2023 年牛客多校第七场题解

出题人：南京外国语学校

1 A Random Addition

题意：给定长度为 n 的数列，初始全为 0。对其中 m 个区间 $[l_i, r_i]$ 执行加 x 操作，x 等概率从 $[0,1]$ 实数集合选取。这些区间包含或不相交。q 次询问整个序列最大值在 $[p,q]$ 的概率对 998 244 353 取模。$1 \le n \le 10^5$，$1 \le m \le 200$，$0 \le p \le q \le 10$。

解法：首先考虑一个数字仅被加一次，则该数字取值的概率密度函数为：

$$P(x) = f_1(x) = \begin{cases} 1, 0 \le x \le 1 \\ 0, \text{otherwise} \end{cases}$$

如果加两次，则该数字的取值概率密度函数就需要使用到卷积：

$$\begin{aligned} f_2(x) &= P(x) * P(x) \\ &= \int_{-\infty}^{+\infty} P(t)P(x-t)\mathrm{d}t \\ &= \begin{cases} x, 0 \le x \le 1 \\ 2-x, 1 < x \le 2 \\ 0, \text{otherwise} \end{cases} \end{aligned}$$

即，两个窗函数的卷积是一个三角波函数。

如果该数字加 k 次，则可以考虑将 $P(x)$ 进行 k 次卷积。使用数学归纳法证明该 $f_k(x)$ 一定是一个 k 段函数，每段函数都可以使用一个多项式表达。假设经过第 $k-1$ 次卷积后整个函数分为 $k-1$ 段，其中第 j （$j \in [1, k-1]$）个分段函数对应区间为 $[j-1, j]$。考虑对这个函数再做一次卷积：

$$\begin{aligned} f_{k,j}(x) &= \int_{-\infty}^{+\infty} f_{k-1,j}(t)P(x-t)\mathrm{d}t \\ &= \int_{x-1}^{x} f_{k-1,j}(t)\mathrm{d}t \\ &= \begin{cases} \int_{j-1}^{x} f_{k-1,j}(t)\mathrm{d}t, x \in [j-1, j] \\ \int_{x-1}^{j} f_{k-1,j}(t)\mathrm{d}t, x \in [j, j+1] \end{cases} \end{aligned}$$

若记 $F(x) = \int_0^x f(t)\mathrm{d}t$，则卷积式可写为：

$$f_{k,j}(x) = \begin{cases} F(x) - F(j-1), x \in [j-1, j] \\ F(j) - F(x-1), x \in [j, j+1] \end{cases}$$

回到本题，由于需要求的是小于等于某个特定数字的概率，因而需要对概率密度函数进行积分得到概率分布函数，不妨直接维护概率分布函数。

由于题目中有个强约束——区间包含或者不相交，因而这些区间组成一个树形结构，可以考虑 dfs 遍历得到每个区间上的概率分布函数。当子树中存在多个不相交的直接子区间时，父节点需要将这些区间的概率密度函数一一通过多项式卷积乘到**对应**分段区间上，然后父区间再自行进行卷积运算。这么做的原因是因为对于两个独立变量的最大值的概率分布函数 $\Pr(\max(X_1, X_2) \le x) = \Pr(X_1 \le x)\Pr(X_2 \le x)$。

