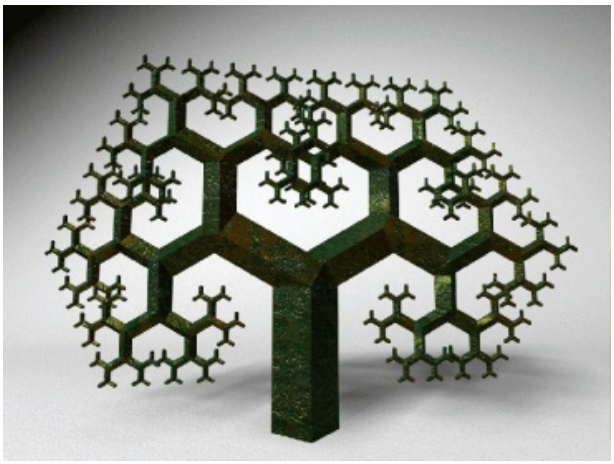
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Т. И. Федоряева

# КОМБИНАТОРНЫЕ АЛГОРИТМЫ

Учебное пособие



Новосибирск
2011

УДК 519.1+510.52+519.683
ББК В127я73-1
Ф337

**Федоряева Т. И.** Комбинаторные алгоритмы: Учебное пособие / Новосиб. гос. ун-т. Новосибирск, 2011. 118 с.

ISBN 978-5-4437-0019-9

Учебное пособие написано на основе курса "Комбинаторные алгоритмы", читаемого автором студентам факультета информационных технологий НГУ. Наряду с теоретическими знаниями даётся описание важнейших комбинаторных алгоритмов над объектами дискретной математики, приводится строгое обоснование рассматриваемых алгоритмов и детально изучается их асимптотическая сложность.

Пособие прежде всего ориентировано на студентов программистских специальностей, которым по роду их занятий приходится заниматься разработкой алгоритмов и анализом их вычислительной сложности. Изучение комбинаторных алгоритмов также будет полезно любому заинтересованному читателю для развития самостоятельных навыков по построению и анализу алгоритмов, для решения задач в области дискретной математики и применения методов дискретного анализа в своей профессиональной деятельности.

Рецензенты:

зав. лабораторией совершенных комбинаторных структур Института математики им. С. Л. Соболева, канд.физ.-мат.наук С. В. Августинович; доцент факультета информационных технологий Новосибирского государственного университета, канд.физ.-мат.наук А. Л. Пережогин

Учебное пособие разработано в соответствии с требованиями ФГОС ВПО к структуре и результатам освоения основных образовательных программ по профессиональному циклу по направлению подготовки "Информатика и вычислительная техника". Издание подготовлено в рамках реализации *Программы развития государственного образовательного учреждения высшего профессионального образования "Новосибирский государственный университет"* на 2009–2018 годы.



ISBN 978-5-4437-0019-9

# Оглавление

# Предисловие

Комбинаторные алгоритмы предназначены для выполнения вычислений на различного рода объектах, возникающих в прикладных комбинаторных задачах и при исследовании дискретных математических структур. Необходимость разработки эффективных, быстрых комбинаторных алгоритмов уже давно не вызывает сомнений. На практике нужны не просто алгоритмы, а хорошие алгоритмы в широком смысле. Одним из основных критериев качества алгоритма является время, необходимое для его выполнения.

Разработке и анализу вычислительной сложности комбинаторных алгоритмов над классическими комбинаторными объектами посвящено настоящее учебное пособие. Наряду с теоретическими знаниями даётся описание таких важнейших алгоритмов, приводится их строгое обоснование и детально изучается асимптотическая сложность рассматриваемых алгоритмов. Мы познакомим читателя с широким кругом понятий и сведений из дискретной математики, необходимых практикующему программисту. Пополним запас примеров нетривиальных алгоритмов над объектами дискретной математики, помогающих существенно обогатить навыки самостоятельного конструирования алгоритмов и сформировать мышление, позволяющее использовать методы дискретного анализа при разработке эффективных алгоритмов для решения практических задач и оценке их сложности.

Для понимания материала учебного пособия требуется знание основных понятий и фактов из дискретной математики и математической логики. Читатель должен обладать минимальным опытом программирования, каждый изучаемый алгоритм снабжен понятным псевдокодом, позволяющим реализовать рассматриваемый алгоритм на доступном языке программирования. При изучении отдельных тем используются основы математического анализа и теории вероятностей.

Практически все рассматриваемые задачи и алгоритмы их решения, разумеется, не являются новыми, однако во многих случаях изложенные доказательства и обоснования оценок сложности оригинальны. Мы попытаемся изложить каждый алгоритм так, чтобы был понятен путь его возникновения. Особое внимание уделяется выявлению интуитивных идей, лежащих в основе алгоритмов, и иллюстрации работы изучаемых алгоритмов на примерах.

Материал учебного пособия организован следующим образом. В первой главе обсуждаются понятие сложности машинных алгоритмов и язык псевдокода, на котором записываются алгоритмы. Даётся асимптотический формализм для оценок времени работы алгоритмов и приводятся примеры асимптотических соотношений. На примере алгоритма $n$-факториального представления числа изучаются введённые понятия и обозначения, исследуется время работы алгоритма.

Вторая глава посвящена задаче генерации комбинаторных объектов, среди которых перестановки, все подмножества заданного конечного множества, двоичные векторы, коды Грея и сочетания из $n$ элементов по $k$. Основное внимание уделено линейным алгоритмам генерации объектов в лексикографическом порядке и в порядке минимального изменения. Для перестановок изучаются понятия индекса относительно лексикографического порядка и вектора инверсий. Детально исследуются линейные алгоритмы генерации перестановок в лексикографическом порядке и Джонсона – Троттера в порядке минимального изменения. Далее рассматриваются алгоритмы порождения двоичных векторов и подмножеств $n$-элементного множества, понятие кода Грея и линейный алгоритм генерации двоично-отраженных кодов Грея. Наконец, в лексикографическом порядке порождаются все сочетания. Для всех изучаемых алгоритмов обосновывается их корректность и оценивается асимптотическая сложность.

Третья глава знакомит с алгоритмами генерации случайных комбинаторных объектов с равномерным распределением и даёт начальное

представление о теории вероятностных алгоритмов. Она содержит алгоритмы построения случайной перестановки, случайного подмножества и случайного сочетания.

В четвёртой главе рассматривается задача порождения разбиений совокупности комбинаторных объектов. Изучаются разбиения чисел и множеств. Обсуждаются упорядоченные и неупорядоченные разбиения числа $n$. Изучаются алгоритмы генерации разбиений числа $n$ в словарном порядке и разбиений $n$-элементного множества, которые имеют асимптотически наилучший порядок.

Пятая глава посвящена задаче сортировки комбинаторных объектов. В ней мы знакомим читателя с понятиями минимального, среднего и максимального времени работы алгоритма. Сначала устанавливаются нижние оценки сложности произвольного алгоритма сортировки сравнением. Далее изучаются алгоритмы сортировки вставками, пузырьковой, быстрой и пирамидальной сортировки. Для каждого из алгоритмов устанавливается асимптотический порядок минимального, среднего и максимального времени их работы. Глава заканчивается изучением линейного алгоритма сортировки подсчётом.

Учебное пособие написано на основе курса "Комбинаторные алгоритмы", читаемого автором студентам факультета информационных технологий Новосибирского государственного университета. Оно прежде всего ориентировано на студентов программистских специальностей, которым по роду их занятий приходится иметь дело с разработкой алгоритмов и анализом их вычислительной сложности. Предоставляемые знания необходимы практикующему программисту, поскольку они существенно обогащают навыки конструирования эффективных алгоритмов. Изучение комбинаторных алгоритмов также будет полезно любому заинтересованному читателю для развития самостоятельных навыков по разработке и анализу алгоритмов, для решения задач в области дискретной математики и применения методов дискретного анализа в своей профессиональной деятельности.

# Глава 1
# Введение

## 1. Машинные алгоритмы и их сложность

Понятие алгоритма, подобно многим фундаментальным понятиям математики, является настолько интуитивно "понятным", насколько и сложным при его строгой формализации и скорее должно рассматриваться как неопределяемое. С неформальной точки зрения под *алгоритмом* часто понимается формально описанная вычислительная процедура, получающая исходные данные, называемые *входными данными* алгоритма, и выдающая результат вычислений на *выход*. Мы также ограничимся таким подходом, оставляя многочисленные известные формализации данного понятия вне рамок настоящего пособия.

Алгоритмы строятся для решения тех или иных вычислительных задач. Формулировка задачи описывает, каким требованиям должны удовлетворять входные и выходные данные, а алгоритм, решающий эту задачу, для каждой входной последовательности находит решение задачи, записываемое в выходные данные. Такой алгоритм, когда для каждого допустимого ввода результатом его работы является требуемый в задаче вывод, называют *корректным*. Некорректный алгоритм для некоторых входных данных может вообще не завершить свою работу или выдать выходные данные, отличные от требуемых в задаче.

При анализе алгоритма решения поставленной задачи нас в первую очередь будет интересовать его трудоёмкость, под которой мы понимаем время выполнения соответствующей программы на ЭВМ. Ясно, что этот показатель существенно зависит от типа используемого компьютера. Чтобы сделать наши выводы о трудоёмкости алгоритмов в достаточной мере универсальными, будем считать, что все вычисления производятся на некой абстрактной вычислительной машине. Такая машина в состоянии выполнять арифметические операции, сравнения, пересылки и операции условной и безусловной передачи управления. Эти

операции считаются *элементарными*. Мы принимаем, что каждая из элементарных операций выполняется за единицу времени (т. е. не учитываем продолжительность времени, затраченного на выполнение самих этих операций), и, следовательно, время работы алгоритма равно числу выполненных им элементарных операций. Память рассматриваемой абстрактной вычислительной машины состоит из неограниченного числа ячеек, к которым имеется прямой доступ. После того, как всем входным данным задачи присвоены конкретные значения, они размещаются в памяти компьютера.

С входными данными алгоритма связано некоторое натуральное число (или набор целочисленных параметров), называемое *размерностью данных*, которое выражает меру их количества. Как определяется размерность данных? Это зависит от вида рассматриваемой задачи. В одних случаях размерностью разумно считать число элементов на входе, например, для задачи сортировки. В других более естественно считать размерностью общее число ячеек, необходимых для размещения всех входных данных в памяти (такой подход является наиболее употребительным). Как уже отмечалось, иногда размерность данных измеряется не одним числом, а несколькими, например, в случае задач для графов с $n$ вершинами и $m$ рёбрами. При изучении конкретных алгоритмов формализация понятия размерности данных, как правило, не вызывает трудностей, поэтому мы не будем останавливаться на её детализации (чаще всего это будет длина входных данных). Отметим также, что в литературе размерность данных иногда называют размерностью самой задачи, для которой разрабатывается алгоритм её решения.

Определим *сложность*, или *трудоёмкость алгоритма* решения данной задачи как функцию $T$ от размерности данных $n$, ставящую в соответствие каждому $n$ наибольшее время $T(n)$ работы алгоритма на входных данных размерности $n$. Заданную таким образом сложность иногда называют *временной сложностью*, в отличие от *сложности по памяти*, определяющей величину объёма памяти, использованного ал-

горитмом, как функцию размерности данных. Анализ эффективности каждого из представленных алгоритмов заключается в выяснении вопроса: как быстро растёт функция $T(n)$ с ростом $n$? Иными словами, нас интересует только асимптотическая сложность, т. е. асимптотическая скорость увеличения времени работы алгоритма, когда размерность данных неограниченно растёт.

Алгоритмы будем записывать на неформальном языке программирования, содержащем обычные общеизвестные конструкции. При этом мы предполагаем, что читатель знаком с одним из языков программирования высокого уровня. Как правило, будем опускать описание типов и переменных, использующихся в алгоритме, которое легко восстанавливается при записи окончательного кода программы. Приведем основные операторы, используемые при написании псевдокода:

- операторные скобки для задания составного оператора

  **begin**
  ...
  **end**;

- операторы цикла

  **for** $i = 0$ **to** $n$ **do** $A(i)$ (выполнять оператор $A(i)$ последовательно для $i = 0, 1, 2, \ldots, n$);

  **for** $i = n$ **downto** $0$ **do** $A(i)$ (выполнять оператор $A(i)$ последовательно для $i = n, n-1, n-2, \ldots, 0$);

  **for** $x \in X$ **do** $A(x)$ (выполнять оператор $A(x)$ для всех элементов $x$ множества $X$ в произвольной последовательности);

  **while** условие $A$ **do** $B$ (выполнять оператор $B$ до тех пор, пока выполняется условие $A$);

- оператор присваивания $x := y$;

- условный оператор **if** условие $A$ **then** $B$ **else** $C$ (если выполняется условие $A$, выполнить оператор $B$, иначе выполнить оператор $C$ );

- оператор $x \leftrightarrow y$ (обмен значениями между $x$ и $y$). В случае отсутствия в используемом языке программирования оператор обмена $\leftrightarrow$ реализуется следующей процедурой $Swap$.

  **Procedure** $Swap(x, y)$;
  $z := x$;
  $x := y$;
  $y := z$

- оператор вывода данных **write**.

# 2. Асимптотический формализм оценок времени работы алгоритмов

Несмотря на то, что функция сложности алгоритма в некоторых случаях может быть определена точно, в большинстве случаев искать её точное значение не имеет смысла. Дело в том, что для данных достаточно большой размерности постоянные множители и слагаемые низшего порядка, участвующие в выражении для такой функции, вносят крайне незначительный вклад и подавляются эффектами, вызванными увеличением размерности данных. Простейший эффект такого рода можно наблюдать, когда функция $T(n)$ сложности алгоритма представляет собой некоторый многочлен $T(n) = a_m\, n^m + a_{m-1}\, n^{m-1} + \ldots + a_1\, n + a_0$. В этом случае роль одночленов $a_i\, n^i$ степени, меньшей степени $m$ самого многочлена, несущественна, и ими можно пренебречь при достаточно большой размерности $n$.

При анализе алгоритмов такой подход оказывается крайне полезным, так как предлагает наглядную характеристику эффективности алгоритма — асимптотический порядок роста функции его сложности и позволяет сравнивать производительность различных алгоритмов. Чтобы сравнить одну величину с другой, во многих случаях достаточно знать не точные, а приближённые их значения, определяющие поведение роста этих величин. Например, известная формула Стирлинга даёт

широко применяемое приближение

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Paul Bachmann в 1894 г. ввел очень удобное обозначение $O$ (читается "*о большое*") для использования в приближенных формулах, до сих пор этот формализм применяется во многих математических дисциплинах, в том числе и при анализе алгоритмов.

**Определение 1.** Неотрицательная функция $f(n)$ *не превосходит по порядку* функцию $g(n)$ (*используем запись* $f(n) = O(g(n))$), если существуют положительные константы $N$ и $c$ такие, что $f(n) \leq c\,g(n)$ для любого $n \geq N$.

В этом случае функция $g(n)$ является асимптотически верхней оценкой функции $f(n)$[1]. Часто встречающемуся выражению "трудоёмкость (сложность) алгоритма есть или равна $O(g(n))$" придаётся именно такой смысл. Это означает, что для некоторой постоянной $c$ алгоритм на произвольном входе размерности $n$ заканчивает работу не более, чем через $c\,g(n)$ элементарных операций для всех достаточно больших $n$. В частности, трудоёмкость $O(1)$ означает, что время работы соответствующего алгоритма не зависит от размерности входа и не превосходит некоторой константы. Алгоритм с трудоёмкостью $O(n)$, где $n$ — размерность входа, называют *линейным*. Алгоритм сложности $O(n^c)$ называется *полиномиальным*. Алгоритм, сложность которого есть $O\left(2^{n^c}\right)$, называется *экспоненциальным*. Здесь $c$ — положительная константа.

Записи с символом $O$ дают верхнюю оценку скорости роста функции, но, вообще говоря, не дают точный порядок роста. Например, если время работы алгоритма есть $O(n^2)$, это не означает, что это же время не может составлять $O(n)$. Для нижних оценок применяется другая запись с символом $\Omega$ (читается "*омега большое*").

[1] С точностью до константы.

**Определение 2.** Неотрицательная функция $f(n)$ *не меньше по порядку* функции $g(n)$ (*используем запись* $f(n) = \Omega(g(n))$ ), если существуют положительные константы $N$ и $c$ такие, что $f(n) \geq c\,g(n)$ для любого $n \geq N$.

В этом случае функция $g(n)$ является асимптотически нижней оценкой функции $f(n)$. И наконец, чтобы точно указать порядок роста функции $f(n)$, не давая точных значений констант, применяется запись с символом $\Theta$ (читается "*тэта большое*").

**Определение 3.** Неотрицательная функция $f(n)$ *асимптотически равна* функции $g(n)$ (*используем запись* $f(n) = \Theta(g(n))$ ), если существуют положительные константы $N, c_1, c_2$ такие, что выполняются неравенства $c_1\,g(n) \leq f(n) \leq c_2\,g(n)$ для всех $n \geq N$.

Запись $f(n) = \Theta(g(n))$ включает в себя две асимптотические оценки: верхнюю и нижнюю. Таким образом, начиная с некоторого $N$ рост функции $f(n)$ полностью соответствует росту функции $g(n)$. В дальнейшем выражению "сложность алгоритма есть $\Theta(g(n))$" будем придавать именно этот смысл.

Отметим, что для функции многих переменных $f(n_1, \ldots, n_k)$ символ $\Lambda(f(n_1, \ldots, n_k))$, $\Lambda \in \{O, \Omega, \Theta\}$ определяется подобным же образом.

Записи с асимптотическими обозначениями $O$, $\Omega$, $\Theta$ очень удобны и часто употребляются в уравнениях, но при этом требуют некоторой осторожности. Дело в том, что используемый знак "=" — это не равенство в обычном смысле, а несимметричное (**!**) отношение включения подмножеств "$\subseteq$". В этом случае с формальной точки зрения запись $O(g(n))$ необходимо рассматривать как множество неотрицательных функций $f(n)$, не превосходящих по порядку функцию $g(n)$, а функцию $f(n)$ — как одноэлементное множество. Таким образом, запись $f(n) = O(g(n))$ означает запись $f(n) \in O(g(n))$, запись $O(n) = O(n^2)$ означает $O(n) \subseteq O(n^2)$ и т. д. В случае арифметических операций над

такими множествами функций $X, Y$ и функцией $f(n)$ множества $X+Y$, $X\,Y$, $f\,X$ определяются следующим естественным образом:

$$X + Y = \{x + y \mid x \in X \,\&\, y \in Y\},$$

$$X\,Y = \{x\,y \mid x \in X \,\&\, y \in Y\},$$

$$f\,X = \{f\,x \mid x \in X\}.$$

Аналогично следует трактовать $\Lambda$-записи в общем случае, когда $\Lambda \in \{O, \Omega, \Theta\}$. Используя введенные определения асимптотических обозначений, легко доказать справедливость свойств, сформулированных в следующем утверждении.

**Теорема 1.** Пусть $f(n)$, $g(n)$, $h(n)$ — неотрицательные функции и $\Lambda \in \{O, \Omega, \Theta\}$. Тогда

(i) $f(n) = \Lambda(f(n))$ (рефлексивность);

(ii) если $f(n) = \Lambda(g(n))$ и $g(n) = \Lambda(h(n))$, то $f(n) = \Lambda(h(n))$ (транзитивность);

(iii) $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$ (обращение);

(iv) $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$ (симметричность);

(v) $f(n) = \Theta(g(n)) \Leftrightarrow (f(n) = O(g(n)) \,\&\, f(n) = \Omega(g(n))$ (эквивалентность);

(vi) $\Lambda(\Lambda(f(n))) = \Lambda(f(n))$ (идемпотентность);

(vii) арифметические операции:

$\Lambda(f(n))\,\Lambda(g(n)) = \Lambda(f(n)\,g(n))$,

$f(n)\,\Lambda(g(n)) = \Lambda(f(n)\,g(n))$,

$\Lambda(f(n)) + \Lambda(g(n)) = \Lambda(f(n) + g(n)) = \Lambda(\max\{f(n), g(n)\})$,

$c\Lambda(f(n)) = \Lambda(cf(n)) = \Lambda(f(n))$, если $c$ — положительная константа.

Условимся, что всюду далее при исследовании асимптотических свойств функций будем использовать символ $c$, возможно с различными индексами, для обозначения положительных констант, не зависящих от аргумента изучаемых функций.

Приведем пример, содержащий несколько иллюстраций введенных понятий и обозначений.

**Пример 1.**

(i) $f(n) = \Theta(n^2)$, где $f(n) = c_1 n + \sum_{i=1}^{n} c_2\, i + c_3$.

Действительно, используя теорему 1 и формулу суммы арифметической прогрессии, получаем $f(n) = O(n) + c_2(n^2 + n)/2 + O(1) = O(n) + O(n^2) + O(n) + O(1) = O(\max\{n, n^2\}) = O(n^2)$. Очевидно, $f(n) \geq c_2\, n^2/2$. Поэтому $f(n) = \Omega(n^2)$. Таким образом, $f(n) = \Theta(n^2)$.

(ii) $c\, n \neq O(\sqrt{n})$.

В самом деле, пусть найдутся положительные константы $c'$ и $N$ такие, что $c\, n \leq c'\sqrt{n}$ для всех $n \geq N$. Тогда $n \leq (c'/c)^{\,2}$ для всех достаточно больших $n$, пришли к очевидному противоречию.

(iii) $c\, n^m \neq \Omega(n^k)$, где $k > m$.

Доказывается аналогично (ii).

(iv) $O(n) + c_1 2^{\,n+c_2} + c_3 = O(2^n)$.

Следует из теоремы 1 и неравенства $2^{\,n} \geq n$.

(v) $H_n = \ln n + \Theta(1) = \Theta(\ln n)$,[2] где

$$H_n = \sum_{i=1}^{n} \frac{1}{i}\,.$$[3]

Для асимптотических оценок гармонических чисел $H_n$ воспользуемся аппроксимациями интеграла произвольной интегрируемой убывающей функции $f(x)$ с помощью конечных сумм:

$$\int_{m}^{n+1} f(x)\, dx \leq \sum_{i=m}^{n} f(i) \leq \int_{m-1}^{n} f(x)\, dx.$$

[2] $\ln n$ — натуральный логарифм.

[3] Конечные суммы $H_n$ известны как *гармонические числа*.

Функция $f(x) = 1/x$ убывает на интервале $(1, +\infty)$. Следовательно,

$$\ln(n+1) - \ln 2 = \int_2^{n+1} \frac{dx}{x} \le \sum_{i=2}^{n} \frac{1}{i} \le \int_1^n \frac{dx}{x} = \ln n.$$

Поэтому $H_n = \ln n + O(1)$ и $H_n \ge \ln n + 1 - \ln 2 = \ln n + \Omega(1)$. Таким образом, $H_n = \ln n + \Theta(1) = \Theta(\ln n)$.

Отметим, что аналогичные сравнения интеграла и конечной суммы для произвольной возрастающей функции приводят к неравенствам

$$\int_{m-1}^{n} f(x)\,dx \le \sum_{i=m}^{n} f(i) \le \int_m^{n+1} f(x)\,dx.$$

Такие оценки используются в нижеприведенных примерах (vi) и (viii).

(vi) $\ln(n!) = \Theta(n \ln n)$.

Действительно, $\ln(n!) = \sum_{i=1}^{n} \ln i$. Отсюда очевидно, $\ln(n!) \le n \ln n = O(n \ln n)$. Для доказательства нижней оценки воспользуемся сравнением интеграла и конечной суммы функции $\ln x$, возрастающей и положительной на интервале $(1, +\infty)$. Имеем

$$\sum_{i=2}^{n} \ln i \ge \int_1^n \ln x\,dx = x \ln x - x \Big|_1^n = n \ln n - n + 1 = \Omega(n \ln n).$$

Следовательно, $\ln(n!) = \Theta(n \ln n)$.

Заметим, что более точные верхнюю и нижнюю оценки величин $n!$ и $\ln(n!)$ можно получить из *формулы Стирлинга*

$$n! = \sqrt{2\,\pi\,n}\left(\frac{n}{e}\right)^n \big(1 + \Theta(1/n)\big).$$

(vii) $\lg(n!) = \Theta(n \lg n)$[4].

Непосредственно следует из примера (vi) и равенства $\lg(n!) = \lg e \, \ln(n!)$.

(viii) $\displaystyle\sum_{i=1}^{n} i \ln i = \frac{n^2}{2} \ln n - \frac{n^2}{4} + O(n \ln n) = O(n^2 \ln n)$.

[4] $\lg n$ — двоичный логарифм.

Действительно, поскольку $f(x) = x \ln x$ — возрастающая функция, как и в примере (v), получаем следующие неравенства:

$$\begin{aligned}\sum_{i=2}^{n-1} i \ln i &\leq \int_2^n x \ln x \, dx = \frac{x^2}{2} \ln x - \frac{x^2}{4} \Big|_2^n = \\ &= \frac{n^2}{2} \ln n - \frac{n^2}{4} - \ln 4 + 1 \leq \frac{n^2}{2} \ln n - \frac{n^2}{4}.\end{aligned}$$

Осталось заметить, что $\sum_{i=1}^{n} i \ln i = n \ln n + \sum_{i=2}^{n-1} i \ln i$.

