Berlekamp–Massey 算法

Berlekamp–Massey 算法是一种用于求数列的最短递推式的算法。给定一个长为 n 的数列，如果它的最短递推式的阶数为 m，则 Berlekamp–Massey 算法能够在 $O(nm)$ 时间内求出数列的每个前缀的最短递推式。最坏情况下 $m=O(n)$，因此算法的最坏复杂度为 $O(n^2)$。

定义

定义一个数列 $\{a_0 \dots a_{n-1}\}$ 的递推式为满足下式的序列 $\{r_0 \dots r_m\}$：

$\sum_{j=0}^{m} r_j a_{i-j} = 0, \forall i \geq m$

其中 $r_0 = 1$。m 称为该递推式的 **阶数**。

数列 $\{a_i\}$ 的最短递推式即为阶数最小的递推式。

做法

与上面定义的稍有不同，这里定义一个新的递推系数 $\{f_0 \dots f_{m-1}\}$，满足：

$a_i = \sum_{j=0}^{m-1} f_j a_{i-j-1}, \forall i \geq m$

容易看出 $f_i = -r_{i+1}$，并且阶数 m 与之前的定义是相同的。

我们可以增量地求递推式，按顺序考虑 $\{a_i\}$ 的每一位，并在递推结果出现错误时对递推系数 $\{f_i\}$ 进行调整。方便起见，以下将前 i 位的最短递推式记为 $F_i = \{f_{i,j}\}$。

显然初始时有 $F_0 = \{\}$。假设递推系数 F_{i-1} 对数列 $\{a_i\}$ 的前 $i-1$ 项均成立，这时对第 i 项就有两种情况：

1. 递推系数对 a_i 也成立，这时不需要进行任何调整，直接令 $F_i = F_{i-1}$ 即可。
2. 递推系数对 a_i 不成立，这时需要对 F_{i-1} 进行调整，得到新的 F_i。

设 $\Delta_i = a_i - \sum_{j=0}^{m} f_{i-1,j} a_{i-j-1}$，即 a_i 与 F_{i-1} 的递推结果的差值。

如果这是第一次对递推系数进行修改，则说明 a_i 是序列中的第一个非零项。这时直接令 F_i 为 i 个 0 即可，显然这是一个合法的最短递推式。

否则设上一次对递推系数进行修改时，已考虑的 $\{a_i\}$ 的项数为 k。如果存在一个序列 $G = \{g_0 \dots g_{m'-1}\}$，满足：

$\sum_{j=0}^{m'-1} g_j a_{i'-j-1} = 0, \forall i' \in [m', i)$

并且 $\sum_{j=0}^{m'-1} g_j a_{i-j-1} = \Delta_i$，那么不难发现将 F_k 与 G 按位分别相加之后即可得到一个合法的递推系数 F_i。

考虑如何构造 G。一种可行的构造方案是令

$G = \{0, 0, \dots, 0, \frac{\Delta_i}{\Delta_k}, -\frac{\Delta_i}{\Delta_k} F_{k-1}\}$

其中前面一共有 $i-k-1$ 个 0，且最后的 $-\frac{\Delta_i}{\Delta_k} F_{k-1}$ 表示将 F_{k-1} 每项乘以 $-\frac{\Delta_i}{\Delta_k}$ 后接在序列后面。

不难验证此时 $\sum_{j=0}^{m'-1} g_j a_{i-j-1} = \Delta_k \frac{\Delta_i}{\Delta_k} = \Delta_i$，因此这样构造出的是一个合法的 G。将 F_i 赋值为 F_k 与 G 逐项相加后的结果即可。

如果要求的是符合最开始定义的递推式 $\{r_i\}$，则将 $\{f_j\}$ 全部取相反数后在最开始插入 $r_0 = 1$ 即可。

从上述算法流程中可以看出，如果数列的最短递推式的阶数为 m，则算法的复杂度为 $O(nm)$。最坏情况下 $m = O(n)$，因此算法的最坏复杂度为 $O(n^2)$。

在实现算法时，由于每次调整递推系数时都只需要用到上次调整时的递推系数 F_k，因此如果只需要求整个数列的最短递推式，可以只存储当前递推系数和上次调整时的递推系数，空间复杂度为 $O(n)$。

