

# 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） $\left\{ \begin{matrix} n \\ k \end{matrix} \right\}$ ，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

$$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = \left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\} + k \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\}$$

边界是  $\left\{ \begin{matrix} n \\ 0 \end{matrix} \right\} = [n=0]$ 。

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  $\left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\}$  种方案；
- 将新元素放入一个现有的非空子集，有  $k \left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\}$  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

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

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

$$F_i = \sum_{j=0}^i (-1)^{i-j} \binom{i}{j} G_j$$

$$= \sum_{j=0}^i (-1)^{i-j} \binom{i}{j} j^n$$

$$= \sum_{j=0}^i \frac{i! (-1)^{i-j} j^n}{j! (i-j)!}$$

考虑  $F_i$  与  $\left\{ \begin{smallmatrix} n \\ i \end{smallmatrix} \right\}$  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  $\left\{ \begin{smallmatrix} n \\ i \end{smallmatrix} \right\}$  的  $i!$  倍。于是

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

## 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  $\left\{ \begin{smallmatrix} n \\ i \end{smallmatrix} \right\}$ 。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。

```
1  #ifndef _FEISTDLIB_POLY_
2  #define _FEISTDLIB_POLY_
3
4  /*
5   * This file is part of the fstdlib project.
6   * Version: Build v0.0.2
7   * You can check for details at
8   https://github.com/FNatsuka/fstdlib
9   */
10
11  #include <algorithm>
12  #include <cmath>
13  #include <cstdio>
14  #include <vector>
15
16  namespace fstdlib {
17
18  using ll = long long;
19  int mod = 998244353, grt = 3;
20
21  class poly {
22  private:
23      std::vector<int> data;
24
25      void out(void) {
26          for (int i = 0; i < (int)data.size(); ++i) printf("%d ",
27 data[i]);
28          puts("");
29      }
30
31  public:
32      poly(std::size_t len = std::size_t(0)) { data =
33 std::vector<int>(len); }
34
35      poly(const std::vector<int> &b) { data = b; }
36
37      poly(const poly &b) { data = b.data; }
38
39      void resize(std::size_t len, int val = 0) { data.resize(len,
40 val); }
41
42      std::size_t size(void) const { return data.size(); }
43
44      void clear(void) { data.clear(); }
45  #if __cplusplus >= 201103L
46      void shrink_to_fit(void) { data.shrink_to_fit(); }
47  #endif
48      int &operator[](std::size_t b) { return data[b]; }
49  }
```

```

50     const int &operator[](std::size_t b) const { return data[b]; }
51
52     poly operator*(const poly &h) const;
53     poly operator*=(const poly &h);
54     poly operator*(const int &h) const;
55     poly operator*=(const int &h);
56     poly operator+(const poly &h) const;
57     poly operator+=(const poly &h);
58     poly operator-(const poly &h) const;
59     poly operator-=(const poly &h);
60     poly operator<<(const std::size_t &b) const;
61     poly operator<=<=(const std::size_t &b);
62     poly operator>>(const std::size_t &b) const;
63     poly operator>>=(const std::size_t &b);
64     poly operator/(const int &h) const;
65     poly operator/=(const int &h);
66     poly operator==(const poly &h) const;
67     poly operator!=(const poly &h) const;
68     poly operator+(const int &h) const;
69     poly operator+=(const int &h);
70     poly inv(void) const;
71     poly inv(const int &h) const;
72     friend poly sqrt(const poly &h);
73     friend poly log(const poly &h);
74     friend poly exp(const poly &h);
75 };
76
77 int qpow(int a, int b, int p = mod) {
78     int res = 1;
79     while (b) {
80         if (b & 1) res = (ll)res * a % p;
81         a = (ll)a * a % p, b >>= 1;
82     }
83     return res;
84 }
85
86 std::vector<int> rev;
87
88 void dft_for_module(std::vector<int> &f, int n, int b) {
89     static std::vector<int> w;
90     w.resize(n);
91     for (int i = 0; i < n; ++i)
92         if (i < rev[i]) std::swap(f[i], f[rev[i]]);
93     for (int i = 2; i <= n; i <= 1) {
94         w[0] = 1, w[1] = qpow(grt, (mod - 1) / i);
95         if (b == -1) w[1] = qpow(w[1], mod - 2);
96         for (int j = 2; j < i / 2; ++j) w[j] = (ll)w[j - 1] * w[1] %
97 mod;
98         for (int j = 0; j < n; j += i)
99             for (int k = 0; k < i / 2; ++k) {
100                 int p = f[j + k], q = (ll)f[j + k + i / 2] * w[k] % mod;
101                 f[j + k] = (p + q) % mod, f[j + k + i / 2] = (p - q +