最后查询仅需要查询根节点的区间（即整个数组）上的概率分布函数即可。如果使用朴素多项式乘法（卷积），总复杂度 $\mathcal{O}(10m^3)$。

关于多项式函数的卷积和乘积，可以参看附录乙 · 乘积、卷积部分。

感谢致远星的代码：

```cpp
#include <bits/stdc++.h>
using namespace std;
const int mod = 998244353;
int n, m, t;
struct qq
{
    int l, r;
    bool operator<(const qq &b)const
    {
        if (r != b.r)
            return r < b.r;
        return l > b.l;
    }
} a[210];
int sta[210], top, inv[1001000];
vector<int> g[210];
struct P
{
    int xs[210], len;
    // 积分
    inline void integral()
    {
        for (int i = len; i >= 0; i--)
            xs[i + 1] = 1ll * xs[i] * inv[i + 1] % mod;
        xs[0] = 0;
        len++;
    }
    // 单点求值（点值）
    inline int eval(int x)
    {
        int ss = 0;
        for (int j = 0, w = 1; j <= len; j++, w = 1ll * w * x % mod)
            (ss += 1ll * w * xs[j] % mod) %= mod;
        return ss;
    }
} dp[210][410], E, C, fz[210];
// dp[i][j]：第i个区间的第j个分段函数区间[j,j+1]的多项式函数表达式
// 注意：dp[i][j]是概率分布函数，即概率密度函数的积分
// 两个多项式函数卷积
P mul(P a, P b)
{
    P c;
    c.len = a.len + b.len;
    for (int i = 0; i <= c.len; i++)
        c.xs[i] = 0;
```

```
    for (int i = 0; i <= a.len; i++)
        for (int j = 0; j <= b.len; j++)
            (c.xs[i + j] += 1ll * a.xs[i] * b.xs[j] % mod) %= mod;
    return c;
}
int c[410][410];
// 反褶，f(x)->f(1-x)
// 虽然f(x)仅在[0,inf]上定义，但是也可以进行定义域的补充，使得它是一个奇函数
P reverseAndShift(P a)
{
    P b;
    b.len = a.len;
    for (int i = 0; i <= b.len; i++)
        b.xs[i] = 0;
    for (int i = 0; i <= a.len; i++)
        (b.xs[0] += a.xs[i]) %= mod, (b.xs[i] -= a.xs[i]) %= mod;
    return b;
}
// 叠加两个多项式函数
void add(P &a, P b)
{
    if (a.len < b.len)
        swap(a, b);
    for (int i = 0; i <= b.len; i++)
        (a.xs[i] += b.xs[i]) %= mod;
}
int d[410];
void dfs(int x)
{
    if (g[x].empty())
    {
        dp[x][0] = E, dp[x][1] = C, d[x] = 1;
        return;
    }
    dp[x][0] = C, d[x] = 0;
    for (int v : g[x])
    {
        dfs(v);
        // 各个子树的概率分布函数要和父节点进行卷积以进行合并
        // 即，当前这一段的概率分布函数为f1(x)，现在需要乘到父节点概率分布函数的这一段上
        for (int z = 0; z <= max(d[x], d[v]); z++)
            fz[z] = mul(dp[x][min(z, d[x])], dp[v][min(z, d[v])]);
        d[x] = max(d[x], d[v]);
        for (int z = 0; z <= d[x]; z++)
            dp[x][z] = fz[z];
    }
    // 考虑对这个区间整体做一次卷积运算
```

```
    if (x)
    {
        // 最右侧的区间平移
        dp[x][d[x] + 1] = dp[x][d[x]], d[x]++;
        // 其他区间由于对1卷积，因而等价于直接积分
        for (int i = 0; i <= d[x]; i++)
            dp[x][i].integral();
        // 前一区间跟后一区间的叠加
        for (int i = d[x]; i; i--)
            add(dp[x][i], reverseAndShift(dp[x][i - 1]));
        // 补足x->inf的时候概率分布函数=1的部分，增加新的一段分段
        dp[x][d[x] + 1] = C, d[x]++;
    }
}
const int T = 1000000;
int S(int x)
{
    int e = x % T;
    x /= T; // 查在哪个区间（分段函数）之中
    e = 1ll * e * inv[T] % mod;
    return dp[0][min(x, d[0])].eval(e);
}
int main()
{
    E.len = 1, E.xs[1] = 1;
    C.len = 0, C.xs[0] = 1;
    inv[1] = 1;
    for (int i = 2; i <= T; i++)
        inv[i] = mod - 1ll * inv[mod % i] * (mod / i) % mod;
    c[0][0] = 1;
    for (int i = 1; i <= 400; i++)
    {
        c[i][0] = 1;
        for (int j = 1; j <= i; j++)
            c[i][j] = (c[i - 1][j - 1] + c[i - 1][j]) % mod;
    }
    scanf("%d%d%d", &n, &m, &t);
    for (int i = 1; i <= m; i++)
        scanf("%d%d", &a[i].l, &a[i].r);
    sort(a + 1, a + m + 1);
    for (int i = 1; i <= m; i++)
    {
        while (top && a[sta[top]].l >= a[i].l)
            g[i].push_back(sta[top]), top--;
        sta[++top] = i;
    }
    for (int i = 1; i <= top; i++)
```

```
        g[0].push_back(sta[i]);
    dfs(0);
    for (int i = 1, l, r; i <= t; i++)
    {
        scanf("%d%d", &l, &r);
        printf("%d\n", ((S(r) - S(l)) % mod + mod) % mod);
    }
    return 0;
}
```

2 C Beautiful Sequence

题意：给定长度为 n 的序列 $\{a\}_{i=1}^{n}$ 的异或差分序列 $\{b\}_{i=1}^{n-1}$，要求 $\{a\}$ 不严格递增且数字范围都在 $[0,2^{30}-1]$ 的第 k 大的序列。$1\le n\le 10^6$，$1\le k<2^{30}$。

