斯特林数

第二类斯特林数（Stirling Number）

为什么先介绍第二类斯特林数

虽然被称作「第二类」，第二类斯特林数却在斯特林的相关著作和具体数学中被首先描述，同时也比第一类斯特林数常用得多。

第二类斯特林数（斯特林子集数）$\begin{Bmatrix}n\\k\end{Bmatrix}$，也可记做 $S(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空子集的方案数。

递推式

$$\begin{Bmatrix}n\\k\end{Bmatrix}=\begin{Bmatrix}n-1\\k-1\end{Bmatrix}+k\begin{Bmatrix}n-1\\k\end{Bmatrix}$$

边界是 $\begin{Bmatrix}n\\0\end{Bmatrix}=[n=0]$。

考虑用组合意义来证明。

我们插入一个新元素时，有两种方案：

- 将新元素单独放入一个子集，有 $\begin{Bmatrix}n-1\\k-1\end{Bmatrix}$ 种方案；
- 将新元素放入一个现有的非空子集，有 $k\begin{Bmatrix}n-1\\k\end{Bmatrix}$ 种方案。

根据加法原理，将两式相加即可得到递推式。

通项公式

$$\begin{Bmatrix}n\\m\end{Bmatrix}=\sum_{i=0}^{m}\frac{(-1)^{m-i}i^n}{i!(m-i)!}$$

使用容斥原理证明该公式。设将 n 个两两不同的元素，划分到 i 个两两不同的集合（允许空集）的方案数为 G_i，将 n 个两两不同的元素，划分到 i 个两两不同的非空集合（不允许空集）的方案数为 F_i。

显然

$$
G_i = i^n
$$

$$
G_i = \sum_{j=0}^{i} \binom{i}{j} F_j
$$

根据二项式反演

$$
\begin{aligned}
F_i &= \sum_{j=0}^{i} (-1)^{i-j} \binom{i}{j} G_j \\
&= \sum_{j=0}^{i} (-1)^{i-j} \binom{i}{j} j^n \\
&= \sum_{j=0}^{i} \frac{i!(-1)^{i-j} j^n}{j!(i-j)!}
\end{aligned}
$$

考虑 F_i 与 $\begin{Bmatrix}n\\i\end{Bmatrix}$ 的关系。第二类斯特林数要求集合之间互不区分，因此 F_i 正好就是 $\begin{Bmatrix}n\\i\end{Bmatrix}$ 的 $i!$ 倍。于是

$$
\begin{Bmatrix}n\\m\end{Bmatrix} = \frac{F_m}{m!} = \sum_{i=0}^{m} \frac{(-1)^{m-i} i^n}{i!(m-i)!}
$$

同一行第二类斯特林数的计算

「同一行」的第二类斯特林数指的是，有着不同的 i，相同的 n 的一系列 $\begin{Bmatrix}n\\i\end{Bmatrix}$。求出同一行的所有第二类斯特林数，就是对 $i = 0..n$ 求出了将 n 个不同元素划分为 i 个非空集的方案数。

根据上面给出的通项公式，卷积计算即可。该做法的时间复杂度为 $O(n \log n)$。

下面的代码使用了名为 `poly` 的多项式类，仅供参考。

实现

```
#ifndef _FEISTDLIB_POLY_
#define _FEISTDLIB_POLY_

/*
 * This file is part of the fstdlib project.
 * Version: Build v0.0.2
 * You can check for details at
https://github.com/FNatsuka/fstdlib
 */

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <vector>

namespace fstdlib {

using ll = long long;
int mod = 998244353, grt = 3;

class poly {
 private:
  std::vector<int> data;

  void out(void) {
    for (int i = 0; i < (int)data.size(); ++i) printf("%d ",
data[i]);
    puts("");
  }

 public:
  poly(std::size_t len = std::size_t(0)) { data =
std::vector<int>(len); }

  poly(const std::vector<int> &b) { data = b; }

  poly(const poly &b) { data = b.data; }

  void resize(std::size_t len, int val = 0) { data.resize(len,
val); }

  std::size_t size(void) const { return data.size(); }

  void clear(void) { data.clear(); }
#if __cplusplus >= 201103L
  void shrink_to_fit(void) { data.shrink_to_fit(); }
#endif
  int &operator[](std::size_t b) { return data[b]; }

