ФУНДАМЕНТАЛЬНЫЕ АЛГОРИТМЫ

АЛГОРИТМЫ ГЕНЕРАЦИИ ПЕРЕСТАНОВОК

ПЛАН ЛЕКЦИИ

1. Теоретические сведения
2. Алгоритм определения знаков перестановок
3. Антилексикографический алгоритм генерации перестановок
4. Генерирование перестановок с минимальным числом транспозиций
5. Генерирование перестановок с минимальным числом транспозиций соседних элементов
6. Тестовые последовательности
7. Генерирование всех подмножеств множества
8. Генерирование *k*-элементных подмножеств множества из *n* элементов
9. Генерирование разбиений множества

1. Теоретические сведения (1)

Перестановкой *n*-элементного множества *X* назовём взаимно однозначную функцию отображения $f: X \to X$. Обычно задаётся в виде таблицы(матрицы).

Пример. $$f(a) = d;\ f(b) = a;\ f(c) = c;\ f(d) = b;\ \Rightarrow \begin{pmatrix} a & b & c & d \\ d & a & c & b \end{pmatrix}$$

Определим суперпозицию функций перестановок следующим образом

$$g = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 3 & 1 & 4 \end{pmatrix};$$

$$f = \begin{pmatrix} 2 & 5 & 3 & 1 & 4 \\ 3 & 4 & 2 & 5 & 1 \end{pmatrix};$$

$$fg = fg(i) = f[g(i)] = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 2 & 5 & 1 \end{pmatrix}$$

Определим ***тождественную перестановку:*** $$e = \begin{pmatrix} 1 & 2 & 3 & \dots & \dots & n \\ 1 & 2 & 3 & \dots & \dots & n \end{pmatrix}; ef = fe = f$$

1. Теоретические сведения (2)

Операция обратной перестановки есть

$$f^{-1} f = f\ f^{-1} = e;$$

Пример: $$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 2 & 1 & 5 \end{pmatrix}; f^{-1} = \begin{pmatrix} 3 & 4 & 2 & 1 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 1 & 2 & 5 \end{pmatrix}$$

Для произвольных перестановок $f, g, h \in S_n$ выполняются условия:

$(fg)h = f(gh);$

$ef = fe = f;$ где S_n – множество всех перестановок.

$f^{-1} f = f\ f^{-1} = e;$

S_n образует симметрическую группу степени n относительно операции суперпозиции со свойствами

$$G \subseteq S_n : \begin{cases} f, g \in G \Rightarrow fg \in G, \\ f \in G \Rightarrow f^{-1} \in G. \end{cases}$$

1. Теоретические сведения (3)

Каждую перестановку можно представить в виде ориентированного графа с множеством вершин мощностью n, в котором $x \to y$ когда $f(x) = y$. Исходит дуга $<x, f(x)>$, а входит $<f^{-1}(x), x>$. Возможно появление замкнутых циклов

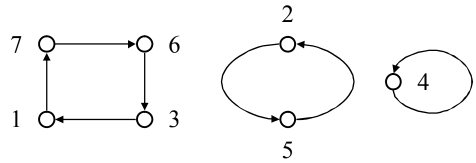
$$a_0^{(i)} \to a_1^{(i)} \to a_2^{(i)} \to \ldots \to a_{n_{i-1}}^{(i)} \to a_0^{(i)}; i = 1,2,\ldots,k.$$

Вся перестановка представима совокупностью циклов

$$f = \left(a_0^{(1)} \quad a_1^{(1)} \quad \ldots \quad a_{n_{1-1}}^{(1)}\right)\left(a_0^{(2)} \quad a_1^{(2)} \quad \ldots \quad a_{n_{2-1}}^{(2)}\right) \ldots \left(a_0^{(k)} \quad a_1^{(k)} \quad \ldots \quad a_{n_{k-1}}^{(k)}\right)$$

Пример для случая

$$f = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 5 & 1 & 4 & 2 & 3 & 6 \end{pmatrix}$$



1. Теоретические сведения (4)

Перестановка содержит циклы [1 7 6 3][2 5][4] тип перестановки $1^1\, 2^1\, 4^1$.