(ix) $\sum_{i=1}^{\lceil n/c \rceil} \lfloor \lg i \rfloor = \Omega(n \lg n)$[5].

Доказательство требуемой нижней оценки проведём с помощью метода разбиения суммы на части. Для $n \geq 2c$ имеем[6]

$$\begin{aligned}\sum_{i=1}^{\lceil n/c \rceil} \lfloor \lg i \rfloor &= \sum_{i=1}^{\lceil n/2c \rceil - 1} \lfloor \lg i \rfloor + \sum_{i=\lceil n/2c \rceil}^{\lceil n/c \rceil} \lfloor \lg i \rfloor \geq \\ &\geq \sum_{i=\lceil n/2c \rceil}^{\lceil n/c \rceil} \lfloor \lg i \rfloor \geq \sum_{i=\lceil n/2c \rceil}^{\lceil n/c \rceil} \lfloor \lg \lceil n/2c \rceil \rfloor \geq \\ &\geq (\lceil n/c \rceil - \lceil n/2c \rceil + 1)(\lg(n/2c) - 1) \geq \\ &\geq \frac{n}{2c}(\lg n - \lg(2c) - 1) = \\ &= \Omega(n)\,\Omega(\lg n) = \Omega(n \lg n).\end{aligned}$$

Несмотря на то, что основное внимание мы уделяем порядку роста времени работы, нельзя забывать, что бóльший порядок роста сложности алгоритма может иметь меньшую мультипликативную постоянную[7], чем малый порядок роста сложности другого алгоритма. В таком случае алгоритм с быстро растущей сложностью

[5] $\lfloor x \rfloor$ — наибольшее целое число, не превосходящее $x$;
$\lceil x \rceil$ — наименьшее целое число, не меньшее $x$.

[6] Здесь использованы неравенства $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$.

[7] Константа $c$ в определении $O(g(n))$.

может оказаться предпочтительнее для задачи с малой размерностью данных и, возможно, для всех интересующих нас задач, например, с практической точки зрения.

## 3. Алгоритм нахождения $n$-факториального представления числа

Для знакомства с неформальным языком программирования, на котором будем записывать псевдокод, а также с понятием сложности алгоритма и используемой при его анализе асимптотической $\Lambda$-символикой рассмотрим алгоритм нахождения $n$-факториального представления числа (при фиксированном неотрицательном целом $n$).

**Определение 4.** *$n$-факториальным представлением* целого неотрицательного числа $m$ называется последовательность целых чисел $(d_0, d_1, \ldots, d_{n-1})$ такая, что $0 \le d_i \le i$ при $i = 0, 1, \ldots, n-1$ и

$$m = d_{n-1}(n-1)! + d_{n-2}(n-2)! + \ldots + d_0 0!.$$

Из курса алгебры известно, что для произвольного целого неотрицательного числа $m < n!$ существует[8] $n$-факториальное представление числа $m$, причём единственное. Рассмотрим следующую задачу: пусть задано значение $n$, а $m$ может быть любым целым неотрицательным числом, не превосходящим $n!$, требуется найти $n$-факториальное представление числа $m$. Эта задача легко решается с помощью простого алгоритма. Неформально опишем его по шагам.

Шаг 0. Полагаем $d_0 = 0$ и $q_0 = m$.

Шаг 1. Делим $q_0$ на 2, находим остаток от деления $d_1$ и частное от деления $q_1 = \lfloor q_0/2 \rfloor$. При этом имеем $0 \le d_1 < 2$.

[8] Последовательность $(d_0, d_1, \ldots, d_{n-1})$ является записью числа $m$ относительно одной из разновидностей смешанных систем счисления.

Переходим к следующему шагу. К шагу $i \geq 1$ будут определены числа $d_0, d_1, \ldots, d_{i-1}$ и $q_0, q_1, \ldots, q_{i-1}$.

Шаг $i$. Делим $q_{i-1}$ на $i+1$, полагаем $d_i$ — остаток от деления и $q_i = \lfloor q_{i-1}/(i+1) \rfloor$. При этом имеем $0 \leq d_i < i+1$.

Шаг $n-1$. На заключительном шаге находим остаток $d_{n-1}$ от деления $q_{n-2}$ на $n$ и частное $q_{n-1} = \lfloor q_{n-2}/n \rfloor$.

Нетрудно доказать, что последовательность $(d_0, d_1, \ldots, d_{n-1})$, вычисленная посредством этого алгоритма, является $n$-факториальным представлением числа $m$. Заметим, что пошаговый процесс можно закончить, как только будет выполнено равенство $q_i = 0$, все оставшиеся числа $q_j$ при $j > i$ будут также равны 0. Кроме того, можно отказаться от массива из элементов $q_i$, достаточно на каждом шаге записывать требуемое значение элемента $q_i$ в единственную переменную $q$. Таким образом, приходим к следующей программной реализации.

**Алгоритм** $FDecomp(n, m)$ **нахождения**
**$n$-факториального представления числа** $m < n!$

$i := 0$;
$d_0 := 0$;
$q := m$;
**while** $q > 0$ **do**
**begin**
  $i := i + 1$;
  $d_i := q$ **mod** $i$;
  $q := \lfloor q/i \rfloor$
**end**;
$i := i + 1$;
**while** $i < n$ **do** $d_i := 0$.

Чему равна сложность $T(n)$ алгоритма $FDecomp(n, m)$ нахождения $n$-факториального представления для произвольного числа $m < n!$?

Оценим время работы алгоритма. Введем следующие обозначения:

- $c_1$ — число элементарных операций, выполняющихся за одну итерацию первого while-цикла;
- $c_2$ — число элементарных операций, выполняющихся за одну итерацию второго while-цикла;
- $c_3$ — число элементарных операций, выполняющихся при инициализации переменных, вне обоих while-циклов;
- $n_1$ — число итераций первого while-цикла;
- $n_2$ — число итераций второго while-цикла.

Тогда время работы алгоритма $FDecomp(n, m)$ равно $c_1 n_1 + 1 + c_2 n_2 + 1 + c_3$, причём константы $c_1, c_2, c_3$ не зависят от $n$, а числа $n_1, n_2$ определяются в зависимости от $m$ и $n_1 + n_2 = n - 1$. Наибольшее число операций будет выполняться для такого числа $m$, когда $d_i > 0$ для любого $i > 0$. Поэтому в худшем случае (по времени работы алгоритма) $n_1 = n - 1$, $n_2 = 0$ и $T(n) = c_1(n - 1) + 2 + c_3 = \Theta(n)$. Причём и в лучшем случае время работы алгоритма $FDecomp(n, m)$ есть $\Theta(n)$, так как выполняется равенство $n_1 + n_2 = n - 1$.

При определении $n$-факториального представления числа $m$ иногда ограничиваются последовательностью $(d_0, d_1, \ldots, d_k)$, для которой $d_k \neq 0$ и $d_{k+1} = \ldots = d_{n-1} = 0$ при $m > 0$ ($k = 0$, если $m = 0$). В этом случае в алгоритме $FDecomp$ необходимо отказаться от последнего while-цикла, в котором присваивается нулевое значение оставшимся членам последовательности $d_i$. Однако это не улучшит сложность алгоритма, поскольку можно выбрать такое число $m$, что $d_i > 0$ для всех $i$, например, $m = \sum_{i=1}^{n-1} i!$. Таким образом, сложность алгоритма $FDecomp(n, m)$ есть $\Theta(n)$.

# Глава 2
# Генерация комбинаторных объектов

В прикладных задачах часто возникает необходимость порождать все элементы некоторого класса комбинаторных объектов. Такого рода задачи решаются с помощью *алгоритмов генерации*. Наряду с обычным выводом требуемых объектов без повторений, эти алгоритмы позволяют одновременно производить анализ объектов, их обработку, отбор и т. п. В этой главе мы познакомимся с различными алгоритмами генерации перестановок, двоичных векторов, всех подмножеств конечного множества, кодов Грея и сочетаний.

Все рассматриваемые методы систематического порождения комбинаторных объектов будут сводиться к выбору начальной конфигурации, задающей первый генерируемый объект, трансформации полученного объекта в следующий и проверке условия окончания, которое определяет момент прекращения вычислений. При этом особый интерес будут представлять *алгоритмы генерации объектов в порядке минимального изменения*, когда два "соседних" порождаемых объекта различаются в подходящем смысле "минимально".

При рассмотрении класса комбинаторных объектов предполагается, что все его объекты имеют некоторую одинаковую количественную меру, предварительно заданную целочисленным параметром (или набором целочисленных параметров), который передается на вход алгоритма генерации. Нас прежде всего будет интересовать время работы алгоритма, требующееся для порождения всего класса объектов, как функция от размерности входных данных. Мы будем стремиться получить асимптотически наилучший алгоритм генерации. В частности, в некоторых алгоритмах можно порождать множество всех требуемых объектов за время, пропорциональное его мощности (при этом, естественно, не учитывается время для вывода на печать самого комбинаторного объекта). В этом случае алгоритм имеет сложность $O(k)$, где $k$ — число порождаемых объектов. Такой *алгоритм генерации* комби-

наторных объектов в литературе часто называют *линейным*. Хотя это название несёт дуализм[9], оно оправдано, поскольку такие алгоритмы генерации имеют асимптотически наилучшую сложность. Мы также будет придерживаться этой терминологии.

# 1. Перестановки и алгоритмы их порождения

**Определение 5.** *Перестановкой* множества $A$ называется произвольное взаимно-однозначное отображение $\alpha : A \to A$.

Обычно перестановка конечного множества $A$ определяется с помощью таблицы с двумя строками, каждая из которых содержит все элементы множества $A$, причем элемент $\alpha(a)$ помещается под элементом $a$. Например, перестановку $\alpha : A \to A$ множества $A = \{a, b, c, d\}$ такую, что $\alpha(a) = d$, $\alpha(b) = a$, $\alpha(c) = c$ и $\alpha(d) = b$, можно записать в виде следующей таблицы

$$\alpha = \begin{pmatrix} a\,b\,c\,d \\ d\,a\,c\,b \end{pmatrix}.$$

Иногда перестановкой называется вторая строка такой таблицы, а сама функция $\alpha : A \to A$, заданная таблицей, называется *подстановкой*. Поскольку порядок элементов множества $A$ будет всегда зафиксирован, мы используем термин перестановка как для обозначения самой функции $\alpha$, так и для обозначения нижней строки таблицы, определяющей это отображение. Таким образом, в указанном примере перестановка есть последовательность $(d, a, c, b)$. В общем случае для $n$-элементного множества $A$ с зафиксированным порядком элементов $a_1, \ldots, a_n$ перестановка — это произвольная последовательность длины $n$ из различных элементов множества $A$. Так, последовательность $(\alpha_1, \ldots, \alpha_n)$, где все элементы $\alpha_i \in A$ различны, есть перестановка.

Обычно природа элементов множества $A$ несущественна, поэтому без уменьшения общности можно считать, что $A = \{1, 2, \ldots, n\}$ (ина-

[9] Сравните с понятием линейного алгоритма из гл. 1.

че необходимо перейти к номерам элементов). Обозначим множество всех перестановок $n$-элементного множества через $S_n$. На множестве $S_n$ определена операция умножения перестановок $\alpha \circ \beta$ как суперпозиция отображений $\alpha$ и $\beta$: $\alpha \circ \beta(i) = \alpha(\beta(i))$. Вообще говоря, эта операция не коммутативна, т. е. $\alpha \circ \beta \neq \beta \circ \alpha$. При этом выполняются следующие аксиомы группы:

- $\forall \alpha \in S_n \forall \beta \in S_n \forall \gamma \in S_n \ \alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma$ (ассоциативность);
- $\exists\, e \in S_n \, \forall \alpha \in S_n \ (\alpha \circ e = e \circ \alpha = \alpha)$ (существование *единичного элемента* $e$);
- $\forall \alpha \in S_n \, \exists\, \alpha^{-1} \in S_n \ (\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = e)$ (существование *обратного элемента* $\alpha^{-1}$).

*Тождественная перестановка* $e$ является единичным элементом, а для

$$e = \begin{pmatrix} 1\,2\,\ldots\,n \\ 1\,2\,\ldots\,n \end{pmatrix}$$

нахождения обратной перестановки $\alpha^{-1}$ достаточно сначала поменять местами строки в таблице, определяющей перестановку $\alpha$, а затем упорядочить столбцы в порядке возрастания по верхним элементам. Таким образом, множество перестановок $S_n$ образует группу относительно операции умножения $\circ$, называемую *симметрической группой степени* $n$. Её порядок, т. е. число элементов множества $S_n$, равен $n!$.

Рассмотрим задачу генерации всех перестановок $n$-элементного множества. Возникновение этой задачи относят к началу XVII в., когда в Англии зародилось особое искусство колокольного боя, основанного, если говорить упрощённо, на выбивании на $n$ различных колоколах всех $n!$ перестановок. Перестановки эти следовало "выбивать по памяти", что способствовало разработке сторонниками этого искусства первых простых методов систематического перечисления всех перестановок без повторений. В Книге рекордов Гиннеса содержится упоминание о вы-

бивании всех 8!=40320 перестановок на 8 колоколах в 1963 г., на это потребовалось 17 часов 58,5 минут.

Далее в этом разделе мы познакомимся с несколькими алгоритмами генерации всех перестановок $n$-элементного множества. Сначала определим лексикографический порядок на множестве $S_n$, индекс перестановки относительно этого порядка и вектор инверсии. Затем рассмотрим алгоритмы порождения перестановок, связанные с этими понятиями. Наконец, детально изучим линейный алгоритм генерации перестановок в лексикографическом порядке и линейный алгоритм Джонсона – Троттера генерации перестановок в порядке минимального изменения.

## 1.1. Индекс перестановки

На множестве всех перестановок $n$-элементного множества определим бинарное отношение $\preceq$ следующим образом:

$$(\alpha_1, \alpha_2, \ldots, \alpha_n) \preceq (\beta_1, \beta_2, \ldots \beta_n) \Leftrightarrow \exists\, k \geq 1\, (\alpha_k < \beta_k \;\&\; \forall i < k\, (\alpha_i = \beta_i)).$$

Очевидно, что отношение $\preceq$ удовлетворяет следующим аксиомам:

- $\forall \alpha\ (\alpha \preceq \alpha)$ (рефлексивность);
- $\forall \alpha \forall \beta\ (\alpha \preceq \beta\ \&\ \beta \preceq \alpha \Rightarrow \alpha = \beta)$ (антисимметричность);
- $\forall \alpha \forall \beta \forall \gamma\ (\alpha \preceq \beta\ \&\ \beta \preceq \gamma \Rightarrow \alpha \preceq \gamma)$ (транзитивность);
- $\forall \alpha \forall \beta\ (\alpha \preceq \beta\ \vee\ \beta \preceq \alpha)$ (сравнимость).

Таким образом, отношение $\preceq$ есть линейный порядок на множестве $S_n$. Такой порядок называется *лексикографическим*[10]. Например, последовательность перестановок из $S_3$, записанная в лексикографическом порядке, имеет вид $123, 132, 213, 231, 312, 321$ (здесь перестановки перечислены в порядке возрастания получающихся чисел).

[10] В общем случае лексикографический порядок определяется на множестве $A^n$ слов конечного алфавита $A$ (с заданным упорядочиванием букв) фиксированной длины $n$.

Ясно, что тождественная перестановка $(1, 2, \ldots, n)$ есть наименьший элемент в $(S_n, \preceq)$ (относительно порядка $\preceq$), а перестановка $(n, n-1,$ $n-2, \ldots, 1)$ — наибольший элемент.

В дальнейшем будем использовать следующие очевидные свойства перестановок. Рассмотрим различные элементы $i_1, \ldots, i_k \in \{1, \ldots, n\}$ и множество $S(i_1, \ldots, i_k)$ всех перестановок $k$-элементного множества $\{i_1, \ldots, i_k\}$. На множестве $S(i_1, \ldots, i_k)$ можно также рассмотреть лексикографический порядок $\preceq$.

**Лемма 1.**

(i) $(S(i_1, \ldots, i_k), \preceq)$ — линейно упорядоченное множество.

(ii) Если $\alpha_1, \ldots, \alpha_k \in \{i_1, \ldots, i_k\}$ и $\alpha_1 > \alpha_2 > \ldots > \alpha_k$, то перестановки $(\alpha_1, \alpha_2, \ldots, \alpha_k)$ и $(\alpha_k, \alpha_{k-1}, \ldots, \alpha_1)$ являются наибольшим и наименьшим элементами линейно упорядоченного множества $(S(i_1, \ldots, i_k), \preceq)$ соответственно.

(iii) Если перестановки $\alpha, \beta, \gamma \in S_n$ имеют общее начало длины $k$ и $\alpha \preceq \gamma \preceq \beta$, то $\alpha_{k+1} \leq \gamma_{k+1} \leq \beta_{k+1}$ и $(\alpha_{k+1}, \ldots, \alpha_n) \preceq (\gamma_{k+1}, \ldots, \gamma_n) \preceq (\beta_{k+1}, \ldots, \beta_n)$.

Рассмотрим метод вычисления номера заданной перестановки в последовательности всех перестановок из $S_n$, записанных в лексикографическом порядке, т. е. установим соответствие между целыми числами $0, 1, 2, \ldots, n! - 1$ и $n!$ перестановками из $S_n$. Для этого индукцией по $n$ определим отображение

$$\mathcal{I}_n : S_n \to \{0, 1, \ldots, n! - 1\}.$$

При $n = 1$ полагаем $\mathcal{I}_1(\alpha) = 0$, где $\alpha = (1)$ и $S_1 = \{\alpha\}$. Пусть отображения $\mathcal{I}_i : S_i \to \{0, 1, \ldots, i! - 1\},\ i = 1, 2, \ldots, n-1$ уже определены. Для произвольной перестановки $\alpha = (\alpha_1, \ldots, \alpha_n) \in S_n$ полагаем

$$\mathcal{I}_n(\alpha) = (\alpha_1 - 1)(n-1)! + \mathcal{I}_{n-1}(\alpha'),$$

где $\alpha'$ — последовательность $n-1$ элементов, полученная из перестанов-

ки $\alpha$ удалением $\alpha_1$ и уменьшением на единицу всех элементов, больших $\alpha_1$. Нетрудно доказать, что $\alpha' \in S_{n-1}$ и $\mathcal{I}_n(\alpha) \leq n! - 1$.

**Теорема 2.** Отображение $\mathcal{I}_n : S_n \to \{0, 1, \ldots, n! - 1\}$ есть изоморфизм линейно упорядоченных множеств $(S_n, \preceq)$ и $(\{0, 1, \ldots, n! - 1\}, \leq)$.

**Доказательство** проведем индукцией по $n$. Базис индукции при $n = 1$ очевиден. Предположим, что утверждение теоремы верно для $n - 1$, и докажем его для $n$. Конечные множества $S_n$ и $\{0, 1, \ldots, n! - 1\}$ равномощны. Поэтому если мы покажем, что $\mathcal{I}_n$ является разнозначным отображением, то $\mathcal{I}_n$ есть биекция. Пусть $\alpha, \beta \in S_n$ и $\mathcal{I}_n(\alpha) = \mathcal{I}_n(\beta)$. Из определения отображения $\mathcal{I}_n$ имеем

$$\mathcal{I}_n(\alpha) = (\alpha_1 - 1)(n-1)! + \mathcal{I}_{n-1}(\alpha'),$$

$$\mathcal{I}_n(\beta) = (\beta_1 - 1)(n-1)! + \mathcal{I}_{n-1}(\beta').$$

Так как $\mathcal{I}_{n-1}(\alpha') < (n-1)!$ и $\mathcal{I}_{n-1}(\beta') < (n-1)!$, числа $\mathcal{I}_{n-1}(\alpha')$ и $\mathcal{I}_{n-1}(\beta')$ имеют некоторые $(n-1)$-факториальные представления $(d_0^{\alpha'}, \ldots, d_{n-2}^{\alpha'})$ и $(d_0^{\beta'}, \ldots, d_{n-2}^{\beta'})$ соответственно. Тогда последовательность $(d_0^{\alpha'}, \ldots, d_{n-2}^{\alpha'}, \alpha_1 - 1)$ является $n$-факториальным представлением числа $\mathcal{I}_n(\alpha)$, а последовательность $(d_0^{\beta'}, \ldots, d_{n-2}^{\beta'}, \beta_1 - 1)$ есть $n$-факториальное представление числа $\mathcal{I}_n(\beta)$. Из единственности такого представления получаем $\alpha_1 = \beta_1$. Следовательно, $\mathcal{I}_{n-1}(\alpha') = \mathcal{I}_{n-1}(\beta')$ и $\alpha' = \beta'$ в силу индукционного предположения. Поэтому $\alpha = \beta$. Таким образом, $\mathcal{I}_n$ есть биекция.

Докажем, что $\mathcal{I}_n$ — изоморфизм. Достаточно показать, что если $\alpha, \beta \in S_n$ и $\alpha \preceq \beta$, то $\mathcal{I}_n(\alpha) \leq \mathcal{I}_n(\beta)$ (т. е. $\mathcal{I}_n$ сохраняет порядок). Пусть $\alpha \preceq \beta$. Тогда $\alpha_1 \leq \beta_1$. Если $\alpha_1 \neq \beta_1$, то $\mathcal{I}_n(\alpha) \leq (\beta_1 - 2)(n-1)! + (n-1)! - 1 = (\beta_1 - 1)(n-1)! - 1 \leq \mathcal{I}_n(\beta)$. Пусть теперь $\alpha_1 = \beta_1$. Тогда $\alpha' \preceq \beta'$, и по индукционному предположению имеем $\mathcal{I}_{n-1}(\alpha') \leq \mathcal{I}_{n-1}(\beta')$. Следовательно, $\mathcal{I}_n(\alpha) \leq \mathcal{I}_n(\beta)$. Теорема 2 доказана.

**Определение 6.** Пусть $\alpha \in S_n$. Целое неотрицательное число $\mathcal{I}_n(\alpha)$ называется *индексом перестановки* $\alpha$.

Теорема 2 показывает, что $\mathcal{I}_n$ является нумерацией всех перестановок из $S_n$, упорядоченных в лексикографическом порядке, а индекс $\mathcal{I}_n(\alpha)$ есть номер перестановки $\alpha \in S_n$ в этой последовательности.

Индуктивное определение индекса перестановки $\alpha \in S_n$ фактически задаёт $n$-факториальное представление $(0, \ldots, \alpha'_1 - 1, \alpha_1 - 1)$ числа $\mathcal{I}_n(\alpha)$, причём по $n$-факториальному представлению $(d_0, \ldots, d_{n-1})$ индекса $\mathcal{I}_n(\alpha)$ также восстанавливается и сама перестановка $\alpha = (\alpha_1, \ldots, \alpha_n)$. Опустим формализацию процедуры $PConstrF(d_0, \ldots, d_{n-1})$, осуществляющей такое восстановление перестановки. Теперь мы можем порождать все перестановки из $S_n$ в лексикографическом порядке следующим образом: изменяя индекс $i$ от 0 до $n!-1$, находим $n$-факториальное представление числа $i$ и по нему восстанавливаем перестановку.

**Алгоритм $PIndex(n)$ генерации всех перестановок из $S_n$ по индексам**

**for** $i = 0$ **to** $n! - 1$ **do**
**begin** % нахождение $n$-факториального
 $FDecomp(n, i)$; % представления $(d_0, \ldots, d_{n-1})$ числа $i$;
 $PConstrF(d_0, \ldots, d_{n-1})$; % восстановление перестановки $\alpha$ по
 **write**$(\alpha_1, \ldots, \alpha_n)$ % $n$-факториальному представлению
**end**.

Время работы алгоритма $PIndex(n)$ есть $\Omega(nn!)$, так как количество итераций for-цикла равно $n!$, и алгоритм $FDecomp(n, i)$ требует времени $\Theta(n)$. Такая сложность не является оптимальной, в следующем разделе мы познакомимся с линейным алгоритмом генерации всех перестановок из $S_n$ в лексикографическом порядке.

## 1.2. Генерация перестановок в лексикографическом порядке

Будем говорить, что перестановка $\beta \in S_n$ *непосредственно следует* за перестановкой $\alpha \in S_n$ относительно лексикографического порядка $\preceq$, если выполняются следующие условия:

- $\alpha \prec \beta$[11],
- не существует такой перестановки $\gamma \in S_n$, что $\alpha \prec \gamma \prec \beta$.

При генерации перестановок в лексикографическом порядке, начиная с тождественной перестановки $(1, 2, \ldots, n)$, требуется переходить от уже построенной перестановки $\alpha = (\alpha_1, \ldots, \alpha_n)$ к непосредственно следующей за ней перестановке $\beta = (\beta_1, \ldots, \beta_n)$ до тех пор, пока не получим наибольшую перестановку $(n, n-1, \ldots, 1)$ (относительно лексикографического порядка).