```

```
  const int &operator[](std::size_t b) const { return data[b]; }

  poly operator*(const poly &h) const;
  poly operator*=(const poly &h);
  poly operator*(const int &h) const;
  poly operator*=(const int &h);
  poly operator+(const poly &h) const;
  poly operator+=(const poly &h);
  poly operator-(const poly &h) const;
  poly operator-=(const poly &h);
  poly operator<<(const std::size_t &b) const;
  poly operator<<=(const std::size_t &b);
  poly operator>>(const std::size_t &b) const;
  poly operator>>=(const std::size_t &b);
  poly operator/(const int &h) const;
  poly operator/=(const int &h);
  poly operator==(const poly &h) const;
  poly operator!=(const poly &h) const;
  poly operator+(const int &h) const;
  poly operator+=(const int &h);
  poly inv(void) const;
  poly inv(const int &h) const;
  friend poly sqrt(const poly &h);
  friend poly log(const poly &h);
  friend poly exp(const poly &h);
};

int qpow(int a, int b, int p = mod) {
  int res = 1;
  while (b) {
    if (b & 1) res = (ll)res * a % p;
    a = (ll)a * a % p, b >>= 1;
  }
  return res;
}

std::vector<int> rev;

void dft_for_module(std::vector<int> &f, int n, int b) {
  static std::vector<int> w;
  w.resize(n);
  for (int i = 0; i < n; ++i)
    if (i < rev[i]) std::swap(f[i], f[rev[i]]);
  for (int i = 2; i <= n; i <<= 1) {
    w[0] = 1, w[1] = qpow(grt, (mod - 1) / i);
    if (b == -1) w[1] = qpow(w[1], mod - 2);
    for (int j = 2; j < i / 2; ++j) w[j] = (ll)w[j - 1] * w[1] %
mod;
    for (int j = 0; j < n; j += i)
      for (int k = 0; k < i / 2; ++k) {
        int p = f[j + k], q = (ll)f[j + k + i / 2] * w[k] % mod;
        f[j + k] = (p + q) % mod, f[j + k + i / 2] = (p - q +
```

```
mod) % mod;
      }
    }
  }

poly poly::operator*(const poly &h) const {
  int N = 1;
  while (N < (int)(size() + h.size() - 1)) N <<= 1;
  std::vector<int> f(this->data), g(h.data);
  f.resize(N), g.resize(N);
  rev.resize(N);
  for (int i = 0; i < N; ++i)
    rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? N >> 1 : 0);
  dft_for_module(f, N, 1), dft_for_module(g, N, 1);
  for (int i = 0; i < N; ++i) f[i] = (ll)f[i] * g[i] % mod;
  dft_for_module(f, N, -1), f.resize(size() + h.size() - 1);
  for (int i = 0, inv = qpow(N, mod - 2); i < (int)f.size();
++i)
    f[i] = (ll)f[i] * inv % mod;
  return f;
}

poly poly::operator*=(const poly &h) { return *this = *this * h;
}

poly poly::operator*(const int &h) const {
  std::vector<int> f(this->data);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * h %
mod;
  return f;
}

poly poly::operator*=(const int &h) {
  for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] *
h % mod;
  return *this;
}

poly poly::operator+(const poly &h) const {
  std::vector<int> f(this->data);
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) %
mod;
  return f;
}

poly poly::operator+=(const poly &h) {
  std::vector<int> &f = this->data;
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) %
mod;
  return f;
```

```
}

poly poly::operator-(const poly &h) const {
  std::vector<int> f(this->data);
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] - h[i] +
mod) % mod;
  return f;
}

poly poly::operator-=(const poly &h) {