Пара $<a_i, a_j>$, $i < j$ называется *инверсией перестановки* $< a_i, \ldots, a_n >$ если $a_i > a_j$.

Знак перестановки есть $\mathrm{sgn}(f) = (-1)^{I(f)}$, где $I(f)$ – число инверсий.

Перестановка является *чётной*, если $\mathrm{sgn}(f) = 1$, или нечётной если $\mathrm{sgn}(f) = -1$.

Перестановка <2, 4, 3, 5, 1> имеет инверсии: <2, 1>, <4, 3>, <4, 1>, <3, 1>, <5, 1>, поэтому нечётная.

Произвольную подстановку с длиной цикла 2 назовём *транспозицией*.

Лемма 1. Произвольную перестановку $f \in S_n$ можно представить в виде суперпозиции $I(f)$ транспозиций соседних элементов.

Лемма 2. Каждая транспозиция есть нечётная перестановка. Знак произвольного цикла длины k равен $(-1)^{k-1}$.

Лемма 3. Знак произвольной перестановки f типа $1^{\lambda_1} \ldots n^{\lambda_n}$ определяется формулой

$$\mathrm{sgn}(f) = (-1)^{\sum_{j=1}^{\left[\frac{n}{2}\right]} \lambda_{2j}}$$

2. Алгоритм определения знаков перестановок

Вход: произвольная перестановка $f \in S_n$, $p[1], \ldots, p[n]$, $p[i]=f[i]$.

Выход: $z = \mathrm{sgn}(f)$

```
Begin
 z :=1;
 for i := 1 to n do Nnew[i] := true;
 for i := 1 to n do (* оператор *)
    if Nnew[i] then (*Найден цикл, содержащий  i*)
     Begin j := p[i]; (* блок *)
           while j != i do
            Begin Nnew[j] := false; z := – z; j := p[j]
            End
     End
End
```

Оператор обеспечивает суммарное число шагов $O(n)$, и блок имеет суммарное число шагов, равное длине всех циклов, то есть тоже $O(n)$.

3. Антилексикографический алгоритм генерации (1)

Лексикографический порядок на множестве $X = \{1, \ldots, n\}$ определяется как $\langle x_1, x_2, \ldots, x_n\rangle < \langle y_1, y_2, \ldots, y_n\rangle \Leftrightarrow$ если $\exists\, k \geq 1$, такое что $y_k \geq x_k$ для каждого $x_l = y_l\;\; l < k$.

Пример $X = \{1, 2, 3\}$

Лексикографический порядок(1)	Антилексикографический порядок (2)
1 2 3	1 2 **3**
1 3 2	2 1 **3**
2 1 3	1 3 **2**
2 3 1	3 1 **2**
3 1 2	2 3 **1**
3 2 1	3 2 **1**

1. В первой перестановке – последовательность возрастающая, во второй – убывающая, первая постановка – обращение первой.
2. Последовательность делится на n блоков, длиной $(n - 1)!$, соответствующих убывающим элементам в последней позиции.

3. Антилексикографический алгоритм генерации (2)

Вход: *n*.
Выход: Последовательность перестановок множества [1, …,n]

```
procedure REVERS(m); (* массив p – глобальный *)
Begin i := 1; j := m;
   while i < j do
     Begin
        pom := p[i] ; p[i] := p[j] ; p[j] := pom;
        i := i + 1; j := j – 1
     End
End (* REVERS*)
procedure ANTYLEX(m);
Begin
   if m = 1 then (* получена новая перестановка *)
     (* фиксация перестановки, вывод или запись *)
   else for i := 1 to m do
        Begin ANTYLEX(m – 1);
```