参考实现

```cpp
vector<int> berlekamp_massey(const vector<int> &a) {
  vector<int> v, last;  // v is the answer, 0-based, p is the module
  int k = -1, delta = 0;

  for (int i = 0; i < (int)a.size(); i++) {
    int tmp = 0;
    for (int j = 0; j < (int)v.size(); j++)
      tmp = (tmp + (long long)a[i - j - 1] * v[j]) % p;

    if (a[i] == tmp) continue;

    if (k < 0) {
      k = i;
      delta = (a[i] - tmp + p) % p;
      v = vector<int>(i + 1);

      continue;
    }

    vector<int> u = v;
    int val = (long long)(a[i] - tmp + p) * power(delta, p - 2) % p;

    if (v.size() < last.size() + i - k) v.resize(last.size() + i - k);

    (v[i - k - 1] += val) %= p;

    for (int j = 0; j < (int)last.size(); j++) {
      v[i - k + j] = (v[i - k + j] - (long long)val * last[j]) % p;
      if (v[i - k + j] < 0) v[i - k + j] += p;
    }

    if ((int)u.size() - i < (int)last.size() - k) {
      last = u;
      k = i;
      delta = a[i] - tmp;
      if (delta < 0) delta += p;
    }
  }

  for (auto &x : v) x = (p - x) % p;
  v.insert(v.begin(), 1);

  return v;  // $\forall i, \sum_{j = 0} ^ m a_{i - j} v_j = 0$
}
```

朴素的 Berlekamp–Massey 算法求解的是有限项数列的最短递推式。如果待求递推式的序列有无限项，但已知最短递推式的阶数上界，则只需取出序列的前 $2m$ 项即可求出整个序列的最短递推式。（证明略）

应用

由于 Berlekamp–Massey 算法的数值稳定性比较差，在处理实数问题时一般很少使用。为了叙述方便，以下均假定在某个质数 p 的剩余系下进行运算。

求向量列或矩阵列的最短递推式

如果要求向量列 $\boldsymbol{v}_i$ 的最短递推式，设向量的维数为 n，我们可以随机一个 n 维行向量 $\mathbf{u}^T$，并计算标量序列 $\{\boldsymbol{u}^T\boldsymbol{v}_i\}$ 的最短递推式。由 Schwartz–Zippel 引理，二者的最短递推式有至少 $1-\frac{n}{p}$ 的概率相同。

求矩阵列 $\{A_i\}$ 的最短递推式也是类似的，设矩阵的大小为 $n\times m$，则只需随机一个 $1\times n$ 的行向量 $\mathbf{u}^T$ 和一个 $m\times 1$ 的列向量 $\boldsymbol{v}$，并计算标量序列 $\{\boldsymbol{u}^TA_i\boldsymbol{v}\}$ 的最短递推式即可。由 Schwartz–Zippel 引理可以类似地得到二者相同的概率至少为 $1-\frac{n+m}{p}$。

优化矩阵快速幂

设 $\boldsymbol{f}_i$ 是一个 n 维列向量，并且转移满足 $\boldsymbol{f}_i=A\boldsymbol{f}_{i-1}$，则可以发现 $\{\boldsymbol{f}_i\}$ 是一个不超过 n 阶的线性递推向量列。（证明略）

我们可以直接暴力求出 $\boldsymbol{f}_0\dots\boldsymbol{f}_{2n-1}$，然后用前面提到的做法求出 $\{\boldsymbol{f}_i\}$ 的最短递推式，再调用 常系数齐次线性递推 即可。

如果要求的向量是 $\boldsymbol{f}_m$，则算法的复杂度是 $O(n^3+n\log n\log m)$。如果 A 是一个只有 k 个非零项的稀疏矩阵，则复杂度可以降为 $O(nk+n\log n\log m)$。但由于算法至少需要 $O(nk)$ 的时间预处理，因此在压力不大的情况下也可以使用 $O(n^2\log m)$ 的线性递推算法，复杂度同样是可以接受的。

求矩阵的最小多项式

方阵 A 的最小多项式是次数最小的并且满足 $f(A)=0$ 的多项式 f。

实际上最小多项式就是 $\{A^i\}$ 的最小递推式，所以直接调用 Berlekamp–Massey 算法就可以了。如果 A 是一个 n 阶方阵，则显然最小多项式的次数不超过 n。