  std::vector<int> &f = this->data;
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] - h[i] +
mod) % mod;
  return f;
}

poly poly::operator<<(const std::size_t &b) const {
  std::vector<int> f(size() + b);
  for (int i = 0; i < (int)size(); ++i) f[i + b] = data[i];
  return f;
}

poly poly::operator<<=(const std::size_t &b) { return *this =
(*this) << b; }

poly poly::operator>>(const std::size_t &b) const {
  std::vector<int> f(size() - b);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = data[i + b];
  return f;
}

poly poly::operator>>=(const std::size_t &b) { return *this =
(*this) >> b; }

poly poly::operator/(const int &h) const {
  std::vector<int> f(this->data);
  int inv = qpow(h, mod - 2);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * inv
% mod;
  return f;
}

poly poly::operator/=(const int &h) {
  int inv = qpow(h, mod - 2);
  for (int i = 0; i < (int)data.size(); ++i) data[i] =
(ll)data[i] * inv % mod;
  return *this;
}

poly poly::inv(void) const {
```

```cpp
  int N = 1;
  while (N < (int)(size() + size() - 1)) N <<= 1;
  std::vector<int> f(N), g(N), d(this->data);
  d.resize(N), f[0] = qpow(d[0], mod - 2);
  for (int w = 2; w < N; w <<= 1) {
    for (int i = 0; i < w; ++i) g[i] = d[i];
    rev.resize(w << 1);
    for (int i = 0; i < w * 2; ++i)
      rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? w : 0);
    dft_for_module(f, w << 1, 1), dft_for_module(g, w << 1, 1);
    for (int i = 0; i < w * 2; ++i)
      f[i] = (ll)f[i] * (2 + mod - (ll)f[i] * g[i] % mod) % mod;
    dft_for_module(f, w << 1, -1);
    for (int i = 0, inv = qpow(w << 1, mod - 2); i < w; ++i)
      f[i] = (ll)f[i] * inv % mod;
    for (int i = w; i < w * 2; ++i) f[i] = 0;
  }
  f.resize(size());
  return f;
}

poly poly::operator==(const poly &h) const {
  if (size() != h.size()) return 0;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 0;
  return 1;
}

poly poly::operator!=(const poly &h) const {
  if (size() != h.size()) return 1;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 1;
  return 0;
}

poly poly::operator+(const int &h) const {
  poly f(this->data);
  f[0] = (f[0] + h) % mod;
  return f;
}

poly poly::operator+=(const int &h) { return *this = (*this) + h; }

poly poly::inv(const int &h) const {
  poly f(*this);
  f.resize(h);
  return f.inv();
}

int modsqrt(int h, int p = mod) { return 1; }
```

```cpp
poly sqrt(const poly &h) {
  int N = 1;
  while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
  poly f(N), g(N), d(h);
  d.resize(N), f[0] = modsqrt(d[0]);
  for (int w = 2; w < N; w <<= 1) {
    g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = d[i];
    f = (f + f.inv(w) * g) / 2;
    f.resize(w);
  }
  f.resize(h.size());
  return f;
}

poly log(const poly &h) {
  poly f(h);
  for (int i = 1; i < (int)f.size(); ++i) f[i - 1] = (ll)f[i] *
i % mod;
  f[f.size() - 1] = 0, f = f * h.inv(), f.resize(h.size());
  for (int i = (int)f.size() - 1; i > 0; --i)
    f[i] = (ll)f[i - 1] * qpow(i, mod - 2) % mod;
  f[0] = 0;
  return f;
}

poly exp(const poly &h) {
  int N = 1;
  while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
  poly f(N), g(N), d(h);
  f[0] = 1, d.resize(N);