解法：不难注意到给定了异或差分数组后，确定了第一个数字就确定了整个数组。由于是异或操作，考虑分位处理。

从高到低的枚举第 k 个二进制位。这时考虑如何通过异或差分数组得到递增条件。如果 $b_i=0$，则 $a_i=a_{i+1}$，符合条件；如果 $b_i=1$，则必须有 $0=a_i<a_{i+1}=1$，要么就是更高的二进制位上已经能够明确分出大小。因而可以考虑维护一个 `vector` 表示这一段仍未分出大小，需要通过更低位去判断，而块间不需要判断。因而块内必须有 $0=a_i<a_{i+1}=1$。如果当前这个二进制位没有约束，则可以 $0,1$。否则就必须强制填 $0,1$。这样的第 k 大只需要把可以变化的位拿出来，将 k 进行二进制表示即可。复杂度 $\mathcal{O}(n\log V)$。

```
#include <bits/stdc++.h>
#define fp(i, a, b) for (int i = a, i##_ = b; i <= i##_; ++i)
#define fd(i, a, b) for (int i = a, i##_ = b; i >= i##_; --i)

using namespace std;
using ll = long long;
const int N = 1e6 + 5;
int n, k, a[N], w[30];
void Solve() {
    scanf("%d%d", &n, &k), --k;
    fp(i, 2, n) scanf("%d", a + i), a[i] ^= a[i - 1];
    vector<int> vec;
    vector<pair<int, int>> s = {{1, n}}, t;
    memset(w, -1, sizeof w);
    fd(j, 29, 0) {
        for (auto [l, r] : s) {
            int fg = 0;
            fp(i, l + 1, r) {
                if ((a[i] >> j & 1) != (a[l] >> j & 1)) {
                    fp(k, i + 1, r)
                        if ((a[k] >> j & 1) != (a[i] >> j & 1))
                            return puts("-1"), void();
                    int x = a[l] >> j & 1;
```

```
                    if (w[j] != -1 && w[j] != x)
                        return puts("-1"), void();
                    w[j] = x, fg = 1;
                    t.push_back({l, i - 1}), t.push_back({i, r});
                    break;
                }
            }
            if (!fg) t.push_back({l, r});
        }
        s = t, t.clear();
    }
    int x = 0;
    fp(i, 0, 29) {
        if (w[i] == -1)
            x |= (k & 1) << i, k >>= 1;
        else x |= w[i] << i;
    }
    if (k) puts("-1");
    else {
        fp(i, 1, n) printf("%d%c", x ^ a[i], " \n"[i == n]);
    }
}
int main() {
    int t = 1;
    scanf("%d", &t);
    while (t--) Solve();
    return 0;
}
```

3 E Star Wars

题意：在一个无向图 $G(n,m)$ 上，可能有重边。一次操作可以执行下面的其中一条：

1. 在 (u,v) 上连接一条边。
2. 去除 (u,v) 连接的一条边。
3. 修改 a_u 权值。
4. 查询 u 相邻节点 a_v 的权值和。

操作次数 $1 \le q \le 3 \times 10^5$，$1 \le n,m \le 3 \times 10^5$。

解法：对于这类维护周围点连接情况的题，很容易想到根号分治，对大点（度数大的点）和小点（度数小的点）进行分开考虑。

对于大点小点的划分，通常来说使用根号为界限——当度数小于 $O\left(\sqrt{n}\right)$ 时，可以暴力去搜周围的点，单次查询复杂度 $O\left(\sqrt{n}\right)$；对于度数较大的点，由于边数有限（假设 n,m 同阶），这样大点数目本身较少，只有 $O\left(\sqrt{n}\right)$ 个。因而可以考虑对大点进行一些特殊的数据结构处理以快速维护。

对于本题，当维护点权值更新时，小点可以暴力外推到周围所有的点，更新它们的答案，让大点的答案被动被小点所更新；大点对大点也可以考虑暴力，因为大点本身不多；大点对小点则采取标记的方式，在大点处打上标记，让小点来被动查询大点的更新情况。