Рассмотрим способ построения такой перестановки $\beta$. Просматриваем справа налево перестановку $\alpha = (\alpha_1, \ldots, \alpha_n)$ в поисках самой правой позиции $i$ такой, что $\alpha_i < \alpha_{i+1}$. Если такой позиции нет, то $\alpha_1 > \alpha_2 > \ldots > \alpha_n$, т. е. $\alpha = (n, n-1, \ldots, 1)$ и генерировать больше нечего. Поэтому считаем, что такая позиция $i$ есть. Значит, $\alpha_i < \alpha_{i+1} > \alpha_{i+2} > \ldots > \alpha_n$. Далее ищем первую позицию $j$ при переходе от позиции $n$ к позиции $i$ такую, что $\alpha_i < \alpha_j$. Тогда $i < j$. Затем меняем местами элементы $\alpha_i$ и $\alpha_j$, а в полученной перестановке $\alpha' = (\alpha'_1, \ldots, \alpha'_n)$ отрезок $\alpha'_{i+1}, \ldots, \alpha'_{n-1}, \alpha'_n$ переворачиваем. Построенную перестановку обозначим через $\beta$.

Например, пусть $\alpha = (2, 6, 5, 8, 7, 4, 3, 1)$. Тогда $\alpha_i = 5$ и $\alpha_j = 7$. Поменяем местами эти элементы, перевернём отрезок $(8, 5, 4, 3, 1)$ и получим перестановку $\beta = (2, 6, 7, 1, 3, 4, 5, 8)$.

**Лемма 2.** Перестановка $\beta$ непосредственно следует за перестановкой $\alpha$ относительно лексикографического порядка.

[11] $\alpha \prec \beta$, если $\alpha \preceq \beta$ и $\alpha \neq \beta$.

**Доказательство.** В силу построения $\beta_s = \alpha'_s = \alpha_s$ для любой позиции $s < i$. Так как $\beta_i = \alpha'_i = \alpha_j > \alpha_i$, то $\alpha \preceq \beta$.

Предположим, что $\alpha \preceq \gamma \preceq \beta$, и покажем, что $\gamma = \alpha$ или $\gamma = \beta$. Так как $\beta_s = \alpha_s$ для всех $s < i$, то из определения лексикографического порядка получаем $\gamma_s = \alpha_s$ при $s < i$. Тогда $\alpha_i \leq \gamma_i \leq \beta_i = \alpha_j$ в силу леммы 1(iii). Предположим, что $\alpha_i \neq \gamma_i$. Тогда $\gamma_i \in \{\alpha_{i+1}, \ldots, \alpha_n\}$. Но $\alpha_j > \alpha_i < \alpha_{i+1} > \alpha_{i+2} > \ldots > \alpha_n$. Следовательно, $\gamma_i \geq \alpha_j$ в силу выбора $j$. Поэтому $\gamma_i = \alpha_j$. Таким образом, $\alpha_i = \gamma_i$ или $\gamma_i = \beta_i$.

Случай 1. $\alpha_i = \gamma_i$. Тогда $(\alpha_{i+1}, \ldots, \alpha_n) \preceq (\gamma_{i+1}, \ldots, \gamma_n)$ по лемме 1(iii). В силу леммы 1(ii) имеем $(\gamma_{i+1}, \ldots, \gamma_n) \preceq (\alpha_{i+1}, \ldots, \alpha_n)$. Поэтому по лемме 1(i) получаем $(\alpha_{i+1}, \ldots, \alpha_n) = (\gamma_{i+1}, \ldots, \gamma_n)$. Таким образом, справедливо равенство $\alpha = \gamma$.

Случай 2. $\gamma_i = \beta_i$. Покажем, что $\alpha'_{i+1}, \ldots, \alpha'_n$ — убывающая последовательность. Действительно, последовательность $\alpha_{i+1}, \alpha_{i+2}, \ldots, \alpha_n$ убывает в силу выбора $i$. Причём $\alpha'_j = \alpha_i$, $j > i$ и $\alpha'_s = \alpha_s$ для всех $s$ таких, что $s > i$ и $s \neq j$. В силу выбора позиции $j$ получаем

$$\alpha'_j = \alpha_i \geq \alpha_{j+1} = \alpha'_{j+1}, \text{ если } j < n;$$
$$\alpha'_{j-1} = \alpha_{j-1} > \alpha_j > \alpha_i = \alpha'_j, \text{ если } j > i+1.$$

Следовательно, $\alpha'_{i+1}, \ldots, \alpha'_n$ — убывающая последовательность. После переворота этой последовательности получим возрастающую последовательность $\beta_{i+1}, \ldots, \beta_n$. Тогда $(\beta_{i+1}, \ldots, \beta_n) \preceq (\gamma_{i+1}, \ldots, \gamma_n)$ по лемме 1(ii) и $(\gamma_{i+1}, \ldots, \gamma_n) \preceq (\beta_{i+1}, \ldots, \beta_n)$ по лемме 1(iii). Поэтому $(\beta_{i+1}, \ldots, \beta_n) = (\gamma_{i+1}, \ldots, \gamma_n)$ по лемме 1(i). Таким образом, $\beta = \gamma$. Лемма 2 доказана.

Перейдем к рассмотрению алгоритма генерации перестановок, в котором применяется описанный способ перестроения перестановки в непосредственно следующую за ней. При программной реализации этого алгоритма используется массив $\alpha$ размерности $n + 1$. В $\alpha_1, \ldots, \alpha_n$ записываем текущую порождаемую перестановку из $S_n$, первоначально тождественную. Значение $\alpha_0$ не изменяется и равно 0, поэтому всегда справедливо неравенство $\alpha_0 < \alpha_1$. Это неравенство гарантирует нахож-

дение самой правой позиции $i \geq 0$ такой, что $\alpha_i < \alpha_{i+1}$. Алгоритм заканчивает работу, когда значение $i$ становится равным 0.

**Алгоритм $PLex(n)$ генерации всех перестановок в лексикографическом порядке**

**for** $j = 0$ **to** $n$ **do** $\alpha_j := j$;
$i := 1$;
**while** $i \neq 0$ **do**
**begin**
  **write** $(\alpha_1, \ldots, \alpha_n)$;
  $i := n - 1$;
  **while** $\alpha_i > \alpha_{i+1}$ **do** $i := i - 1$;
  $j := n$;
  **while** $\alpha_j < \alpha_i$ **do** $j := j - 1$;
  $Swap(\alpha_i, \alpha_j)$;
  $k := i + 1$;
  $m := i + \lfloor (n - i)/2 \rfloor$;
  **while** $k \leq m$ **do**
  **begin**
    $Swap(\alpha_k, \alpha_{n-k+i+1})$;
    $k := k + 1$
  **end**
**end**.

**Пример 2.** При $n = 3$ процесс работы алгоритма $PLex(n)$ генерации перестановок из $S_3$ в лексикографическом порядке представлен следующей последовательностью перестроений перестановок $\alpha^i$:

$\alpha^1 = (1, 2, 3)$, $\alpha_i^1 = 2$, $\alpha_j^1 = 3$;
$\alpha^2 = (1, 3, 2)$, $\alpha_i^2 = 1$, $\alpha_j^2 = 2$;
$\alpha^3 = (2, 1, 3)$, $\alpha_i^3 = 1$, $\alpha_j^2 = 3$;
$\alpha^4 = (2, 3, 1)$, $\alpha_i^4 = 2$, $\alpha_j^4 = 3$;
$\alpha^5 = (3, 1, 2)$, $\alpha_i^5 = 1$, $\alpha_j^5 = 2$;
$\alpha^6 = (3, 2, 1)$, $i = 0$.

**Теорема 3.** Алгоритм $PLex(n)$ корректен и строит все перестановки из $S_n$ без повторений в лексикографическом порядке за время $O(n!)$.

**Доказательство.** Используя лемму 2, нетрудно обосновать корректность алгоритма $PLex(n)$.

Оценим сложность $T(n)$ алгоритма $PLex(n)$. Рассмотрим разбиение множества $S_n$ на $n$ подмножеств $S_n^k$, $k = 1, \ldots, n$. Множество $S_n^k$ состоит из всех перестановок, на первом месте которых стоит число $k$. Тогда $S_n^k$ содержит $(n-1)!$ перестановок. Относительно лексикографического порядка каждая перестановка из $S_n^k$ предшествует произвольной перестановке из $S_n^m$ при $k < m$, а упорядочивание на $S_n^k$ соответствует лексикографическому упорядочиванию множества всех перестановок $S(\{1, \ldots, n\} \setminus \{k\})$. Таким образом, последовательность $\alpha^1, \alpha^2, \ldots, \alpha^{n!}$ всех перестановок из $S_n$, упорядоченных лексикографически, разбивается на следующие $n$ блоков:

$$\underbrace{(1,\ldots)\cdots(1,\ldots)}_{S_n^1}\cdots\underbrace{(k,\ldots)\cdots(k,\ldots)}_{S_n^k}\cdots\underbrace{(n,\ldots)\cdots(n,\ldots)}_{S_n^n}\ .$$

В силу леммы 1(ii) перестановки $(k, 1, 2, \ldots, k-1, k+1, \ldots, n)$ и $(k, n, n-1, \ldots, k+1, k-1, \ldots, 1)$ являются первой и последней перестановкой из $S_n^k$ соответственно. Введём следующие обозначения:

- $t_0^n$ — число операций, выполняемых в алгоритме $PLex(n)$ до печати перестановки $\alpha^1$;
- $t_i^n$ — число операций, выполняемых в алгоритме $PLex(n)$, начиная с печати $\alpha^i$ и до печати $\alpha^{i+1}$;
- $t_{n!}^n$ — число операций, выполняемых в алгоритме $PLex(n)$, начиная с печати $\alpha^{n!}$ и до окончания работы программы.

Тогда из алгоритма $PLex(n)$ получаем

$$\begin{aligned} T(n) &= \sum_{i=0}^{n!} t_i^n = t_0^n + \sum_{i=1}^{n!} t_i^n = O(n) + \sum_{k=1}^{n} \sum_{i:\,\alpha^i \in S_n^k} t_i^n = \\ &= O(n) + \sum_{k=1}^{n} (t_{k(n-1)!}^n + \sum_{i:\,\alpha^i \in S_n^k \setminus \{\alpha^{k(n-1)!}\}} t_i^n), \end{aligned}$$

$$\sum_{i:\,\alpha^i \in S_n^k \setminus \{\alpha^{k(n-1)!}\}} t_i^n = T(n-1) - c_1 n,$$

где константа $c_1$ не зависит от $n$. Подсчитаем $t_{k(n-1)!}^n$ — число операций начиная с печати последней перестановки $(k, n, n-1, \ldots, k+1, k-1, \ldots, 1)$ из $k$-го блока $S_n^k$ до печати первой перестановки $(k+1, 1, 2, \ldots, k, k+2, \ldots, n)$ из $(k+1)$-го блока $S_n^{k+1}$. В этом случае будем менять местами числа $k$ и $k+1$, а затем переворачивать последовательность $n, n-1, \ldots, k+2, k, k-1, \ldots, 1$. Теперь понятно, что

$$t_{k(n-1)!}^n = c_2 n + c_3 k,$$

где константы $c_2, c_3$ не зависят от $n$ и $k$. Следовательно,

$$\sum_{k=1}^{n} t_{k(n-1)!}^n = \sum_{k=1}^{n} (c_2 n + c_3 k) = O(n^2).$$

Таким образом, получаем

$$\begin{aligned} T(n) &= O(n) + O(n^2) + \sum_{k=1}^{n} (\, T(n-1) - c_1 n) = \\ &= n\, T(n-1) + O(n^2). \end{aligned}$$

Поскольку мы исследуем асимптотическую оценку сложности алгоритма $PLex(n)$, можно считать, что $T(n) = n\, T(n-1) + c\, n^2$ для некоторой константы $c$. Решим это рекуррентное соотношение. Сделаем замену

$$T(n) = P_n - c\, n.$$

Тогда $P_n = n\,P_{n-1} + 2\,c\,n$. Следовательно,

$$\frac{P_n}{n!} = \frac{P_{n-1}}{(n-1)!} + \frac{2\,c}{(n-1)!}\ .$$

Поэтому при $n > 1$ имеем

$$\frac{P_n}{n!} = P_1 + \sum_{i=1}^{n-1} \frac{2\,c}{i!} = O(1).$$

Таким образом, справедливы соотношения $P_n = O(n!)$ и $T(n) = O(n!)$. Теорема 3 доказана.

## 1.3. Порождение перестановок через векторы инверсий

Пусть $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ есть перестановка из $S_n$.

**Определение 7.** Пара $(\alpha_i, \alpha_j)$ называется *инверсией перестановки* $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$, если $i < j$ и $\alpha_i > \alpha_j$.

Инверсию $(\alpha_i, \alpha_j)$ будем также называть *$j$-инверсией* перестановки $\alpha$, тем самым явно указывая номер меньшего элемента в инверсии $(\alpha_i, \alpha_j)$.

**Определение 8.** *Вектором инверсий перестановки* $\alpha \in S_n$ называется последовательность целых чисел $(d_1, d_2, \ldots, d_n)$ такая, что $d_j$ есть число $j$-инверсий перестановки $\alpha$.

Другими словами, $d_j$ есть число элементов перестановки $\alpha$, бóльших $\alpha_j$ и находящихся левее $\alpha_j$ в последовательности $(\alpha_1, \ldots, \alpha_n)$,

$$d_j = \mid \{\alpha_i \mid i < j \ \&\ \alpha_i > \alpha_j\} \mid \ [12].$$

Например, для перестановки $(4, 3, 5, 2, 1, 7, 8, 6, 9)$ вектором инверсий будет вектор $(0, 1, 0, 3, 4, 0, 0, 2, 0)$.

[12] $|X|$ — мощность множества $X$.

Вектор инверсий содержит информацию о структуре "беспорядка" перестановки. Такая информация оказывается полезной при разработке различных алгоритмов обработки данных. Именно этим объясняется интерес к алгоритму генерации всех перестановок $n$-элементного множества через векторы инверсий, хотя сложность такого алгоритма, как будет показано далее, хуже сложности рассмотренного алгоритма $PLex(n)$ генерации перестановок в лексикографическом порядке.

Пусть $(d_1, \ldots, d_n)$ — вектор инверсий перестановки $\alpha \in S_n$. Очевидно, $0 \leq d_j < j$. Определим отображение $\mathcal{V}_n : S_n \to D_n$, полагая $\mathcal{V}_n(\alpha) = (d_1, \ldots, d_n)$, где $D_n = \{(d_1, \ldots, d_n) \,|\, 0 \leq d_j < j,\, j = 1, \ldots, n\}$.

**Теорема 4.** Отображение $\mathcal{V}_n : S_n \to D_n$ является биекцией, причём любая перестановка $\alpha \in S_n$ однозначно восстанавливается по её вектору инверсий $\mathcal{V}_n(\alpha)$.

**Доказательство.** Очевидно, что множества $D_n$, $S_n$ равномощны и содержат $n!$ элементов. Поэтому достаточно показать, что $\mathcal{V}_n$ является сюръективным отображением. Опишем метод, как для произвольного вектора $d = (d_1, \ldots, d_n) \in D_n$ построить перестановку $\alpha = (\alpha_1, \ldots, \alpha_n) \in S_n$ такую, что $\mathcal{V}_n(\alpha) = d$, тем самым всё будет доказано.

Действительно, определим множество $I_n = \{1, 2, \ldots, n\}$. Расположим его элементы в порядке возрастания и $(d_n + 1)$-й элемент с конца этой возрастающей последовательности обозначим через $\alpha_n$. Тогда среди чисел из $I_n$ имеется ровно $d_n$ чисел, больших $\alpha_n$. Поэтому в произвольной перестановке $\alpha$ вида $\alpha = (\ldots, \alpha_n)$ число $n$-инверсий будет равно $d_n$. Далее определим множество $I_{n-1} = I_n \setminus \{\alpha_n\}$, т. е. вычеркнем $\alpha_n$. Расположим его элементы в порядке возрастания и $(d_{n-1} + 1)$-й элемент с конца этой возрастающей последовательности обозначим через $\alpha_{n-1}$. Тогда среди чисел из множества $I_{n-1}$ имеется ровно $d_{n-1}$ чисел, больших $\alpha_{n-1}$. Следовательно, в произвольной перестановке $\alpha = (\ldots, \alpha_{n-1}, \alpha_n)$ чисел, больших $\alpha_{n-1}$ и не равных $\alpha_n$, будет

ровно $d_{n-1}$. Поэтому в перестановке $\alpha$ число $(n-1)$-инверсий есть $d_{n-1}$ и число $n$-инверсий равно $d_n$.

На шаге $i$ уже будут определены множество $I_{i+1}$ и вычеркнутые из множества $\{1, 2, \ldots, n\}$ элементы $\alpha_n, \alpha_{n-1}, \ldots, \alpha_{i+1}$. Полагаем

$$I_i = I_{i+1} \setminus \{\alpha_{i+1}\} = I_n \setminus \{\alpha_n, \alpha_{n-1}, \ldots, \alpha_{i+1}\}.$$

Далее снова расположим элементы множества $I_i$ в порядке возрастания и $(d_i + 1)$-й элемент с конца этой возрастающей последовательности обозначим через $\alpha_i$. Тогда в произвольной перестановке $\alpha$ вида $(\ldots, \alpha_i, \alpha_{i+1}, \ldots, \alpha_n)$ число $j$-инверсий будет равно $d_j$ для всех $j \geq i$. Продолжим этот процесс до шага $i = 1$. В итоге получим перестановку $\alpha = (\alpha_1, \ldots, \alpha_n)$, число $i$-инверсий которой равно $d_i$ для любого $i$. Теорема 4 доказана.

**Пример 3.** Процесс восстановления перестановки $\alpha = (\alpha_1, \ldots, \alpha_5)$, имеющей вектор инверсий $d = (0, 0, 2, 1, 1)$, выглядит так:

$i = 5$, $I_5 = \{1, 2, 3, 4, 5\}$, $d_5 + 1 = 2$, $\alpha_5 = 4$;
$i = 4$, $I_4 = \{1, 2, 3, 5\}$, $d_4 + 1 = 2$, $\alpha_4 = 3$;
$i = 3$, $I_3 = \{1, 2, 5\}$, $d_3 + 1 = 3$, $\alpha_3 = 1$;
$i = 2$, $I_2 = \{2, 5\}$, $d_2 + 1 = 1$, $\alpha_2 = 5$;
$i = 1$, $I_1 = \{2\}$, $d_1 + 1 = 1$, $\alpha_1 = 2$;
$\alpha = (2, 5, 1, 3, 4)$.

Формализуем данный алгоритм в виде псевдокода. Очевидным образом можно сэкономить используемую память и уменьшить число операций, отказавшись от создания множества $I_i$ и упорядочивания его элементов на каждом шаге $i$. Вместо этого достаточно хранить только лишь метки для уже вычеркнутых чисел из множества $\{1, 2, \ldots, n\}$ и добавлять на шаге $i$ метку для числа $\alpha_i$. Будем использовать массив $\phi$ размерности $n$ с булевыми значениями его элементов true и false для создания таких меток.

**Алгоритм** $PConstrV(d_1, \ldots, d_n)$ **восстановления перестановки** $\alpha = (\alpha_1, \ldots, \alpha_n)$ **по её вектору инверсий** $d = (d_1, \ldots, d_n)$

**for** $k = 1$ **to** $n$ **do** $\phi_k :=$ **true**;
**for** $i = n$ **downto** 1 **do**
**begin**
  $j := 1$;
  $m := n$;
  **while** $j \le d_i$ **or** $\phi_m =$ **false do**
  **begin**
    **if** $\phi_j =$ **true then** $j := j + 1$;
    $m := m - 1$
  **end**;
  $\alpha_i := m$;
  $\phi_m :=$ **false**
**end**.

Оценим сложность $T(n)$ алгоритма $PConstrV(d_1, \ldots, d_n)$. Число итераций for-циклов равно $n$. Поскольку число итераций while-цикла не превосходит n, имеем $T(n) \le n + n(c_1 + nc_2 + c_3) = O(n^2)$. Значит, $T(n) = O(n^2)$. Докажем, что $T(n) = \Theta(n^2)$. Действительно, рассмотрим вектор $d = (0, 1, \ldots, n-1)$, являющийся вектором инверсий перестановки $\alpha = (n, n-1, \ldots, 2, 1)$. Из алгоритма следует, что $I_n = \{1, \ldots, n\}$, $I_{n-1} = \{2, \ldots, n\}, \ldots, I_i = \{n - i + 1, \ldots, n\}$ и $\phi_{n-i+1} = \ldots = \phi_n = true$. Поэтому для любого $i = n, n-1, \ldots, 1$ число итераций while-цикла (при фиксированном $i$) равно $d_i = i - 1$. Следовательно,

$$T(n) \ge n + \sum_{i=n}^{1}(c_1 + d_i c_2 + c_3) = \Omega(n^2).$$

Таким образом, сложность алгоритма $PConstrV(d_1, \ldots, d_n)$ есть $\Theta(n^2)$.

Алгоритм $PConstrV(d_1, \ldots, d_n)$ восстановления перестановки по её вектору инверсий даёт очевидный способ генерации всех перестановок

из $S_n$. Сначала порождаем вектор инверсий $d = (d_1, \ldots, d_n) \in D_n$, а затем по вектору инверсий $d$ восстанавливаем перестановку $\alpha = (\alpha_1, \ldots, \alpha_n) \in S_n$. По теореме 4 множество векторов инверсий всех перестановок из $S_n$ есть $D_n$. С другой стороны, $D_n$ есть множество $n$-факториальных представлений всех целых неотрицательных чисел, меньших $n!$. Таким образом, приходим к следующему алгоритму генерации перестановок.

**Алгоритм $PVInv(n)$ генерации всех перестановок через векторы инверсий**

**for** $i = 0$ **to** $n! - 1$ **do**
**begin** % нахождение $n$-факториального
$FDecomp(n, i)$; % представления $(d_1, \ldots, d_n)$ числа $i$;
$PConstrV(d_1, \ldots, d_n)$; % восстановление перестановки $\alpha$ по
**write**$(\alpha_1, \ldots, \alpha_n)$ % вектору инверсий $(d_1, \ldots, d_n)$
**end**.

Очевидно, что время работы алгоритма $PVInv(n)$ генерации перестановок через векторы инверсий есть $\Omega(nn!)$. Поэтому линейный алгоритм $PLex(n)$ генерации перестановок в лексикографическом порядке предпочтительнее. Однако алгоритмы, использующие векторы инверсий, представляют интерес, поскольку информация, содержащаяся в векторе инверсий, настолько богата, что позволяет полностью восстановить саму перестановку.

## 1.4. Алгоритм Джонсона – Троттера генерации перестановок

Мы рассмотрели несколько алгоритмов генерации перестановок из $S_n$. При этом переход от предыдущей перестановки к следующей требовал, вообще говоря, большого числа перемещений элементов исходной перестановки. Так, в лучшем из рассмотренных алгоритмов $PLex(n)$ мы

выделяли два элемента, меняли их местами, а затем переворачивали конечный отрезок перестановки. Поэтому естественно желание получить алгоритм генерации, в котором соседние перестановки будут различаться настолько мало, насколько это возможно. Для того, чтобы такое различие было минимально возможным, любая генерируемая перестановка должна отличаться от предшествующей транспозицией двух соседних элементов. Возможно ли таким образом породить все перестановки без повторений? Оказывается, такую последовательность перестановок легко построить рекурсивно по $n$.

При $n = 1$ последовательность из единственной перестановки (1) будет требуемой. Предположим, что имеется последовательность $\sigma^1, \ldots, \sigma^i, \ldots, \sigma^{(n-1)!}$ всех перестановок из $S_{n-1}$ такая, что каждая следующая $\sigma^{i+1}$ получается из предыдущей $\sigma^i$ перестановкой двух соседних элементов. Построим последовательность $\pi^i$, $i = 1, \ldots, n!$ всех перестановок из $S_n$ с таким же свойством. Расширим каждую перестановку $\sigma^i = (\sigma^i_1, \ldots, \sigma^i_{n-1})$, последовательно вставляя элемент $n$ на каждое из $n$ возможных мест: перед элементом $\sigma^i_1$, между элементами $\sigma^i_j$ и $\sigma^i_{j+1}$, после элемента $\sigma^i_{n-1}$. При этом элемент $n$ вставляем в $\sigma^1$ в направлении справа налево, а в $\sigma^2$ — слева направо:

$$(\sigma^1_1, \ldots, \sigma^1_{n-1}, n), \ldots, (n, \sigma^1_1, \ldots, \sigma^1_{n-1}),$$

$$(n, \sigma^2_1, \ldots, \sigma^2_{n-1}), \ldots, (\sigma^2_1, \ldots, \sigma^2_{n-1}, n)$$

и т. д., при переходе от $\sigma^i$ к $\sigma^{i+1}$ меняем направление вставки элемента $n$ на обратное. Тогда внутри $i$-го *блока* ($n$ перестановок из $S_n$, построенных из $\sigma^i \in S_{n-1}$) каждая следующая перестановка получается из предыдущей перестановкой двух соседних элементов, один из которых есть $n$. При переходе от последней перестановки в $i$-м блоке к первой в следующем $(i+1)$-м блоке по индукционному предположению также переставляются только два элемента. Таким образом, для построенной последовательности перестановок $\pi^i \in S_n$, $i = 1, \ldots, n!$ выполняется требуемое свойство.

**Пример 4.** На рис. 1 приведено рекурсивное построение последовательности всех перестановок из $S_4$ в порядке минимального изменения.