```

```

102     mod) % mod;
103     }
104 }
105 }
106
107 poly poly::operator*(const poly &h) const {
108     int N = 1;
109     while (N < (int)(size() + h.size() - 1)) N <= 1;
110     std::vector<int> f(this->data), g(h.data);
111     f.resize(N), g.resize(N);
112     rev.resize(N);
113     for (int i = 0; i < N; ++i)
114         rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? N >> 1 : 0);
115     dft_for_module(f, N, 1), dft_for_module(g, N, 1);
116     for (int i = 0; i < N; ++i) f[i] = (ll)f[i] * g[i] % mod;
117     dft_for_module(f, N, -1), f.resize(size() + h.size() - 1);
118     for (int i = 0, inv = qpow(N, mod - 2); i < (int)f.size();
119 ++i)
120         f[i] = (ll)f[i] * inv % mod;
121     return f;
122 }
123
124 poly poly::operator*=(const poly &h) { return *this = *this * h;
125 }
126
127 poly poly::operator*(const int &h) const {
128     std::vector<int> f(this->data);
129     for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * h %
130 mod;
131     return f;
132 }
133
134 poly poly::operator*=(const int &h) {
135     for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] *
136 h % mod;
137     return *this;
138 }
139
140 poly poly::operator+(const poly &h) const {
141     std::vector<int> f(this->data);
142     if (f.size() < h.size()) f.resize(h.size());
143     for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) %
144 mod;
145     return f;
146 }
147
148 poly poly::operator+=(const poly &h) {
149     std::vector<int> &f = this->data;
150     if (f.size() < h.size()) f.resize(h.size());
151     for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) %
152 mod;
153     return f;

```

```

154 }
155
156 poly poly::operator-(const poly &h) const {
157     std::vector<int> f(this->data);
158     if (f.size() < h.size()) f.resize(h.size());
159     for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] - h[i] +
160 mod) % mod;
161     return f;
162 }
163
164 poly poly::operator--(const poly &h) {
165     std::vector<int> &f = this->data;
166     if (f.size() < h.size()) f.resize(h.size());
167     for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] - h[i] +
168 mod) % mod;
169     return f;
170 }
171
172 poly poly::operator<<(const std::size_t &b) const {
173     std::vector<int> f(size() + b);
174     for (int i = 0; i < (int)size(); ++i) f[i + b] = data[i];
175     return f;
176 }
177
178 poly poly::operator<=(const std::size_t &b) { return *this =
179 (*this) << b; }
180
181 poly poly::operator>>(const std::size_t &b) const {
182     std::vector<int> f(size() - b);
183     for (int i = 0; i < (int)f.size(); ++i) f[i] = data[i + b];
184     return f;
185 }
186
187 poly poly::operator>>=(const std::size_t &b) { return *this =
188 (*this) >> b; }
189
190 poly poly::operator/(const int &h) const {
191     std::vector<int> f(this->data);
192     int inv = qpow(h, mod - 2);
193     for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * inv
194 % mod;
195     return f;
196 }
197
198 poly poly::operator/=(const int &h) {
199     int inv = qpow(h, mod - 2);
200     for (int i = 0; i < (int)data.size(); ++i) data[i] =
201 (ll)data[i] * inv % mod;
202     return *this;
203 }
204
205 poly poly::inv(void) const {