  for (int w = 2; w < N; w <<= 1) {
    f.resize(w), g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = d[i];
    f = f * (g + 1 - log(f));
    f.resize(w);
  }
  f.resize(h.size());
  return f;
}

struct comp {
  long double x, y;

  comp(long double _x = 0, long double _y = 0) : x(_x), y(_y) {}

  comp operator*(const comp &b) const {
    return comp(x * b.x - y * b.y, x * b.y + y * b.x);
  }

  comp operator+(const comp &b) const { return comp(x + b.x, y +
b.y); }
```

```cpp

  comp operator-(const comp &b) const { return comp(x - b.x, y - b.y); }

  comp conj(void) { return comp(x, -y); }
};

const int EPS = 1e-9;

template <typename FLOAT_T>
FLOAT_T fabs(const FLOAT_T &x) {
  return x > 0 ? x : -x;
}

template <typename FLOAT_T>
FLOAT_T sin(const FLOAT_T &x, const long double &EPS = fstdlib::EPS) {
  FLOAT_T res = 0, delt = x;
  int d = 0;
  while (fabs(delt) > EPS) {
    res += delt, ++d;
    delt *= -x * x / ((2 * d) * (2 * d + 1));
  }
  return res;
}

template <typename FLOAT_T>
FLOAT_T cos(const FLOAT_T &x, const long double &EPS = fstdlib::EPS) {
  FLOAT_T res = 0, delt = 1;
  int d = 0;
  while (fabs(delt) > EPS) {
    res += delt, ++d;
    delt *= -x * x / ((2 * d) * (2 * d - 1));
  }
  return res;
}

const long double PI = std::acos((long double)(-1));

void dft_for_complex(std::vector<comp> &f, int n, int b) {
  static std::vector<comp> w;
  w.resize(n);
  for (int i = 0; i < n; ++i)
    if (i < rev[i]) std::swap(f[i], f[rev[i]]);
  for (int i = 2; i <= n; i <<= 1) {
    w[0] = comp(1, 0), w[1] = comp(cos(2 * PI / i), b * sin(2 * PI / i));
    for (int j = 2; j < i / 2; ++j) w[j] = w[j - 1] * w[1];
    for (int j = 0; j < n; j += i)
      for (int k = 0; k < i / 2; ++k) {
        comp p = f[j + k], q = f[j + k + i / 2] * w[k];
```

```
        f[j + k] = p + q, f[j + k + i / 2] = p - q;
      }
    }
  }

class arbitrary_module_poly {
 private:
  std::vector<int> data;

  int construct_element(int D, ll x, ll y, ll z) const {
    x %= mod, y %= mod, z %= mod;
    return ((ll)D * D * x % mod + (ll)D * y % mod + z) % mod;
  }

 public:
  int mod;

  arbitrary_module_poly(std::size_t len = std::size_t(0),
                        int module_value = 1e9 + 7) {
    mod = module_value;
    data = std::vector<int>(len);
  }

  arbitrary_module_poly(const std::vector<int> &b, int
module_value = 1e9 + 7) {
    mod = module_value;
    data = b;
  }

  arbitrary_module_poly(const arbitrary_module_poly &b) {
    mod = b.mod;
    data = b.data;
  }

  void resize(std::size_t len, const int &val = 0) {
data.resize(len, val); }

  std::size_t size(void) const { return data.size(); }

  void clear(void) { data.clear(); }
#if __cplusplus >= 201103L
  void shrink_to_fit(void) { data.shrink_to_fit(); }
#endif
  int &operator[](std::size_t b) { return data[b]; }

  const int &operator[](std::size_t b) const { return data[b]; }

  arbitrary_module_poly operator*(const arbitrary_module_poly
&h) const;
  arbitrary_module_poly operator*=(const arbitrary_module_poly
&h);
  arbitrary_module_poly operator*(const int &h) const;
```

```
  arbitrary_module_poly operator*=(const int &h);
  arbitrary_module_poly operator+(const arbitrary_module_poly
&h) const;
