Генерация комбинаторных объектов

В данной лекции начнем знакомство с генерацией (порождением) комбинаторных объектов (множеств и последовательностей), структура которых определена в каждой конкретной задаче. Количество объектов зависит от их размера, как правило, экспоненциально, поэтому алгоритмы перебора, т.е. порождения всех или всех допустимых в задаче объектов обычно имеют ***экспоненциальную сложность***.

Все рассматриваемые методы систематического порождения комбинаторных объектов будут сводиться к выбору ***начальной конфигурации***, задающей первый генерируемый объект, трансформации полученного объекта в следующий и проверке условия окончания, которое определяет момент прекращения вычислений. При этом особый интерес будут представлять *алгоритмы генерации объектов в порядке минимального изменения,* когда два "соседних" порождаемых объекта различаются в подходящем смысле "минимально".

Нас прежде всего будет интересовать время работы алгоритма, требующееся для порождения всего класса объектов, как функция от размерности входных данных. Мы будем стремиться получить асимптотически наилучший алгоритм генерации. В частности, в некоторых алгоритмах можно порождать множество всех требуемых объектов за время, пропорциональное его мощности. В этом случае алгоритм имеет сложность ***0(k)***, где - ***k*** число порождаемых объектов. Такой *алгоритм генерации* комбинаторных объектов в литературе часто называют *линейным.*

Схема перебора всегда будет одинакова:

во-первых, надо установить ПОРЯДОК на элементах, подлежащих перечислению (в частности, определить, какой из них будет первым, а какой последним);

во-вторых, научиться переходить от произвольного элемента к непосредственно следующему за ним, данное требование очень важно, т.к. возвращаться к пропущенным в процессе перебора элементам будет сложнее, чем задать алгоритм сплошного перебора.

Наиболее естественным способом упорядочения составных объектов является ***лексикографический порядок***, принятый в любом словаре (сначала сравниваются первые буквы слов, потом вторые и т.д.) - именно его мы и будем чаще всего использовать. А вот процедуру получения следующего элемента придется каждый раз изобретать заново.

Две последовательности **a**=$a_1, a_2, \ldots, a_n$ и **b**=$b_1, b_2, \ldots, b_n$ упорядочены ***лексикографически***, если последовательность **a** предшествует последовательности **b**, т.е. для некоторого s их начальные отрезки длины s равны, а $(s+1)$-ый член последовательности a меньше.

Пример

a=(4,2,3,1,5,6), **b**=(4,2,3,5,1,6).

Здесь $s=3$ (первые три элемента совпадают), а четвертый элемент последовательности **a** меньше элемента последовательности **b** (5>1).

Генерация всех перестановок в лексикографическом порядке

Перестановки (без повторений) – это последовательности, которые содержат все элементы одного и того же множества по одному разу, но отличаются порядком.

Обозначим множество всех перестановок n – элементного множества через S_n.

Пример. Для последовательности **a,b,c** существуют следующие перестановки, перечисленные в лексикографическом порядке:

S_3={a,b,c; a,c,b; b,a,c; b,c,a; c,a,b; c,b,a}.

Пример. Задача о коммивояжере. Классическая формулировка задачи известна уже более 200 лет: имеются n городов, расстояния между которыми заданы; коммивояжеру необходимо выйти из какого-либо города, посетить остальные n-1 городов точно по одному разу и вернуться в исходный город. При этом маршрут коммивояжера должен быть минимальной длины (стоимости). Написать программу решения задачи.

Количество перестановок элементов n-элементного множества S_n определяется формулой $P_n = n! = 1*2*3*\ldots*n$.

Действительно, на первое место может быть помещен любой из N элементов (всего вариантов N), на вторую позицию (N-1) элементов, итого вариантов N*(N-1), и если продолжить данную последовательность для всех мест, то получим: N*(N-1)*(N-2)* ... *1; Для рассмотренного примера n=3 и P_n=3!=6.

Задача. Напечатать все перестановки чисел $a_1,a_2,...,a_n$ (то есть последовательности длины n, в которые каждое из этих чисел a_i, i=1,2,…,n входит по одному разу.

Очевидно, что вместо перестановки чисел $a_1,a_2,...,a_n$ можно рассмотреть все перестановки чисел 1,2,…,n, которые соответствуют индексам последовательности $a_1,a_2,...,a_n$, т.е. вместо исходного массива будем использовать массив p_1, $p_2,...,p_n$, где p_i=i, i=1,2,…,n.