```

```

206     int N = 1;
207     while (N < (int)(size() + size() - 1)) N <= 1;
208     std::vector<int> f(N), g(N), d(this->data);
209     d.resize(N), f[0] = qpow(d[0], mod - 2);
210     for (int w = 2; w < N; w <= 1) {
211         for (int i = 0; i < w; ++i) g[i] = d[i];
212         rev.resize(w < 1);
213         for (int i = 0; i < w * 2; ++i)
214             rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? w : 0);
215         dft_for_module(f, w < 1, 1), dft_for_module(g, w < 1, 1);
216         for (int i = 0; i < w * 2; ++i)
217             f[i] = (ll)f[i] * (2 + mod - (ll)f[i] * g[i] % mod) % mod;
218         dft_for_module(f, w < 1, -1);
219         for (int i = 0, inv = qpow(w < 1, mod - 2); i < w; ++i)
220             f[i] = (ll)f[i] * inv % mod;
221         for (int i = w; i < w * 2; ++i) f[i] = 0;
222     }
223     f.resize(size());
224     return f;
225 }
226
227 poly poly::operator==(const poly &h) const {
228     if (size() != h.size()) return 0;
229     for (int i = 0; i < (int)size(); ++i)
230         if (data[i] != h[i]) return 0;
231     return 1;
232 }
233
234 poly poly::operator!=(const poly &h) const {
235     if (size() != h.size()) return 1;
236     for (int i = 0; i < (int)size(); ++i)
237         if (data[i] != h[i]) return 1;
238     return 0;
239 }
240
241 poly poly::operator+(const int &h) const {
242     poly f(this->data);
243     f[0] = (f[0] + h) % mod;
244     return f;
245 }
246
247 poly poly::operator+=(const int &h) { return *this = (*this) +
248 h; }
249
250 poly poly::inv(const int &h) const {
251     poly f(*this);
252     f.resize(h);
253     return f.inv();
254 }
255
256 int modsqrt(int h, int p = mod) { return 1; }
257

```

```

258 poly sqrt(const poly &h) {
259     int N = 1;
260     while (N < (int)(h.size() + h.size() - 1)) N <= 1;
261     poly f(N), g(N), d(h);
262     d.resize(N), f[0] = modsqrt(d[0]);
263     for (int w = 2; w < N; w <= 1) {
264         g.resize(w);
265         for (int i = 0; i < w; ++i) g[i] = d[i];
266         f = (f + f.inv(w) * g) / 2;
267         f.resize(w);
268     }
269     f.resize(h.size());
270     return f;
271 }
272
273 poly log(const poly &h) {
274     poly f(h);
275     for (int i = 1; i < (int)f.size(); ++i) f[i - 1] = (ll)f[i] *
276 i % mod;
277     f[f.size() - 1] = 0, f = f * h.inv(), f.resize(h.size());
278     for (int i = (int)f.size() - 1; i > 0; --i)
279         f[i] = (ll)f[i - 1] * qpow(i, mod - 2) % mod;
280     f[0] = 0;
281     return f;
282 }
283
284 poly exp(const poly &h) {
285     int N = 1;
286     while (N < (int)(h.size() + h.size() - 1)) N <= 1;
287     poly f(N), g(N), d(h);
288     f[0] = 1, d.resize(N);
289     for (int w = 2; w < N; w <= 1) {
290         f.resize(w), g.resize(w);
291         for (int i = 0; i < w; ++i) g[i] = d[i];
292         f = f * (g + 1 - log(f));
293         f.resize(w);
294     }
295     f.resize(h.size());
296     return f;
297 }
298
299 struct comp {
300     long double x, y;
301
302     comp(long double _x = 0, long double _y = 0) : x(_x), y(_y) {}
303
304     comp operator*(const comp &b) const {
305         return comp(x * b.x - y * b.y, x * b.y + y * b.x);
306     }
307
308     comp operator+(const comp &b) const { return comp(x + b.x, y +
309 b.y); }