  arbitrary_module_poly operator+=(const arbitrary_module_poly
&h);
  arbitrary_module_poly operator-(const arbitrary_module_poly
&h) const;
  arbitrary_module_poly operator-=(const arbitrary_module_poly
&h);
  arbitrary_module_poly operator<<(const std::size_t &b) const;
  arbitrary_module_poly operator<<=(const std::size_t &b);
  arbitrary_module_poly operator>>(const std::size_t &b) const;
  arbitrary_module_poly operator>>=(const std::size_t &b);
  arbitrary_module_poly operator/(const int &h) const;
  arbitrary_module_poly operator/=(const int &h);
  arbitrary_module_poly operator==(const arbitrary_module_poly
&h) const;
  arbitrary_module_poly operator!=(const arbitrary_module_poly
&h) const;
  arbitrary_module_poly inv(void) const;
  arbitrary_module_poly inv(const int &h) const;
  friend arbitrary_module_poly sqrt(const arbitrary_module_poly
&h);
  friend arbitrary_module_poly log(const arbitrary_module_poly
&h);
};

arbitrary_module_poly arbitrary_module_poly::operator*(
    const arbitrary_module_poly &h) const {
  int N = 1;
  while (N < (int)(size() + h.size() - 1)) N <<= 1;
  std::vector<comp> f(N), g(N), p(N), q(N);
  const int D = std::sqrt(mod);
  for (int i = 0; i < (int)size(); ++i)
    f[i].x = data[i] / D, f[i].y = data[i] % D;
  for (int i = 0; i < (int)h.size(); ++i) g[i].x = h[i] / D,
g[i].y = h[i] % D;
  rev.resize(N);
  for (int i = 0; i < N; ++i)
    rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? N >> 1 : 0);
  dft_for_complex(f, N, 1), dft_for_complex(g, N, 1);
  for (int i = 0; i < N; ++i) {
    p[i] = (f[i] + f[(N - i) % N].conj()) * comp(0.50, 0) *
g[i];
    q[i] = (f[i] - f[(N - i) % N].conj()) * comp(0, -0.5) *
g[i];
  }
  dft_for_complex(p, N, -1), dft_for_complex(q, N, -1);
  std::vector<int> r(size() + h.size() - 1);
  for (int i = 0; i < (int)r.size(); ++i)
    r[i] = construct_element(D, p[i].x / N + 0.5, (p[i].y +
q[i].x) / N + 0.5,
```

```
                             q[i].y / N + 0.5);
  return arbitrary_module_poly(r, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator*=(
    const arbitrary_module_poly &h) {
  return *this = *this * h;
}

arbitrary_module_poly arbitrary_module_poly::operator*(const int
&h) const {
  std::vector<int> f(this->data);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * h %
mod;
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator*=(const
int &h) {
  for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] *
h % mod;
  return *this;
}

arbitrary_module_poly arbitrary_module_poly::operator+(
    const arbitrary_module_poly &h) const {
  std::vector<int> f(this->data);
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) %
mod;
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator+=(
    const arbitrary_module_poly &h) {
  if (size() < h.size()) resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) data[i] = (data[i] +
h[i]) % mod;
  return *this;
}

arbitrary_module_poly arbitrary_module_poly::operator-(
    const arbitrary_module_poly &h) const {
  std::vector<int> f(this->data);
  if (f.size() < h.size()) f.resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + mod -
h[i]) % mod;
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator-=(
