МИНОБРНАУКИ РОССИИ
ФГБОУ ВО «СГУ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

РЕКУРСИВНЫЕ АЛГОРИТМЫ. УПР 3.2

ЛАБОРАТОРНАЯ РАБОТА

студента 3 курса 331 группы
направления 10.05.01 — Компьютерная безопасность
факультета КНиИТ
Никитина Арсения Владимировича

Проверил
доцент ______________ А. Н. Гамова

Саратов 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной работе будут рассмотрены принципы рекурсивных алгоритмов, в частности алгоритма получения всех возможных перестановок элементов заданного множества без использования дополнительной памяти.

1 Постановка задачи

Напишите процедуру формирования на том же месте всех $n!$ перестановок для n элементов $a_1, ..., a_n$, то есть без дополнительного массива. После формирования очередной перестановки можно, например, обратиться к процедуре Q (с параметром), которая напечатает полученную перестановку.

Указание: задачу формирования всех перестановок элементов $a_1, ..., a_m$ можно считать состоящей из m подзадач формирования всех перестановок для элементов $a_1, ..., a_{m-1}$, за которыми следует a_m. Причем в i-й подзадаче вначале меняются местами элементы a_1 и a_m

2 Рекурсивный объект

Рекурсивным называется объект, частично состоящий или определяемый с помощью самого себя.

Мощность рекурсивного определения заключается в том, что оно позволяет с помощью конечного высказывания определить бесконечное вычисление, причем программа не будет содержать явных повторений. Однако рекурсивные алгоритмы лучше всего использовать, если в решаемой задаче, вычисляемой функции или структуре обрабатываемых данных рекурсия уже присутствует явно. В общем виде рекурсивную программу P можно выразить как некоторую композицию P из множества операторов S (не содержащих P) и самой P:

$$P = P[S, P]$$

Для выражения рекурсивных программ необходимо и достаточно иметь понятие процедуры или подпрограммы, поскольку они позволяют дать любому оператору имя, с помощью которого к нему можно обращаться. Если некоторая процедура P содержит явную ссылку на саму себя, то ее называют *прямо рекурсивной*. Если же P ссылается на другую процедуру Q, содержащую (прямую или косвенную) ссылку на P, то P называют *косвенно рекурсивной*.

Подобно операторам цикла, рекурсивные процедуры могут приводить к не заканчивающимся вычислениям, и, поэтому на эту проблему следует особо обратить внимание. Очевидно основное требование, чтобы рекурсивное обращение к P управлялось некоторым условием B, которое в какой-то момент становится ложным.

3 Описание алгоритма получения всех перестановок множества

Как и было сказано в указании к реализации алгоритма, для того, чтобы не было лишних затрат по памяти, можно сразу выводить текущую перестановку в консоль, что и реализовано с помощью вспомогательной функции $print\_permuration$. Данная функция получает массив, в котором переставлены некоторые элементы и выводит в консоль порядковый номер текущей перестановки (благодаря глобальной переменной $current$), а также саму перестановку $current\_permutation$. Данную функцию требуется вызвать до вызова основной рекурсивной функции $next\_permutation$ для того, чтобы вывести в консоль самую первую перестановку.

Самая первая перестановка получается путем сортировки входного множества.

Затем происходит единственный вызов из основной части программы рекурсивной функции $next\_set$ от отсортированного множества и от его размера.

Абсолютно все перестановки получаются в лексикографическом порядке.

Итак, алгоритм получения всех перестановок выглядит следующим образом:

1. Необходимо просмотреть текущую перестановку справа налево и при этом следить за тем, чтобы каждый следующий элемент перестановки (элемент с большим номером) был не более чем предыдущий (элемент с меньшим номером). Как только данное соотношение будет нарушено необходимо остановиться и отметить текущее число (позиция 2).
2. Снова просмотреть пройденный путь справа налево пока не дойдем до первого числа, которое больше чем отмеченное на предыдущем шаге.
3. Затем нужно поменять местами два полученных элемента.
4. Теперь в части массива, которая размещена справа от позиции 1 надо отсортировать все числа в порядке возрастания.
5. Поскольку до этого они все были уже записаны в порядке убывания необходимо эту часть подпоследовательность просто перевернуть.
6. Затем происходит вызов процедуры печати в консоль полученной перестановки с увеличением глобальной переменной $current$ на единицу.
7. Затем рекурсивный вызов функции получения следующей перестановки.