```



```

310
311     comp operator-(const comp &b) const { return comp(x - b.x, y -
312 b.y); }
313
314     comp conj(void) { return comp(x, -y); }
315 };
316
317 const int EPS = 1e-9;
318
319 template <typename FLOAT_T>
320 FLOAT_T fabs(const FLOAT_T &x) {
321     return x > 0 ? x : -x;
322 }
323
324 template <typename FLOAT_T>
325 FLOAT_T sin(const FLOAT_T &x, const long double &EPS =
326 fstdlib::EPS) {
327     FLOAT_T res = 0, delt = x;
328     int d = 0;
329     while (fabs(delt) > EPS) {
330         res += delt, ++d;
331         delt *= -x * x / ((2 * d) * (2 * d + 1));
332     }
333     return res;
334 }
335
336 template <typename FLOAT_T>
337 FLOAT_T cos(const FLOAT_T &x, const long double &EPS =
338 fstdlib::EPS) {
339     FLOAT_T res = 0, delt = 1;
340     int d = 0;
341     while (fabs(delt) > EPS) {
342         res += delt, ++d;
343         delt *= -x * x / ((2 * d) * (2 * d - 1));
344     }
345     return res;
346 }
347
348 const long double PI = std::acos((long double)(-1));
349
350 void dft_for_complex(std::vector<comp> &f, int n, int b) {
351     static std::vector<comp> w;
352     w.resize(n);
353     for (int i = 0; i < n; ++i)
354         if (i < rev[i]) std::swap(f[i], f[rev[i]]);
355     for (int i = 2; i <= n; i <= 1) {
356         w[0] = comp(1, 0), w[1] = comp(cos(2 * PI / i), b * sin(2 *
357 PI / i));
358         for (int j = 2; j < i / 2; ++j) w[j] = w[j - 1] * w[1];
359         for (int j = 0; j < n; j += i)
360             for (int k = 0; k < i / 2; ++k) {
361                 comp p = f[j + k], q = f[j + k + i / 2] * w[k];

```

```

362         f[j + k] = p + q, f[j + k + i / 2] = p - q;
363     }
364 }
365 }
366
367 class arbitrary_module_poly {
368 private:
369     std::vector<int> data;
370
371     int construct_element(int D, ll x, ll y, ll z) const {
372         x %= mod, y %= mod, z %= mod;
373         return ((ll)D * D * x % mod + (ll)D * y % mod + z) % mod;
374     }
375
376 public:
377     int mod;
378
379     arbitrary_module_poly(std::size_t len = std::size_t(0),
380                          int module_value = 1e9 + 7) {
381         mod = module_value;
382         data = std::vector<int>(len);
383     }
384
385     arbitrary_module_poly(const std::vector<int> &b, int
386 module_value = 1e9 + 7) {
387         mod = module_value;
388         data = b;
389     }
390
391     arbitrary_module_poly(const arbitrary_module_poly &b) {
392         mod = b.mod;
393         data = b.data;
394     }
395
396     void resize(std::size_t len, const int &val = 0) {
397 data.resize(len, val); }
398
399     std::size_t size(void) const { return data.size(); }
400
401     void clear(void) { data.clear(); }
402 #if __cplusplus >= 201103L
403     void shrink_to_fit(void) { data.shrink_to_fit(); }
404 #endif
405     int &operator[](std::size_t b) { return data[b]; }
406
407     const int &operator[](std::size_t b) const { return data[b]; }
408
409     arbitrary_module_poly operator*(const arbitrary_module_poly
410 &h) const;
411     arbitrary_module_poly operator*=(const arbitrary_module_poly
412 &h);
413     arbitrary_module_poly operator*(const int &h) const;