3. Антилексикографический алгоритм генерации (3)

```
if i < m then
      Begin
        pom := p[i] ; p[i] := p[m] ; p[m] := pom;
        REVERS(m – 1)
      End
End;
Begin
for i := 1 to n do
                    p[i] := i;
ANTYLEX(n)
End
```

4. Генерирование за минимальное число транспозиций (1)

Вход: *n*.
Выход: Последовательность перестановок множества [1, …,n]

```
procedure PERM(m);(*  Массив p глобальный *)
Begin
    if m = 1 then (* получена новая перестановка *)
      (* фиксация перестановки, вывод или запись *)
    else
      for i := 1 to m do
        Begin
         PERM(m – 1);
         if m > i then
           Begin
           pom := p[B(m,i)] ; p[B(m,i)] := p[m] ; p[m] := pom
           End
         End
End (* PERM *)
```

4. Генерирование за минимальное число транспозиций (2)

```
procedure B(m, i); (*  расчёт адреса перестановки *)
Begin
    if ((m mod 2) = 1) and (m > 2) then
       if i < m – 1 then B := i
       else B := m – 2
    else B := m – 1
End (*  B*)

 Begin
for i := 1 to n do
             p[i] := i;
PERM(n)
End
```


Johnson S.M. Generation of permutations by adjacent transpositions. Math. Comp., 1963, 17, s. 282 – 285.

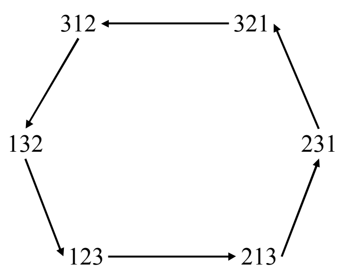
Trotter H.F. Perm (Algorithm 115), Comm. ACM, 1962, 5, s. 434 – 435.

5. Генерирование с минимальным числом транспозиций соседних элементов (1)

Идея: Пуст имеем перестановки {[2,3], [3, 2]}, $n = 3$.

Вставка 1(единицы)

1	2	3
2	**1**	3
2	3	**1**
3	2	**1**
3	**1**	2
1	3	2



1. Единица перемещается между первой и последней позициями взад и вперёд $(n – 1)!$ раз.
2. Каждая новая перестановка образуется транспозицией *самого меньшего* из элементов, который *не находится в граничном положении*, то есть имеет левого и правого соседа.

5. Генерирование с минимальным числом транспозиций соседних элементов (2)

Вход: *n*.
Выход: Последовательность перестановок множества [1, …,n]

```
Begin
for i := 1 to n do
  Begin
   p[i] := i;
   c[i] := 1;
   pr[i] := true (* Признак переноса – вперёд (истина) или назад (ложь) *)
  end
c[n] := 0; (*  Для того, чтобы c[i] ≠ n – i + 1для i = n в цикле while c[i] … *)
(* фиксация перестановки, вывод или запись *)
i := 1;
while i < n do
  Begin i := 1; x := 0;
   while c[i] = n – i + 1 do
      Begin
        pr[i] := not pr[i];
        c[i] := 1;
```

5. Генерирование с минимальным числом транспозиций соседних элементов (3)

```
      if pr[i] then x := x + 1;
      i := i +1
      end
      if i < n then
           Begin (* выполняется транспозиция *)
            if pr[i] then k := x + c[i]
                   else k := n + x + 1 – c[i] – i;
           pom := p[k + 1] ; p[k + 1] := p[k] ; p[k] := pom;
 (* фиксация перестановки, вывод или запись *)
           end
  end
end
```