当边数发生变化时，需要动态调整大小点的划分，当一个小点连接大量的边后需要晋升为大点以加速操作。同时根据上述规则同步更新每个点到周围点的答案。

感谢HDU-T04的提交。

```cpp
#include <bits/stdc++.h>
#define pb push_back
#define fir first
#define sec second
#define ll long long
using namespace std;
typedef pair<int, int> pii;
const int N = 3e5 + 10, K = sqrt(N);
// 块阈值为K
const int P = (1 << 30);
int n, q, tot, d[N], cnt[N];
ll w[N], res[N];
vector<pii> e[N], G[N];
map<int, int> mp[N];
int TOT;
map<int, int> idx;
void add(int x, int y, int id)
{
    if (d[x] >= K)
        G[y].pb({x, id});
    else
        e[x].pb({y, id});
}
// 由小点晋升为大点
void upd(int x)
{
    for (auto E : e[x])
    {
        G[E.fir].pb({x, E.sec});
        if (cnt[E.sec])
            res[x] += w[E.fir];
    }
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0), cout.tie(0);
    cin >> n >> q;
    int lst = 0;
```

```
    while (q--)
    {
        int opt, x, y;
        cin >> opt >> x, x ^= lst;
        if (!idx.count(x))
            idx[x] = ++TOT, x = TOT;
        else
            x = idx[x];
        if (opt == 4)
        {
            // 大点等于被动更新值+本地值
            if (d[x] >= K)
            {
                cout << res[x] + w[x] << endl;
                lst = (res[x] + w[x]) % P;
            }
            else // 小点暴力搜周围的点
            {
                ll ans = w[x];
                for (auto E : e[x])
                    if (cnt[E.sec])
                        ans += w[E.fir];
                cout << ans << endl;
                lst = ans % P;
            }
        }
        else
        {
            cin >> y, y ^= lst;
            if (opt == 1)
            {
                // 先维护边
                if (!idx.count(y))
                    idx[y] = ++TOT, y = TOT;
                else
                    y = idx[y];
                if (x == y)
                    continue;
                if (y < x)
                    swap(x, y);
                int id;
                if (!mp[x].count(y))
                {
                    mp[x][y] = ++tot, id = tot;
                    add(x, y, id), add(y, x, id);
                }
                else
```

```
                id = mp[x][y];
            cnt[id]++;
            if (cnt[id] == 1)
            {
                // x为大点：被动更新
                if (d[x] >= K)
                    res[x] += w[y];
                if (d[y] >= K)
                    res[y] += w[x];
            }
            d[x]++, d[y]++;
            if (d[x] == K)
                upd(x);
            if (d[y] == K)
                upd(y);
        }
        else if (opt == 2)
        {
            if (!idx.count(y))
                idx[y] = ++TOT, y = TOT;
            else
                y = idx[y];

            if (x == y)
                continue;
            if (y < x)
                swap(x, y);
            int id = mp[x][y];
            cnt[id]--;
            if (!cnt[id])
            {
                // 回退
                if (d[x] >= K)
                    res[x] -= w[y];
                if (d[y] >= K)
                    res[y] -= w[x];
            }
        }
        else
        {
            w[x] += y;
            for (auto E : G[x])
                if (cnt[E.sec])
                    res[E.fir] += y;
        }
    }
}
```

```
    return 0;
}
```

4 F Counting Sequences

题意：问有多少个长度为 n 的序列 $\{A\}=\{a_1,a_2,\cdots,a_n\}$ 满足，它与自身循环右移一位的序列 $\{B\}=\{a_2,a_3,\cdots,a_n,a_1\}$ 异或得到的序列 $\{C\}$ 中，每个数字二进制表示中 1 的数目有 k 个，且 $0\le a_i<2^m$。$1\le n<998\,244\,353$，$1\le m\le 10^8$，$1\le k\le 5\times 10^4$。

解法：由于是异或操作，因而每位独立。考虑对于一个特定的二进制位，原序列 $\{A\}$ 与右移一位的 $\{B\}$ 异或后得到的异或结果序列 $\{C\}$ 性质，有：

1. 一个异或结果序列对应于原来的两个序列。不难注意到 $c_i=a_{i+1}\oplus a_i$，因而 $\{C\}$ 等价于 $\{A\}$ 的异或差分序列。如果设定 a_1，则整个序列都可以通过逐步递推得到，并且 a_1 任意。因而 a_1 的两种取值对应于原序列的两种取值。
2. 如果异或结果序列的 1 个数为奇数，则原序列无解。因为 $c_i=a_i\oplus a_{i+1}$（此处 $a_{n+1}=a_1$），则 $\bigoplus_{i=1}^{n}c_i=\bigoplus_{i=1}^{n}a_ia_{i+1}=0$，因为每个数字都出现了两次。因而 c_i 中 1 个数必然为偶数。

因而对于一个二进制位，异或结果序列在这一位上产生 k 个 1 的方案可以用一个多项式 f 表达：

$$f(x)=\sum_{i=0}^{\left\lfloor\frac{n}{2}\right\rfloor}2\binom{n}{2i}x^{2i}$$

即，奇数个 1 的方案不存在，偶数个 1 的方案等价于在 n 个数中选出 k 个数字的方案乘以 2。

由于此处需要计算 m 个二进制位上得到 1 个数总和为 k 的方案，由于各行独立，因而使用乘法原理将每一行的多项式乘起来，不难得到答案为 $[x^k]f^m(x)$。由于求的目标项系数仅为 k 次，因而进行快速幂的多项式 f 也不需要保留到 n 次，而仅需要 k 次即可，更高次项显然不会对答案有任何贡献。使用多项式快速幂（多项式 exp 和多项式 ln 配合）即可在 $\mathcal{O}(k\log m)$ 的复杂度内求解。

关于生成函数的一些基础知识，请参看附录 · 甲 生成函数入门部分。

```
int main()
{
    long long n, m, k;
    cin >> n >> m >> k;
    Poly f(k + 1);
    long long C = 1;
    for (int i = 0; i <= k; i++)
    {
        if (i % 2 == 0)
            f[i] = C;
        C = C * (n - i) % P;
        C = C * inv[i + 1] % P;
    }
    Poly ans = f.pow(m, k + 1);
    long long cur = ans[k];
    printf("%d", cur * fpow(2, m) % P);
```

```
    return 0;
}
```

5 G Cyperation

题意：给定长度为 n 的环 $\{a\}_{i=1}^{n}$，每次可以选择环上最近距离为 k 的两点 i,j 使得 a_i 和 a_j 都减 1。问能不能经过若干次操作让整个环上数字清零。多测，$1 \le T \le 10^6$，$2 \le \sum n \le 10^6$，$1 \le k \le n$。