    const arbitrary_module_poly &h) {
```

```
  if (size() < h.size()) resize(h.size());
  for (int i = 0; i < (int)h.size(); ++i)
    data[i] = (data[i] + mod - h[i]) % mod;
  return *this;
}

arbitrary_module_poly arbitrary_module_poly::operator<<(
    const std::size_t &b) const {
  std::vector<int> f(size() + b);
  for (int i = 0; i < (int)size(); ++i) f[i + b] = data[i];
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator<<=(const
std::size_t &b) {
  return *this = (*this) << b;
}

arbitrary_module_poly arbitrary_module_poly::operator>>(
    const std::size_t &b) const {
  std::vector<int> f(size() - b);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = data[i + b];
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator>>=(const
std::size_t &b) {
  return *this = (*this) >> b;
}

arbitrary_module_poly arbitrary_module_poly::inv(void) const {
  int N = 1;
  while (N < (int)(size() + size() - 1)) N <<= 1;
  arbitrary_module_poly f(1, mod), g(N, mod), h(*this), f2(1,
mod);
  f[0] = qpow(data[0], mod - 2, mod), h.resize(N), f2[0] = 2;
  for (int w = 2; w < N; w <<= 1) {
    g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = h[i];
    f = f * (f * g - f2) * (mod - 1);
    f.resize(w);
  }
  f.resize(size());
  return f;
}

arbitrary_module_poly arbitrary_module_poly::inv(const int &h)
const {
  arbitrary_module_poly f(*this);
  f.resize(h);
  return f.inv();
}
```

```
arbitrary_module_poly arbitrary_module_poly::operator/(const int
&h) const {
  int inv = qpow(h, mod - 2, mod);
  std::vector<int> f(this->data);
  for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * inv
% mod;
  return arbitrary_module_poly(f, mod);
}

arbitrary_module_poly arbitrary_module_poly::operator/=(const
int &h) {
  int inv = qpow(h, mod - 2, mod);
  for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] *
inv % mod;
  return *this;
}

arbitrary_module_poly arbitrary_module_poly::operator==(
    const arbitrary_module_poly &h) const {
  if (size() != h.size() || mod != h.mod) return 0;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 0;
  return 1;
}

arbitrary_module_poly arbitrary_module_poly::operator!=(
    const arbitrary_module_poly &h) const {
  if (size() != h.size() || mod != h.mod) return 1;
  for (int i = 0; i < (int)size(); ++i)
    if (data[i] != h[i]) return 1;
  return 0;
}

arbitrary_module_poly sqrt(const arbitrary_module_poly &h) {
  int N = 1;
  while (N < (int)(h.size() + h.size() - 1)) N <<= 1;
  arbitrary_module_poly f(1, mod), g(N, mod), d(h);
  f[0] = modsqrt(h[0], mod), d.resize(N);
  for (int w = 2; w < N; w <<= 1) {
    g.resize(w);
    for (int i = 0; i < w; ++i) g[i] = d[i];
    f = (f + f.inv(w) * g) / 2;
    f.resize(w);
  }
  f.resize(h.size());
  return f;
}

arbitrary_module_poly log(const arbitrary_module_poly &h) {
  arbitrary_module_poly f(h);
  for (int i = 1; i < (int)f.size(); ++i) f[i - 1] = (ll)f[i] *
```

```
i % f.mod;
  f[f.size() - 1] = 0, f = f * h.inv(), f.resize(h.size());
  for (int i = (int)f.size() - 1; i > 0; --i)
    f[i] = (ll)f[i - 1] * qpow(i, f.mod - 2, f.mod) % f.mod;
  f[0] = 0;
  return f;
}

using m_poly = arbitrary_module_poly;
}  // namespace fstdlib

#endif
```