```

```

414     arbitrary_module_poly operator*=(const int &h);
415     arbitrary_module_poly operator+(const arbitrary_module_poly
416     &h) const;
417     arbitrary_module_poly operator+=(const arbitrary_module_poly
418     &h);
419     arbitrary_module_poly operator-(const arbitrary_module_poly
420     &h) const;
421     arbitrary_module_poly operator-=(const arbitrary_module_poly
422     &h);
423     arbitrary_module_poly operator<<=(const std::size_t &b) const;
424     arbitrary_module_poly operator<<=(const std::size_t &b);
425     arbitrary_module_poly operator>>=(const std::size_t &b) const;
426     arbitrary_module_poly operator>>=(const std::size_t &b);
427     arbitrary_module_poly operator/(const int &h) const;
428     arbitrary_module_poly operator/=(const int &h);
429     arbitrary_module_poly operator==(const arbitrary_module_poly
430     &h) const;
431     arbitrary_module_poly operator!=(const arbitrary_module_poly
432     &h) const;
433     arbitrary_module_poly inv(void) const;
434     arbitrary_module_poly inv(const int &h) const;
435     friend arbitrary_module_poly sqrt(const arbitrary_module_poly
436     &h);
437     friend arbitrary_module_poly log(const arbitrary_module_poly
438     &h);
439 };
440
441 arbitrary_module_poly arbitrary_module_poly::operator*(
442     const arbitrary_module_poly &h) const {
443     int N = 1;
444     while (N < (int)(size() + h.size() - 1)) N <= 1;
445     std::vector<comp> f(N), g(N), p(N), q(N);
446     const int D = std::sqrt(mod);
447     for (int i = 0; i < (int)size(); ++i)
448         f[i].x = data[i] / D, f[i].y = data[i] % D;
449     for (int i = 0; i < (int)h.size(); ++i) g[i].x = h[i] / D,
450     g[i].y = h[i] % D;
451     rev.resize(N);
452     for (int i = 0; i < N; ++i)
453         rev[i] = (rev[i >> 1] >> 1) | (i & 1 ? N >> 1 : 0);
454     dft_for_complex(f, N, 1), dft_for_complex(g, N, 1);
455     for (int i = 0; i < N; ++i) {
456         p[i] = (f[i] + f[(N - i) % N].conj()) * comp(0.50, 0) *
457     g[i];
458         q[i] = (f[i] - f[(N - i) % N].conj()) * comp(0, -0.5) *
459     g[i];
460     }
461     dft_for_complex(p, N, -1), dft_for_complex(q, N, -1);
462     std::vector<int> r(size() + h.size() - 1);
463     for (int i = 0; i < (int)r.size(); ++i)
464         r[i] = construct_element(D, p[i].x / N + 0.5, (p[i].y +
465     q[i].x) / N + 0.5,

```

```

466         q[i].y / N + 0.5);
467     return arbitrary_module_poly(r, mod);
468 }
469
470 arbitrary_module_poly arbitrary_module_poly::operator*=(
471     const arbitrary_module_poly &h) {
472     return *this = *this * h;
473 }
474
475 arbitrary_module_poly arbitrary_module_poly::operator*(const int
476 &h) const {
477     std::vector<int> f(this->data);
478     for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * h %
479 mod;
480     return arbitrary_module_poly(f, mod);
481 }
482
483 arbitrary_module_poly arbitrary_module_poly::operator*=(const
484 int &h) {
485     for (int i = 0; i < (int)size(); ++i) data[i] = (ll)data[i] *
486 h % mod;
487     return *this;
488 }
489
490 arbitrary_module_poly arbitrary_module_poly::operator+(
491     const arbitrary_module_poly &h) const {
492     std::vector<int> f(this->data);
493     if (f.size() < h.size()) f.resize(h.size());
494     for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + h[i]) %
495 mod;
496     return arbitrary_module_poly(f, mod);
497 }
498
499 arbitrary_module_poly arbitrary_module_poly::operator+=(
500     const arbitrary_module_poly &h) {
501     if (size() < h.size()) resize(h.size());
502     for (int i = 0; i < (int)h.size(); ++i) data[i] = (data[i] +
503 h[i]) % mod;
504     return *this;
505 }
506
507 arbitrary_module_poly arbitrary_module_poly::operator-(
508     const arbitrary_module_poly &h) const {
509     std::vector<int> f(this->data);
510     if (f.size() < h.size()) f.resize(h.size());
511     for (int i = 0; i < (int)h.size(); ++i) f[i] = (f[i] + mod -
512 h[i]) % mod;
513     return arbitrary_module_poly(f, mod);
514 }
515
516 arbitrary_module_poly arbitrary_module_poly::operator-=(
517     const arbitrary_module_poly &h) {

