Строки – 3: перестановки и скобочные последовательности

Denis Bakin

План

1. Рекурсивная генерация перестановок (идея «заморозки префикса»)
2. Алгоритм Нарайаны: следующая лексикографическая перестановка
3. Получение перестановки по номеру
4. Правильные скобочные последовательности: определение и генерация
5. Подсчет количества ПСП: рекуррентная формула Каталана

Рекурсивная генерация перестановок: идея

- На каждом уровне рекурсии фиксируем (замораживаем) очередную позицию
- После фиксации позиции `pos` префикс `a[0..pos]` уже не меняется внутри этой ветки
- Для фиксации перебираем, какой из «свободных» элементов поставить в `a[pos]`
- Когда зафиксированы все `n` позиций, получена готовая перестановка

Дерево рекурсии

Рекурсивная генерация: шаги

Пусть `pos` — первая незафиксированная позиция.

1. Если `pos == n`, выводим текущую перестановку
2. Иначе перебираем `j` от `pos` до `n - 1`
3. Меняем `a[pos]` и `a[j]` местами (фиксируем новый элемент)
4. Рекурсивно вызываем генерацию для `pos + 1`
5. Меняем обратно местами `a[pos]` и `a[j]`

Рекурсивная генерация перестановок (C++)

```cpp
#include <iostream>
#include <string>
#include <algorithm>

void generate(int last_idx, int length, std::string& s) {
    if (last_idx == length - 1) { // Вывод очередной перестановки
        // Вывод очередной перестановки
        return;
    }
    for (int j = last_idx; j < length; ++j) {  // Запускаем процесс обмена
        std::swap(s[last_idx], s[j]);          // a[t] со всеми последующими
        generate(last_idx + 1, length, s);     // Рекурсивный вызов
        std::swap(s[last_idx], s[j]);
    }
}
```

Алгоритм Нарайаны (next permutation)

Найти следующую перестановку в лексикографическом порядке:

1. Найти максимальный i, где $a_i < a_{i+1}$

Если шага 1 не существует, текущая перестановка последняя (убывающая).

Алгоритм Нарайаны (next permutation)

Найти следующую перестановку в лексикографическом порядке:

1. Найти максимальный i, где $a_i < a_{i+1}$
2. Найти максимальный $j > i$, где $a_j > a_i$

Если шага 1 не существует, текущая перестановка последняя (убывающая).

Алгоритм Нарайаны (next permutation)

Найти следующую перестановку в лексикографическом порядке:

1. Найти максимальный i, где $a_i < a_{i+1}$
2. Найти максимальный $j > i$, где $a_j > a_i$
3. Поменять a_i и a_j местами

Если шага 1 не существует, текущая перестановка последняя (убывающая).

Алгоритм Нарайаны (next permutation)

Найти следующую перестановку в лексикографическом порядке:

1. Найти максимальный i, где $a_i < a_{i+1}$
2. Найти максимальный $j > i$, где $a_j > a_i$
3. Поменять a_i и a_j местами
4. Развернуть суффикс $a[i+1], \dots, a_{n-1}$

Если шага 1 не существует, текущая перестановка последняя (убывающая).

Пример Нарайаны: 1 2 3 4

Реализация nextPermutation за $\mathcal{O}(n)$

```cpp
#include <algorithm>
#include <vector>
bool nextPermutation(std::vector<int>& a) {
    int i = static_cast<int>(a.size()) - 2;
    while (i >= 0 && a[i] >= a[i + 1])
        --i;

    if (i < 0)
        return false; // следующей перестановки нет

    int j = static_cast<int>(a.size()) - 1;
    while (a[j] <= a[i])
        --j;
    std::swap(a[i], a[j]);
    std::reverse(a.begin() + i + 1, a.end());
    return true;
}
```

STL для работы с перестановками

```
namespace std {
    bool next_permutation(it_first, it_last);
    bool prev_permutation(it_first, it_last);

    bool is_permutation(it11, it12,
                        it21, it22);
}
```

- `next_permutation` и `prev_permutation` работают на диапазоне по итераторам
- Для полного перебора обычно начинают с отсортированной последовательности