解法：首先特判全 0 和 $k > \left\lfloor \frac{n}{2} \right\rfloor$ 的情况。首先可以考虑将能够一起操作的数字通过重新排列放置到一起，由于每个数字只能至多和两个数字一起减 1，因而必定可以成环。整个大环就可以分成若干个独立的小环单独处理。

这时考虑环 $a_0, a_1, \cdots, a_k$，设 a_0 和 a_1 一起减了 x 次。则 a_1 再想要减到 0，a_1 就必须和 a_2 一起减 $a_1 - x$ 次，同理可以推得 a_2 和 a_3 一起减 $a_2 - a_1 + x$ 次，依次类推。

不难注意到每传递一轮 x 的系数正负号翻转一次。可以把环上这每个次数的一次函数表达式写出。显然这些次数都是非负数，因而维护一个合法的 x 区间 $[l, r]$ 即可。同时，当环长为偶数时，经过一圈的递推可以得到 a_0 和 a_1 要一起减 $x + b$ 次。如果 $b \neq 0$ 则无解；如果环长为奇数，则一圈递推后 a_0 和 a_1 的次数为 $b - x$。这时需要满足 $2x = b$，可以解出固定的整数 x（b 为奇数则直接无解）。带入检查这个 x 是否合法即可。

```
#include <bits/stdc++.h>
#define fp(i, a, b) for (int i = a, i##_ = int(b); i <= i##_; ++i)
#define fd(i, a, b) for (int i = a, i##_ = int(b); i >= i##_; --i)

using namespace std;
using ll = long long;
const int N = 2e6 + 5;
ll a[N];
int n, k, vis[N];
void Clear() { fp(i, 0, n - 1) vis[i] = 0; }
struct node {
    ll a, b;
    node operator-(const node &x) const
    {
        return {a - x.a, b - x.b};
    }
    node operator+(const node &x) const
    {
        return {a + x.a, b + x.b};
    }
};
bool Checker(vector<long long> &num) {
    int m = num.size();
    vector<node> cur(m);
    cur[0] = (node){1ll, 0ll};
    ll l = 0, r = num[0];
    fp(i, 0, m - 1)
        cur[(i + 1) % m] = (node){0ll, num[(i + 1) % m]} - cur[i];
```

```
    fp(i, 0, m - 1)
        if (cur[i].a == -1) // -x+b>=0 x<=b
            r = min(r, cur[i].b);
        else if (cur[i].a == 1) // x+b>=0 x>=-b
            l = max(l, -cur[i].b);
        else assert(false);
    if (m % 2) {
        if (cur[0].b % 2)
            return false;
        ll x = cur[0].b / 2;
        // printf("CUR X:%lld\n", x);
        if (x < l || x > r)
            return false;
    }
    else {
        if (cur.back().b != num[0])
            return false;
    }
    return l <= r;
}
void Solve() {
    scanf("%d%d", &n, &k);
    fp(i, 0, n - 1) scanf("%lld", &a[i]);
    bool zero = 1;
    fp(i, 0, n - 1) if (a[i]) zero = 0;
    if (zero) return (void)puts("YES");
    if (k > n / 2)
        return (void)puts("NO");
    bool fg = true;
    fp(i, 0, n - 1)
        if (!vis[i]) {
            vector<long long> cur;
            int pos = i;
            while (!vis[pos]) {
                cur.push_back(a[pos]);
                vis[pos] = 1;
                pos = (pos + k) % n;
            }
            fg &= Checker(cur);
        }
    if (fg) puts("YES");
    else puts("NO");
}
int main() {
    int t = 1;
    scanf("%d", &t);
    while (t--) Solve(), Clear();
```

```
    return 0;
}
```

6 I We Love Strings

题意：给定 n 个仅含 `0,1,?` 的正则串，`?` 可以匹配一个 `0` 或 `1`，问有多少个 01 串可以被至少一个正则串匹配。$1 \le n \le 400$，$\sum |s_i| \le 400$。

解法：由于 $\sum |s_i| \le 400$，不难想到根号分治——首先根据串长对串进行分类。对于串长小于等于 20 的，可以考虑直接枚举这 2^k 个串，依次观察是否和这些正则串匹配；当串长长的时候，正则串个数不多。这时可以用 f_S 表示至少满足 S 集合内的正则串的串个数是多少。暴力计算 f_S——枚举 S 中有哪些正则串，合并这些串的要求后求出 f_S 表示至少满足 S 集合内所有正则串的串个数。该形式等价于高维前缀和的逆变换，可以采用 SOS dp 的方式求解。本题可以参看链接中第二个例题的解法。