瓶颈在于求出 A^i，因为如果直接每次做矩阵乘法的话复杂度会达到 $O(n^4)$。但考虑到求矩阵列的最短递推式时实际上求的是 $\{\boldsymbol{u}^TA^i\boldsymbol{v}\}$ 的最短递推式，因此我们只要求出 $A^i\boldsymbol{v}$ 就行了。

假设 A 有 k 个非零项，则复杂度为 $O(kn+n^2)$。

求稀疏矩阵行列式

如果能求出方阵 A 的特征多项式，则常数项乘上 $(-1)^n$ 就是行列式。但是最小多项式不一定就是特征多项式。

实际上如果把 A 乘上一个随机对角阵 B，则 AB 的最小多项式有至少 $1-\frac{2n^2-n}{p}$ 的概率就是特征多项式。最后再除掉 $\det B$ 就行了。

设 A 为 n 阶方阵，且有 k 个非零项，则复杂度为 $O(kn+n^2)$。

求稀疏矩阵的秩

设 A 是一个 $n\times m$ 的矩阵，首先随机一个 $n\times n$ 的对角阵 P 和一个 $m\times m$ 的对角阵 Q, 然后计算 $QAPA^TQ$ 的最小多项式即可。

实际上不用调用矩阵乘法，因为求最小多项式时要用 $QAPA^TQ$ 乘一个向量，所以我们依次把这几个矩阵乘到向量里就行了。答案就是最小多项式除掉所有 x 因子后剩下的次数。

设 A 有 k 个非零项，且 $n\le m$，则复杂度为 $O(kn+n^2)$。

解稀疏方程组

问题：已知 $A\mathbf{x}=\mathbf{b}$, 其中 A 是一个 $n\times n$ 的 **满秩** 稀疏矩阵，$\mathbf{b}$ 和 $\mathbf{x}$ 是 $1\times n$ 的列向量。$A,\mathbf{b}$ 已知，需要在低于 n^ω 的复杂度内解出 x。

做法：显然 $\mathbf{x} = A^{-1}\mathbf{b}$。如果我们能求出 $\{A^i\mathbf{b}\}(i \ge 0)$ 的最小递推式 $\{r_0 \dots r_{m-1}\}(m \le n)$, 那么就有结论

$A^{-1}\mathbf{b} = -\frac{1}{r_{m-1}} \sum_{i=0}^{m-2} A^i\mathbf{b}r_{m-2-i}$

（证明略）

因为 A 是稀疏矩阵，直接按定义递推出 $\mathbf{b} \dots A^{2n-1}\mathbf{b}$ 即可。

同样地，设 A 中有 k 个非零项，则复杂度为 $O(kn + n^2)$。

参考实现

```cpp
vector<int> solve_sparse_equations(const vector<tuple<int, int, int>> &A,
                                   const vector<int> &b) {
  int n = (int)b.size();  // 0-based

  vector<vector<int>> f({b});

  for (int i = 1; i < 2 * n; i++) {
    vector<int> v(n);
    auto &u = f.back();

    for (auto [x, y, z] : A)  // [x, y, value]
      v[x] = (v[x] + (long long)u[y] * z) % p;

    f.push_back(v);
  }

  vector<int> w(n);
  mt19937 gen;
  for (auto &x : w) x = uniform_int_distribution<int>(1, p - 1)(gen);

  vector<int> a(2 * n);
  for (int i = 0; i < 2 * n; i++)
    for (int j = 0; j < n; j++) a[i] = (a[i] + (long long)f[i][j] * w[j]) % p;

  auto c = berlekamp_massey(a);
  int m = (int)c.size();

  vector<int> ans(n);

  for (int i = 0; i < m - 1; i++)
    for (int j = 0; j < n; j++)
      ans[j] = (ans[j] + (long long)c[m - 2 - i] * f[i][j]) % p;

  int inv = power(p - c[m - 1], p - 2);

  for (int i = 0; i < n; i++) ans[i] = (long long)ans[i] * inv % p;

  return ans;
}
```

例题

1. LibreOJ #163. 高斯消元 2