6. Тестовые последовательности

01	**1 2 3 4**	**1 2 3 4**	**1 2 3 4**
02	**2 1 3 4**	**2 1 3 4**	**2 1 3 4**
03	**1 3 2 4**	**2 3 1 4**	**2 3 1 4**
04	**3 1 2 4**	**3 2 1 4**	**2 3 4 1**
05	**2 3 1 4**	**3 1 2 4**	**3 2 4 1**
06	**3 2 1 4**	**1 3 2 4**	**3 2 1 4**
07	**1 2 4 3**	**4 3 2 1**	**3 1 2 4**
08	**2 1 4 3**	**3 4 2 1**	**1 3 2 4**
09	**1 4 2 3**	**3 2 4 1**	**1 3 4 2**
10	**4 1 2 3**	**2 3 4 1**	**3 1 4 2**
11	**2 4 1 3**	**2 4 3 1**	**3 4 1 2**
12	**4 2 1 3**	**4 2 3 1**	**3 4 2 1**
13	**1 3 4 2**	**4 1 3 2**	**4 3 2 1**
14	**3 1 4 2**	**1 4 3 2**	**4 3 1 2**
15	**1 4 3 2**	**1 3 4 2**	**4 1 3 2**
16	**4 1 3 2**	**3 1 4 2**	**1 4 3 2**
17	**3 4 1 2**	**3 4 1 2**	**1 4 2 3**
18	**4 3 1 2**	**4 3 1 2**	**4 1 2 3**
19	**2 3 4 1**	**4 2 1 3**	**4 2 1 3**
20	**3 2 4 1**	**2 4 1 3**	**4 2 3 1**
21	**2 4 3 1**	**2 1 4 3**	**2 4 3 1**
22	**4 2 3 1**	**1 2 4 3**	**2 4 1 3**
23	**3 4 2 1**	**1 4 2 3**	**2 1 4 3**
24	**4 3 2 1**	**4 1 2 3**	**1 2 4 3**

7. Генерирование всех подмножеств множества (1)

Каждое n-элементное множество $X = \{x_1, x_2, \ldots, x_n\}$ имеет ровно 2^n подмножеств.

$$Y \subseteq X, \qquad b_i = \begin{cases} 0, & y_i \notin Y, \\ 1, & y_i \in Y. \end{cases}$$

Перестановка соответствует двоичному числу $0 \le r \le 2^{n-1} \equiv r = \sum_{i=0}^{n-1} b_i \cdot 2^i$

Данный подход особенно эффективен для реализации в машинных кодах.

Наблюдение.

Если последовательность

$$C_1, C_2, \ldots, C_m$$

содержит все $m = 2^k$ последовательности длины k, а C_i отличается от C_{i+1} в точности по одной координате ($i = 1,\ldots,m-1$), то последовательность

$$C_10, C_20, \ldots, C_m0, C_m1, C_{m-1}1, \ldots, C_11$$

содержит все бинарные последовательности длины $k + 1$, отличающиеся по одной координате. $C_1, C_2, \ldots, C_{2 \cdot n}$ называется ***бинарным кодом Грея***.

7. Генерирование всех подмножеств множества (2)

Вход: *n*

Выход: Подмножества, каждая в бинарном представлении *B*[1], …, *B*[*n*].

```
Begin
for i := 1 to n do B[i] := 0; (* пустое множество *)
i := 0; (* число до сих пор сгенерированных подмножеств *)
repeat
   (* фиксация перестановки, вывод или запись B *)
     i := i +1; p := 1; j := i;
     while j mod 2 = 0 do
        Begin (* J·2^(p − 1) = i *)
          j := j / 2; p := p +1;
        end; (* следующий элемент *)
      if p <= n then B[p] := 1 – B[p]; (* смена позиции p *)
until p > n
end
```