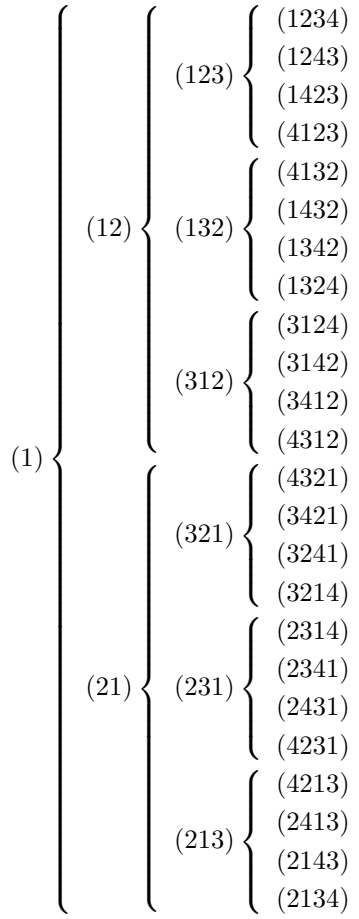



Рис. 1. Рекурсивное построение перестановок из $S_4$

Описанный рекурсивный алгоритм, генерирующий последовательность перестановок из $S_n$, применённый непосредственно, имеет огромный недостаток: последовательность перестановок строится "целиком" и требует хранения всех перестановок из $S_{n-1}, S_{n-2}, \ldots$ Очевидно, такой

алгоритм использовал бы огромный объём памяти, поэтому он неприменим. Мы изучим нерекурсивную модификацию этого метода — алгоритм Джонсона – Троттера.

Получим рассмотренный выше порядок перестановок $n$ элементов без явной генерации перестановок для меньших значений $n$. Это можно сделать, связав с каждой компонентой $\pi_i$ перестановки $\pi = (\pi_1, \ldots, \pi_n)$ её *направление*. Будем указывать направление при помощи стрелки $\rightarrow$ ("вправо") или $\leftarrow$ ("влево") над рассматриваемой компонентой перестановки. Перестановку $\pi$ вместе с заданными направлениями её компонент обозначим через $\overrightarrow{\pi}$. Будем писать $\overrightarrow{\pi} = \overrightarrow{\sigma}$, если $\pi = \sigma$ и направления соответствующих компонент совпадают. Если $\pi \in S_n$, то через $\pi \setminus \{n\}$ обозначим перестановку из $S_{n-1}$, получающуюся из $\pi$ удалением элемента $n$. Для перестановки $\pi \in S_n$ число $m \in \{1, \ldots, n\}$ будем называть *кандидатом для перемещения*, если стрелка над $m$ в $\overrightarrow{\pi}$ указывает на меньшее соседнее число. Такое соседнее число будем называть *дублёром числа* $m$ и обозначать $m^*$. Например, для перестановки $\pi = (5, 3, 4, 2, 1)$ рассмотрим следующую расстановку направлений $\overrightarrow{\pi} = (\overleftarrow{5}, \overleftarrow{3}, \overleftarrow{4}, \overrightarrow{2}, \overleftarrow{1})$. Тогда числа $4, 2$ — кандидаты для перемещения, а остальные $5, 3, 1$ не являются кандидатами. Для числа 4 дублером является число 3, а число 1 есть дублер числа 2.

В рассматриваемом алгоритме в перестановке $\overrightarrow{\pi}$ будем менять местами наибольший кандидат для перемещения $m$ с его дублёром $m^*$. В дальнейшем наибольший кандидат для перемещения в перестановке $\overrightarrow{\pi}$ будем просто называть *переставляемым элементом*.

**Замечание 1.** Для любой перестановки $\overrightarrow{\pi}$ число 1 не является кандидатом для перемещения.

**Замечание 2.** Для перестановки $\overrightarrow{\pi}$, $\pi \in S_n$ число $n$ не является переставляемым элементом тогда и только тогда, когда первая компонента $\overrightarrow{\pi}$ есть $\overleftarrow{n}$ или последняя компонента $\overrightarrow{\pi}$ есть $\overrightarrow{n}$.

Перейдем к описанию алгоритма Джонсона – Троттера генерации всех перестановок в порядке минимального изменения.

**Алгоритм Джонсона – Троттера** $PMin(n)$

$\overrightarrow{\pi} := (\overleftarrow{1}, \overleftarrow{2}, \ldots, \overleftarrow{n})$;
$m := 0$;
**while** $m \neq 1$ **do**
**begin**
**write**$(\pi_1, \ldots, \pi_n)$;
$m := n$;
**while** ($m$ не кандидат для перемещения в $\pi$ **and** $m > 1$) **do** $m := m-1$;
$\pi$: $m \leftrightarrow m^*$; (считаем $1^* = 1$)
$\overrightarrow{\pi}$: над всеми элементами перестановки $\pi$, большими $m$, меняем стрелку на противоположную по направлению
**end**.

**Пример 5.** Рассмотрим работу алгоритма $PMin(n)$ при $n = 3$:
$\overrightarrow{\pi}^1 = (\overleftarrow{1}, \overleftarrow{2}, \overleftarrow{3})$, $m = 3$, $m^* = 2$;
$\overrightarrow{\pi}^2 = (\overleftarrow{1}, \overleftarrow{3}, \overleftarrow{2})$, $m = 3$, $m^* = 1$;
$\overrightarrow{\pi}^3 = (\overleftarrow{3}, \overleftarrow{1}, \overleftarrow{2})$, $m = 2$, $m^* = 1$;
$\overrightarrow{\pi}^4 = (\overrightarrow{3}, \overleftarrow{2}, \overleftarrow{1})$, $m = 3$, $m^* = 2$;
$\overrightarrow{\pi}^5 = (\overleftarrow{2}, \overrightarrow{3}, \overleftarrow{1})$, $m = 3$, $m^* = 1$;
$\overrightarrow{\pi}^6 = (\overleftarrow{2}, \overleftarrow{1}, \overrightarrow{3})$, $m = 1$.

**Лемма 3.** Алгоритм $PMin(n)$ корректен и строит все перестановки из $S_n$ без повторений в порядке минимального изменения.

**Доказательство.** Обозначим через $\overrightarrow{\pi}^i$ последовательность чисел $\pi^i = (\pi^i_1, \ldots, \pi^i_n)$, полученную в результате $i$-й печати при работе алгоритма $PMin(n)$ вместе с направлениями, определенными в этот момент для компонент $\pi^i_1, \ldots, \pi^i_n$. Поскольку $\overrightarrow{\pi}^1 = (\overleftarrow{1}, \overleftarrow{2}, \ldots, \overleftarrow{n})$ и $\overrightarrow{\pi}^i$ получается из $\overrightarrow{\pi}^{i-1}$ перестановкой двух соседних элементов, $\pi^i \in S_n$ для всех $i$. Нам потребуется следующее свойство.

**Лемма 4.** Если $\overrightarrow{\pi}^i$ имеет переставляемый элемент $m \neq n$, то определены перестановки $\overrightarrow{\pi}^{i+1}, \ldots, \overrightarrow{\pi}^{i+n}$, число $n$ является переставляемым элементом в перестановке $\overrightarrow{\pi}^j$ при $j = i+1, \ldots, i+n-1$ и не является переставляемым элементом в перестановке $\overrightarrow{\pi}^{i+n}$. Кроме того, $\overrightarrow{\pi}^{i+1} \setminus \{n\} = \overrightarrow{\pi}^{i+2} \setminus \{n\} = \ldots = \overrightarrow{\pi}^{i+n} \setminus \{n\}$.

**Доказательство** леммы 4. По замечанию 1 имеем $m \neq 1$. По замечанию 2 перестановка $\overrightarrow{\pi}^i$ имеет вид $(\overleftarrow{n}, \ldots)$ или $(\ldots, \overrightarrow{n})$. В теле внешнего while-цикла в перестановке $\overrightarrow{\pi}^i$ элементы $m$, $m^*$ меняются местами, и над всеми элементами, бóльшими чем $m$, стрелка меняется на противоположную. Полученная перестановка есть $\overrightarrow{\pi}^{i+1}$. Так как $n > m > m^*$, перестановка $\overrightarrow{\pi}^{i+1}$ имеет вид $(\overrightarrow{n}, \ldots)$ или $(\ldots, \overleftarrow{n})$. По замечанию 2 число $n$ является переставляемым элементом в $\overrightarrow{\pi}^{i+1}$. Далее снова по замечанию 2 число $n$ будет переставляемым элементом в $\overrightarrow{\pi}^{i+2}, \ldots, \overrightarrow{\pi}^{i+n-1}$. Поэтому определена перестановка $\overrightarrow{\pi}^{i+n}$, которая будет иметь вид $(\ldots, \overrightarrow{n})$ или $(\overleftarrow{n}, \ldots)$. Понятно, что направления всех чисел в перестановках $\overrightarrow{\pi}^{i+1}, \ldots, \overrightarrow{\pi}^{i+n}$ совпадают. Лемма 4 доказана.

Теперь индукцией по $n$ покажем, что алгоритм $PMin(n)$ корректен, строит все перестановки из $S_n$ без повторений, и для любого $i \neq n!$ перестановка $\overrightarrow{\pi}^i$ имеет переставляемый элемент, а $\overrightarrow{\pi}^{n!}$ не имеет кандидата для перемещения. Базис индукции $n = 1$ очевиден. Пусть для $n-1$ всё доказано. Рассмотрим работу алгоритмов $PMin(n)$ и $PMin(n-1)$ одновременно. В силу замечания 2 алгоритмом $PMin(n)$ будут напечатаны следующие $n$ перестановок:

$$\overrightarrow{\pi}^1 = (\overleftarrow{1}, \overleftarrow{2}, \ldots, \overleftarrow{n-1}, \overleftarrow{n}),$$
$$\overrightarrow{\pi}^2 = (\overleftarrow{1}, \overleftarrow{2}, \ldots, \overleftarrow{n}, \overleftarrow{n-1}),$$
$$\vdots$$
$$\overrightarrow{\pi}^n = (\overleftarrow{n}, \overleftarrow{1}, \overleftarrow{2}, \ldots, \overleftarrow{n-1}).$$

Так как

$$\overrightarrow{\pi}^1 \setminus \{n\} = \ldots = \overrightarrow{\pi}^n \setminus \{n\} = (\overleftarrow{1}, \overleftarrow{2}, \ldots, \overleftarrow{n-1})$$

есть первая напечатанная алгоритмом $PMin(n-1)$ перестановка, по индукционному предположению $\overrightarrow{\pi}^n \setminus \{n\}$ имеет переставляемый элемент $m_1$, при условии $(n-1)! \neq 1$, и $m_1 \neq n$. Тогда $m_1$ — переставляемый элемент в $\overrightarrow{\pi}^n$, и по лемме 4 определены следующие $n$ перестановок $\overrightarrow{\pi}^{n+1}, \ldots, \overrightarrow{\pi}^{2n} \in S_n$, причём

$$\overrightarrow{\pi}^{n+1} \setminus \{n\} = \ldots = \overrightarrow{\pi}^{2n} \setminus \{n\}$$

является второй напечатанной алгоритмом $PMin(n-1)$ перестановкой. По индукционному предположению $\overrightarrow{\pi}^{2n} \setminus \{n\}$ имеет переставляемый элемент $m_2$, при условии $(n-1)! \neq 2$, и перестановка $\overrightarrow{\pi}^{2n}$ имеет вид $(\overleftarrow{n}, \ldots)$ или $(\ldots, \overrightarrow{n})$ по лемме 4 и замечанию 2. Поэтому $m_2$ является переставляемым элементом в $\overrightarrow{\pi}^{2n}$ и $m_2 \neq n$. В общем случае в $i$-м блоке определены следующие $n$ перестановок $\overrightarrow{\pi}^{(i-1)n+1}, \ldots, \overrightarrow{\pi}^{(i-1)n+n} \in S_n$,

$$\overrightarrow{\pi}^{(i-1)n+1} \setminus \{n\} = \ldots = \overrightarrow{\pi}^{in} \setminus \{n\}$$

есть $i$-я напечатанная алгоритмом $PMin(n-1)$ перестановка и $\overrightarrow{\pi}^{in}$ имеет переставляемый элемент $m_i \neq n$, при условии $(n-1)! \neq i$. При $i = (n-1)! - 1$ перестановка $\overrightarrow{\pi}^{in}$ имеет переставляемый элемент $m_{n!-n} \neq n$. По лемме 4 определены следующие $n$ перестановок $\overrightarrow{\pi}^{n!-n+1}, \ldots, \overrightarrow{\pi}^{n!} \in S_n$, причём число $n$ не является переставляемым элементом в $\overrightarrow{\pi}^{n!}$. По индукционному предположению перестановка $\overrightarrow{\pi}^{n!} \setminus \{n\}$ из $S_{n-1}$ не имеет кандидата для перемещения. Поэтому перестановка $\overrightarrow{\pi}^{n!}$ также не имеет кандидата для перемещения. Таким образом, для любого $i \neq n!$ перестановка $\overrightarrow{\pi}^i$ имеет переставляемый элемент, а $\overrightarrow{\pi}^{n!}$ не имеет кандидата для перемещения. После печати перестановки $\overrightarrow{\pi}^{n!}$ и выхода из внутреннего while-цикла переменная $m$ примет значение 1. Поэтому алгоритм $PMin(n)$ остановит свою работу.

В результате работы алгоритма $PMin(n)$ напечатанные перестановки $\pi^i$, $i = 1, \ldots, n!$ разбиты на $(n-1)!$ блоков по $n$ штук в каждом. В любом $i$-м блоке перестановки различаются между собой положением числа $n$, причём $\pi^{in} \setminus \{n\}$ является $i$-й напечатанной перестановкой

при работе алгоритма $PMin(n-1)$. По индукционному предположению получаем $\pi^{in} \setminus \{n\} \neq \pi^{i'n} \setminus \{n\}$ при $i \neq i'$. Следовательно, перестановки в разных блоках различны. Таким образом, мы получили все $n!$ перестановок из $S_n$ без повторений. Лемма 4 доказана.

Перейдём к программной реализации алгоритма Джонсона – Тротера. Чтобы задать перестановку с направлениями её компонент $\overrightarrow{\pi}$, введём два массива $\pi$ и $\tau$, где $\pi$ — собственно сама перестановка и $\tau$ — массив из чисел $-1, +1$. Причём $\tau_i$ указывает направление числа $\pi_i$: $+1$ означает стрелку $\rightarrow$, а $-1$ означает стрелку $\leftarrow$ . Для формализации условия внутреннего while-цикла по числу $m$ требуется найти его дублёра $m^*$. Если мы знаем $number(m)$, номер позиции числа $m$ в перестановке $\pi$ и направление $\tau_m$ числа $m$, то дублёр определяется просто как $m^* = \pi_{number(m)+\tau_m}$. Поэтому для хранения номера числа $m$ в перестановке $\pi$ введём ещё один массив $\sigma = (\sigma_1, \ldots, \sigma_n)$ такой, что $\sigma_i = number(i)$ — номер позиции числа $i$ в перестановке $\pi$. Другими словами, $\sigma$ — обратная перестановка к $\pi$. Тогда для любого $m$ имеем $\pi_{\sigma_m} = m$ и $\pi_{\sigma_m+\tau_m} = m^*$. Условие, что $m$ не кандидат для перемещения в $\overrightarrow{\pi}$, означает, что $\pi_{\sigma_m+\tau_m} > m$, за исключением двух ситуаций, когда мы находимся в крайних позициях следующего вида:

$\pi = (\overleftarrow{m}, \ldots)$, $\sigma_m = 1$, $\tau_m = -1$;

$\pi = (\ldots, \overrightarrow{m})$, $\sigma_m = n$, $\tau_m = +1$.

Поэтому нужно доопределить значения $\pi_0$ и $\pi_{n+1}$ так, чтобы в этих ситуациях число $m$ не являлось кандидатом для перемещения. Например, полагаем $\pi_0 = \pi_{n+1} = n + 1$. Тогда $\pi_0 > m$ и $\pi_{n+1} > m$ для для любого $m \leq n$.

В условии внутреннего while-цикла есть неравенство $m > 1$. Его можно исключить, если при $m = 1$ произойдёт выход из этого цикла. Это означает, что неравенство $\pi_{\sigma_m+\tau_m} > m$ не должно выполняться при $m = 1$. Так как $\pi_{\sigma_1} = 1$, для этого достаточно определить $\tau_1 = 0$. При этом ничего не нарушится, поскольку по замечанию 1 число 1 не является кандидатом для перемещения в любой перестановке $\overrightarrow{\pi}$.

**Алгоритм $PMin(n)$ генерации всех перестановок в порядке минимального изменения**

**for** $i = 1$ to $n$ **do**
**begin**
$\pi_i := i$;
$\tau_i := -1$;
$\sigma_i := i$
**end**;
$\pi_0 := n + 1$;
$\pi_{n+1} := n + 1$;
$\tau_1 := 0$;
$m := 0$;
**while** $m \neq 1$ **do**
**begin**
**write**$(\pi_1, \ldots, \pi_n)$;
$m := n$;
**while** $\pi_{\sigma_m + \tau_m} > m$ **do**
**begin**
$\tau_m := -\tau_m$;
$m := m - 1$
**end**;
$Swap(\pi_{\sigma_m}, \pi_{\sigma_m + \tau_m})$;
$Swap(\sigma_m, \sigma_{\pi_{\sigma_m} + \tau_m})$
**end**.

**Теорема 5.** Алгоритм $PMin(n)$ корректен и строит все перестановки из $S_n$ без повторений в порядке минимального изменения за время $O(n!)$.

**Доказательство.** Ввиду леммы 3 остается только оценить сложность $T(n)$ алгоритма $PMin(n)$. Из доказательства леммы 3 вытекает, что последовательность $\pi^1, \pi^2, \ldots, \pi^{n!}$ всех перестановок из $S_n$,

напечатанных в ходе работы алгоритма $PMin(n)$, разбивается на $n$-элементные блоки $S_n^k$, $k = 1, \ldots, (n-1)!$. Введём обозначения:

- $I_n$ — число сравнений, выполняемых в условии внутреннего while-цикла во время работы алгоритма $PMin(n)$;
- $t_n^i$ — число сравнений, выполняемых в условии внутреннего while-цикла во время работы алгоритма $PMin(n)$, начиная с печати перестановки $\pi^i$ и до печати $\pi^{i+1}$, $i < n!$;
- $t_n^{n!}$ — число сравнений, выполняемых в условии внутреннего while-цикла во время работы алгоритма $PMin(n)$, начиная с печати перестановки $\pi^{n!}$ и до окончания работы алгоритма.

Индукцией по $n$ докажем, что

$$I_n = \sum_{i=1}^{n} i!.$$

Действительно, базис индукции $n = 1$ очевиден. Пусть $n \geq 2$ и для $n-1$ это равенство справедливо. Имеем

$$I_n = \sum_{i=1}^{n!} t_n^i = \sum_{k=1}^{(n-1)!} \sum_{i:\, \pi^i \in S_n^k} t_n^i.$$

Из доказательства леммы 3 следует, что число $n$ является переставляемым элементом в первых $n-1$ перестановках каждого блока $S_n^k$, $\pi^{kn} \setminus \{n\}$ есть $k$-я напечатанная перестановка во время работы алгоритма $PMin(n-1)$, причём переставляемые элементы в перестановках $\pi^{kn}$ и $\pi^{kn} \setminus \{n\}$ совпадают и не равны $n$. Следовательно, $t_n^{kn} = 1 + t_{n-1}^k$ и $t_n^i = 1$ для любого $i$ такого, что $\pi^i \in S_n^k$ и $i \neq kn$. Учитывая индукционное предположение, получаем

$$I_n = \sum_{k=1}^{(n-1)!} (n - 1 + t_n^{kn}) = \sum_{k=1}^{(n-1)!} (n + t_{n-1}^k) = n! + I_{n-1} = \sum_{i=1}^{n} i!.$$

Очевидно, $T(n) = O(n) + O(I_n)$. Так как $I_n = \sum_{i=1}^{n} i! = n! + o(n!)$[13], получаем $T(n) = O(n!)$. Теорема 5 доказана.

Более детальное сравнение сложности линейных алгоритмов генерации $PLex(n)$ и $PMin(n)$ показывает, что алгоритм $PMin(n)$ быстрее. Кроме того, алгоритм $PMin(n)$ интересен тем, что перестановки выписываются в порядке минимального изменения, а это очень важно, если с порождаемой перестановкой в требуемом алгоритме необходимо производить какие-либо вычисления, связанные с элементами самой перестановки. В этом случае имеется возможность использовать результаты вычислений, полученные для предыдущей перестановки, отличающейся от следующей только лишь транспозицией соседних элементов.

## 2. Подмножества конечного множества

Пусть $\mathcal{P}(A)$ — множество всех подмножеств $n$-элементного множества $A = \{a_0, a_1, \ldots, a_{n-1}\}$. Мощность множества $\mathcal{P}(A)$ равна $2^n$. Существует биективное отображение $f : \mathcal{P}(A) \to E^n$ из множества всех подмножеств $\mathcal{P}(A)$ в $n$-мерный единичный куб $E^n$,[14] определяемое по правилу $f(B) = \chi^B$ для произвольного подмножества $B \subseteq A$, где

$$\chi^B = (\chi_0^B, \ldots, \chi_{n-1}^B),$$

$$\chi_i^B = \begin{cases} 1, & \text{если } a_i \in B, \\ 0, & \text{если } a_i \notin B. \end{cases}$$

Таким образом, задача генерации всех подмножеств заданного $n$ элементного множества $A$ сводится к задаче порождения всех $n$-разрядных двоичных последовательностей. Также отметим, что генерация всех двоичных векторов длины $n$ по сути есть прохождение всех вершин $n$-мерного единичного куба без повторений.

[13] $f(n) = o(g(n))$, если $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

[14] Здесь $E^n$ — множество вершин единичного куба, т. е. множество всех $n$-разрядных двоичных последовательностей.

Для любого целого положительного числа $a$ существует его единственное двоичное представление $(b_{n-1}b_{n-2}\dots b_1b_0)_2 = a$, где $b_i$ определяются из следующих соотношений:

$$a = b_0 + b_1 2 + b_2 2^2 + \dots + b_{n-1} 2^{n-1},$$

$$b_i \in \{0, 1\},\ i = 0, \dots, n-1.$$

Поэтому для любого $a < 2^n$ существует единственное двоичное $n$-разрядное представление. Множество двоичных векторов длины $n$ с лексикографическим порядком $(E^n, \preceq)$ есть линейно упорядоченное множество с наименьшим элементом $(0, 0, \dots, 0)$ и наибольшим элементом $(1, 1, \dots, 1)$, причём $(b_{n-1} \dots b_0)_2$ есть номер вектора $(b_{n-1}, \dots, b_0)$ относительно этого упорядочивания. Поэтому все $n$-разрядные двоичные последовательности можно сгенерировать следующим образом. Пробегая значение индекса $i$ от 0 до $2^n - 1$, по числу $i$ восстанавливаем его двоичное представление длины $n$, т. е. двоичный вектор с номером $i$ в лексикографическом порядке. Однако такой алгоритм генерации не будет линейным, так как время восстановления двоичного представления длины $n$ не ограничено константой. Поэтому требуется другой алгоритм генерации двоичных векторов.

## 2.1. Генерация двоичных векторов и подмножеств

Будем порождать все двоичные векторы длины $n$ в лексикографическом порядке, начиная с наименьшего элемента. Как перейти от заданного вектора $b = (b_{n-1}, \dots, b_0)$ к непосредственно следующему вектору в лексикографическом порядке? Просматривая справа налево вектор $b$, ищем самую правую позицию $i$ такую, что $b_i = 0$. Запишем в эту $i$-ю позицию 1, а все элементы $b_j$, $j < i$, стоящие справа от $b_i$, полагаем равными 0. Очевидно, что мы получим требуемый вектор.

Для программной реализации этого алгоритма дополним массив $b = (b_{n-1}, \dots, b_0)$ элементом $b_n$. При инициализации полагаем $b_n = 0$. Для всех порождаемых последовательностей при поиске правой позиции $i$

элемент $b_n$ не изменяется, за исключением ситуации, когда генерируется последний вектор $(1, 1, \ldots, 1)$, $i = n$ и $b_n$ станет равным 1. Поэтому равенство $b_n = 1$ будет условием остановки алгоритма. Программная реализация данного алгоритма приведена в следующем псевдокоде.

**Алгоритм $VLex(n)$ генерации всех двоичных векторов длины $n$**

**for** $i = 0$ **to** $n$ **do** $b_i := 0$;
**while** $b_n \neq 1$ **do**
**begin**
  **write** $(b_{n-1}, b_{n-2}, \ldots, b_0)$;
  $i := 0$;
  **while** $b_i = 1$ **do**
  **begin**
    $b_i := 0$;
    $i := i + 1$
  **end**;
  $b_i := 1$
**end**.

Оценим время $T(n)$ работы алгоритма $VLex(n)$. Пусть $W^n$ — число проверок условия внутреннего while-цикла и $W_i^n$ — число проверок равенства $b_i = 1$. Очевидно, что $T(n) = O(n) + O(W^n)$ и $W^n = \sum_{i=0}^{n} W_i^n$. Проверка условия $b_i = 1$ осуществляется тогда и только тогда, когда $c_j = 1$ для любого $j < i$, где $(c_{n-1}, c_{n-2}, \ldots, c_0)$ — последний вектор, выведенный на печать перед проверкой этого условия. Следовательно,

$$W_i^n = |\{(c_{n-1}, c_{n-2}, \ldots, c_0) \in E^n | \, \forall j < i \; c_j = 1\}| = 2^{n-i},$$

$$W^n = \sum_{i=0}^{n} 2^{n-i} = 2^{n+1} - 1 = O(2^n).$$

Поэтому $T(n) = O(n) + O(2^n) = O(2^n)$. Таким образом, алгоритм $VLex(n)$ генерации двоичных векторов линеен.

