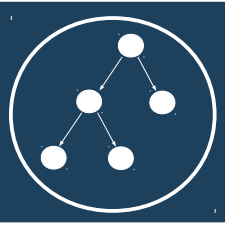


Data Structures and Algorithms

Алгоритмы.
Генерация перестановок в лексикографическом порядке.
Алгоритм Нарайаны.

Лексикографический порядок последовательностей

Лексикографический порядок — отношение линейного порядка на множестве последовательностей.

1) Последовательность **A** предшествует последовательности **B** (A < B), если первые n элементов из этих последовательностей совпадают, а n +1 элемент **B** больше n+1 элемента **A**.

1, 2, 3, 4, 5 - **A**

n, n+1

1, 2, 3, 5, 4 - **B**

n, n+1

$$A_{n+1} < B_{n+1} \Rightarrow A < B$$

2) Последовательность **A** является началом последовательности **B.**

1, 2, 3 - **A**

1, 2, 3, 5, 4 - **B**

A

$$A < B$$

Лексикографический порядок перестановок

Если рассматривать перестановки одинаковой длинны то для них лексикографический порядок можно сформулировать так: перестановка A предшествует перестановке B (A < B), если первые n элементов из этих перестановок совпадают, а n +1 элемент B больше n+1 элемента A.

1, 2, 3, 4, 5 - **A**

n n+1

1, 2, 3, 5, 4 - **B**

n n+1

$$A_{n+1} < B_{n+1} \Rightarrow A < B$$

Генерация всех перестановок в лексикографическом порядке

Для генерации всех перестановок в лексикографическом порядке, можно использовать алгоритм Нарайаны. И хотя сам алгоритм предназначен для получения следующей перестановки в лексикографическом порядке, его не сложно дополнить для генерации всех перестановок длинной n.

К его достоинствам можно отнести простоту реализации (не рекурсивный), относительно малый расход дополнительной памяти.

Сведение о алгоритме

Алгоритм Нарайаны

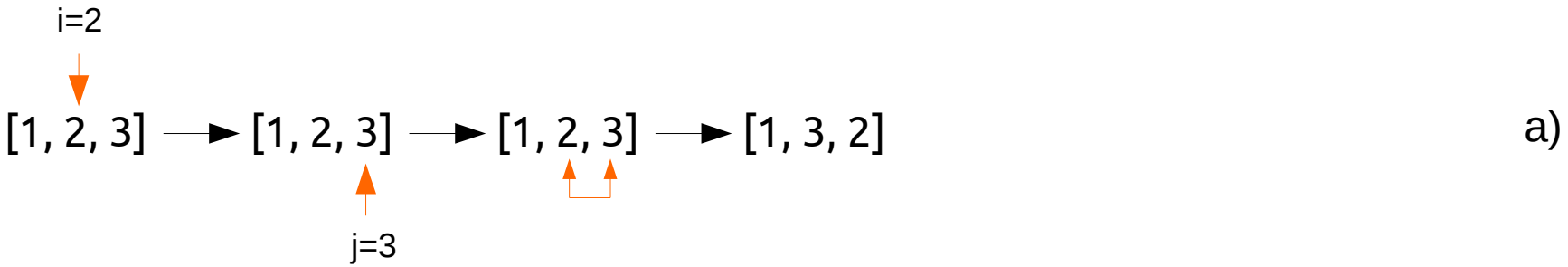
Сложность по времени в наихудшем случае	O(n)
Затраты памяти	O(n)