实现

```
int main() {
  scanf("%d", &n);
  fact[0] = 1;
  for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i %
mod;
  exgcd(fact[n], mod, ifact[n], ifact[0]),
      ifact[n] = (ifact[n] % mod + mod) % mod;
  for (int i = n - 1; i >= 0; --i) ifact[i] = (ll)ifact[i + 1] *
(i + 1) % mod;
  poly f(n + 1), g(n + 1);
  for (int i = 0; i <= n; ++i)
    g[i] = (i & 1 ? mod - 1ll : 1ll) * ifact[i] % mod,
    f[i] = (ll)qpow(i, n) * ifact[i] % mod;
  f *= g, f.resize(n + 1);
  for (int i = 0; i <= n; ++i) printf("%d ", f[i]);
  return 0;
}
```

同一列第二类斯特林数的计算

「同一列」的第二类斯特林数指的是，有着不同的 i，相同的 k 的一系列 $\begin{Bmatrix}i\\k\end{Bmatrix}$。求出同一列的所有第二类斯特林数，就是对 $i=0..n$ 求出了将 i 个不同元素划分为 k 个非空集的方案数。

利用指数型生成函数计算。

一个盒子装 i 个物品且盒子非空的方案数是 $[i>0]$。我们可以写出它的指数型生成函数为 $F(x)=\sum\limits_{i=1}^{+\infty}\frac{x^i}{i!}=\mathrm{e}^x-1$。经过之前的学习，我们明白 $F^k(x)$ 就是 i 个有标号物品放到 k 个有标号盒子里的指数型生成函数，那么除掉 $k!$ 就是 i 个有标号物品放到 k 个无标号盒子里的指数型生成函数。

$\begin{Bmatrix}i\\k\end{Bmatrix}=\frac{\left[\frac{x^i}{i!}\right]F^k(x)}{k!}$，$O(n\log n)$ 计算多项式幂即可。

另外，$\exp F(x)=\sum\limits_{i=0}^{+\infty}\frac{F^i(x)}{i!}$ 就是 i 个有标号物品放到任意多个无标号盒子里的指数型生成函数（EXP 通过每项除以一个 $i!$ 去掉了盒子的标号）。这其实就是贝尔数的生成函数。

这里涉及到很多「有标号」「无标号」的内容，注意辨析。

实现

```
int main() {
  scanf("%d%d", &n, &k);
  poly f(n + 1);
  fact[0] = 1;
  for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i % mod;
  for (int i = 1; i <= n; ++i) f[i] = qpow(fact[i], mod - 2);
  f = exp(log(f >> 1) * k) << k, f.resize(n + 1);
  int inv = qpow(fact[k], mod - 2);
  for (int i = 0; i <= n; ++i)
    printf("%lld ", (ll)f[i] * fact[i] % mod * inv % mod);
  return 0;
}
```

第一类斯特林数（Stirling Number）

第一类斯特林数（斯特林轮换数）$\begin{bmatrix}n\\k\end{bmatrix}$，也可记做 $s(n,k)$，表示将 n 个两两不同的元素，划分为 k 个互不区分的非空轮换的方案数。

一个轮换就是一个首尾相接的环形排列。我们可以写出一个轮换 $[A,B,C,D]$，并且我们认为 $[A,B,C,D]=[B,C,D,A]=[C,D,A,B]=[D,A,B,C]$，即，两个可以通过旋转而互相得到的轮换是等价的。注意，我们不认为两个可以通过翻转而相互得到的轮换等价，即 $[A,B,C,D]\neq[D,C,B,A]$。

递推式

$$\begin{bmatrix}n\\k\end{bmatrix}=\begin{bmatrix}n-1\\k-1\end{bmatrix}+(n-1)\begin{bmatrix}n-1\\k\end{bmatrix}$$

边界是 $\begin{bmatrix}n\\0\end{bmatrix}=[n=0]$。

该递推式的证明可以考虑其组合意义。

我们插入一个新元素时，有两种方案：

- 将该新元素置于一个单独的轮换中，共有 $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$ 种方案；
- 将该元素插入到任何一个现有的轮换中，共有 $(n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$ 种方案。

根据加法原理，将两式相加即可得到递推式。

通项公式

第一类斯特林数没有实用的通项公式。

同一行第一类斯特林数的计算

类似第二类斯特林数，我们构造同行第一类斯特林数的生成函数，即

$$F_n(x) = \sum_{i=0}^{n} \begin{bmatrix} n \\ i \end{bmatrix} x^i$$

根据递推公式，不难写出

$$F_n(x) = (n-1)F_{n-1}(x) + xF_{n-1}(x)$$

于是

$$F_n(x) = \prod_{i=0}^{n-1}(x+i) = \frac{(x+n-1)!}{(x-1)!}$$

这其实是 x 的 n 次上升阶乘幂，记做 $x^{\overline{n}}$。这个东西自然是可以暴力分治乘 $O(n\log^2 n)$ 求出的，但用上升幂相关做法可以 $O(n\log n)$ 求出，详情见 多项式平移 | 连续点值平移。