Учитывая биекцию $f : \mathcal{P}(A) \to E^n$, перевод алгоритма $VLex(n)$ на язык подмножеств множества $\{a_0, a_1, \ldots, a_{n-1}\}$ приводит к следующему алгоритму генерации всех подмножеств (здесь дополнительно рассматривается фиктивный элемент $a_n \notin \{a_0, \ldots, a_{n-1}\}$).

**Алгоритм $SLex(n)$ генерации подмножеств $n$-элементного множества $\{a_0, \ldots, a_{n-1}\}$**

$B := \emptyset$;
**while** $a_n \notin B$ **do**
**begin**
  **write**$(B)$;
  $i := 0$;
  **while** $a_i \in B$ **do**
  **begin**
    $B := B \setminus \{a_i\}$;
    $i := i + 1$
  **end**;
  $B := B \cup \{a_i\}$
**end**.

**Пример 6.** Генерация двоичных векторов $b^i$ длины $n = 3$ и подмножеств $B^i$ множества $A = \{a_0, a_1, a_2\}$, $i = 1, \ldots, 8$.

$b^1 = (0, 0, 0)$, $B^1 = \emptyset$, $i = 0$;
$b^2 = (0, 0, 1)$, $B^2 = \{a_2\}$, $i = 1$;
$b^3 = (0, 1, 0)$, $B^3 = \{a_1\}$, $i = 0$;
$b^4 = (0, 1, 1)$, $B^4 = \{a_1, a_2\}$, $i = 2$;
$b^5 = (1, 0, 0)$, $B^5 = \{a_0\}$, $i = 0$;
$b^6 = (1, 0, 1)$, $B^6 = \{a_0, a_2\}$, $i = 1$;
$b^7 = (1, 1, 0)$, $B^7 = \{a_0, a_1\}$, $i = 0$;
$b^8 = (1, 1, 1)$, $B^8 = A$, $i = 3$.

## 2.2. Коды Грея и алгоритм их генерации

В этом разделе мы познакомимся с алгоритмом генерации всех $n$-разрядных двоичных векторов в порядке минимального изменения. Рассмотренные алгоритмы $VLex(n)$, $SLex(n)$ порождения двоичных векторов и подмножеств не обладают этим свойством (см. пример 6), так как минимальные изменения при переходе от одного множества к другому соответствуют добавлению или удалению ровно одного элемента, а в терминах двоичных векторов это означает, что последовательные векторы должны отличаться в одном разряде. Такие наборы двоичных векторов известны как коды Грея.

**Определение 9.** *$n$-разрядным кодом Грея* называется упорядоченная последовательность $2^n$ различных двоичных $n$-разрядных векторов (*кодовых слов*), последовательные векторы которой различаются в одном разряде. Код Грея называется *циклическим*, если его первый и последний векторы также различаются в одном разряде.

В этих терминах задача генерации всех $n$-разрядных двоичных векторов в порядке минимального изменения есть задача построения $n$-разрядного кода Грея. Очевидно, что всякий $n$-разрядный код Грея определяет гамильтонов путь в $n$-мерном единичном кубе $E^n$, и наоборот, всякий гамильтонов путь в $E^n$ задаёт $n$-разрядный код Грея. Аналогичное соответствие имеется между циклическим кодом Грея и гамильтоновым циклом. Единичный куб $E^n$ гамильтонов при $n \geq 2$. Одно из построений гамильтонова цикла может быть получено индуктивно: в двух копиях $E^n$, разбивающих куб $E^{n+1}$, выбираются такие циклы и достраиваются до требуемого цикла $(n+1)$-мерного куба. Эти рассуждения показывают существование различных (с точностью до циклических сдвигов) циклических кодов Грея при $n \geq 3$.

Рассмотрим другой способ задания циклического кода Грея. Ввиду цикличности начальным кодовым словом считаем вектор $(0, 0, \ldots, 0)$.

**Определение 10.** Пусть $G(n) = G_0, G_1, \ldots, G_{2^n-1}$ — циклический $n$-разрядный код Грея, $t_i$ — номер позиции (считая справа налево), меняющейся при переходе от кодового слова $G_{i-1}$ к $G_i$, $i = 1, \ldots, 2^n - 1$. Последовательность $T_n = t_1, t_2, \ldots, t_{2^n-1}$ называется *последовательностью переходов* кода Грея $G(n)$.

Например, для циклического кода Грея $G(2) = 00, 10, 11, 01$ последовательность переходов есть $2, 1, 2$. Очевидно, что циклический код Грея однозначно определяется по своей последовательности переходов.

Рассмотрим *двоично-отражённые коды Грея* $G(n)$, определяемые рекурсивным образом:

$$G(1) = 0, 1$$

$$G(n+1) = 0G_0, 0G_1, \ldots, 0G_{2^n-1}, 1G_{2^n-1}, \ldots, 1G_1, 1G_0, \text{ где}$$

$$G(n) = G_0, G_1, \ldots, G_{2^n-1}.$$

Очевидно, что так определенная последовательность $G(n)$ является циклическим $n$-разрядным кодом Грея. Докажем, что последовательность его переходов $T_n$ рекурсивно вычисляется следующим образом:

$$T_1 = 1$$

$$T_{n+1} = T_n, n+1, T_n.$$

Действительно, из определения $G(n)$ имеем $T_1 = 1$ и $T_{n+1} = T_n, n+1, T_n^*$, где $T_n^*$ получается переворачиванием последовательности $T_n$. Очевидно, что $T_1^* = T_1$ и $T_n^* = (T_{n-1}^*)^*, n, T_{n-1}^* = T_n$. Поэтому последовательность $T_{n+1}$ имеет требуемый вид.

Известно другое рекурсивное определение тех же кодов Грея $G(n)$: $G(1) = 0, 1$; $G(n+1) = G_0 0, G_0 1, G_1 1, G_1 0, G_2 0, G_2 1, \ldots, G_n 1, G_n 0$ и последовательности его переходов $T_1 = 1$, $T_{n+1} = 1, t_1 + 1, 1, t_2 + 1, 1, \ldots, 1, t_{2^n-1} + 1, 1$, где $T_n = t_1, t_2, \ldots, t_{2^n-1}$ (иными словами, все компоненты $T_n$ увеличиваются на 1, а затем 1 вставляется в начало, в конец и между элементами полученной последовательности).

Перейдём к алгоритму генерации двоично-отражённого кода Грея $G(n)$. На основе его индуктивного определения очевиден рекурсивный алгоритм порождения $G(n)$. Но такой алгоритм будет использовать память, необходимую для хранения $G(n-1)$, $G(n-2)$, ... Нам требуется алгоритм, последовательно порождающий элементы кода $G(n)$ и использующий лишь ограниченный объём памяти. Для генерации $G(n)$ достаточно эффективно порождать его последовательность переходов $T_n = t_1, t_2, \ldots, t_{2^n-1}$, так как, зная предыдущее кодовое слово $G_i = (g_n g_{n-1} \ldots g_1)$, можно перейти к $G_{i+1}$, просто полагая $g_{t_i} := \overline{g}_{t_i}$.[15]

Теперь построим алгоритм порождения последовательности переходов $T_n$. Будем использовать стек[16] $A_n$. Определим $k$-е состояние стека $A_n^k$. В пустой стек последовательно добавим числа $n, n-1, \ldots, 2, 1$. Полученное состояние есть $A_n^1$, и число 1 является верхним элементом стека. Далее при переходе от $k$-го состояния к $(k+1)$-му состоянию удалим верхний элемент $i$ из стека $A_n^k$, и если $i \neq 1$, то последовательно добавим элементы $i-1, i-2, \ldots, 1$. Получим $(k+1)$-е состояние стека $A_n^{k+1}$. Верхний элемент стека $A_n$ при его $k$-ом состоянии обозначим через $a_n^k$. Индукцией по $n$ нетрудно доказать следующие свойства.

**Лемма 5.**

(i) $A_n^k \neq \emptyset$ и $a_{n+1}^k = a_n^k \neq n+1$ для всех $k < 2^n$;

(ii) $A_n^{2^n-1}$ содержит единственное число 1;

(iii) $A_n^{2^n} = \emptyset$ и $a_{n+1}^{2^n} = n+1$;

(iv) в непустом стеке $A_n^k$ все элементы расположены в порядке строгого возрастания сверху вниз;

(v) $T_n = a_n^1, a_n^2, \ldots, a_n^{2^n-1}$.

[15] $f(n) = \overline{n}$ — булева функция "отрицание".

[16] Стек (англ. stack — стопка) — структура данных, в которой организован последовательный доступ к элементам по правилу "первым пришёл — первым ушёл". Добавление элемента, называемое *проталкиванием*, возможно только в *вершину стека* (добавленный элемент становится первым сверху). Удаление элемента, называемое *выталкиванием*, также возможно только из вершины стека, при этом второй сверху элемент становится верхним. Доступ имеется только к верхнему элементу стека.

В силу леммы 5(v) последовательность выталкиваемых элементов $a_n^k$, $k = 1, \ldots, 2^n - 1$ порождает последовательность переходов $T_n$ двоично-отражённого кода Грея $G(n)$, причем для каждого состояния, за исключением $k$-го при $k = 2^n$, стек $A_n$ будет непустым в силу свойств (i) и (iii) леммы 5. Поэтому равенство $A_n^k = \emptyset$ служит условием окончания процесса выталкиваний. Для программной реализации это условие требуется сформулировать, например, в терминах выталкиваемого элемента. Для этого перейдём к стеку $A_{n+1}$. В силу леммы 5 процесс выталкиваний элементов $a_{n+1}^k$ из стека $A_{n+1}$ до тех пор, пока $a_{n+1}^k$ не станет равен $n+1$, порождает ту же последовательность $T_n$.

Далее, мы стремимся получить линейный алгоритм. Это означает, что среднее число операций, необходимых для генерирования каждого следующего элемента из последовательности переходов, должно быть ограничено константой, не зависящей от $n$. В рассмотренном процессе число изменяемых элементов в стеке $A_{n+1}$ зависит от значения верхнего выталкиваемого элемента, т. е. оно непостоянно. Эту зависимость можно устранить с помощью перехода от стека $A_{n+1}$ к такому массиву $B$, в котором его $k$-е состояние $B^k$ будет полностью соответствовать состоянию стека $A_{n+1}^k$, а при смене состояний будет изменяться только фиксированное число элементов. Воспользуемся свойством (iv) леммы 5 и вместо записи самих элементов в стек в массив будем записывать указатели на нижеследующие числа стека. Именно, введём массив $B = (b_0, b_1, \ldots, b_n)$ и определим его $k$-е состояние $B^k = (b_0^k, \ldots, b_n^k)$ при $k \leq 2^n$ следующим образом. Полагаем $b_0^k = a_{n+1}^k$ и для $j = 1, \ldots, n$ пусть $b_j^k$ равен элементу стека $A_{n+1}^k$, находящемуся под числом $j$, если $j$ есть в стеке $A_{n+1}^k$, в противном случае полагаем $b_j^k = j + 1$. Заметим, что элемент $b_j^k$ определён корректно, так как ниже любого числа $j \leq n$ из стека всегда есть следующий элемент (по крайней мере, $n+1$ находится на дне стека $A_n^k$), и число $j$ входит в стек не более одного раза в силу леммы 5(iv). Теперь нетрудно доказать, что при переходе от $k$-го состояния стека $A_{n+1}^k$ к его $(k+1)$-му состоянию $A_{n+1}^{k+1}$ состояние

массива $B^k$ изменяется по следующему правилу:

$$b_0^{k+1} = \begin{cases} b_i^k, & \text{если } i = 1, \\ 1, & \text{если } i \neq 1; \end{cases}$$

$$b_j^{k+1} = \begin{cases} j+1, & \text{если } i = j = 1, \\ b_j^k, & \text{если } i = 1 \text{ и } j \neq 1, \\ b_j^k, & \text{если } i \neq 1 \text{ и } i < j \leq n, \\ j+1, & \text{если } i \neq 1 \text{ и } j = i, \\ b_i^k, & \text{если } i \neq 1 \text{ и } j = i-1, \\ j+1 = b_j^k, & \text{если } i \neq 1 \text{ и } 1 \leq j < i-1; \end{cases}$$

$$i = b_0^k = a_{n+1}^k.$$

Таким образом, мы приходим к следующему алгоритму.

**Алгоритм $TranSeq(n)$ генерации последовательности переходов двоично-отражённого кода Грея $G(n)$**

**for** $i = 0$ **to** $n$ **do** $b_i := i + 1$;
$i := 0$;
**while** $i \neq n + 1$ **do**
**begin**
  **write**$(b_0)$;
  $i := b_0$;
  $b_0 := 1$;
  $b_{i-1} := b_i$;
  $b_i := i + 1$
**end**.

**Лемма 6.** Алгоритм $TranSeq(n)$ корректен и строит последовательность переходов двоично-отражённого кода Грея $G(n)$.

**Доказательство.** В силу леммы 5 последовательность $T_n$ порождается в результате выталкивания верхних элементов $a^k_{n+1}$ стека $A_{n+1}$ до тех пор, пока $a^k_{n+1} \neq n+1$. Элемент $a^k_{n+1}$ есть первый элемент $k$-го состояния $B^k$ массива $B = (b_0, \ldots, b_n)$. Причём $B^1 = (1, 2, \ldots, n+1)$ и $B^{k+1}$ определяется через $B^k$ с помощью вышеуказанной формулы для $b^{k+1}_j$. Эта формула показывает, что в массиве $B$ только три элемента изменяются по следующему правилу:

$$b_0^{k+1} = \begin{cases} b_i^k, & \text{если } i = 1, \\ 1, & \text{если } i \neq 1; \end{cases}$$

$$b^{k+1}_{i-1} = b^k_i, \text{ если } i \neq 1;\ b^{k+1}_i = i+1;\ i = b^k_0,$$

что соответствует выполнению операторов тела while-цикла.
Лемма 6 доказана.

### Алгоритм $Vmin(n)$ генерации двоично-отражённого кода Грея $G(n)$

**for** $i = 0$ **to** $n-1$ **do** $g_i := 0$;
**for** $i = 0$ **to** $n$ **do** $b_i := i+1$;
$i := 0$;
**while** $i \neq n+1$ **do**
**begin**
  **write**$(g_n, g_{n-1}, \ldots, g_1)$;
  $i := b_0$;
  $g_i := \overline{g}_i$;
  $b_0 := 1$;
  $b_{i-1} := b_i$;
  $b_i := i+1$
**end**.

**Теорема 6.** Алгоритм $Vmin(n)$ корректен и строит двоично-отражённый код Грея $G(n)$ за время $O(2^n)$.

**Доказательство.** Корректность алгоритма следует из леммы 6. Так как код Грея $G(n)$ содержит $2^n$ кодовых слов, и число операций в алгоритме, необходимых для генерирования каждого следующего кодового слова, ограничено константой, не зависящей от $n$, данный алгоритм является линейным. Теорема 6 доказана.

# 3. Генерация сочетаний в лексикографическом порядке

В предыдущих разделах мы рассмотрели алгоритмы генерации всех подмножеств заданного $n$-элементного множества. Часто в задачах возникает необходимость генерировать не все подмножества, а лишь те, которые удовлетворяют некоторым ограничениям. Одним из таких общих ограничений является мощность подмножеств. Познакомимся с алгоритмом генерации всех подмножеств фиксированной мощности $k$.

**Определение 11.** *Сочетанием из $n$ элементов по $k$* называется неупорядоченная выборка $k$ элементов из заданных $n$ элементов.

Мы будем генерировать все сочетания из $n$ по $k$ для заданного $n$-элементного множества $A$. Число сочетаний из $n$ по $k$ равно биномиальному коэффициенту

$$C_n^k = \frac{n!}{k!(n-k)!}\,.$$

Поэтому лучшая сложность, которую можно ожидать для алгоритма генерации всех сочетаний, есть $O(C_n^k)$. Без ограничения общности можно предполагать $A = \{1, 2, \ldots, n\}$. Произвольное сочетание из $n$ по $k$ удобно представить в виде конечной последовательности длины $k$ из чисел, упорядоченных по возрастанию слева направо. Все такие сочетания, как последовательности, естественно порождать в лексикографическом порядке. Например, при $n = 5$ и $k = 3$ последовательность всех сочетаний в лексикографическом порядке следующая: 123, 124, 125, 134, 135, 145, 234, 235, 245, 345.

Очевидно, что первый элемент в лексикографическом порядке есть сочетание $(1, 2, \ldots, k)$, а последний — $(n-k+1, n-k+2, \ldots, n-1, n)$. Остаётся определить по данному сочетанию $a = (a_1, \ldots, a_k)$ вид непосредственно следующего сочетания $b = (b_1, \ldots, b_k)$. Такое сочетание $b$ получается в результате нахождения самого правого элемента $a_m$, который ещё не достиг своего возможного максимального значения, его увеличения на 1, а затем присвоения всем элементам справа от этого элемента новых возможных наименьших значений. Для этого необходимо найти самый правый элемент $a_m$ такой, что чисел больших, чем $a_m + 1$ найдётся по крайней мере $k - m$ штук. Таким образом,

$$b = (a_1, \ldots, a_{m-1}, a_m + 1, a_m + 2, \ldots, a_m + k - m + 1), \text{ где } m = m(a);$$

$$m(a) = \max\{i \mid a_i < n - k + i,\ 1 \le i \le k\}.$$

Очевидно, что число $m(a)$ определено тогда и только тогда, когда сочетание $a$ не является наибольшим относительно лексикографического порядка. Предположим, что $b$ не наибольшее сочетание, и вычислим $m(b)$. Из определения $m(b)$ следует, что если $b_k < n$, то $m(b) = k$. Пусть теперь $b_k = n$. Тогда $b_{k-1} = n-1, \ldots, b_m = n-k+m$. Но $b_m = a_m + 1$, поэтому $b_{m-1} = a_{m-1} < a_m = b_m - 1 = n - k + m - 1$. Следовательно, $m(b) = m - 1 = m(a) - 1$. Таким образом, доказана следующая

**Лемма 7.** Пусть $a = (a_1, \ldots, a_k)$, $b = (b_1, \ldots, b_k)$ — сочетания из $n$ по $k$, упорядоченные по возрастанию слева направо. Если $b$ непосредственно следует за $a$ относительно лексикографического порядка, то

$$b_i = \begin{cases} a_i, & \text{если } 1 \le i < m(a), \\ a_{m(a)} + i - m(a) + 1, & \text{если } m(a) \le i \le k, \end{cases}$$

причём если $b$ не является наибольшим сочетанием, то

$$m(b) = \begin{cases} m(a) - 1, & \text{если } b_k = n, \\ k, & \text{если } b_k < n. \end{cases}$$

**Алгоритм $CLex(n)$ генерации сочетаний из $n$ по $k$ в лексикографическом порядке**

**for** $i = 1$ **to** $k$ **do** $a_i := i$;
**if** $k = n$ **then** $m := 1$
**else** $m := k$;
**while** $m \neq 0$ **do**
**begin**
**write**$(a_1, \ldots, a_k)$;
**if** $a_k = n$ **then** $m := m - 1$
**else** $m := k$;
**if** $m \neq 0$ **then for** $i = m$ **to** $k$ **do** $a_i := a_m + i - m + 1$
**end**.

**Теорема 7.** Алгоритм $CLex(n)$ корректен и генерирует все сочетания из $n$ по $k$ без повторений в лексикографическом порядке за время

$$\frac{n+1}{n+1-k}\, C_n^k\, O(1).$$

**Доказательство.** В случае $k = n$ имеем единственное сочетание $a = (1, 2, \ldots, k)$ из $n$ по $n$, причём $a_k = n$. При инициализации значение $m$ равно 1. Поэтому после печати сочетания $a$ в теле while-цикла имеем $m = 0$, и, следовательно, алгоритм закончит работу.

Далее считаем $k < n$. В силу леммы 7 для обоснования корректности алгоритма достаточно заметить, что при инициализации значение $m$ равно $m(1, \ldots, k) = k$, и после печати наибольшего сочетания $b$ алгоритм останавливается. Обозначим через $a$ сочетание $(n-k, n-k+2, \ldots, n-1, n)$, непосредственно предшествующее $b$. Тогда $m(a) = 1$ и $b_k = n$. Следовательно, после печати сочетания $b$ переменная $m$ примет значение 0, и алгоритм закончит работу.

Оценим время работы алгоритма $CLex(n)$. Очевидно, что число итераций while-цикла равно $C_n^k + 1$. Поэтому сложность алгоритма есть

$k + C_n^k\, O(1) + I_{n,k}\, O(1)$, где $I_{n,k}$ — число итераций внутреннего for-цикла. Подсчитаем $I_{n,k}$. Сначала заметим, что для любого сочетания $a = (a_1, \ldots, a_k)$, не являющегося наибольшим, $m(a) = \max\{i \,|\, a_i \neq n - k + i,\ 1 \leq i \leq k\}$. Это равенство следует из возрастания элементов $a_i$ слева направо. Рассмотрим разбиение $A_0, A_1, \ldots, A_k$ множества всех сочетаний из $n$ по $k$, где

$$A_0 = \{(n-k+1, \ldots, n-1, n)\},$$

$$A_j = \{a = (a_1, \ldots, a_k) \,|\, a \text{ — сочетание и } m(a) = j\},\ 1 \leq j \leq k.$$

После печати произвольного сочетания $a \notin A_0$ внутренний for-цикл выполняется $k - m(a) + 1$ раз, а для наибольшего сочетания такой цикл не выполняется. Следовательно, $I_{n,k} = \sum_{j=1}^{k} |A_j|(k-j+1)$. Для любого сочетания $a \in A_j$ имеем $a_j < n-k+j$, $a_{j+1} = n-k+j+1, \ldots, a_{n-1} = n-1$, $a_n = n$. Следовательно, $|A_j| = C_{n-k+j-1}^{j}$ и

$$I_{n,k} = \sum_{j=1}^{k} C_{n-k+j-1}^{j}(k-j) + C_n^k - 1,$$

Далее, используя свойства биномиальных коэффициентов, получаем

$$\begin{aligned}
\sum_{j=1}^{k} C_{n-k+j-1}^{j}(k-j) &= \sum_{i=0}^{k} C_{n-i-1}^{k-i} - k = \\
&= kC_n^k - \sum_{i=0}^{k-1}(k-i)C_{n-i-1}^{k-i} - k = \\
&= kC_n^k - (n-k)\sum_{i=0}^{k-1} C_{n-i-1}^{k-i-1} - k = \\
&= kC_n^k - (n-k)C_n^{k-1} - k = \frac{k}{n+1-k}\, C_n^k - k.
\end{aligned}$$

Таким образом, справедливо равенство

$$I_{n,k} = \frac{n+1}{n+1-k}\, C_n^k - k - 1.$$

Так как $C_n^k \leq C_n^k\,(n+1)/(n+1-k)$, время работы алгоритма $CLex(n)$ есть $O(1)C_n^k\,(n+1)/(n+1-k)$. Теорема 7 доказана.

Заметим, что $(n+1)/(n+1-k) < 2$ при $k \leq (n/2)$, и следовательно алгоритм $CLex(n)$ генерации сочетаний является линейным. Однако $(n+1)/(n+1-k) = O(n)$ при $n-k = o(n)$. В этом случае можно применить алгоритм $CLex(n)$ для генерации сочетаний из $n$ по $n-k$, а затем полученные сочетания "дополнить" до сочетаний из $n$ по $k$.

# Глава 3
# Генерация случайных комбинаторных объектов

Рассмотрим конечную совокупность $\Xi$ комбинаторных объектов заданного типа (например, перестановки, подмножества, сочетания) и случайную величину $\xi$, принимающую значения из $\Xi$. Считаем, что задана вероятностная мера. Поэтому для любого объекта $\sigma \in \Xi$ определена вероятность $\mathsf{P}(\xi = \sigma)$, а следовательно, и дискретная функция распределения случайной величины $\xi$. Алгоритм, на выходе которого случайным образом получается некоторый комбинаторный объект из $\Xi$, задаёт случайную величину со значениями из $\Xi$. Общая задача состоит в построении алгоритма генерации случайного комбинаторного объекта $\xi \in \Xi$ с заданной функцией распределения. Мы будем рассматривать только равномерные распределения, и сгенерированный случайный объект с равномерным распределением называем *случайным объектом*. Равномерное распределение в этом случае означает равновероятность получения любого комбинаторного объекта из $\Xi$, т. е. $\mathsf{P}(\xi = \sigma) = 1/|\Xi|$ для каждого объекта $\sigma \in \Xi$.

## 1. Алгоритм построения случайной перестановки

Задача генерации таких случайных комбинаторных объектов, как случайный пароль заданной длины, случайный лабиринт, случайная конфигурация, как правило, приводит к задаче генерации случайной перестановки (либо случайной перестановки с какими-либо ограничениями).