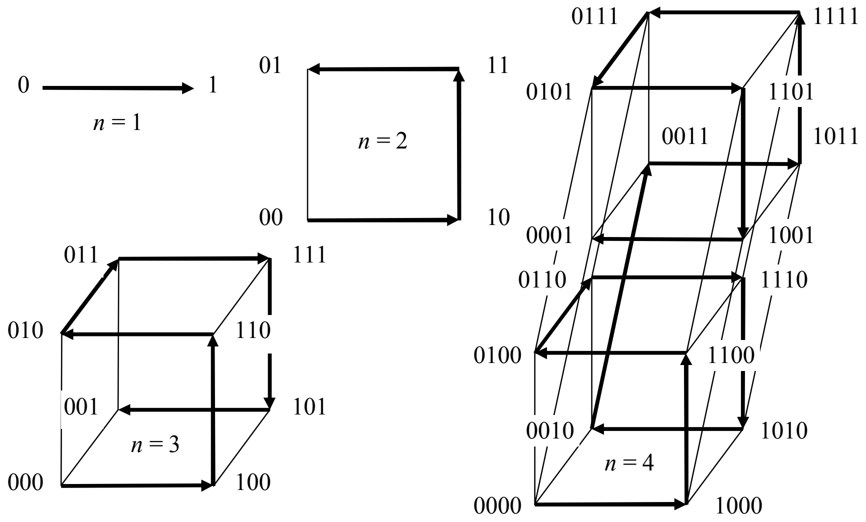
7. Генерирование всех подмножеств множества (3)

ГЕНЕРИРУЕМАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ ($n = 4$)

0 0 0 0
1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 0
1 1 1 0
1 0 1 0
0 0 1 0
0 0 1 1
1 0 1 1
1 1 1 1
0 1 1 1
0 1 0 1
1 1 0 1
1 0 0 1
0 0 0 1

7. Генерирование всех подмножеств множества (4)

ДВОИЧНЫЕ n-МЕРНЫЕ КУБЫ



8. Генерирование k-элементных подмножеств множества из n элементов (1)

Пусть множество из n элементов представлено как $X = \{1, 2, \dots, n\}$.

При лексикографическом порядке за последовательностью $\langle a_1, \dots, a_k \rangle$ непосредственно следует последовательность $\langle b_1, \dots, b_k \rangle = \langle a_1, \dots, a_{p-1}, a_p + 1, a_p + 2, \dots, a_p + k - p + 1 \rangle$, где $p = \max\{i\colon a_i < n - k + 1\}$.

А за последовательностью $\langle b_1, \dots, b_k \rangle$ непосредственно следует $\langle c_1, \dots, c_k \rangle = \langle b_1, \dots, b_{q-1}, b_q + 1, b_q + 2, \dots, b_q + k - q + 1 \rangle$, в котором

$$q = \begin{cases} p-1, & b_k = n, \\ k, & b_k < n. \end{cases}$$

Предположим, что $\langle a_1, \dots, a_k \rangle$ и $\langle b_1, \dots, b_k \rangle$ отличаются от последней последовательности $\langle n - k - p + 1, \dots, n \rangle$

8. Генерирование k-элементных подмножеств множества из n элементов (2)

Вход: n, k

Выход: набор k-элементных подмножеств в количестве C_n^m.

```
Begin
 for i := 1 to k do A[i] := i; (* 1-е подмножество *)
 p := k;
 while p <= 1 do
  Begin
  (* Фиксация набора, запись массива A  *)
    if A[k] = n then p := p – 1 else p := k;
    if p >= k downto p do A[i] := A[p] + i – p + 1
  end
end
```

Для $n = 6$, $k = 4$

1 2 3 4
1 2 3 5
1 2 3 6
1 2 4 5
1 2 4 6
1 2 5 6
1 3 4 5
1 3 4 6
1 4 5 6
2 3 4 5
2 3 4 6
2 3 5 6
2 4 5 6
4 3 5 6

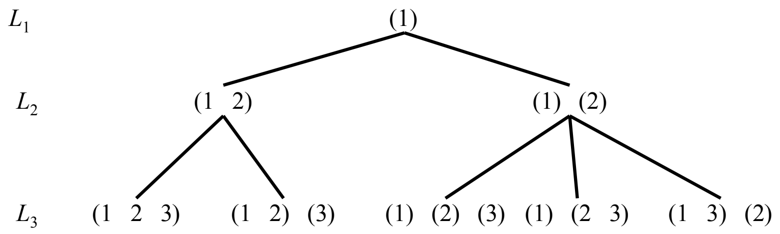
9. Генерирование разбиений множества (1)