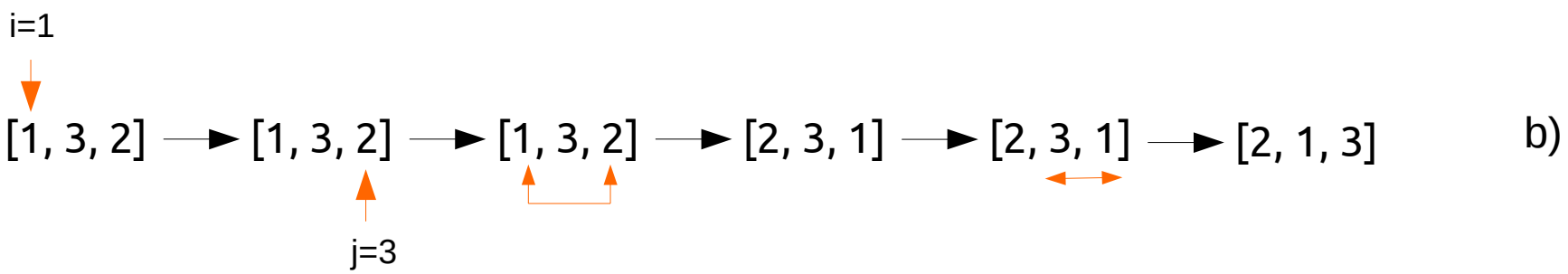
Внимание!! Это сложность генерации **одной** (следующей в лексикографическом порядке) перестановки. Генерация всех перестановок имеет факториальную сложность.

Описание алгоритма Нарайаны

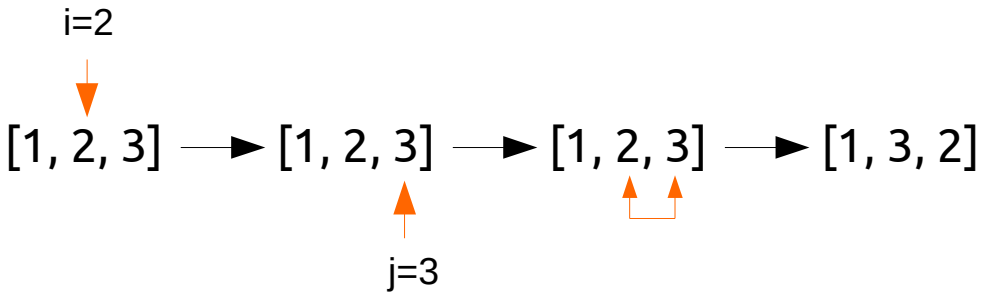
1) Найти такой максимальный индекс i для которого $A_i < A_{i+1}$ (оптимально выполнять поиск с конца перестановки).
2) Найти максимальный индекс j для которого $A_j > A_i$ (оптимально выполнять поиск с конца перестановки).
3) Выполнить обмен A_i и A_j элемента местами
4) Записать последовательность $A_{i+1},\ldots, A_n$ в обратном порядке

Графическое пояснение





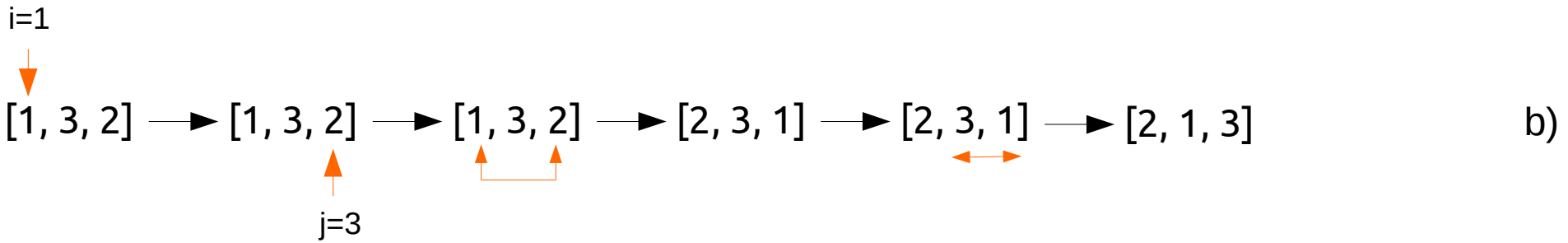
Описание алгоритма



a)

На первом шаге ищем максимальный индекс элемента сосед которого справа больше. Поиск выполняется с конца последовательности. Это индекс 2. После чего ищем максимальный индекс элемента который больше чем элемент индекс которого найден на шаге 1. Это индекс 3. Проводим обмен. После этого нужно провести реверс последовательности от индекса 2 и до конца. Но после второго индекса идет только один элемент, так что его реверс ничего не изменит.

Описание алгоритма



На первом шаге ищем максимальный индекс элемента сосед которого справа больше. Поиск выполняется с конца последовательности. Это индекс 1. После чего ищем максимальный индекс элемента который больше чем элемент индекс которого найден на шаге 1. Это индекс 3. Проводим обмен. После этого нужно провести реверс последовательности от индекса 1 и до конца. После реверса мы получаем новую перестановку.