```

```

518     if (size() < h.size()) resize(h.size());
519     for (int i = 0; i < (int)h.size(); ++i)
520         data[i] = (data[i] + mod - h[i]) % mod;
521     return *this;
522 }
523
524 arbitrary_module_poly arbitrary_module_poly::operator<<(<
525     const std::size_t &b) const {
526     std::vector<int> f(size() + b);
527     for (int i = 0; i < (int)size(); ++i) f[i + b] = data[i];
528     return arbitrary_module_poly(f, mod);
529 }
530
531 arbitrary_module_poly arbitrary_module_poly::operator<=<(<const
532     std::size_t &b) {
533     return *this = (*this) << b;
534 }
535
536 arbitrary_module_poly arbitrary_module_poly::operator>>(<
537     const std::size_t &b) const {
538     std::vector<int> f(size() - b);
539     for (int i = 0; i < (int)f.size(); ++i) f[i] = data[i + b];
540     return arbitrary_module_poly(f, mod);
541 }
542
543 arbitrary_module_poly arbitrary_module_poly::operator>>=<(<const
544     std::size_t &b) {
545     return *this = (*this) >> b;
546 }
547
548 arbitrary_module_poly arbitrary_module_poly::inv(void) const {
549     int N = 1;
550     while (N < (int)(size() + size() - 1)) N <=< 1;
551     arbitrary_module_poly f(1, mod), g(N, mod), h(*this), f2(1,
552     mod);
553     f[0] = qpow(data[0], mod - 2, mod), h.resize(N), f2[0] = 2;
554     for (int w = 2; w < N; w <=< 1) {
555         g.resize(w);
556         for (int i = 0; i < w; ++i) g[i] = h[i];
557         f = f * (f * g - f2) * (mod - 1);
558         f.resize(w);
559     }
560     f.resize(size());
561     return f;
562 }
563
564 arbitrary_module_poly arbitrary_module_poly::inv(const int &h)
565     const {
566     arbitrary_module_poly f(*this);
567     f.resize(h);
568     return f.inv();
569 }

```

```

570
571 arbitrary_module_poly arbitrary_module_poly::operator/(const int
572 &h) const {
573     int inv = qpow(h, mod - 2, mod);
574     std::vector<int> f(this->data);
575     for (int i = 0; i < (int)f.size(); ++i) f[i] = (ll)f[i] * inv
576 % 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

```

1  int main() {
2      scanf("%d", &n);
3      fact[0] = 1;
4      for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i %
5      mod;
6      exgcd(fact[n], mod, ifact[n], ifact[0]),
7          ifact[n] = (ifact[n] % mod + mod) % mod;
8      for (int i = n - 1; i >= 0; --i) ifact[i] = (ll)ifact[i + 1] *
9      (i + 1) % mod;
10     poly f(n + 1), g(n + 1);
11     for (int i = 0; i <= n; ++i)
12         g[i] = (i & 1 ? mod - 1ll : 1ll) * ifact[i] % mod,
13         f[i] = (ll)qpow(i, n) * ifact[i] % mod;
14     f *= g, f.resize(n + 1);
15     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  $\left\{ \begin{smallmatrix} i \\ k \end{smallmatrix} \right\}$ 。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