Каждое разбиение π множества $\{1, 2, \dots, n\}$ однозначно определяет разбиение π_{n-1} множества $\{1, 2, \dots, n-1\}$ путём удаления n из π. Если дано разбиение $\Omega = \{B_1, B_2, \dots, B_k\}$ множества $\{1, 2, \dots, n-1\}$, то найдутся все разбиения π множества $\{1, 2, \dots, n\}$, такие что $\pi_{n-1} = \Omega$:

$$\begin{gathered} B_1 \cup \{n\}, B_2, \dots, B_k; \\ B_1, B_2 \cup \{n\}, \dots, B_k; \\ \dots \\ B_1, B_2, \dots, B_k \cup \{n\}; \\ B_1, B_2, \dots, B_k, \{n\}. \end{gathered} \qquad (*)$$

Если дан список L_{n-1} всех разбиений $\{1, 2, \dots, n-1\}$, то список L_n всех разбиений $\{1, 2, \dots, n\}$ будем создавать, заменяя каждое разбиение Ω в списке L_{n-1} на последовательность (*).

9. Генерирование разбиений множества (2)



Построение списков для множества {1, 2, 3}

Для работы алгоритма понадобятся следующие переменные-массивы $1 \le i \le n$

PRED[*i*] – содержит номер предыдущего блока по отношению к блоку *i*.

SLED[*i*] – содержит номер следующего блока, равен 0, если блок *i* последний.

BLOK[*i*] – номер блока, содержащего элемент *i*.

FORV[*i*] – направление движения, равен true, если *i* движется вперёд.

9. Генерирование разбиений множества (3)

Вход: *n*

Выход: последовательностей всех разбиений множества

```
Begin
for i := 1 to n do (* поместить i  в первый блок*)
  Begin BLOK[i] := 1; FORV[i] := true;
  end;
SLED[1] := 0;
(* Фиксация разбиения *)
j := n; (* j = активный элемент *)
while j > 1 do
    Begin k := BLOK[j];
      if FORV[j] then (* j  движется вперёд*)
        Begin
          if SLED[k] = 0 then (* k ест последний блок *)
            Begin SLED[k] := j; PRED[j] := k; SLED[j] := 0
            end;
```

9. Генерирование разбиений множества (3)

```
        if SLED[k] > j then (* j  образует новый блок*)
          Begin PRED[j] := k; SLED[j] := SLED[k];
                PRED[SLED[j]] := j; SLED[k] := j
          end;
        BLOK[j] := SLED[k]
      end
    else (* j  движется назад *)
      Begin BLOK[j] := PRED[k];
        if k = j then (* j  образует одноэлементный блок *)
         if SLED[k] = 0 then SLED[PRED[k]] := 0
          else Begin SLED[PRED[k]] := SLED[k];
                     PRED[SLED[k]] := PRED[k]
               end
      end
      (* Фиксация разбиения *)
```

9. Генерирование разбиений множества (4)

```
    j := n;
    while (j > 1) and ((FORV[j] and (BLOK[j] = j) or
                 (not FORV[j] and (BLOK[j] = 1))) do
        Begin
         FORV[j] := not FORV[j];
         j := j – 1;
        end
    end
end
```

1

2

3

9. Генерирование разбиений множества (5)

(1 2 3 4)
(1 2 3) (4)
(1 2) (3) (4)
(1 2) (3 4)
(1 2 4) (3)
(1 4) (2) (3)
(1) (2 4) (3)
(1) (2) (3 4)
(1) (2) (3) (4)
(1) (2 3) (4)
(1) (2 3 4)
(1 4) (2 3)
(1 3 4) (2)
(1 3) (2 4)
(1 3) (2) (4)