Как для заданного $n$-элементного множества $A$ получить случайную перестановку? Случайность в данном случае означает равновероятность получения любой из $n!$ возможных перестановок множества $A$. Одна из поверхностных идей состоит в том, чтобы выбрать случайное целое число от 0 до $n!-1$ и по нему восстановить саму перестановку. Однако это восстановление потребует порядка $n^2$ операций. Такой подход неэффективен, так как случайную перестановку можно сгенерировать за линейное время $O(n)$.

Определим линейный алгоритм генерации случайной перестановки множества $A$. Начиная с произвольной перестановки $(a_1, \ldots, a_n)$ множества $A$, построим последовательность перестановок $\alpha^{n+1}, \alpha^n, \ldots, \alpha^1$, где $\alpha^{n+1} = (a_1, \ldots, a_n)$ и каждая $\alpha^i$ получается из перестановки $\alpha^{i+1} = (\alpha_1^{i+1}, \ldots, \alpha_n^{i+1})$ перестановкой $i$-го и $k$-го элементов перестановки $\alpha^{i+1}$. Причём $k$ выбирается как значение функции $rand(1, i)$, порождающей случайное целое число из интервала $[1, i]$ с равномерным распределением. Таким образом, мы определяем

$$\begin{gathered}
\alpha^{n+1} = (a_1, \ldots, a_n); \\
\alpha^n : \alpha_n^{n+1} \leftrightarrow \alpha_{k_n}^{n+1},\ k_n = rand(1, n); \\
\alpha^{n-1} : \alpha_{n-1}^{n} \leftrightarrow \alpha_{k_{n-1}}^{n},\ k_{n-1} = rand(1, n-1); \\
\vdots \\
\alpha^1 : \alpha_1^2 \leftrightarrow \alpha_{k_1}^2,\ k_1 = rand(1, 1) = 1.
\end{gathered}$$

Формализуем этот алгоритм в виде следующего простого псевдокода.

**Алгоритм** $PRand(n)$ **генерации случайной перестановки** $(\alpha_1, \ldots, \alpha_n)$ **множества** $\{a_1, \ldots, a_n\}$

**for** $i = 1$ **to** $n$ **do** $\alpha_i := a_i$;
**for** $i = n$ **downto** 1 **do**
**begin**
  $k := rand(1, i)$;
  $Swap(\alpha_i, \alpha_k)$
**end**.

**Теорема 8.** Алгоритм $PRand(n)$ строит случайную перестановку множества $\{a_1, \ldots, a_n\}$ за время $O(n)$.

**Доказательство.** Очевидно, что алгоритм $PRand(n)$ требует порядка $n$ операций. Индукцией по $n$ докажем, что для любой начальной перестановки $(a_1, \ldots, a_n)$ множества $A$ на выходе алгоритма $PRand(n)$ равновероятно получение любой из $n!$ перестановок множества $A$.

Базис индукции при $n = 1$ очевиден. Пусть утверждение справедливо для $n - 1$. Полагаем $\alpha^{n+1} = (a_1, \ldots, a_n)$ и через $\alpha^i = (\alpha_1^i, \ldots, \alpha_n^i)$ обозначим содержимое массива $\alpha$ после выполнения итерации второго for-цикла, соответствующей значению $i$. Тогда $\alpha^1$ есть результат работы алгоритма $PRand(n)$. Так как $\alpha^i$ получается из $\alpha^{i+1}$ перестановкой двух элементов, для любого $i = n+1, n, \ldots, 1$ имеем $\alpha^i \in S(a_1, \ldots, a_n)$, где $S(a_1, \ldots, a_n)$ — множество всех перестановок множества $A$. Следовательно, $\alpha^1 \in S(a_1, \ldots, a_n)$. Рассмотрим произвольную перестановку $\beta = (\beta_1, \ldots, \beta_n) \in S(a_1, \ldots, a_n)$. Пусть $\alpha' = (\alpha_1^1, \ldots, \alpha_{n-1}^1)$ и $\beta' = (\beta_1, \ldots, \beta_{n-1})$. Тогда

$$\mathsf{P}(\alpha^1 = \beta) = \mathsf{P}(\alpha_n^1 = \beta_n \,\&\, \alpha' = \beta') = \mathsf{P}(\alpha_n^1 = \beta_n)\ \mathsf{P}(\alpha' = \beta' \,|\, \alpha_n^1 = \beta_n)^{17}.$$

Из алгоритма понятно, что $\alpha_n^n = \alpha_n^{n-1} = \ldots = \alpha_n^1$. Поэтому $\alpha'$ есть результат работы алгоритма $PRand(n-1)$ генерации случайной перестановки множества $\{\alpha_1^n, \ldots, \alpha_{n-1}^n\}$ с начальной перестановкой $(\alpha_1^n, \ldots, \alpha_{n-1}^n)$. Если предположить $\alpha_n^1 = \beta_n$, то $\beta' \in S(\alpha_1^n, \ldots, \alpha_{n-1}^n)$. Тогда по индукционному предположению имеем

$$\mathsf{P}(\alpha' = \beta' \,|\, \alpha_n^1 = \beta_n) = \frac{1}{(n-1)!}\ .$$

Далее, существует индекс $m$ такой, что $\beta_n = a_m$. Следовательно,

$$\begin{aligned} \mathsf{P}(\alpha_n^1 = \beta_n) &= \mathsf{P}(\alpha_n^n = \beta_n) = \mathsf{P}(a_{rand(1,n)} = \beta_n) = \\ &= \mathsf{P}(a_{rand(1,n)} = a_m) = \mathsf{P}(rand(1,n) = m) = \frac{1}{n}\,. \end{aligned}$$

[17] $\mathsf{P}(A|B)$ — условная вероятность события $A$ при условии события $B$.

Таким образом, получаем

$$\mathsf{P}(\alpha^1 = \beta) = \frac{1}{n!}\,\frac{1}{(n-1)!} = \frac{1}{n!}\ .$$

Теорема 8 доказана.

## 2. Алгоритм генерации случайного подмножества и сочетания

Пусть $A = \{a_0, \ldots, a_{n-1}\}$ — $n$-элементное множество. Задача генерации случайного подмножества сводится к задаче порождения случайной двоичной последовательности длины $n$. Такая последовательность может быть получена в результате последовательного нахождения случайных целых чисел $b_i$ из интервала $[0, 1]$ при выполнении цикла

$$\textbf{for } i = 0 \textbf{ to } n-1 \textbf{ do } b_i = rand(0, 1).$$

Последовательность $(b_0, \ldots, b_{n-1})$ будет случайной. Действительно, для произвольного двоичного вектора $c = (c_0, \ldots, c_{n-1})$ имеем

$$\begin{aligned}\mathsf{P}(b = c) &= \mathsf{P}(b_0 = c_0 \,\&\ldots\&\, b_{n-1} = a_{n-1}) = \\ &= \prod_{i=0}^{n-1} \mathsf{P}(b_i = c_i) = \frac{1}{2^{\,n}} = \frac{1}{|\mathcal{P}(A)|}\,.\end{aligned}$$

Сложность описанной процедуры генерации случайного подмножества множества $A$ есть $O(n)$.

Рассмотрим задачу генерации случайного сочетания из $n$ элементов множества $A$ по $k$. Требуется построить случайное $k$-элементное подмножество $B \subseteq A$. Случайным образом выберем первый элемент $a_{r_1} \in A$, затем случайным образом выберем второй элемент $a_{r_2} \in A \setminus \{a_{r_1}\}$ из оставшихся $n-1$ элементов и продолжим этот процесс до тех пор, пока не выберем $a_{r_k} \in A \setminus \{a_{r_1}, \ldots, a_{r_{k-1}}\}$. Очевидно, полученное подмножество $B = \{a_{r_1}, \ldots, a_{r_k}\}$ является сочетанием из $n$ по $k$. Пусть $B' = \{b'_1, \ldots, b'_k\}$ — произвольное сочетание. Вычислим вероятность $\mathsf{P}(B = B')$. Обозначим $\mathcal{E}_1 = \langle\, a_{r_1} \in B' \,\rangle$, $\mathcal{E}_2 = \langle\, a_{r_2} \in B' \,|\, a_{r_1} \in B' \,\rangle$, $\ldots$,

$\mathcal{E}_k = \langle\, a_{r_k} \in B' \,|\, a_{r_1}, \ldots, a_{r_{k-1}} \in B' \,\rangle$. Тогда

$$\begin{aligned} \mathsf{P}(B = B') &= \mathsf{P}(a_{r_1} \in B' \;\&\; \ldots \;\&\; a_{r_k} \in B') = \\ &= \mathsf{P}(\mathcal{E}_1)\; \mathsf{P}(\mathcal{E}_2) \cdots \mathsf{P}(\mathcal{E}_k) = \\ &= \frac{k}{n}\,\frac{k-1}{n-1} \cdots \frac{1}{n-k+1} = \frac{1}{C_n^k}\,. \end{aligned}$$

Таким образом, в процессе описанной процедуры равновероятно получение любого сочетания из $n$ по $k$.

Перейдём к программной реализации требуемого алгоритма. Исходное множество зададим массивом $A = (a_1, \ldots, a_n)$ и случайное сочетание из $n$ по $k$ будем формировать на первых $k$ местах массива $A$. При этом мы сохраним исходное множество $\{a_1, \ldots, a_n\}$, но порядок элементов в массиве $A$ будет изменён. Предположим, что в $(j-1)$-м состоянии массива $A$ первые $j-1$ элементов $a_1^{j-1}, \ldots, a_{j-1}^{j-1}$ являются элементами требуемого сочетания $B$, а $a_j^{j-1}, \ldots, a_n^{j-1}$ есть оставшиеся элементы массива $A$. Среди этих оставшихся элементов случайным образом выберем элемент $a_i^{j-1}$, $j \le i \le n$ и поменяем местами $a_i^{j-1}$ и $a_j^{j-1}$. Тем самым придём к $j$-му состоянию массива $A$. Через $k$ шагов получим требуемое сочетание $B = (a_1^k, \ldots, a_k^k)$.

**Алгоритм генерации случайного $k$-элементного сочетания $n$-элементного множества $A = (a_1, \ldots, a_n)$ на первых $k$ местах**

**for** $j = 1$ **to** $k$ **do**
**begin**
  $i := rand(j, n)$;
  $Swap(a_i, a_j)$
**end**.

Сложность данного алгоритма есть $O(k)$.

Если нам требуется сохранить первоначальный порядок элементов исходного множества $A$, рассмотрим дополнительный массив $M = (m_1, \ldots, m_n)$. В $M$ будем записывать номера элементов множества $A$.

Вместо перестановки элементов $a_i$ и $a_j$, $j \le i \le n$ массива $A$ в соответствии с описанным алгоритмом переставляем номера $m_i$ и $m_j$ этих элементов. При этом достаточно изменять только номер $m_i$ с большим индексом $i$. Требуемое $k$-элементное сочетание записываем в массив $B = (b_1, \ldots, b_k)$. Программная реализация данного алгоритма приведена в следующем простом псевдокоде.

**Алгоритм генерации случайного $k$-элементного сочетания $B = (b_1, \ldots, b_k)$ $n$-элементного множества $A = (a_1, \ldots, a_n)$**

**for** $j = 1$ **to** $n$ **do** $m_j := j$;
**for** $j = 1$ **to** $k$ **do**
**begin**
  $i := rand(j, n)$;
  $b_j := a_{m_i}$;
  $m_i := m_j$
**end**.

# Глава 4
# Разбиения чисел и множеств

В этой главе мы рассмотрим линейные алгоритмы генерации разбиений числа $n$ и разбиений $n$-элементного множества.

## 1. Упорядоченные и неупорядоченные разбиения числа $n$

**Определение 12.** *Разбиением целого положительного числа* $n$ называется представление $n$ в виде суммы $n = x_1 + x_2 + \ldots + x_k$ целых положительных чисел $x_i$, $i = 1, \ldots, k$.

Если дополнительно фиксируется порядок слагаемых $x_1, \ldots, x_k$, последовательность $(x_1, \ldots, x_k)$ называется *упорядоченным разбиением*

*числа* $n$. Существует взаимно-однозначное соответствие между упорядоченными разбиениями числа $n$ на $k$ слагаемых и $(k-1)$-элементными подмножествами $(n-1)$-элементного множества, именно упорядочиванию соответствует расстановка $k-1$ пробелов в $n-1$ промежутках между $n$ единицами, записанными подряд. Поэтому число всех упорядоченных разбиений на $k$ слагаемых есть $C_{n-1}^{k-1}$. Следовательно, число всех упорядоченных разбиений числа $n$ равно $\sum_{k=1}^{n} C_{n-1}^{k-1} = 2^{n-1}$, и существует взаимно-однозначное соответствие между всеми упорядоченными разбиениями числа $n$ и всеми подмножествами $(n-1)$-элементного множества. Таким образом, задача генерации упорядоченных разбиений числа $n$ на $k$ слагаемых сводится к уже решённой задаче генерации сочетаний из $n-1$ по $k-1$, а задача генерации всех упорядоченных разбиений числа $n$ сводится к решённой задаче генерации всех подмножеств $(n-1)$-элементного множества.

Теория *неупорядоченных разбиений* значительно сложнее. Напомним, что при неупорядоченном разбиении два разбиения числа $n$ считаются равными, если они отличаются лишь порядком слагаемых. Например, для $n = 4$ разбиения $2+1+1,\ 1+2+1,\ 1+1+2$ равны. Класс равных неупорядоченных разбиений числа $n$ однозначно задается последовательностью $(a_1, \ldots, a_k)$ такой, что $n = a_1 + \ldots + a_k$ и $a_1 \geq a_2 \geq \ldots \geq a_k > 0$, $k \geq 1$. В дальнейшем будем рассматривать только неупорядоченные разбиения, принимая эту интерпретацию. Обозначим число разбиений числа $n$ на $k$ слагаемых через $p_k(n)$, а число всех разбиений числа $n$ через $p(n)$. Очевидно, $p(n) = \sum_{k=1}^{n} p_k(n)$. Для числа $p(n)$ известно следующее асимптотическое выражение[18]:

$$p(n) \sim \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}\ ^{19}.$$

[18] Получено G. H. Hardy и S. Ramanujan [8, т. 4; 21].

[19] $f(n) \sim g(n)$, если $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

## 2. Генерация разбиений числа $n$ в словарном порядке

На множестве целочисленных последовательностей определим *словарный порядок* $\preceq$[20]. Пусть $a = (a_1, \ldots, a_q)$ и $b = (b_1, \ldots, b_s)$ — произвольные целочисленные последовательности, возможно разной длины.

**Определение 13.** $(a_1, \ldots, a_q) \preceq (b_1, \ldots, b_s)$, если выполняется хотя бы одно из условий: либо $q \leq s$ и $a_i = b_i$ для любого $i \leq q$, либо существует $p \leq \min(s, q)$ такое, что $a_p < b_p$ и $a_i = b_i$ для всех $i < p$.

Бинарное отношение $\preceq$ является линейным порядком. Для разбиений числа $n$ определение словарного порядка несколько упрощается.

**Замечание 3.** Если $a$, $b$ — разбиения числа $n$, то

$$a \prec b \Leftrightarrow \exists\, p \leq \min(s, q)\ (a_p < b_p\ \&\ \forall\, i < p\ (a_i = b_i)).$$

Разбиения числа $n$ будем порождать в словарном порядке. Ясно, что первым разбиением (наименьшим элементом) в словарном порядке будет последовательность $(1, 1, \ldots, 1)$ длины $n$, а последним (наибольшим элементом) — одноэлементная последовательность $(n)$. Выясним вид разбиения, непосредственно следующего за разбиением $a = (a_1, \ldots, a_q)$ в словарном порядке. Найдём такую самую правую позицию $p < q$, в которой число $a_p$ можно увеличить на 1, сохранив свойство убывания последовательности. Далее сумму $\left(\sum_{i=p+1}^{q} a_i - 1\right)$ оставшихся слагаемых за вычетом 1 представим в виде суммы единиц. Нетрудно доказать, что так определенное разбиение $b$ непосредственно следует за $a$.

При переходе от $a$ к $b$ число необходимых изменений над последовательностью $a$ есть величина переменная, зависящая от $n$. Эту зависимость можно устранить, если перейти к другому способу задания разбиения $a = (a_1, \ldots, a_q)$. Упорядочим все различные числа $a_{i_1}, \ldots, a_{i_k}$ среди

[20] Сравните с лексикографическим порядком.

$a_1, \ldots, a_q$ в порядке убывания $a_{i_1} > \ldots > a_{i_k}$. Пусть $m_j$ — число вхождений $a_{i_j}$ в разбиение $a$ и $m = (m_1, \ldots, m_k)$. Ясно, что разбиение $a$ числа $n$ однозначно определяется парой последовательностей $(a_{i_1}, \ldots, a_{i_k})$ и $(m_1, \ldots, m_k)$. Поэтому в дальнейшем всякое разбиение $a$ числа $n$ будем записывать в виде $a = (m_1 \cdot a_1, \ldots, m_k \cdot a_k)$, где $a_1 > \ldots > a_k > 0$, $m_i > 0$, $i = 1, \ldots, k$, $1 \leq k \leq n$ и $n = \sum_{i=1}^{k} m_i a_i$. Элемент $m_i \cdot a_i$ представляет собой последовательность $a_i, a_i, \ldots, a_i$ длины $m_i$ и называется *блоком разбиения*. Очевидно, что разбиение $a$ является наименьшим при $k = 1$ и $m_k = n$, а наибольшим при $k = m_k = 1$. Такое представление разбиений числа $n$ исключает необходимость поиска позиции $p$ при просмотре справа налево текущего разбиения. Как показывает следующая лемма, для перестроения разбиения $a$ в непосредственно следующее разбиение $b$ потребуется изменить не более двух блоков.

**Лемма 8.** Пусть $a = (m_1 \cdot a_1, \ldots, m_k \cdot a_k)$ — разбиение числа $n$ и разбиение $b$ непосредственно следует за $a$ в словарном порядке. Тогда

(i) если $m_k = 1$, то $k \geq 2$ и

$$b = (m_1 \cdot a_1, \ldots, m_{k-2} \cdot a_{k-2}, 1 \cdot (a_{k-1} + 1), \Sigma' \cdot 1);$$

(ii) если $m_k \geq 2$, $k \geq 2$ и $a_{k-1} = a_k + 1$, то

$$b = (m_1 \cdot a_1, \ldots, m_{k-2} \cdot a_{k-2}, (m_{k-1} + 1) \cdot a_{k-1}, \Sigma \cdot 1);$$

(iii) если $m_k \geq 2$, $k \geq 2$ и $a_{k-1} \neq a_k + 1$, то

$$b = (m_1 \cdot a_1, \ldots, m_{k-1} \cdot a_{k-1}, 1 \cdot (a_k + 1), \Sigma \cdot 1);$$

(iv) если $k = 1$, то $b = (1 \cdot (a_k + 1), \Sigma \cdot 1)$.

Здесь $\Sigma' = m_k\, a_k + m_{k-1} a_{k-1} - (a_{k-1} + 1)$ и $\Sigma = m_k\, a_k - (a_k + 1)$.

**Доказательство.** При переходе от $a$ к $b$ необходимо самый правый возможный элемент разбиения $a$ увеличить на 1. Такой элемент $x$ будет первым элементом блока, в который он входит. Рассмотрим два возможных случая.

Случай 1. $m_k = 1$. Элемент $a_k$ нельзя увеличить, сохранив разбиение числа $n$. Так как $a$ не наибольшее разбиение, имеем $k \geq 2$. Поскольку

$a_{k-1}+a_k \geq a_k+1$, то $x = a_{k-1}$ и $x$ является первым элементом $(k-1)$-го блока. Этот элемент увеличиваем на 1, а сумму оставшихся слагаемых $\Sigma' = m_k\, a_k + m_{k-1}a_{k-1} - (a_{k-1}+1)$ представляем в виде суммы единиц.

Случай 2. $m_k \geq 2$. Тогда $m_k\, a_k \geq a_k + 1$. Следовательно, $x = a_k$ и $x$ является первым элементом $k$-го блока. Этот элемент увеличиваем на 1, а сумму оставшихся слагаемых $\Sigma = m_k\, a_k - (a_k + 1)$ представляем в виде суммы единиц.

В каждом из этих случаев остаётся пересчитать значения $a_i$ и $m_i$ для не более, чем двух изменившихся блоков. Лемма 8 доказана.

**Замечание 4.** Выбирая значение $a_0$ такое, что $a_0 \neq a_1+1$, в лемме 8 можно исключить (iv) и условие $k \geq 2$ в (ii),(iii), сохранив справедливым утверждение леммы.

**Пример 7.** Разбиения числа $n = 7$ в словарном порядке:
$(7 \cdot 1) = (1, 1, 1, 1, 1, 1, 1)$,
$(1 \cdot 2, 5 \cdot 1) = (2, 1, 1, 1, 1, 1)$,
$(2 \cdot 2, 3 \cdot 1) = (2, 2, 1, 1, 1)$,
$(3 \cdot 2, 1 \cdot 1) = (2, 2, 2, 1)$,
$(1 \cdot 3, 4 \cdot 1) = (3, 1, 1, 1, 1)$,
$(1 \cdot 3, 1 \cdot 2, 2 \cdot 1) = (3, 2, 1, 1)$,
$(1 \cdot 3, 2 \cdot 2) = (3, 2, 2)$,
$(2 \cdot 3, 1 \cdot 1) = (3, 3, 1)$,
$(1 \cdot 4, 3 \cdot 1) = (4, 1, 1, 1)$,
$(1 \cdot 4, 1 \cdot 2, 1 \cdot 1) = (4, 2, 1)$,
$(1 \cdot 4, 1 \cdot 3) = (4, 3)$,
$(1 \cdot 5, 2 \cdot 1) = (5, 1, 1)$,
$(1 \cdot 5, 1 \cdot 2) = (5, 2)$,
$(1 \cdot 6, 1 \cdot 1) = (6, 1)$,
$(1 \cdot 7) = (7)$.

Перейдём к программной реализации рассмотренного алгоритма.

**Алгоритм $PartN(n)$ генерации разбиений числа $n$ словарном порядке**

```
m_0 := 2;
a_0 := 1 − n;
a_1 := 1;
m_1 := n;
k := 1;
while k ≠ 0 do
begin
    write(m_1 · a_1, ..., m_k · a_k);
    Σ := m_k a_k;
    if m_k = 1 then begin
                        k := k − 1;
                        Σ := Σ + m_k a_k − (a_k + 1);
                        m_k := 1;
                        a_k := a_k + 1
                    end
               else begin
                        Σ := Σ − (a_k + 1);
                        if a_{k−1} = a_k + 1 then begin
                                                      k := k − 1;
                                                      m_k := m_k + 1
                                                  end
                                             else begin
                                                      a_k := a_k + 1;
                                                      m_k := 1
                                                  end
                    end;
    if Σ ≠ 0 then begin
                      k := k + 1;
                      a_k := 1;
                      m_k := Σ
                  end
end.
```

**Теорема 9.** Алгоритм $PartN(n)$ корректен и генерирует все разбиения числа $n$ без повторений в словарном порядке за время $O(p(n))$.

**Доказательство.** Нетрудно проверить, что значения переменных $k$, $m_i$, $a_i$, $i = 1, \ldots, k$ при инициализации соответствуют наименьшему разбиению $(n \cdot 1)$, а в теле while-цикла вычисляются согласно формулам из леммы 8(i)-(iii). Через $a_0^*$, $m_0^*$, $a_1^*$ соответственно обозначим значения переменных $a_0$, $m_0$, $a_1$ при их инициализации. Тогда[21]

$$n + (m_0^* - 1)\, a_0^* - 1 = 0;$$

$$a_0^* \neq a_1^* + 1, \text{ если } n > 1;$$

$$a_0^* < 1 \text{ или } a_0^* > n.$$

Из алгоритма понятно, что переменная $a_0$ не изменяет своего значения $a_0^*$ вплоть до тех пор, пока не будет выполнен оператор печати $\mathrm{write}(m_1 \cdot a_1, \ldots, m_k \cdot a_k)$ при $k = m_k = 1$. Кроме того, $0 < a_1 < n$ для любого разбиения $(m_1 \cdot a_1, \ldots, m_k \cdot a_k)$ числа $n$, за исключением случая, когда $k = m_k = 1$. Используя лемму 8 и замечание 4, по индукции получаем, что после печати текущего разбиения числа $n$, начиная с разбиения $(n \cdot 1)$, будет напечатано непосредственно следующее в словарном порядке разбиение $(m_1 \cdot a_1, \ldots, m_k \cdot a_k)$. Причём в момент его печати будет выполняться неравенство $a_0 \neq a_1 + 1$, за исключением печати наибольшего разбиения $(1 \cdot n)$, когда $k = m_k = 1$. Теперь из алгоритма понятно, что переменная $m_0$ не изменит своего первоначального значения $m_0^*$ вплоть до печати разбиения $(1 \cdot n)$.

Рассмотрим работу тела while-цикла после печати разбиения $(1, n)$. Имеем $k = m_1 = 1$, $a_1 = n$, $a_0 = a_0^*$, $m_0 = m_0^*$. После выполнения первого условного оператора получим $k = 0$ и $\Sigma = m_1 a_1 + m_0^*\, a_0^* - (a_0^* + 1) = 0$. Поэтому переменная $k$ не изменит своего нулевого значения после выполнения последнего условного оператора. Таким образом, алгоритм $PartN(n)$ закончит работу.