Алгоритм 1.

1. От конца к началу перестановки ищем первый элемент a_i такой, что $p_i < p_{i+1}$.
Запоминаем его индекс в ***r***.
2. От элемента ***r+1*** до конца ищем последний элемент, больший, чем p_r, и запоминаем его индекс – ***k***.
3. Меняем местами элементы с номерами r и k.
4. Теперь в части массива, которая размещена справа от позиции r (после обмена) надо отсортировать все числа в порядке возрастания. Благодаря тому, что до этого они все были уже записаны в порядке убывания необходимо данную подпоследовательность просто перевернуть. Всю группу элементов от (r+1)-го до n-го попарно меняем местами: (r+1)-й элемент с n-м, (r+2)-й с (n-1)-м и т.д.

Пример. Предположим, что необходимо получить все перестановки чисел 1,2,3,4,5,6 в лексикографическом порядке, первой перестановкой будет (1,2,3,4,5,6), а последней (6,5,4,3,2,1). Предположим, что на некотором шаге работы была получена перестановка P = (3,4,6,5,2,1). Здесь 4<6 и первым элементом большим чем 4 справа является 5. Меняем местами 4 и 5, имеем (3,5,6,4,2,1). Теперь все элементы стоящие за 5, записываем в обратном порядке P=(3,5,1,2,4,6). Которая будет рассматриваться при генерации следующей перестановки.

Алгоритм 1. PLex(n)

```
#include <iostream>
using namespace std;
typedef int vec[20];
vec p; int n;
void print()
{
 for (int i=1;i<=n;i++)
cout<<p[i]<<" ";
 cout<<endl;
}
void swap(int &a, int &b){
 int r;
 r=a; a=b; b=r;
}
```

```
void main() {
      int i, j, k;
  cin>>n;
 for (i=1;i<=n;i++) p[i]=i;
  print();
  do {
     for (i=n-1; i>=1, p[i] > p[i + 1]; i--);
        if (i==0) break;
        for (j=n; p[j]<p[i]; j--);
        swap(p[i], p[j]);
        for (k = 1; k<=(n - i) / 2; k++ )
        swap(p[i + k], p[n + 1 - k]);
        print();
  } while (true);
}
```

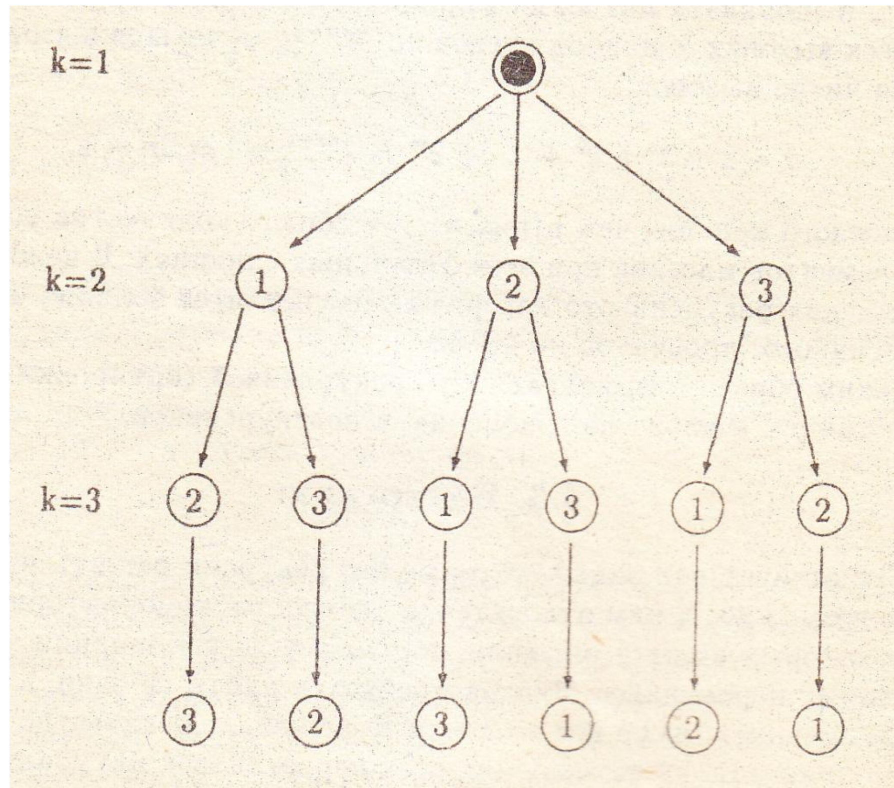
Рекурсивный алгоритм.

