it++)

{

if (it->cap > 0 && level[it->to] == -1)

{

level[it->to] = level[v] + 1;

q.push(it->to);

}

}

}

return level[t] != -1;

}

int dinic\_dfs(int s, int t, int flow)

{

if (s == t)

return flow;

for (int& i = cur[s]; i < G[s].size(); i++)

{

Edge& e = G[s][i];

if (e.cap > 0 && level[e.to] > level[s])

{

int d = dinic\_dfs(e.to, t, min(flow, e.cap));

if (d > 0)

{

e.cap -= d;

G[e.to][e.rev].cap += d;

return d;

}

}

}

return 0;

}

int dinic(int s, int t)

{

int maxflow = 0, flow;

while (bfs(s, t))

{

memset(cur, 0, sizeof(cur));

while (flow = dinic\_dfs(s, t, inf))

maxflow += flow;

}

return maxflow;

}

/\*\*

\* 4.2 最大权闭合子图

\*

\* 闭合子图就是给定一个有向图, 从中选择一些点组成一个点集V

\* 对于V中任意一个点，其后续节点都仍然在V中

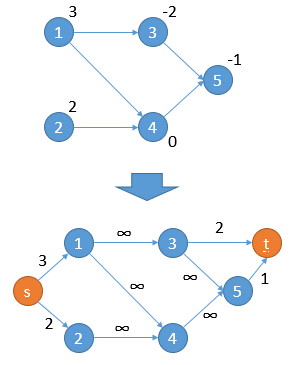
\* 对于一般的图来说: 首先建立源点s和汇点t

\* 将源点s与所有权值为正的点相连, 容量为权值

\* 将所有权值为负的点与汇点t相连, 容量为权值的绝对值

\* 权值为0的点不做处理

\* 同时将原来的边容量设置为无穷大



\* 最大权闭合子图的权值等于所有正权点之和减去最小割

\* \*/

/\*\*

\* 4.3 二分图的性质

\*

\* 最小点覆盖数 = 最大匹配数

\* 最小边覆盖数 = 最大点独立集 = 顶点数 - 最大匹配数

\* 最小路径覆盖, 即用尽量少的不相交简单路径覆盖有向无环图G的所有顶点

\* 将每一个顶点i拆成两个顶点Xi和Yi

\* 然后根据原图中边的信息, 从X部往Y部引边, 所有边的方向都是由X部到Y部

\* 最小路径覆盖数＝原图G的顶点数 - 二分图的最大匹配数

\* \*/

// ==========杂项==========

/\*\*

\* 5.1 矩阵快速幂

\*

\* 利用矩阵表示线性递推关系, 结合快速幂算法快速求出

\* \*/

const int mSize = 4;

long long mod;

struct Matrix

{

long long v[mSize][mSize];

friend Matrix operator\* (const Matrix& a, const Matrix& b)

{

Matrix c;

for (int i = 0; i < mSize; i++)

for (int j = 0; j < mSize; j++)

{

c.v[i][j] = 0;

for (int k = 0; k < mSize; k++)

c.v[i][j] += a.v[i][k] \* b.v[k][j] % mod;

c.v[i][j] %= mod;

}

return c;

}

friend Matrix operator^ (Matrix x, long long n)

{

Matrix ans;

for (int i = 0; i < mSize; i++)

for (int j = 0; j < mSize; j++)

ans.v[i][j] = (i == j);

while (n)

{

if (n & 1)

ans = ans \* x;

x = x \* x;

n >>= 1;

}

return ans;

}

};

/\*\*

\* 5.2 数位动态规划

\*

\* 将数字分解, 从高位开始动态规划

\* limit标记表示当前为是否受到限制, 不受限即可取0~9

\* start标记表示当前的数是否已经开始, 如果每一位都选取0则未开始

\* 这两个标记需要根据题目要求来加, 并不是一定需要

\* \*/

long long dp[20][2][2];

int digit[20];

long long dfs(int pos, int s1, int s2, bool limit, bool start)

{

if (pos == -1)

return f(s1, s2);

if (!limit && start && dp[pos][s1][s2] != -1)

return dp[pos][s1][s2];

int up = limit ? digit[pos] : 9;

long long ans = 0;

if (!start)

{

for (int i = 0; i <= up; i++)

ans += dfs(pos - 1, g1(s1), g1(s2), limit && i == up, start || i != 0);

}

else

{

for (int i = 0; i <= up; i++)

ans += dfs(pos - 1, g2(s1), g2(s1), limit && i == up, start || i != 0);

}

if (!limit && start)

dp[pos][s1][s2] = ans;

return ans;

}

long long solve(long long num)

{

int idx = 0;

while (num)

{

digit[idx++] = num % 10;

num /= 10;

}

return dfs(idx - 1, 0, 0, true, false);

}