[21] Здесь рассмотрена более общая ситуация, чем в алгоритме $PartN(n)$, чтобы понять роль значений $a_0^*$, $m_0^*$.

Поскольку число операций, необходимых для перехода от одного разбиения к непосредственно следующему, ограничено константой, не зависящей от $n$, данный алгоритм генерации $PartN(n)$ является линейным. Теорема 9 доказана.

Заметим, что в алгоритме $PartN(n)$ значения переменных $a_0$, $m_0$ при инициализации могут быть заданы произвольным образом при условии выполнения трёх соотношений, приведенных в доказательстве теоремы 9. В частности, можно определить $a_0^* = n + 1$ и $m_0^* = 0$.

# 3. Разбиения конечного множества

**Определение 14.** Семейство множеств $\mathcal{A} = \{\mathcal{A}_i\}_{i \in I}$, где $I$ — некоторое множество индексов, называется *разбиением множества* $A$, если $A = \bigcup\limits_{i \in I} \mathcal{A}_i$, $\mathcal{A}_i \bigcap \mathcal{A}_j = \emptyset$ при $i \neq j$ и $\mathcal{A}_i \neq \emptyset$ для любых $i, j \in I$. При этом подмножества $\mathcal{A}_i$, $i \in I$ называются *блоками* разбиения $\mathcal{A}$.

Для конечного множества $A$ в качестве множества индексов $I$ рассматриваются конечные множества $\{1, 2, \ldots, k\}$ и число $k$ называется *числом блоков разбиения*. Число всех разбиений[22] $n$-элементного множества называется *числом Белла* и обозначается $B_n$. Более формально, $B_n = |\,\Pi(A)|$, где $\Pi(A)$ — множество всех разбиений $n$-элементного множества $A$. Число разбиений $n$-элементного множества на $k$ блоков называется *числом Стирлинга второго рода* и обозначается $\mathcal{S}(n, k)$. При этом принимают $B_0 = 1$ и $\mathcal{S}(0, 0) = 1$. Очевидно, что $B_n = \sum_{k=0}^{n} \mathcal{S}(n, k)$.

Для чисел Белла справедливо следующее рекуррентное выражение через биномиальные коэффициенты:

$$B_{n+1} = \sum_{i=0}^{n} C_n^i B_i, \;\; B_0 = 1.$$

[22] Подсчитываются неупорядоченные разбиения, когда порядок блоков $\mathcal{A}_1, \ldots, \mathcal{A}_k$ разбиения $\mathcal{A}$ не важен. Два разбиения считаются равными, если они равны как множества.

Действительно, имеется $B_{n+1}$ разбиений $(n+1)$-элементного множества $A = \{1, 2, \ldots, n+1\}$. Рассмотрим произвольное разбиение $\mathcal{A} \in \Pi(A)$. Число $n+1$ принадлежит некоторому $(i+1)$-элементному блоку разбиения $\mathcal{A}$, где $0 \le i \le n$. Очевидно, что существует $C_n^i$ возможностей для выбора такого блока. Оставшееся множество из $n-i$ чисел может быть разбито $B_{n-i}$ способами. Суммируя по всем допустимым $i$, получаем $B_{n+1} = \sum_{i=0}^{n} C_n^i B_{n-i} = \sum_{i=0}^{n} C_n^i B_i$.

Для числа $B_n$ известно следующее асимптотическое выражение[23]:

$$B_n \sim \Big(\frac{n}{\ln n}\Big)^n.$$

# 4. Генерация разбиений $n$-элементного множества

Познакомимся с алгоритмом генерации всех разбиений $n$-элементного множества $A = \{1, 2, \ldots, n\}$ в порядке минимального изменения. Пусть $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ — разбиение множества $A$ и $a_i$ — наименьший элемент блока $\mathcal{A}_i$. Упорядочим элементы $a_i$, $i = 1, \ldots, k$ в порядке возрастания $a_{i_1} < \ldots < a_{i_k}$. В соответствии с таким порядком определим упорядочение блоков разбиения $\mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_k}$. Другими словами, разбиение $\mathcal{A}$ представим в виде последовательности блоков $\mathcal{A} = (\mathcal{A}_{i_1}, \ldots, \mathcal{A}_{i_k})$, упорядоченной по возрастанию наименьших элементов, содержащихся в блоке. В дальнейшем под разбиением будем понимать именно такую последовательность. Наименьший элемент блока будем называть *номером блока*. Отметим, что номера соседних блоков, вообще говоря, не являются соседними числами.

Идея алгоритма, который мы будем рассматривать, близка к идее алгоритма генерации перестановок в порядке минимального изменения. Каждое следующее разбиение будет получаться из предыдущего в результате минимального изменения разбиения, а именно посредством "перемещения" некоторого элемента из одного блока в соседний. Под

[23] См., например, [8, т. 4].

"перемещением" элемента мы понимаем удаление этого элемента из блока (что может привести к удалению всего одноэлементного блока) и либо добавление его в соседний блок, либо создание из него одноэлементного соседнего блока.

Построим список всех разбиений $\mathcal{A}^i$, $i = 1, \ldots, B_n$ $n$-элементного множества $A = \{1, \ldots, n\}$ без повторений со следующими свойствами:

- для произвольного $x \in A$ в любом разбиении $\mathcal{A}^i$ над элементом $x$ указано возможное направление перемещения при помощи стрелки $\rightarrow$ ("вправо") или $\leftarrow$ ("влево");
- каждое следующее разбиение $\mathcal{A}^{i+1}$ получается из предыдущего $\mathcal{A}^i$ перемещением одного элемента $x_i$ из некоторого блока в соседний блок. Перемещение осуществляется в направлении, указанном в разбиении $\mathcal{A}^i$ над этим элементом $x_i$;
- $\mathcal{A}^1 = (\{\overrightarrow{1}, \overrightarrow{2}, \ldots, \overrightarrow{n}\})$.

Построение проведем индукцией по $n$. Для одноэлементного множества $A$ существует единственное разбиение, состоящее из одного блока с единственным элементом 1. Для элемента 1 определим направление $\overrightarrow{1}$. Теперь предположим, что $\mathcal{B}^i$, $i = 1, \ldots, B_{n-1}$ — список всех разбиений $(n-1)$-элементного множества $\{1, \ldots, n-1\}$ с указанными свойствами. Каждому разбиению $\mathcal{B}^i = (\mathcal{B}^i_1, \ldots, \mathcal{B}^i_{k_i})$ при нечётном $i$ сопоставим последовательность $k_i + 1$ разбиений $n$-элементного множества $A$, последовательно добавляя в каждый блок $\mathcal{B}^i_j$ слева направо $(j = 1, \ldots, k_i)$ элемент $\overrightarrow{n}$ и на последнем шаге (при $j = k_i + 1$) добавляя к разбиению $\mathcal{B}^i$ одноэлементный блок $\{\overrightarrow{n}\}$. Каждому разбиению $\mathcal{B}^i = (\mathcal{B}^i_1, \ldots, \mathcal{B}^i_{k_i})$ при чётном $i$ сопоставим последовательность $k_i + 1$ разбиений $n$-элементного множества $A$, добавляя на первом шаге (при $j = k_i + 1$) к разбиению $\mathcal{B}^i$ одноэлементный блок $\{\overleftarrow{n}\}$ и далее последовательно добавляя в каждый блок $\mathcal{B}^i_j$ справа налево $(j = k_i, \ldots, 1)$ элемент $\overleftarrow{n}$. Нетрудно доказать, что полученный список разбиений $n$-элементного множества $A$ обладает требуемыми свойствами.

Описанный метод построения списка разбиений $\mathcal{A}^i$, $i = 1, \ldots, B_n$ даёт рекурсивный алгоритм генерации всех разбиений $n$-элементного множества. Однако такой алгоритм потребует большого объёма памяти. Чтобы устранить этот недостаток и отказаться от рекурсии, выясним, какой элемент $x_i \in A$ перемещается в $\mathcal{A}^i$ при переходе к следующему разбиению $\mathcal{A}^{i+1}$ и каким образом. Введём следующие обозначения:

- $Block(x)$ — номер блока, содержащего элемент $x$. Очевидно, что элементы $x, y$ принадлежат одному блоку разбиения тогда и только тогда, когда $Block(x) = Block(y)$. Более того, последовательность $(Block(1), \ldots, Block(n))$ полностью определяет разбиение;

- $Next(i)$ — номер блока, непосредственно следующего за блоком с номером $i$. Полагаем $Next(i) = n + 1$, если $i$ — номер последнего блока;

- $Prev(i)$ — номер предыдущего блока для блока с номером $i$. Полагаем $Prev(i) = 0$, если $i$ — номер первого блока;

- $Dir(x)$ — число, определяющее возможное направление перемещения элемента $x$. $Dir(x) = 1$, если $x$ перемещается вправо, и $Dir(x) = -1$, если $x$ перемещается влево.

Используя рекурсивное определение списка разбиений $\mathcal{A}^i$, $i = 1, \ldots, B_n$ множества $A = \{1, \ldots, n\}$, индукцией по $n$ нетрудно доказать следующие свойства разбиений этого списка.

**Лемма 9.** Пусть $x \in A = \{1, \ldots, n\}$. Для любого построенного разбиения $\mathcal{A}^i = (\mathcal{A}^i_1, \ldots, \mathcal{A}^i_{k_i})$, $i = 1, \ldots, B_n$ множества $A$ справедливы следующие свойства:

(i) номер первого блока $\mathcal{A}^i_1$ равен 1;

(ii) элемент $x$ перемещается тогда и только тогда, когда $x$ есть наибольшее число, удовлетворяющее одному из следующих двух условий: либо $Block(x) \neq x$ и $Dir(x) = 1$, либо $Block(x) \neq 1$ и $Dir(x) = -1$;

(iii) если $x$ перемещается, то создаётся одноэлементный блок $\{x\}$ тогда и только тогда, когда $Dir(x) = 1$ и $\mathrm{Next}(Block(x)) > x$;

(iv) если $x$ перемещается, то удаляется одноэлементный блок $\{x\}$ тогда и только тогда, когда $Dir(x) = -1$ и $Block(x) = x$;

(v) если $x$ перемещается и $y \in A$, то направление элемента $y$ меняется на противоположное тогда и только тогда, когда $y > x$;

(vi) при $i = B_n$ для любого элемента $x \in A$ не выполняется каждое из двух условий, указанных в (ii).

Лемма 9 даёт конструктивный способ нахождения перемещаемого элемента $x$, а также условия создания и удаления одноэлементного блока $\{x\}$ при перемещении $x$. Остаётся только переопределить значения $Next(i)$, $Prev(i)$, $Block(x)$, $Dir(x)$ для изменившихся блоков при перемещении $x$ из одного блока в соседний блок. Описанный процесс построения разбиений реализован в нижеприведённом алгоритме $PartS(n)$.

**Пример 8.** Генерация всех разбиений множества $\{1, 2, 3, 4\}$:
$(\overrightarrow{1}\,\overrightarrow{2}\,\overrightarrow{3}\,\overrightarrow{4})$, $x = 4$, $i = 1$, $Next(i) > x$;
$(\overrightarrow{1}\,\overrightarrow{2}\,\overrightarrow{3})(\overrightarrow{4})$, $x = 3$, $i = 1$, $Next(i) > x$;
$(\overrightarrow{1}\,\overrightarrow{2})(\overrightarrow{3})(\overleftarrow{4})$, $x = 4$, $i = 4$, $Block(x) = x$;
$(\overrightarrow{1}\,\overrightarrow{2})(\overrightarrow{3}\,\overleftarrow{4})$, $x = 4$, $i = 3$, $Block(x) \neq x$;
$(\overrightarrow{1}\,\overrightarrow{2}\,\overleftarrow{4})(\overrightarrow{3})$, $x = 2$, $i = 1$, $Next(i) > x$;
$(\overrightarrow{1}\,\overrightarrow{4})(\overrightarrow{2})(\overleftarrow{3})$, $x = 4$, $i = 1$, $Next(i) \not> x$;
$(\overrightarrow{1})(\overrightarrow{4}\,\overrightarrow{2})(\overleftarrow{3})$, $x = 4$, $i = 2$, $Next(i) \not> x$;
$(\overrightarrow{1})(\overrightarrow{2})(\overleftarrow{3}\,\overrightarrow{4})$, $x = 4$, $i = 3$, $Next(i) > x$;
$(\overrightarrow{1})(\overrightarrow{2})(\overleftarrow{3})(\overrightarrow{4})$, $x = 3$, $i = 3$, $Block(x) = x$;
$(\overrightarrow{1})(\overrightarrow{2}\,\overleftarrow{3})(\overleftarrow{4})$, $x = 4$, $i = 4$, $Block(x) = x$;
$(\overrightarrow{1})(\overrightarrow{2}\,\overleftarrow{3}\,\overleftarrow{4})$, $x = 4$, $i = 2$, $Block(x) \neq x$;
$(\overrightarrow{1}\,\overleftarrow{4})(\overrightarrow{2}\,\overleftarrow{3})$, $x = 3$, $i = 2$, $Block(x) \neq x$;
$(\overrightarrow{1}\,\overrightarrow{4}\,\overleftarrow{3})(\overrightarrow{2})$, $x = 4$, $i = 1$, $Next(i) \not> x$;
$(\overrightarrow{1}\,\overleftarrow{3})(\overrightarrow{2}\,\overrightarrow{4})$, $x = 4$, $i = 2$, $Next(i) > x$;
$(\overrightarrow{1}\,\overleftarrow{3})(\overrightarrow{2})(\overrightarrow{4})$, $x = 1$.

**Алгоритм $PartS(n)$ генерации всех разбиений множества $\{1, 2, \ldots, n\}$**

```
for i = 1 to n do
begin
  Block(i) := 1;
  Dir(i) := 1
end;
Next(1) := n + 1;
Prev(1) := 0;
x := 0;
while x ≠ 1 do
begin
  write(Block(1), ..., Block(n));
  x := n;
  while (x > 1) &
     ((Block(x) = x & Dir(x) = 1) ∨ (Block(x) = 1 & Dir(x) = −1)) do
  begin
    Dir(x) := −Dir(x);
    x := x − 1
  end;
  i := Block(x);
  if Dir(x) = 1
   then begin if Next(i) > x then
                    begin if Next(i) = n + 1 then Next(x) := n + 1
                                             else begin
                                                    Next(x) := Next(i);
                                                    Prev(Next(i)) := x
                                                  end;
                          Next(i) := x;
                          Prev(x) := i
                    end;
                Block(x) := Next(i)
        end;
   else begin if Block(x) = x then
                     if Next(i) = n + 1 then Next(Prev(i)) := n + 1
                                        else begin
                                               Next(Prev(i)) := Next(i);
                                               Prev(Next(i)) := Prev(i)
                                               end;
                Block(x) := Prev(i)
        end
end.
```

**Теорема 10.** Алгоритм $PartS(n)$ корректен и генерирует все разбиения $n$-элементного множества без повторений за время $O(B_n)$.

**Доказательство.** Рассмотрим построенный список $\mathcal{A}^i$, $i = 1, \ldots, B_n$ всех разбиений $n$-элементного множества $A$ без повторений. Очевидно, что разбиение $\mathcal{A}^1 = (\{\overrightarrow{1}, \ldots, \overrightarrow{n}\})$ определяется при инициализации переменных $Block(i)$, $Dir(i)$. В силу построения списка $\mathcal{A}^i$, $i = 1, \ldots, B_n$ каждое следующее разбиение $\mathcal{A}^{i+1}$ получается из предыдущего $\mathcal{A}^i$ в результате перемещения некоторого элемента $x_i$ в соседний блок в направлении, соответствующем значению переменной $Dir(x_i)$. Согласно лемме 9 (i)–(ii) перемещаемый элемент $x_i$ находится после окончания работы внутреннего while-цикла. В соответствии с леммой 9 (v) в этом цикле также меняются направления всех элементов, больших $x_i$, на противоположные. Далее в зависимости от значения переменной $Dir(x_i)$ осуществляется перемещение элемента $x_i$ в соседний блок.

Если $x_i$ перемещается вправо, то значение переменной $Block(x_i)$ изменяется на $Next(Block(x_i))$ в случае, когда блок с номером $Block(x_i)$ не является последним, и на $x_i$, если это последний блок. Когда при этом создаётся одноэлементный блок $\{x_i\}$ (согласно лемме 9 (iii) это полностью определяется условием $Next(Block(x_i)) > x_i$), дополнительно перенумеровываются изменившиеся блоки путём переопределения значений переменных $Next$ и $Prev$. Если $x_i$ перемещается влево, то значение переменной $Block(x_i)$ изменяется на $Prev(Block(x_i))$. Когда при этом удаляется одноэлементный блок $\{x_i\}$ (согласно лемме 9 (iv) это полностью определяется условием $Block(x_i) = x_i$), дополнительно перенумеровываются изменившиеся блоки.

В силу леммы 9 (vi) после печати последнего разбиения $\mathcal{A}^{B_n}$ и выполнения всех итераций внутреннего while-цикла переменная $x$ примет значение 1. Следовательно, алгоритм закончит работу.

Оценим время работы $T(n)$ алгоритма $PartS(n)$. Число проверок условия внутреннего while-цикла обозначим через $I_n$. Индукцией по $n$

докажем, что выполняется равенство

$$I_n = \sum_{i=1}^{n} B_i.$$

Действительно, при $n = 1$ имеем $I_1 = B_1 = 1$. Пусть для $n-1$ формула верна. Учитывая рекурсивное определение списка $\mathcal{A}^i$ всех разбиений $n$-элементного множества через список $\mathcal{B}^j$ всех разбиений $(n-1)$-элементного множества, нетрудно доказать равенство $I_n = B_n + I_{n-1}$. Применяя индукционное предположение, получаем требуемое выражение для $I_n$. В силу рекуррентной формулы для чисел Белла имеем

$$\sum_{i=1}^{n-1} B_i \leq \sum_{i=0}^{n-1} C_{n-1}^{i} B_i = B_n.$$

Следовательно, $I_n = O(B_n)$. Очевидно, что $T(n) = O(n) + O(I_n) = O(B_n)$. Теорема 10 доказана.

# Глава 5
# Сортировка комбинаторных объектов

Во многих компьютерных приложениях требуется переразместить заданную совокупность комбинаторных объектов в соответствии с некоторым заранее определённым порядком. Например, необходимо расположить данные по алфавиту или по возрастанию номеров либо отсортировать товары по ценам. Всё это разновидности задачи сортировки, которой посвящена настоящая глава.

Сначала задача сортировки рассматривается с теоретической точки зрения, чтобы получить некоторое представление об ожидаемой эффективности алгоритмов сортировки. Для оценки эффективности оказываются полезными понятия минимального, среднего и максимального времени работы алгоритма. Мы установим нижние оценки времени работы в худшем и среднем случае для алгоритмов сортировки, основанных на принципе попарного сравнения элементов входной последовательности, для этого познакомимся со свойствами бинарных деревьев. Любая сортировка такого вида уже в среднем требует времени $\Omega(n \lg n)$.

Далее детально изучим алгоритмы сортировки вставками, пузырьковой, быстрой и пирамидальной сортировки. Для каждого из рассмотренных алгоритмов будет обоснован асимптотический порядок минимального, среднего и максимального времени их работы. В классе алгоритмов сортировки сравнением быстрая сортировка со средним временем работы $\Theta(n \lg n)$ является асимптотически оптимальной в среднем, а пирамидальная сортировка со сложностью $\Theta(n \lg n)$ — в худшем случае. Линейную сложность можно получить, если при сортировке использовать не только сравнения входных элементов, а, например, специальные представления сортируемых записей или какую-либо дополнительную информацию о порядке расположения или природе входных элементов. В качестве примера такого алгоритма изучается алгоритм сортировки подсчётом со временем работы $\Theta(n)$.

# 1. Задача сортировки

Сортируемые объекты, как правило, являются записями, содержащими одно или несколько полей. Пусть задана последовательность таких записей $x_1, \ldots, x_n$ и на множестве $X$ всех записей, входящих в эту последовательность[24], определен некоторый линейный порядок $\leq$. *Задача сортировки* состоит в переразмещении записей в порядке возрастания, более формально в построении упорядочивания $x_{\pi_1} \leq x_{\pi_2} \leq \cdots \leq x_{\pi_n}$ заданной последовательности записей $x_1, \ldots, x_n$ относительно линейного порядка $\leq$. При этом последовательность $\pi = (\pi_1, \ldots, \pi_n)$ будет перестановкой $n$-элементного множества $\{1, 2, \ldots, n\}$. В алгоритмах сортировки мы будем получать саму упорядоченную последовательность записей, а не упорядочивающую перестановку $\pi$. Хотя иногда бывает удобнее сначала получить перестановку $\pi$, а затем по ней выписать упорядоченную последовательность записей.

Программное задание сортируемых записей часто приводит к тому, что в каждой записи имеется (добавляется) специальное отведённое по-

[24] Записи $x_1, \ldots, x_n$ не обязательно различные.

ле, называемое *ключом*, и записи упорядочиваются относительно линейного порядка для ключей[25]. Поэтому в дальнейшем, если не оговорено особо, считаем, что $X$ есть множество ключей с заданным линейным порядком $\leq$. При работе алгоритма сортировки записи могут сортироваться на месте, т. е. в той же области, которая была отведена для входной последовательности данных, или с использованием дополнительной памяти. В первом случае переразмещение записей должно происходить внутри входной сортируемой последовательности $x_1, \ldots, x_n$. Такое ограничение основано на предположении, что число записей настолько велико, что во время сортировки не допускается перенос их в другую область памяти. При этом, конечно, разрешается использование ограниченного количества ячеек памяти для размещения значений используемых переменных.

Ранее для оценки эффективности алгоритмов мы ввели понятие сложности или трудоёмкости алгоритма в худшем случае. Однако для многих приложений с практической точки зрения оказывается полезным оценить не только максимальное время, но и минимальное, среднее и ожидаемое время работы алгоритма. D. E. Knuth назвал минимальное время временем для оптимистов, среднее и ожидаемое время — для специалистов по теории вероятностей, а максимальное время — временем для пессимистов.

*Минимальное время* $T_{\min}(n)$ — это наименьшее время работы алгоритма на входах размерности $n$. *Максимальное время* $T_{\max}(n)$ (или сложность алгоритма) — это наибольшее время работы алгоритма на входах размерности $n$.

Пусть $\mathbf{a}_1, \ldots, \mathbf{a}_k$ — все входные данные размерности $n$ для алгоритма $\mathcal{A}$ и $\mathsf{P}_i$ — вероятность того, что на вход алгоритма $\mathcal{A}$ подаётся последовательность данных $\mathbf{a}_i$, $0 \leq \mathsf{P}_i \leq 1$, $\sum_{i=1}^{k} \mathsf{P}_i = 1$. При фиксированном $n$ мы определили вероятностное распределение входных данных

[25] Обычно поле "ключ" имеет целый тип данных, и упорядочивание рассматривается относительно естественного линейного порядка для чисел.

для алгоритма $\mathcal{A}$. Рассмотрим время $T_{\mathcal{A}}(n)$ работы алгоритма $\mathcal{A}$ как случайную величину, принимающую одно из возможных значений $t_1, \ldots, t_k$, где $t_i$ — время работы алгоритма $\mathcal{A}$ на входных данных $\mathbf{a}_i$. *Ожидаемое время* $\mathbf{E}(T_{\mathcal{A}})$ работы алгоритма $\mathcal{A}$ — это математическое ожидание случайной величины $T_{\mathcal{A}}(n)$, т. е.

$$\mathbf{E}(T_{\mathcal{A}}) = \sum_{i=1}^{k} t_i \, \mathsf{P}_i \, .$$

*Среднее время* $T_{\text{ave}}(n)$ — это усреднённое время работы алгоритма по всем входным данным размерности $n$, т. е.

$$T_{\text{ave}}(n) = \frac{1}{k} \sum_{i=1}^{k} t_i.$$

Заметим, что если входные данные равновероятны, то среднее и ожидаемое время работы алгоритма совпадают. Рассмотрение понятия ожидаемого времени обусловлено тем, что во многих случаях входные данные, обеспечиваюшие плохую сложность алгоритма в общем случае, для конкретного класса задач либо не подаются на вход вообще, либо настолько редко появляются, что для этого класса задач ожидаемое время работы алгоритма существенно меньше, чем его сложность.

При определении ожидаемого и среднего времени мы предполагали конечность множества входных данных размерности $n$ для всех $n$. Однако эти понятия обобщаются и на случай бесконечных множеств, если для любого $n$ имеется разбиение множества всех входных данных размерности $n$ на подмножества $\mathbf{A}_1, \ldots, \mathbf{A}_k$ такие, что время работы алгоритма $\mathcal{A}$ на каждом входе из фиксированного класса данных $\mathbf{A}_i$ постоянно и принимает значение $t_i$, причём задано распределение вероятностей $\mathsf{P}_i$, $i = 1, \ldots, k$, где $\mathsf{P}_i$ — вероятность того, что на вход алгоритма $\mathcal{A}$ подаются входные данные из класса $\mathbf{A}_i$. Тогда ожидаемое время $\mathbf{E}(T_{\mathcal{A}})$ работы алгоритма $\mathcal{A}$ определяется как функция от размерности данных $n$ аналогичным образом. В случае равновероятности поступления входных данных из классов $\mathbf{A}_1, \ldots, \mathbf{A}_k$ для всех $n$ мы получаем понятие среднего времени $T_{\text{ave}}(n)$.