$F(x) = \sum_{i=1}^{+\infty} \frac{x^i}{i!} = e^x - 1$ 。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。

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

另外,  $\exp F(x) = \sum_{i=0}^{+\infty} \frac{F^i(x)}{i!}$  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

```
1  int main() {
2      scanf("%d%d", &n, &k);
3      poly f(n + 1);
4      fact[0] = 1;
5      for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i %
6      mod;
7      for (int i = 1; i <= n; ++i) f[i] = qpow(fact[i], mod - 2);
8      f = exp(log(f >> 1) * k) << k, f.resize(n + 1);
9      int inv = qpow(fact[k], mod - 2);
10     for (int i = 0; i <= n; ++i)
11         printf("%lld ", (ll)f[i] * fact[i] % mod * inv % mod);
12     return 0;
13 }
```

## 第一类斯特林数 (Stirling Number)

**第一类斯特林数** (斯特林轮换数)  $\left[ \begin{matrix} n \\ k \end{matrix} \right]$ , 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

$$\left[ \begin{matrix} n \\ k \end{matrix} \right] = \left[ \begin{matrix} n-1 \\ k-1 \end{matrix} \right] + (n-1) \left[ \begin{matrix} n-1 \\ k \end{matrix} \right]$$

边界是  $\left[ \begin{matrix} n \\ 0 \end{matrix} \right] = [n = 0]$ 。

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

```
1  int main() {
2      scanf("%d%d", &n, &k);
3      fact[0] = 1;
4      for (int i = 1; i <= n; ++i) fact[i] = (ll)fact[i - 1] * i %
5      mod;
6      ifact[n] = qpow(fact[n], mod - 2);
7      for (int i = n - 1; i >= 0; --i) ifact[i] = (ll)ifact[i + 1] *
8      (i + 1) % mod;
9      poly f(n + 1);
10     for (int i = 1; i <= n; ++i) f[i] = (ll)fact[i - 1] * ifact[i]
11     % mod;
12     f = exp(log(f >> 1) * k) << k, f.resize(n + 1);
13     for (int i = 0; i <= n; ++i)
14         printf("%lld ", (ll)f[i] * fact[i] % mod * ifact[k] % mod);
15     return 0;
16 }
```

## 应用

### 上升幂与普通幂的相互转化

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

则可以利用下面的恒等式将上升幂转化为普通幂：

$$x^{\bar{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k$$

如果将普通幂转化为上升幂，则有下列的恒等式：

$$x^n = \sum_k \left\{ \begin{matrix} n \\ k \end{matrix} \right\} (-1)^{n-k} x^{\bar{k}}$$

### 下降幂与普通幂的相互转化

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

则可以利用下面的恒等式将普通幂转化为下降幂：

$$x^n = \sum_k \left\{ \begin{matrix} n \\ k \end{matrix} \right\} x^{\underline{k}}$$

如果将下降幂转化为普通幂，则有下列的恒等式：

$$x^{\underline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^k$$

## 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

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

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](#)

🔧 本页面最近更新：2024/10/9 22:38:42，[更新历史](#)

✎ 发现错误？想一起完善？ [在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[Ir1d](#), [Enter-tainer](#), [Fei Natsuka](#), [StudyingFather](#), [Xeonacid](#), [MegaOwler](#), [sshwy](#), [Tiphereth-A](#), [iamtwz](#), [ksyx](#), [caijianhong](#), [CCXXI](#), [Great-designer](#), [H-J-Granger](#), [isdanni](#),

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

© 本页面的全部内容在 **CC BY-SA 4.0** 和 **SATA** 协议之条款下提供，附加条款亦可能应用