最后统计对非空集合求和即可。复杂度 $\mathcal{O}\left(n2^{\sqrt{n}}\right)$。

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 400, LIM = 20, P = 998244353;
string s[N + 5];
long long th[N + 5];
class Solution
{
    vector<string> p;
    long long bigSolver(int len, int size)
    {
        vector<long long> f(1 << size);
        f[0] = th[len];
        string base = "";
        for (int j = 0; j < len; j++)
            base += "?";
        for (int i = 1; i < 1 << size; i++)
        {
            string cur = base;
            bool flag = 1;
            for (int j = 0; j < size && flag; j++)
                if (i >> j & 1)
                {
                    for (int k = 0; k < len; k++)
                        if (p[j][k] != '?')
                        {
                            if (cur[k] == '?')
                                cur[k] = p[j][k];
                            else if (cur[k] != p[j][k])
                            {
                                flag = 0;
```

```
                            break;
                        }
                    }
                }
            if (flag)
                f[i] = th[count(cur.begin(), cur.end(), '?')];
        }
        // 调整枚举顺序即可（通过hack的样例和官方数据）
        // 当然建议可以使用最朴素的容斥，如链接所示
        for (int j = 0; j < size; j++)
            for (int i = 0; i < 1 << size; i++)
                if (i >> j & 1)
                    f[i ^ (1 << j)] = (f[i ^ (1 << j)] - f[i] + P) % P;
        long long ans = 0;
        for (int i = 1; i < 1 << size; i++)
            ans = (ans + f[i]) % P;
        return ans;
    }
    long long smallSolver(int len)
    {
        long long ans = 0;
        for (int i = 0; i < 1 << len; i++)
        {
            bool flag = 0;
            for (auto x : p)
            {
                bool cur = 1;
                for (int j = 0; j < len; j++)
                {
                    if (x[j] == '?')
                        continue;
                    if (x[j] != (i >> j & 1) + '0')
                    {
                        cur = 0;
                        break;
                    }
                }
                if (cur)
                {
                    flag = 1;
                    break;
                }
            }
            if (flag)
                ans++;
        }
        return ans;
```

```
    }

public:
    void insert(string s)
    {
        p.push_back(s);
    }
    long long query()
    {
        if (p.empty())
            return 0;
        int len = p[0].length();
        if (len <= LIM)
            return smallSolver(len);
        else
            return bigSolver(len, p.size());
    }
} S[N + 5];
int main()
{
    th[0] = 1;
    for (int i = 1; i <= N; i++)
        th[i] = th[i - 1] * 2 % P;
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
    {
        cin >> s[i];
        S[s[i].length()].insert(s[i]);
    }
    long long ans = 0;
    for (int i = 1; i <= N; i++)
        ans = (ans + S[i].query()) % P;
    printf("%lld", ans);
    return 0;
}
```

7 K Set

题意：给定长度为 n 的序列 $\{a\}_{i=1}^{n}$，求下式：

$$\sum_{S\subseteq\{a\}_{i=1}^{n}} |S| \left(\min_{x\in S} x\right) \left(\max_{y\in S} y\right) \left(\bigoplus_{z\in S} z\right)$$

$1 \le n \le 10^6$，$0 \le a_i < 2^{30}$。

解法：首先对序列排序，那么考虑固定区间左右端点就可以确定 min 和 max。由于出现异或，仍然考虑在这一位上拆位计算，即：

$$\sum_{k=0}^{29} 2^k \sum_{S\in\{a\}_{i=1}^{n}} |S| \left(\min_{x\in S} x\right) \left(\max_{y\in S} y\right) \left[[2^k]\left(\bigoplus_{z\in S} z\right) = 1\right]$$

考虑枚举右端点，观察所有区间异或和为 1 的左端点和集合大小的乘积。考虑用 f 数组维护所有合法方案中当前异或和是 0 还是 1 的 $\min x$ 的和，g 数组表示所有合法方案的 $|S|\min x$ 的和，进行递推。

每当加入一个数字 k，假设当前它在这一位上二进制位是 y，那么对于 f（截至目前枚举到的数字中，所有合法选择状态下 $\min x$ 的和）的更新，有三种情况：

1. 仅选择当前数字。则 $f_y \leftarrow k$。
2. 不选择当前数字。则 $f_0 \leftarrow f_0$，$f_1 \leftarrow f_1$。
3. 之前所有的方案加入当前的数字，则 $\min x$ 本身不变，但是所有方案下 $\min x$ 的和发生变化。则新的 $f_0 \leftarrow f_x$，$f_1 \leftarrow f_{x\oplus 1}$。

对于 g，同样分为这三类：

1. 仅选择当前数字。则 $g_y \leftarrow k \times 1$。
2. 不选择当前数字。则 $g_0 \leftarrow g_0$，$g_1 \leftarrow g_1$。
3. 之前所有的方案加入当前的数字，则 $|S|\min x$ 本身要变成 $(|S|+1)\min x$。则新的 $g_0 \leftarrow f_x + g_x$，$g_1 \leftarrow f_{x\oplus 1} + g_{x\oplus 1}$。

按照上式递推更新即可。总复杂度 $\mathcal{O}(n \log V)$。