1. В качестве первого выбираем любое число из чисел 1, 2, …, n;
2. В качеств второго числа выбираем любое из чисел 1, 2, …, n, кроме того числа, которое выбрано первым;
3. Третьим числом выбираем одно из чисел, которое не выбрано первым или вторым;

и т.д.

Этот процесс продолжаем до тех пор, пока не выберем последнее, n-ое число для перестановки.

Чтобы получить все возможные перестановки, надо на каждом этапе выбора k-го числа последовательно перебирать все допустимые числа, однако перейти к следующему варианту можно, только если найдены полные перестановки для всех предыдущих вариантов. Такой процесс показан в схеме:



Схем
а

В соответствии со схемой вначале будет выбрана перестановка 1 2 3, затем 1 3 2 и т.д.

При выборе очередного числа движение по схеме идёт по стрелке вниз, а при отказе от этого выбора – обратное движение (бэктрекинг). Этот процесс легко реализовать рекурсивным алгоритмом.

Алгоритм 2. Рекурсия	

Алгоритм 2. Рекурсия

```
#include <iostream>
using namespace std;
typedef int vec[20];
vec p, r; int n;
void print()
{
 for (int i=1;i<=n;i++)
cout<<p[i]<<" ";
 cout<<endl;
}
```

```
void per(int k){
 int i;
 for (int i=1;i<=n;i++) {
       if (r[i]==0) {
      r[i]=1; p[k]=i;  if (k==n)
print(); else per(k+1);
      r[i]=0;
       }
 }
}
void main() {
      memset(r,0,sizeof(r));
      cin>>n;
      per(1);
}
```

Время выполнения алгоритма определяется, прежде всего, количеством сгенерированных перестановок $O(n!)$. Например, $10!=3628800$.

Алгоритм Джонсона — Троттера генерации перестановок

Мы рассмотрели несколько алгоритмов генерации перестановок. При этом переход от предыдущей перестановки к следующей требовал, вообще говоря, большого числа перемещений элементов исходной пе рестановки. Так, в лучшем из рассмотренных алгоритмов PLex(n) мы выделяли два элемента, меняли их местами, а затем переворачивали конечный отрезок перестановки. Поэтому естественно желание получить алгоритм генерации, в котором соседние перестановки будут различаться настолько мало, насколько это возможно. Для того, чтобы такое различие было минимально возможным, любая генерируемая перестановка должна отличаться от предшествующей транспозицией двух соседних элементов. Возможно ли таким образом породить все перестановки без повторений? Оказывается, такую последовательность перестановок

При n = 1 последовательность из единственной перестановки (1) будет требуемой. Предположим, что имеется последовательность $\sigma_i^1, \dots, \sigma_i^i, .., \sigma_i^{(n-1)!}$ всех перестановок из S_{n-1} – такая, что каждая следующая σ^{i+1} получается из предыдущей σ^i перестановкой двух соседних элементов. Построим последовательность $\pi^i, i = 1,2, \dots, n!$ всех перестановок из S_n с таким же свойством. Расширим каждую перестановку $\sigma^i = (\sigma_1^i, \sigma_2^i, .., \sigma_{n-1}^i)$, последовательно вставляя элемент ***n*** на каждое из ***n*** возможных мест: перед элементом σ_1^i, между элементами σ_j^i и σ_{j+1}^i после элемента σ_{n-1}^i. При этом элемент *n* вставляем в σ^1 в направлении справа налево, а в σ^2 — слева направо:

$$(\sigma_1^1, \sigma_2^1, .., \sigma_{n-1}^1, n), \dots, (n, \sigma_1^1, \sigma_2^1, .., \sigma_{n-1}^1),$$
$$(n, \sigma_1^2, \sigma_2^2, .., \sigma_{n-1}^2), \dots, (\sigma_1^2, \sigma_2^2, .., \sigma_{n-1}^2, n)$$

и т. д., при переходе от σ^i к σ^{i+1} меняем направление вставки элемента *n* на обратное. Тогда внутри i-го *блока (n* перестановок из S_n, построенных из $\sigma^i \in S_{n-1}$) каждая следующая перестановка получается из предыдущей перестановкой двух соседних элементов, один из которых есть ***n***.

При переходе от последней перестановки в *i*-м блоке к первой в следующем (i + 1)-м блоке по индукционному предположению также переставляются только два элемента. Таким образом, для построенной последовательности перестановок $\pi^i \in S_n, i = 1, \ldots n!$ выполняется требуемое свойство.