Скрипт: сортировка + полный перебор

```cpp
#include <algorithm>
#include <vector>

void printVector(const std::vector<int>& a);

int main() {
    std::vector<int> a = {4, 1, 3, 2};
    std::sort(a.begin(), a.end());

    printVector(a); // первая (минимальная) перестановка
    while (std::next_permutation(a.begin(), a.end())) {
        printVector(a);
    }
}
```

Перестановка по номеру: идея групп

Перестановка по номеру: идея групп

Для `n = 5`:

- Всего перестановок: `5! = 120`
- По первому элементу получаем 5 групп, каждая размера `4! = 24`
- Внутри каждой такой группы по второму элементу: 4 подгруппы размера `3! = 6`
- На каждом шаге размер группы равен факториалу от числа оставшихся позиций

Перестановка по номеру: шаг вычисления

Пусть `num` — номер (с нуля), `group_len = (remaining - 1)!`.

Тогда на текущем шаге:

$$\text{idx} = \left\lfloor \frac{\text{num}}{\text{group\_len}} \right\rfloor, \qquad \text{num} = \text{num} \bmod \text{group\_len}$$

- Выбираем `idx`-й элемент среди еще не использованных
- Если элементы изначально `1..n`, то в первом приближении это выглядит как `elem = num / group_len + 1`

Пример: n = 5, num = 42 (нумерация с нуля)

Свободные элементы: [1, 2, 3, 4, 5]

- group_len = 4! = 24, idx = 42 / 24 = 1 → берем 2, num = 42 % 24 = 18
- group_len = 3! = 6, idx = 18 / 6 = 3 → берем 5, num = 18 % 6 = 0
- group_len = 2! = 2, idx = 0 / 2 = 0 → берем 1, num = 0
- group_len = 1! = 1, idx = 0 / 1 = 0 → берем 3
- Остался 4

Итоговая перестановка: 2 5 1 3 4.

ПСП: определение (один тип скобок)

Правильные последовательности задаются рекурсивно:

- Пустая последовательность корректна
- Если A корректна, то (A) корректна
- Если A и B корректны, то AB корректна

Мы рассматриваем только круглые скобки.

ПСП: баланс скобок

Пусть `balance = open - close`.

Для корректной последовательности:

- На любом префиксе `balance >= 0`
- В конце всей строки `balance = 0`

Это и есть удобный критерий проверки и генерации.

Генерация ПСП рекурсией

Генерация ПСП рекурсией

- Можно добавить `'('`, если `open < n`
- Можно добавить `')'`, если `close < open`
- Когда длина стала `2n`, получена корректная последовательность

Генерация ПСП рекурсией

```
#include <iostream>
#include <string>
void generateBrackets(int n, int open, int close, std::string& cur) {
    if (open + close == 2 * n) {
        std::cout << cur << '\n';
        return;
    }
    if (open < n) {
        cur.push_back('(');
        generateBrackets(n, open + 1, close, cur);
        cur.pop_back();
    }
    if (close < open) {
        cur.push_back(')');
        generateBrackets(n, open, close + 1, cur);
        cur.pop_back();
    }
}
```

Сколько ПСП длины 2n?

Обозначим через C_n число правильных скобочных последовательностей с n открывающими скобками.

Сколько ПСП длины 2n?

Обозначим через C_n число правильных скобочных последовательностей с n открывающими скобками.

Любая непустая корректная последовательность единственным образом представляется как:

$$(A)B$$

где A и B — корректные.

Рекуррентная формула Каталана

Если в `A` ровно `k` пар скобок, то в `B` ровно `n - k - 1`.

Значит:

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-k-1}$$

База: `C_0 = 1`.

Первые значения:

- `C_0 = 1`
- `C_1 = 1`
- `C_2 = 2`
- `C_3 = 5`
- `C_4 = 14`
- `C_5 = 42`

Итоги

- Рекурсивная генерация перестановок фиксирует префикс на каждом уровне
- Алгоритм Нарайаны строит следующую перестановку за линейное время
- Нумерация перестановок опирается на разбиение на факториальные группы
- ПСП удобно описываются через баланс и рекурсивные правила построения
- Количество ПСП задается рекуррентной формулой Каталана