Создание перестановок

Во-первых, давайте начнем с перестановок. Перестановка — это перестановка последовательности таким образом, чтобы она имела другой порядок.

Как мы знаем из математики, **для последовательности из n элементов существует n! разные перестановки** . `н!` называется факториальной операцией:

н! = 1 *2* ... * п

Так, например, для последовательности `[1, 2, 3]` имеется шесть перестановок:

```
[1, 2, 3]

[1, 3, 2]

[2, 1, 3]

[2, 3, 1]

[3, 1, 2]

[3, 2, 1]
```

Факториал растет очень быстро — для последовательности из 10 элементов у нас есть 3 628 800 различных перестановок! В этом случае **мы говорим о перестановочных последовательностях, где каждый отдельный элемент отличается** .

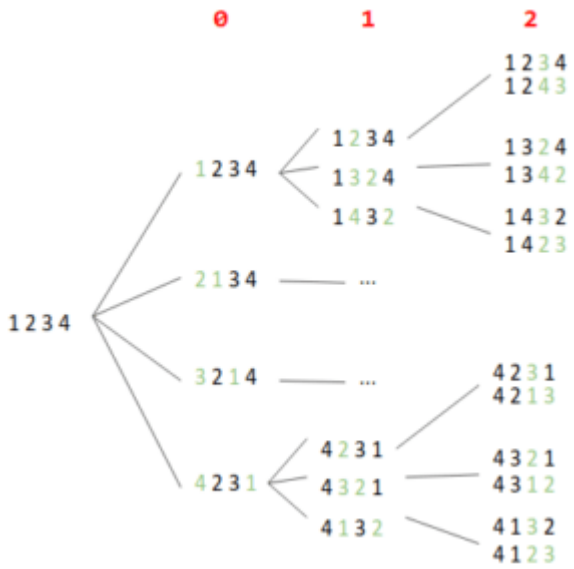
2.1. Алгоритм

Это хорошая идея подумать о генерации перестановок рекурсивным способом. **Введем понятие государства. Он будет состоять из двух вещей: текущей перестановки и индекса обрабатываемого в данный момент элемента.**

Единственная работа, которую нужно сделать в таком состоянии, — поменять местами элемент со всеми оставшимися и выполнить переход в состояние с измененной последовательностью и индексом, увеличенным на единицу.

Проиллюстрируем на примере.

Мы хотим сгенерировать все перестановки для последовательности из четырех элементов — `[1, 2, 3, 4]` . Таким образом, будет 24 перестановки. На иллюстрации ниже представлены частичные шаги алгоритма:



Каждый узел дерева можно понимать как состояние. Красные цифры вверху обозначают индекс обрабатываемого в данный момент элемента. Зеленые цифры в узлах обозначают свопы.

Итак, начинаем в состоянии `[1, 2, 3, 4]` с индексом, равным нулю. Мы меняем местами первый элемент с каждым элементом, включая первый, который ничего не меняет местами, и переходим к следующему состоянию.

Теперь наши желаемые перестановки расположены в последнем столбце справа.

2.2. Java-реализация
Алгоритм, написанный на Java, короткий:

```
private static void permutationsInternal(List<Integer> sequence,
List<List<Integer>> results, int index) {
    if (index == sequence.size() - 1) {
        permutations.add(new ArrayList<>(sequence));
    }

    for (int i = index; i < sequence.size(); i++) {
        swap(sequence, i, index);
        permutationsInternal(sequence, permutations, index + 1);
        swap(sequence, i, index);
    }
}
```

Наша функция принимает три параметра: обрабатываемую в данный момент последовательность, результаты (перестановки) и индекс обрабатываемого в данный момент элемента.

Первое, что нужно сделать, это проверить, достигли ли мы последнего элемента. Если это так, мы добавляем последовательность в список результатов.

Затем в цикле for мы выполняем обмен, делаем рекурсивный вызов метода, а затем обмениваем элемент обратно.

Последняя часть представляет собой небольшой трюк с производительностью: мы можем все время работать с одним и тем же объектом `последовательности` , не создавая новую последовательность для каждого рекурсивного вызова.