```cpp
#include <bits/stdc++.h>
#define fp(i, a, b) for (int i = a, i##_ = b; i <= i##_; ++i)
#define fd(i, a, b) for (int i = a, i##_ = b; i >= i##_; --i)

using namespace std;
using ll = long long;
const int N = 1e6 + 5, P = 998244353;
int n, a[N];
void Solve() {
    scanf("%d", &n);
    fp(i, 1, n) scanf("%d", a + i);
    sort(a + 1, a + n + 1);
    int ans = 0;
    fp(j, 0, 30) {
        vector<ll> f = {0, 0}, g = f, w;
        fp(i, 1, n) {
            int x = a[i] >> j & 1, sum;
            w = {0, 0}, w[x] = a[i], sum = (w[1] + f[x ^ 1] + g[x ^ 1]) % P;
            ans = (ans + (1ll << j) * a[i] % P * sum) % P;
            g = {(g[0] + w[0] + f[x] + g[x]) % P, (g[1] + sum) % P};
            f = {(f[0] + w[0] + f[x]) % P, (f[1] + w[1] + f[x ^ 1]) % P};
        }
```

```
    }
    printf("%d\n", ans);
}
int main() {
    int t = 1;
    // scanf("%d", &t);
    while (t--) Solve();
    return 0;
}
```

8 L Misaka Mikoto's Dynamic KMP Problem

题意：给定串 S，一次操作可以执行下面的其中一条：

1. 修改 S 中一个字符。
2. 给定串 T，问 S 在 T 中出现多少次，以及 S 的 border 长度。输出它们的乘积。

操作次数 $1 \le q \le 10^6$，$\sum |T|, S \le 2 \times 10^6$。强制在线。

解法：这题最大需要注意到的点就是输出的是匹配次数乘以 border 长度。因而当 $|t| < |s|$ 时，border 长度是无需计算的因为匹配次数必然为 0；而当 t 长的时候，暴力匹配 s 和 t，由于限制了 $\sum |t|$，因而这样暴力执行计算的次数不会很多，至多每次 $|t| = |s|$。这样总复杂度仅为 $\mathcal{O}(2\sum |T|)$ 。

```
#include <bits/stdc++.h>
using namespace std;
class KMP
{
    vector<int> nx;
    vector<long long> b;

public:
    KMP(vector<long long> &b)
    {
        this->b = b;
        int n = b.size();
        int j = 0;
        nx.resize(n);
        for (int i = 1; i < n; i++)
        {
            while (j > 0 && b[i] != b[j])
                j = nx[j - 1];
            if (b[i] == b[j])
                j++;
            nx[i] = j;
        }
    }
```

```
    int find(vector<long long> &a)
    {
        int n = b.size(), m = a.size();
        int j = 0;
        long long ans = 0;
        for (int i = 0; i < m; i++)
        {
            while (j > 0 && a[i] != b[j])
                j = nx[j - 1];
            if (a[i] == b[j])
                j++;
            if (j == n)
            {
                //匹配位点:i-n+1
                ans++;
                j = nx[j - 1];
            }
        }
        return ans;
    }
    int getBorder()
    {
        return nx.back();
    }
};
int main()
{
    long long b, mod, x, pos;
    int n, q, op;
    scanf("%d%d%lld%lld", &n, &q, &b, &mod);
    vector<long long> s(n);
    for (int i = 0; i < n; i++)
        scanf("%lld", &s[i]);
    long long cur = 1, ans = 0, lastans = 0;
    while (q--)
    {
        scanf("%d", &op);
        if (op == 1)
        {
            scanf("%lld%lld", &pos, &x);
            s[(pos ^ lastans) % n] = x ^ lastans;
        }
        else
        {
            cur = cur * b % mod;
            scanf("%lld", &pos);
            vector<long long> t(pos);
```

```
            for (auto &x : t)
            {
                scanf("%lld", &x);
                x ^= lastans;
            }
            if (t.size() < s.size())
            {
                lastans = 0;
                continue;
            }
            KMP solve(s);
            lastans = 1ll * solve.find(t) * solve.getBorder();
            ans = (ans + lastans % mod * cur % mod) % mod;
        }
    }
    printf("%lld", ans);
    return 0;
}
```

9 M Writing Books

题意：q 次询问 $[1,n]$ 中数位个数的和。$1 \le q \le 10^5$，$1 \le n \le 10^9$。

解法：枚举 $[10^k, 10^{k+1}-1]$，这里面每个数字数位个数都是 k。枚举 $k \in [1,9]$ 即可。复杂度 $\mathcal{O}(q \log V)$。

```
#include <bits/stdc++.h>
#define fp(i, a, b) for (int i = a, i##_ = b; i <= i##_; ++i)
#define fd(i, a, b) for (int i = a, i##_ = b; i >= i##_; --i)

using namespace std;
using ll = long long;
const int N = 2e5 + 5;
int n; ll f[10];
void Solve() {
    ll x, k = 0, i = 10;
    scanf("%lld", &x);
    for (; i <= x; i *= 10, ++k);
    // printf("%d %d %lld\n", k, x - i / 10 + 1, f[k - 1] + (x - i / 10 + 1) * (k + 1));
    printf("%lld\n", f[k - 1] + (x - i / 10 + 1) * (k + 1));
}
int main() {
    f[0] = 9;
    for (ll i = 1, k = 90; i <= 9; ++i, k *= 10)
        f[i] = f[i - 1] + (ll)k * (i + 1);
    // fp(i, 0, 9) printf("%lld ", f[i]);puts("");
    int t = 1;
```