同一列第一类斯特林数的计算

仿照第二类斯特林数的计算，我们可以用指数型生成函数解决该问题。注意，由于递推公式和行有关，我们不能利用递推公式计算同列的第一类斯特林数。

显然，单个轮换的指数型生成函数为

$$F(x) = \sum_{i=1}^{n} \frac{(i-1)!x^i}{i!} = \sum_{i=1}^{n} \frac{x^i}{i}$$

它的 k 次幂就是 $\begin{bmatrix} i \\ k \end{bmatrix}$ 的指数型生成函数，$O(n\log n)$ 计算即可。

实现

```
int main() {
  scanf("%d%d", &n, &k);
  fact[0] = 1;
  for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i %
mod;
  ifact[n] = qpow(fact[n], mod - 2);
  for (int i = n - 1; i >= 0; --i) ifact[i] = (ll)ifact[i + 1] *
(i + 1) % mod;
  poly f(n + 1);
  for (int i = 1; i <= n; ++i) f[i] = (ll)fact[i - 1] * ifact[i]
% mod;
  f = exp(log(f >> 1) * k) << k, f.resize(n + 1);
  for (int i = 0; i <= n; ++i)
    printf("%lld ", (ll)f[i] * fact[i] % mod * ifact[k] % mod);
  return 0;
}
```

应用

上升幂与普通幂的相互转化

我们记上升阶乘幂 $x^{\overline{n}}=\prod_{k=0}^{n-1}(x+k)$。

则可以利用下面的恒等式将上升幂转化为普通幂：

$$x^{\overline{n}}=\sum_{k}{n \brack k}x^k$$

如果将普通幂转化为上升幂，则有下面的恒等式：

$$x^n=\sum_{k}{n \brace k}(-1)^{n-k}x^{\overline{k}}$$

下降幂与普通幂的相互转化

我们记下降阶乘幂 $x^{\underline{n}}=\dfrac{x!}{(x-n)!}=\prod_{k=0}^{n-1}(x-k)$。

则可以利用下面的恒等式将普通幂转化为下降幂：

$$x^n=\sum_{k}{n \brace k}x^{\underline{k}}$$

如果将下降幂转化为普通幂，则有下面的恒等式：

$$x^{\underline{n}}=\sum_{k}{n \brack k}(-1)^{n-k}x^k$$

多项式下降阶乘幂表示与多项式点值表示的关系

在这里，多项式的下降阶乘幂表示就是用

$$f(x)=\sum_{i=0}^{n} b_i x^{\underline{i}}$$

的形式表示一个多项式，而点值表示就是用 $n+1$ 个点

$$(i,a_i), i=0..n$$

来表示一个多项式。

显然，下降阶乘幂 b 和点值 a 间满足这样的关系：

$$a_k=\sum_{i=0}^{n} b_i k^{\underline{i}}$$

即

$$\begin{aligned}a_k&=\sum_{i=0}^{n}\frac{b_i k!}{(k-i)!}\\ \frac{a_k}{k!}&=\sum_{i=0}^{k} b_i \frac{1}{(k-i)!}\end{aligned}$$

这是一个卷积形式的式子，我们可以在 $O(n\log n)$ 的时间复杂度内完成点值和下降阶乘幂的互相转化。

习题

- HDU3625 Examining the Rooms
- UOJ540 联合省选 2020 组合数问题
- UOJ269 清华集训 2016 如何优雅地求和

参考资料与注释

1. Stirling Number of the First Kind - Wolfram MathWorld
2. Stirling Number of the Second Kind - Wolfram MathWorld

Konano, LLLMMKK, Menci, purple-vine, shuzhouliu, YanagiOrigami