Также может быть хорошей идеей скрыть первый рекурсивный вызов в фасадном методе:

```
public static List<List<Integer>> generatePermutations(List<Integer>
sequence) {
    List<List<Integer>> permutations = new ArrayList<>();
    permutationsInternal(sequence, permutations, 0);
    return permutations;
}
```

Имейте в виду, что показанный алгоритм будет работать только для последовательностей уникальных элементов! Применение того же алгоритма к последовательностям с повторяющимися элементами даст нам повторения.

Теперь пришло время заняться комбинациями. Мы определяем его следующим образом:

k -комбинация множества S - это подмножество из k различных элементов из `S`, где порядок элементов не имеет значения.
Количество k -комбинаций описывается биномиальным коэффициентом:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Так, например, для множества `[a, b, c]` имеем три 2 -комбинации:

```
[a, b]

[a, c]

[b, c]
```

Комбинации имеют множество комбинаторных применений и объяснений. В качестве примера предположим, что у нас есть футбольная лига, состоящая из 16 команд. Сколько разных совпадений мы можем увидеть?

Ответ

$\binom{16}{2}$

, который оценивается как 120.

4.1. Алгоритм

Концептуально мы сделаем что-то похожее на предыдущий алгоритм для наборов мощности. У нас будет рекурсивная функция, состояние которой состоит из индекса обрабатываемого в данный момент элемента и аккумулятора.

Опять же, у нас есть одно и то же решение для каждого состояния: добавить элемент в аккумулятор? Однако на этот раз у нас есть дополнительное ограничение — наш аккумулятор не может содержать более k элементов .

Стоит отметить, что биномиальный коэффициент не обязательно должен быть огромным числом. Например:

$$\binom{100}{2}$$

равно 4950, а

$$\binom{100}{50}$$

имеет 30 цифр!

4.2. Java-реализация

Для простоты мы предполагаем, что элементы в нашем множестве являются целыми числами.

Давайте взглянем на реализацию алгоритма в Java:

```
private static void combinationsInternal(
  List<Integer> inputSet, int k, List<List<Integer>> results,
ArrayList<Integer> accumulator, int index) {
  int needToAccumulate = k - accumulator.size();
  int canAcculumate = inputSet.size() - index;

  if (accumulator.size() == k) {
      results.add(new ArrayList<>(accumulator));
  } else if (needToAccumulate <= canAcculumate) {
      combinationsInternal(inputSet, k, results, accumulator, index + 1);
      accumulator.add(inputSet.get(index));
      combinationsInternal(inputSet, k, results, accumulator, index + 1);
      accumulator.remove(accumulator.size() - 1);
  }
}
```

На этот раз наша функция имеет пять параметров: набор входных данных, параметр k , список результатов, аккумулятор и индекс текущего обрабатываемого элемента.

Начнем с определения вспомогательных переменных:

- `needToAccumulate` — указывает, сколько еще элементов нам нужно добавить в наш аккумулятор, чтобы получить правильную комбинацию.
- `canAcculumate` — указывает, сколько еще элементов мы можем добавить в наш аккумулятор

Теперь мы проверяем, равен ли размер нашего аккумулятора `k` . Если это так, то мы можем поместить скопированный массив в список результатов.

В другом случае, **если у нас еще достаточно элементов в оставшейся части набора, мы делаем два отдельных рекурсивных вызова: с помещением в аккумулятор текущего обрабатываемого элемента и без него.** Эта часть аналогична тому, как мы создали набор мощности ранее.

Конечно, этот метод можно было бы написать так, чтобы он работал немного быстрее. Например, позже мы могли бы объявить переменные `needToAccumulate` и `canAcculumate` . Тем не менее, мы сосредоточены на удобочитаемости.

Опять же, фасадный метод скрывает реализацию:

```
public static List<List<Integer>> combinations(List<Integer> inputSet, int k)
{
    List<List<Integer>> results = new ArrayList<>();
    combinationsInternal(inputSet, k, results, new ArrayList<>(), 0);
    return results;}
```