4 Результаты тестирования программы

```
Введите количество элементов множества
4
Введите последовательно без пробелов n элементов
45 32 54 65
Исходное множество: 32, 45, 54, 65
Всевозможные перестановки множества:
1: 32, 45, 54, 65
2: 32, 45, 65, 54
3: 32, 54, 45, 65
4: 32, 54, 65, 45
5: 32, 65, 45, 54
6: 32, 65, 54, 45
7: 45, 32, 54, 65
8: 45, 32, 65, 54
9: 45, 54, 32, 65
10: 45, 54, 65, 32
11: 45, 65, 32, 54
12: 45, 65, 54, 32
13: 54, 32, 45, 65
14: 54, 32, 65, 45
15: 54, 45, 32, 65
16: 54, 45, 65, 32
17: 54, 65, 32, 45
18: 54, 65, 45, 32
19: 65, 32, 45, 54
20: 65, 32, 54, 45
21: 65, 45, 32, 54
22: 65, 45, 54, 32
23: 65, 54, 32, 45
24: 65, 54, 45, 32
```

Рисунок 1

5 Программная реализация алгоритма

```
a = []


def swap(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp

current = 1

def print_permutation(current_permutation):
        global current
        print(f'{current}: ', end='')
        print(*current_permutation, sep=', ')
        current += 1


def next_set(a, n):

    j = n - 2
    while j != -1 and a[j] >= a[j+1]:
        j -= 1

    if j == -1:
        return

    k = n - 1
    while a[j] >= a[k]:
        k -= 1
    swap(a, j, k)
    l = j + 1
    r = n - 1

    while l < r:
        swap(a, l, r)
        l += 1
        r -= 1
    print_permutation(a)
    return next_set(a, n)

```

```
print('Введите количество элементов множества')
n = int(input())
print(f'Введите последовательно без пробелов n элементов')

a = sorted(list(map(lambda x: int(x), input().split())))

print('Исходное множество: ', end='')
print(*a, sep=', ')
print('Всевозможные перестановки множества:')


print_permutation(a)
next_set(a, n)
```

6 Оценка работы алгоритма

Для исследования производительности рассмотренного алгоритма рассмотрим его составляющие.

Так как рекурсивная функция получения перестановок получает перестановки в лексикографическом порядке, требуется вызвать ее от отсортированного в возрастающем порядке массива, поэтому изначально выполняется сортировка массива с помощью встроенной функции $sorted()$, использующей оптимизированный алгоритм быстрой сортировки. Поэтому данную часть программы можно асимптотически оценить в $O(n \cdot \log n)$.

Затем происходит печать первой перестановки (исходного множества в отсортированном по возрастанию порядке), что занимает $O(n)$.

Затем происходит вызов рекурсивной функции $next\_permuration$, асимптотика которого равна $O(n!)$, но так как внутри нее происходит вызов функции печати перестановки, асимптотика которого равна $O(n)$, то в итоге общая асимптотика равна $O(n \cdot n!)$.

Получаем общую асимптотику получения всех перестановок:

$$O((n \cdot n! + n \cdot \log n) = O(n^2 \cdot (n-1)!)$$

ЗАКЛЮЧЕНИЕ

Итак, в данной работе был рассмотрен алгоритм получения всех $n!$ перестановок множества и вывода их в консоль без использования дополнительной памяти с помощью прямо рекурсивной функции. Произведена оценка его работы. Асимптотика работы данного алгоритма составляет $O(n \cdot n!)$.