```
    scanf("%d", &t);
    while (t--) Solve();
    return 0;
}
```

10 附录

在阅读下列背景知识前，最好有一些 信号与系统 的背景知识。

10.1 甲 生成函数入门

在研究离散时间信号的时候，会提到一种变换：Z 变换。对于一个单边离散时间信号 $f(t)$，会定义它的 Z 变换为 $\mathscr{Z}(z)=\sum\limits_{i=0}^{+\infty}f(i)z^i$，用一个函数来维护一个序列的性质。

考虑用一个函数来维护序列。首先考虑一个最基本的问题：

> 一个盒子中有 n 个完全相同的球，从中拿出 k 个球的方案数是多少？

这个问题的答案是显然的：$\binom{n}{k}$。当 $k=0$ 时答案是 $\binom{n}{0}$，$k=1$ 时答案为 $\binom{n}{1}$，依次类推。考虑把 $k=0,1,\cdots,n-1,n,n+1,\cdots$ 的答案记为一个数列 $\{f\}$，可以得到：

$$\binom{n}{0},\binom{n}{1},\cdots,\binom{n}{k},\cdots,\binom{n}{n-1},\binom{n}{n},0,0,\cdots$$

如果我们不希望每次都书写这么长的序列来记录这个问题的答案，可以考虑引入一个新的符号 x——它不可以和一般的数字进行加减乘除计算，当且仅当 x 的指数相同的时候才可以进行系数的加减运算。和 Z 变换相同，我们进行同样的变化：

$$\binom{n}{0}x^0+\binom{n}{1}x^1+\cdots+\binom{n}{k}x^k+\cdots+\binom{n}{n-1}x^{n-1}+\binom{n}{n}x^n+0x^{n+1}+0x^{n+2}\cdots$$

这时我们会发现这个求和式可以使用二项式定理变成 $(x+1)^n$。因而我们这时就可以使用一个简单的函数 $(x+1)^n$ 来去记录这个问题的一般答案。如果我们想知道这个序列的第 k 项，我们就可以去查询它的第 k 次项的系数，这个系数就表达了这个问题的答案。

但是引入这个记号还不足以带来足够的方便，我们希望这个式子它本身具有一定的组合意义。回到原问题：n 个盒子里面拿出 k 个球的方案，这时我们观察我们的 $(x+1)^n$，问题可以等价变换于——给定 n 个括号，每个括号里面可以选择 1 或者 x，问从这 n 个括号中选择 k 个 x 的方案是什么。这两个问题显然完全等价，因而不难发现选 k 个 x 的方案等价于 $(x+1)^n$ 的 x^k 项系数。我们尝试将 $(x+1)^n$ 和我们的原问题进行映射。首先原问题中每个球取或者不取是相互独立的，且每个球只有取或者不取两种方案。而 $(x+1)^n$ 中的乘法 $(x+1)\cdot(x+1)\cdots(x+1)$ 本质对应于各个球取或者不取的独立性，$x+1$ 对应于当前球取或者不取的两种方案，只能二择其一。即多项式上定义的乘法运算，对应了乘法原理；而加法运算对应了加法原理。至此，任何排列组合问题都可以使用多项式上的这两种运算进行定义和表示。

通常来说，一般的计数问题得到的多项式函数是平凡的，不能用一些很简单的标记方法合并。但是使用 FFT、NTT 等 $\mathcal{O}(n\log n)$ 的时间复杂度优秀的多项式卷积算法就可以快速计算一些卷积递推式，以达到快速计算的目的。

10.2 乙 乘积、卷积

在信号上，严格定义的多项式（离散时间信号）乘积和卷积运算如下：

1. 乘积：

$$f(x)\cdot g(x)=f(x)g(x)=\sum_{i=-\infty}^{+\infty}f_ig_ix^i$$

2. 卷积：

$$f*g(x)=\int_{-\infty}^{+\infty}f(\tau)g(x-\tau)\mathrm{d}\tau=\sum_{i=-\infty}^{+\infty}\sum_{j=-\infty}^{+\infty}f_jg_{i-j}x^j$$

不难看出，我们日常使用的多项式乘积运算本质对应于多项式卷积运算，而乘积其实实际上是各项系数一一相乘。之所以通常的乘积其实是用卷积算出来的，是因为数字的乘积本质是一个 $x=10$ 的多项式卷积——如 $232\times147=(2x^2+3x+2)(x^2+4x+7)|_{x=10}$，而且 x^k 系数得到也是通过 $\sum_{i=0}^{k}x^ix^{k-i}$ 逐项乘起来然后相加得到的。