Применения алгоритма Нарайаны для генерации всех перестановок

Для генерации всех перестановок длинной n с использованием алгоритма Нарайаны, можно использовать такой алгоритм.

1) Начинаем с базовой перестановки 1, 2, 3… n.
2) Ищем максимальный индекс элемента сосед которого справа больше его. Если такого индекса нет прекращаем работу алгоритма.
3) Генерируем следующую перестановку используя алгоритм Нарайаны.
4) Используем полученную перестановку в пункте 2.

Python

Реализация алгоритма на Python

Python

Реализация алгоритма на Python

```
def find_max_index(permutation):
    for i in range(len(permutation) - 2, -1, -1):
        if permutation[i] < permutation[i+1]:
            return i
    return -1
```

Функция поиска максимального индекса элемента сосед которого справа больше

```
def find_index_bigger_element(permutation, element):
    for i in range(len(permutation) - 1, -1, -1):
        if permutation[i] > element:
            return i
    return -1
```

Функция поиска максимального индекса элемента большего чем на найденном индексе

Реализация алгоритма на Python

```
def swap(permutation, i, j):
    permutation[i], permutation[j] = permutation[j], permutation[i]
```

Функция обмена двух элементов последовательности

```
def reverse_permutation(permutation, index):
    n = len(permutation)
    permutation = permutation[:index+1:] + permutation[n - 1:index:-1]
    return permutation
```

Функция для реверса части последовательности

Реализация алгоритма на Python

```
def permutation_generator(n):
    permutation = list(range(1, n+1))
    print(permutation)
    index = find_max_index(permutation)
    while index != -1:
        element = permutation[index]
        swap_index = find_index_bigger_element(permutation, element)
        swap(permutation, index, swap_index)
        permutation = reverse_permutation(permutation, index)
        print(permutation)
        index = find_max_index(permutation)
```

Функция генерации перестановок длинной n

Java

Реализация алгоритма на Java

Реализация алгоритма на Java

```
public static int findMaxIndex(int[] permutation) {
    for (int i = permutation.length - 2; i >= 0; i--) {
        if (permutation[i] < permutation[i + 1]) {
            return i;
        }
    }
return -1;
}
```

Функция поиска максимального индекса элемента сосед которого справа больше

```
public static int findIndexBiggerElement(int[] permutation, int element) {
    for (int i = permutation.length - 1; i >= 0; i--) {
        if (permutation[i] > element) {
            return i;
        }
    }
return -1;
}
```

Функция поиска максимального индекса элемента большего чем на найденном индексе

Реализация алгоритма на Java

```java
public static void swap(int[] permutation, int i, int j) {
    int temp = permutation[i];
    permutation[i] = permutation[j];
    permutation[j] = temp;
}
```

Функция обмена двух элементов последовательности

```java
public static void reverse(int[] permutation, int index) {
    int shift = index + 1;
    for (int i = 0; i < (permutation.length - shift) / 2; i++) {
        int temp = permutation[shift + i];
        permutation[shift + i] = permutation[permutation.length - i - 1];
        permutation[permutation.length - i - 1] = temp;
    }
}
```

Функция для реверса части последовательности

Реализация алгоритма на Java

```
public static void permutationGenerator(int n) {
    int[] permutation = new int[n];
    for (int i = 0; i < permutation.length; i++) {
        permutation[i] = i + 1;
    }
    System.out.println(Arrays.toString(permutation));
    int index = findMaxIndex(permutation);
    for (; index != -1;) {
        int element = permutation[index];
        int swapIndex = findIndexBiggerElement(permutation, element);
        swap(permutation, index, swapIndex);
        reverse(permutation, index);
        System.out.println(Arrays.toString(permutation));
        index = findMaxIndex(permutation);
    }
}
```

Функция генерации перестановок длинной n

Список литературы

1) Дональд Кнут. «Искусство программирования, том 4, выпуск 2. Генерация всех кортежей и перестановок.» : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2008. - 160 с. ISBN 978-5-8459-1164-3. [53 -54]