Описанный рекурсивный алгоритм, генерирующий последовательность перестановок из S_n применённый непосредственно, имеет огромный недостаток: последовательность перестановок строится "целиком" и требует хранения всех перестановок из S_{n-1} , S_{n-2},… . Очевидно, такой алгоритм использовал бы огромный объём памяти, поэтому он неприменим.

<table>
<tr><td rowspan="24">(1)</td><td rowspan="12">(12)</td><td rowspan="4">(123)</td><td>(1234)</td></tr>
<tr><td>(1243)</td></tr>
<tr><td>(1423)</td></tr>
<tr><td>(4123)</td></tr>
<tr><td rowspan="4">(132)</td><td>(4132)</td></tr>
<tr><td>(1432)</td></tr>
<tr><td>(1342)</td></tr>
<tr><td>(1324)</td></tr>
<tr><td rowspan="4">(312)</td><td>(3124)</td></tr>
<tr><td>(3142)</td></tr>
<tr><td>(3412)</td></tr>
<tr><td>(4312)</td></tr>
<tr><td rowspan="12">(21)</td><td rowspan="4">(321)</td><td>(4321)</td></tr>
<tr><td>(3421)</td></tr>
<tr><td>(3241)</td></tr>
<tr><td>(3214)</td></tr>
<tr><td rowspan="4">(231)</td><td>(2314)</td></tr>
<tr><td>(2341)</td></tr>
<tr><td>(2431)</td></tr>
<tr><td>(4231)</td></tr>
<tr><td rowspan="4">(213)</td><td>(4213)</td></tr>
<tr><td>(2413)</td></tr>
<tr><td>(2143)</td></tr>
<tr><td>(2134)</td></tr>
</table>

Как по номеру определить перестановку относительно того порядка, разумеется, который введен на множестве перестановок? Остановимся на лексикографическом порядке. Нумерация начинается с 0.

Рассмотрим идею решения на примере. Пусть n=8 и дан номер L, равный 37021. Найдем соответствующую перестановку. Пусть на первом месте записана единица. Таких перестановок 7!, или 5040 (1*******). При 2 тоже 5040 (2*******). Итак, 37021 / 5040=7. Следовательно, первая цифра в перестановке 8. Новое значение L = 1741 (37021 % 5040=1741). Продолжим рассуждения. Оформим, как обычно, их в виде таблицы

i	L	$P_{старая}$	P	L / P	L % P	$P_{новая}$
1	37021	********	7!=5040	7	1741	8*******
2	1741	8*******	6!=720	2	301	83******
3	301	83******	5!=120	2	61	834*****
4	61	834*****	4!=24	2	13	8345****
5	13	8345****	3!=6	2	1	83456***
6	1	83456***	2!=2	0	1	834561**
7	1	834561**	1!=1	1	0	8345617*
8	0	8345617*	0!=1	0	0	83456172

Обратим внимание на третью строку, в которой на третье место записывается цифра 4. То, что записываются не цифры 1 и 2, очевидно: их требуется пропустить. Цифра три «занята», поэтому записываем 4. Точно также в строках 4, 5 и 7.

По перестановке получить ее номер, не выполняя генерацию множества перестановок. Начнем с примера.

Пусть n=8 и дана перестановка 5 3 8 7 1 4 6 2.

Схема:

7!*<количество цифр в перестановке на 1-м месте, идущих до цифры 5, с учетом занятых цифр – ответ 4>

6!*< количество цифр в перестановке на 2-м месте, идущих до цифры 3, с учетом занятых цифр – ответ 2>

5!*< количество цифр в перестановке на 3-м месте, идущих до цифры 8, с учетом занятых цифр – ответ 5>

4!*< количество цифр в перестановке на 4-м месте, идущих до цифры 7, с учетом занятых цифр – ответ 4>

3!*< количество цифр в перестановке на 5-м месте, идущих до цифры 1, с учетом занятых цифр – ответ 0>

2!*< количество цифр в перестановке на 6-м месте, идущих до цифры 4, с учетом занятых цифр – ответ 1>

1!*< количество цифр в перестановке на 7-м месте, идущих до цифры 6, с учетом занятых цифр – ответ 1>

Итак,

7!*4+6!*2+5!*5+4!*4+3!*0+2!*1+1!*1=5040*4+720*2+120*5+24*4+6*0+2*1+1*1 = 22299.