Для каждого из изучаемых алгоритмов сортировки мы будем исследовать асимптотический порядок минимального $T_{\min}(n)$, среднего $T_{\text{ave}}(n)$ и максимального $T_{\max}(n)$ времени работы, при этом под размерностью данных $n$ понимается число сортируемых ключей. Поведение алгоритма сортировки, не учитывающего природу сортируемых ключей, полностью определяется относительным расположением ключей в последовательности входных данных, а не их конкретной величиной. Поэтому при изучении ожидаемого времени все последовательности ключей $x_1, \ldots, x_n$ фиксированной длины, имеющие один и тот же порядок элементов по отношению друг к другу, естественно объединить в один класс, а при рассмотрении среднего времени будем считать равновероятным появление на входе последовательностей ключей из разных классов. Так, при $n = 5$ последовательности данных $2, 3, 1, 1, 5$ и $17, 20, 10, 10, 25$ будут принадлежать одному классу данных, при $n = 3$ имеется ровно 13 классов, образующих разбиение множества всех последовательностей длины 3, а вероятность появления входной последовательности из любого такого класса равна $1/13$. Кроме того, для вычисления минимального, среднего и максимального времени можно ограничиться входными данными $x_1, \ldots, x_n$ размерности $n$, составленными из ключей абстрактного $n$-элементного линейно упорядоченного множества $X$ (вообще говоря, ключи $x_i$ могут повторяться)[26]. В дальнейшем считаем, что никакие два ключа в этой последовательности не имеют одинаковых значений, т. е. если $i \neq j$, то либо $x_i < x_j$, либо $x_i > x_j$. Это ограничение[27] упростит дальнейший анализ без потери общности, и при наличии равных ключей корректность рассматриваемых алгоритмов сортировки не нарушится.

[26] Иными словами, если допускаются равные ключи, входными данными размерности $n$ будут все перестановки множества ключей $X = \{x_1, \ldots, x_n\}$ с повторениями.

[27] При этом для алгоритма сортировки входными данными размерности $n$ будут все возможные перестановки множества ключей $X = \{x_1, \ldots, x_n\}$. В действительности условие отсутствия равных ключей мы используем только при анализе среднего времени работы.

# 2. Нижние оценки сложности алгоритма сортировки сравнением

Прежде чем переходить к нижним оценкам эффективности алгоритмов сортировки, познакомимся с понятием бинарного дерева и рассмотрим некоторые свойства бинарных деревьев.

**Определение 15.** *Корневое дерево* — это граф, рекурсивно определяемый следующим образом:

1) одна вершина есть дерево, эта вершина является *корнем* дерева и не имеет *сыновей*;

2) пусть $D_1, \ldots, D_k$, $k \geq 1$ — деревья с корнями $v_1, \ldots, v_k$ соответственно, причём множества вершин этих деревьев попарно не пересекаются. Соединим рёбрами новую вершину $v$ с каждой из вершин $v_1, \ldots, v_k$. Полученный граф есть корневое дерево с корнем $v$ и *поддеревьями* $D_1, \ldots, D_k$, *соответствующими вершине* $v$. Для вершины $v$ *сыновьями* будут вершины $v_1, \ldots, v_k$, при этом сама вершина $v$ является *родителем* каждой из вершин $v_1, \ldots, v_k$. Для остальных вершин сыновья и родители остаются прежними;

3) всякое корневое дерево строится по одному из правил 1 или 2.

Для произвольной вершины $v$ корневого дерева существует единственный простой путь от корня до вершины $v$, длина этого пути называется *глубиной вершины* $v$, а вершины этого пути — *предками* $v$. Вершина корневого дерева, не имеющая сыновей, называется его *концевой вершиной*. *Высота корневого дерева* — это наибольшая длина простого пути от корня дерева до его концевых вершин. Высота поддерева с корнем $v$ (вершинами этого поддерева являются вершина $v$ и все её потомки) называется *высотой вершины* $v$.

**Определение 16.** *Бинарное* (или *двоичное*) *дерево* — это корневое дерево, у которого каждая вершина имеет не более двух сыновей, один из которых называется *левым*, а другой — *правым*. Бинарное дерево,

в котором все вершины глубины меньшей, чем высота дерева, имеют ровно два сына, называется *полным бинарным деревом*.

В дальнейшем нам потребуются свойства бинарных деревьев, сформулированные в следующих трёх леммах.

**Лемма 10.** Пусть $D$ — произвольное $n$-вершинное бинарное дерево высоты $h$. Тогда

(i) в дереве $D$ число вершин высоты $i$ не превосходит $2^{h-i}$, где $i \leq h$;

(ii) $n \leq 2^{h+1} - 1$;

(iii) для полного бинарного дерева $D$ в неравенствах из (i) и (ii) достигается равенство;

(iv) $h \geq \lfloor \lg n \rfloor$;

(v) $n \geq 2k - 1$, где $k$ — число концевых вершин дерева $D$.

Доказательство (iii) нетрудно провести, используя индукцию по высоте дерева $h$ и формулу суммы геометрической прогрессии. Утверждения (i), (ii) вытекают из (iii), а (iv) непосредственно следует из (ii). Утверждение (v) доказывается индукцией по числу вершин $n$.

**Лемма 11.** Пусть $D$ — $n$-вершинное бинарное дерево высоты $h$, в котором каждая вершина глубины меньшей, чем $h-1$ имеет ровно два сына. Тогда $h = \lfloor \lg n \rfloor$.

**Доказательство.** При $n = 1$ утверждение очевидно. Пусть $n \geq 2$. Нетрудно понять, что дерево $D$ содержит полное бинарное дерево $D'$ высоты $h - 1$ и, по крайней мере, одну вершину, не принадлежащую дереву $D'$. Следовательно, $n \geq 2^h - 1 + 1 = 2^h$ по лемме 10 (iii). Значит, $h \leq \lfloor \lg n \rfloor$. С другой стороны, $h \geq \lfloor \lg n \rfloor$ по лемме 10 (iv). Лемма 11 доказана.

**Лемма 12.** Пусть $D$ — $n$-вершинное бинарное дерево с $k$ концевыми вершинами, $H(D)$ — сумма глубин всех его концевых вершин и $F(D)$ —

сумма глубин всех его вершин. Тогда

$$H(D) \geq k \lg k, \;\; F(D) \geq \sum_{j=1}^{n} \lfloor \lg j \rfloor .$$

**Доказательство.** Оценку числа $H(D)$ получим индукцией по высоте $h$ дерева $D$. Базис индукции при $h = 0$ очевиден. Пусть $h > 0$ и вершина $v$ — корень дерева $D$. Возможны два случая.

Случай 1. Корень $v$ имеет единственного сына. Пусть $D'$ — поддерево, соответствующее вершине $v$. Тогда $H(D) = H(D') + k$. По индукционному предположению $H(D') \geq k \lg k$, так как $D'$ имеет $k$ концевых вершин, и высота дерева $D'$ равна $h - 1$. Следовательно, $H(D) \geq k \lg k$.

Случай 2. Корень $v$ имеет два сына. Пусть $D_1, D_2$ — поддеревья, соответствующие вершине $v$, и $k_i$ — число концевых вершин дерева $D_i$. Тогда $H(D) = H(D_1) + k_1 + H(D_2) + k_2$ и $k = k_1 + k_2$. Используя индукционное предположение, получаем $H(D) \geq f(k_1) + k$, где $f(x) = x \lg x + (k - x) \lg(k - x)$. Рассмотрим функцию $f(x)$ на интервале $(0, k)$. Значение $f(k/2)$ есть экстремум функции $f(x)$, и вторая производная $f''(x)$ положительна. Следовательно, $f(k/2)$ есть минимум функции $f(x)$. Поэтому $H(D) \geq f(k/2) + k = k \lg k$.

Получим оценку числа $F(D)$ индукцией по $n$. Пусть $v$ — вершина $D$ глубины $s(v)$, равной высоте дерева $D$. Рассмотрим бинарное $(n - 1)$-вершинное дерево $D'$, получающееся из $D$ удалением вершины $v$. По лемме 10 (iv) и индукционному предположению получаем

$$F(D) = s(v) + F(D') \geq \lfloor \lg n \rfloor + F(D') \geq \sum_{j=1}^{n} \lfloor \lg j \rfloor .$$

Лемма 12 доказана.

Среди алгоритмов сортировки выделяется большой класс алгоритмов, использующих при сортировке только сравнение ключей входных записей. Такие алгоритмы будем называть *алгоритмами сортировки сравнением*. Любой алгоритм $\mathcal{A}$ сортировки сравнением задаёт бинарное дерево $D_{\mathcal{A}}$, определяемое следующим образом. Пусть $y_1, \ldots, y_n$ —

различные переменные, значения которых будут соответствовать входной последовательности ключей. Рассмотрим бинарное дерево, в котором представлены все операции сравнения ключей, выполняющиеся во время работы алгоритма $\mathcal{A}$, при этом все другие операции, например, перемещение данных и вычисления, мы игнорируем. Каждая неконцевая вершина этого дерева соответствует сравнению некоторых переменных $y_i$ и $y_j$, сделанному алгоритмом в процессе его выполнения. Такую вершину отмечаем меткой $y_i : y_j$, $i \leq j$. Из этой вершины выходит не более двух рёбер, представляющих два возможных результата сравнения. При этом левому поддереву соответствуют дальнейшие сравнения, выполняющиеся во время работы алгоритма при $y_i \leq y_j$, а правому поддереву — сравнения, которые нужно выполнить при $y_i > y_j$. Ребро, соответствующее невозможным соотношениям между уже рассмотренными переменными, в дереве не допускается. В этом случае либо левое, либо правое поддерево будет пустым, а для любой входной последовательности ключей соответствующая часть вычислений алгоритмом $\mathcal{A}$ никогда не выполняется. Каждая концевая вершина дерева $D_{\mathcal{A}}$ помечена перестановкой $(\pi_1, \ldots, \pi_n)$ множества $\{1, 2, \ldots, n\}$, которая определяет окончательное упорядочивание $y_{\pi_1} \leq \ldots \leq y_{\pi_n}$ соответствующих ключей входной последовательности. Полученное размеченное бинарное дерево $D_{\mathcal{A}}$ называется *деревом решений* алгоритма $\mathcal{A}$. Очевидно, что выполнение алгоритма $\mathcal{A}$ сортировки заданной входной последовательности значений переменных $y_1, \ldots, y_n$ соответствует прохождению пути от корня дерева решений $D_{\mathcal{A}}$ до одной из его концевых вершин.

Отметим следующие свойства дерева решений $D_{\mathcal{A}}$:

• бинарное дерево $D_{\mathcal{A}}$ имеет ровно $n!$ концевых вершин;

• из корня дерева $D_{\mathcal{A}}$ к каждой его концевой вершине проходит единственный путь, соответствующий работе алгоритма $\mathcal{A}$ на подходящей входной последовательности размерности $n$.

Действительно, в силу определения дерева решений каждая его концевая вершина помечена некоторой перестановкой $\pi$ множества

$\{1, 2, \ldots, n\}$. Очевидно, что различным концевым вершинам соответствуют различные перестановки. Пусть $\pi$ — произвольная перестановка множества $\{1, 2, \ldots, n\}$ и $y_1 \leq \ldots \leq y_n$ — упорядочивание входной последовательности ключей $x_1, \ldots, x_n$. Корректный алгоритм сортировки сравнением должен произвести сортировку любой входной последовательности размерности $n$. При работе алгоритма $\mathcal{A}$ на входной последовательности $y_{\pi^{-1}(1)}, \ldots, y_{\pi^{-1}(n)}$ выполненным последовательным сравнениям соответствует путь от корня дерева $D_{\mathcal{A}}$ до концевой вершины, помеченной перестановкой $\pi$. Следовательно, число концевых вершин дерева $D_{\mathcal{A}}$ равно числу всех перестановок $n$-элементного множества. Остаётся заметить, что в каждом дереве существует единственный путь, соединяющий произвольные две вершины.

**Теорема 11.** Сложность любого алгоритма сортировки сравнением есть $\Omega(n \lg n)$.

**Доказательство.** Пусть $\mathcal{A}$ — произвольный алгоритм сортировки сравнением. При получении нижних оценок сложности $T_{\max}(n)$ алгоритма $\mathcal{A}$ можно ограничиться входными данными размерности $n$, состоящими из различных ключей. Путь из корня дерева решений $D_{\mathcal{A}}$ в произвольную его концевую вершину соответствует работе алгоритма $\mathcal{A}$ на подходящей входной последовательности размерности $n$. Длина этого пути равна числу выполненных сравнений, потребовавшихся для сортировки входной последовательности. Поэтому высота $h$ дерева $D_{\mathcal{A}}$ есть число операций сравнений, выполняющихся алгоритмом $\mathcal{A}$ при его работе на некоторой входной последовательности размерности $n$. Следовательно, $T_{\max}(n) \geq h$. Число концевых вершин бинарного дерева $D_{\mathcal{A}}$ не превосходит $2^h$ в силу леммы 10 (ii),(v). Дерево решений $D_{\mathcal{A}}$ имеет $n!$ концевых вершин. Поэтому $h \geq \lg(n!)$. В силу примера 1 (vii) получаем $T_{\max}(n) = \Omega(n \lg n)$. Теорема 11 доказана.

Теорема 11 дает нижнюю оценку порядка максимального времени работы произвольного алгоритма сортировки сравнением. На самом де-

ле, аналогичная оценка справедлива и для среднего времени.

**Теорема 12.** Среднее время работы любого алгоритма сортировки сравнением есть $\Omega(n \lg n)$.

**Доказательство.** Пусть $\mathcal{A}$ — произвольный алгоритм сортировки сравнением с множеством ключей $X = \{x_1, \ldots, x_n\}$. Входная последовательность размерности $n$ представляет собой произвольную перестановку множества $X$, их число равно $n!$. Сумму глубин всех концевых вершин дерева решений $D_{\mathcal{A}}$ обозначим через $H(D_{\mathcal{A}})$. Как и в теореме 11, в силу свойств дерева решений $D_{\mathcal{A}}$ получаем $T_{\text{ave}}(n) \geq H(D_{\mathcal{A}})/n!$, где $T_{\text{ave}}(n)$ — среднее время работы алгоритма $\mathcal{A}$ на входных данных размерности $n$. Причём дерево $D_{\mathcal{A}}$ имеет $n!$ концевых вершин. Следовательно, $T_{\text{ave}}(n) \geq \lg(n!) = \Omega(n \lg n)$ в силу леммы 12 и примера 1 (vii). Теорема 12 доказана[28].

**Следствие 1.** В предположении равновероятности входных данных ожидаемое время работы любого алгоритма сортировки сравнением есть $\Omega(n \lg n)$.

## 3. Алгоритм сортировки вставками и оценки времени его работы

Рассматриваемый алгоритм называется *сортировкой вставками*. Этот метод сортировки последовательности ключей $x_1, \ldots, x_n$ состоит в следующем. На $i$-м шаге ключ $x_i$ вставляется в нужную позицию среди уже упорядоченных ключей $x_1, \ldots, x_{i-1}$. После этой вставки первые $i$

[28] Если допускаются равные ключи, работу алгоритма $\mathcal{A}$ необходимо представить в виде *тернарного* дерева решений $D_{\mathcal{A}}$, в котором из каждой неконцевой вершины с меткой $y_i : y_j$ выходит три ребра, соответствующие трём возможным исходам операции сравнения $y_i < y_j$, $y_i = y_j$ и $y_i > y_j$, а концевые вершины помечены перестановками с повторениями, определяющими окончательное упорядочивание. При этом средняя глубина концевых вершин произвольного тернарного дерева, имеющего $k$ концевых вершин, не меньше, чем $\log_3 k$.

ключей исходной последовательности будут упорядочены. Для поиска требуемой позиции ключа $x_i$ временно сохраняем его в дополнительной переменной $x$, далее ключи $x_j$, $j = i-1, \ldots, 1$ последовательно сравниваются с $x$ и сдвигаются вправо. Этот процесс продолжается до тех пор, пока $x < x_j$. В найденную таким образом позицию помещаем элемент $x_i$. Чтобы обеспечить остановку этого процесса, удобно ввести дополнительный ключ $x_0$, значение которого меньше значения любого ключа $x_k$, $k = 1, \ldots, n$. Программная реализация описанного алгоритма приведена в следующем псевдокоде.

**Алгоритм сортировки вставками последовательности** $x_1, \ldots, x_n$

```
x_0 := −∞;
for i = 2 to n do
begin
    x := x_i;
    j := i − 1;
    while x < x_j do
    begin
        x_{j+1} := x_j;
        j := j − 1
    end;
    x_{j+1} := x
end.
```

Здесь мы постулировали существование константы $-\infty$, меньшей значения любого ключа, встречающегося в рассматриваемой задаче. Если такую константу указать трудно, можно модифицировать алгоритм и отказаться от её использования. Для этого необходима только дополнительная проверка $j > 0$ в условии while-цикла.

**Пример 9.** Записи извержений знаменитых вулканов имеют два поля: название вулкана и год его извержения. Вулкан Пили — 1902 г., Этна — 1669 г., Кракатау — 1883 г., Агунг — 1963 г., Св.Елена — 1980 г., Везувий — 79 г. Отсортируем эти записи по году извержения вулканов. На рис. 2 приведена последовательность ключей после выполнения итерации for-цикла для указанного значения счётчика итераций $i$. При выполнении этой итерации ключ $x_i$ вставляется в позицию с номером $j+1$. В итоге получим список вулканов в порядке года их извержения: Везувий, Этна, Кракатау, Пили, Агунг, Св. Елена.

| $i$ | $j+1$ | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ |
|---|---|---|---|---|---|---|---|---|
| | | $-\infty$ | 1902 | 1669 | 1883 | 1963 | 1980 | 79 |
| 2 | 1 | $-\infty$ | 1669 | 1902 | 1883 | 1963 | 1980 | 79 |
| 3 | 2 | $-\infty$ | 1669 | 1883 | 1902 | 1963 | 1980 | 79 |
| 4 | 4 | $-\infty$ | 1669 | 1883 | 1902 | 1963 | 1980 | 79 |
| 5 | 5 | $-\infty$ | 1669 | 1883 | 1902 | 1963 | 1980 | 79 |
| 6 | 1 | $-\infty$ | 79 | 1669 | 1883 | 1902 | 1963 | 1980 |

Рис. 2. Сортировка вставками

Корректность алгоритма сортировки вставками вытекает из следующего свойства: после итерации for-цикла для выбранного значения $i$, ключи $x_1, \ldots, x_i$ будут отсортированы, а оставшиеся ключи останутся на прежних местах. Это свойство легко доказать индукцией по $i$.

Оценим минимальное $T_{\min}(n)$ и максимальное $T_{\max}(n)$ время работы алгоритма сортировки вставками. Пусть $T(n)$ — время работы алгоритма на входной последовательности $x_1, \ldots, x_n$ и $d_i$ — число элементов $x_j$, бóльших $x_i$ и находящихся в последовательности $x_1, \ldots, x_n$ левее $x_i$. Тогда $0 \le d_i < i$. Заметим, что для фиксированного в for-цикле значения переменной $i$ проверка условия while-цикла выполняется $d_i + 1$ раз. Действительно, после предыдущей итерации for-цикла мы имеем последовательность $x_{k_1}, \ldots, x_{k_{i-1}}, x_i, \ldots, x_n$, в которой первые $i-1$ ключей —

это отсортированная по возрастанию последовательность $x_1, \ldots, x_{i-1}$. Среди этих $i-1$ ключей ровно $d_i$ элементов, бóльших $x_i$, которые идут подряд и непосредственно предшествуют $x_i$. Поэтому проверка условия while-цикла будет осуществляться $d_i$ раз при перемещении этих $d_i$ элементов вправо и последний раз при выходе из while-цикла. Теперь непосредственно из алгоритма получаем

$$\begin{aligned} T(n) &= c_1 + (n-1)c_2 + \sum_{i=2}^{n}(d_i+1) + c_3\sum_{i=2}^{n} d_i = \\ &= c_1' + c_2' n + c_3' \sum_{i=2}^{n} d_i. \end{aligned}$$

Следовательно, время работы алгоритма сортировки вставками на возрастающей входной последовательности, когда $d_i = 0$ при $i = 1, \ldots, n$, является минимальным временем, а на строго убывающей входной последовательности, когда

$$\sum_{i=2}^{n} d_i = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2},$$

будет максимальным. Поэтому $T_{\min}(n) = \Theta(n)$ и $T_{\max}(n) = \Theta(n^2)$.

Оценим среднее время $T_{\text{ave}}(n)$. Предположим, что все входные данные равновероятны, т. е. любая перестановка ключей $x_1, \ldots, x_n$ равновероятна на входе алгоритма. Тогда $T_{\text{ave}}(n) = \mathbf{E}(T)$, и мы будем оценивать ожидаемое время $\mathbf{E}(T)$. Имеем

$$\mathbf{E}(T) = \Theta(n) + \Theta(1)\,\mathbf{E}(\sum_{i=2}^{n} d_i).$$

$$\mathbf{E}(\sum_{i=2}^{n} d_i) = \sum_{i=2}^{n} \mathbf{E}(d_i).$$

Величина $d_i$ принимает одно из значений $0, 1, \ldots, i-1$. В силу равновероятности входных данных получаем

$$\mathbf{E}(d_i) = \sum_{j=0}^{i-1} j\,\mathsf{P}(d_i = j) = \sum_{j=0}^{i-1} \frac{j}{i} = \frac{i-1}{2}.$$

Поэтому выполняется следующее соотношение

$$\mathbf{E}(T) = \Theta(n) + \Theta(1)\,\frac{n(n-1)}{4}.$$

Следовательно, $T_{\text{ave}}(n) = \Theta(n^2)$. Таким образом, доказана

**Теорема 13.** Минимальное время работы алгоритма сортировки вставками равно $\Theta(n)$, среднее и максимальное время есть $\Theta(n^2)$.

## 4. Алгоритм пузырьковой сортировки и оценки времени его работы

Следующий простой алгоритм сортировки сравнением называется *алгоритмом пузырьковой сортировки*. Чтобы описать основную идею этого метода, запишем сортируемые ключи в массив, "расположенный вертикально". При сортировке ключи с бóльшими значениями, относительно заданного линейного порядка $\leq$, будем располагать выше ключей с меньшими значениями. Ключи с бóльшими значениями "всплывают" вверх наподобие пузырька. При первом проходе снизу вверх первый ключ сравнивается с непосредственно следующим. Если внизу оказывается бóльший ключ, соответствующие ключи меняем местами. При встрече бóльшего ключа верхний ключ становится эталоном для сравнения, и все последующие ключи сравниваются с этим новым бóльшим ключом. В результате первого прохода всего массива самый большой ключ окажется в самом верху, т. е. всплывает. Во время второго прохода снизу вверх находится ключ со вторым по величине значением, который помещается под ключом, найденным при первом проходе массива, т. е. на вторую сверху позицию, и т. д. Отметим, что во время второго и последующих проходов массива нет необходимости просматривать ключи, найденные за предыдущие проходы. Поэтому в алгоритме используем переменную $k$, значение которой при каждом проходе устанавливается равным наибольшему индексу $i$ такому, что все ключи $x_{i+1}, x_{i+2}, \ldots, x_n$ уже находятся на своих окончательных позициях.

**Алгоритм пузырьковой сортировки последовательности $x_1, \ldots, x_n$**

$k := n$;
**while** $k \neq 0$ **do**
**begin**
  $i := 0$;
  **for** $j = 1$ to $k - 1$ **do**
  **if** $x_j > x_{j+1}$ **then begin**
    $Swap(x_j, x_{j+1})$;
    $i := j$
  **end**;
  $k := i$
**end**.

**Пример 10.** Выполним пузырьковую сортировку ключей, записанных в первый столбец таблицы (см. рис. 3), бóльшие ключи располагаем выше меньших. На рис. 3 приведена последовательность ключей перед итерацией while-цикла, соответствующей указанному значению переменной $k$. При выполнении этой итерации (прохода массива) выделенные ключи всплывают. Отсортированная в результате последовательность ключей записана в последнем столбце.

По сравнению с алгоритмом сортировки вставками, метод пузырьков оказывается более сложным, но, тем не менее, асимптотический порядок минимального, среднего и максимального времени работы этого алгоритма оказываются такими же [29]. Минимальное время имеет порядок $n$ и достигается на уже отсортированной входной последовательности, максимальное время имеет порядок $n^2$ и достигается на входной последовательности, отсортированной в обратном порядке, а среднее время имеет порядок $n^2$.

[29] Детальный анализ показывает, что метод пузырьковой сортировки требует примерно в два раза больше времени, чем алгоритм сортировки вставками [8, т. 3].

| $k=8$ | $k=7$ | $k=6$ | $k=5$ | $k=4$ | $k=0$ |
|---|---|---|---|---|---|
| $\overline{42}$ | 64 | 64 | 64 | 64 | 64 |
| 61 | $\overline{42}$ | 61 | 61 | 61 | 61 |
| 33 | **61** | $\overline{42}$ | 56 | 56 | 56 |
| **64** | 33 | **56** | $\overline{42}$ | 53 | 53 |
| 17 | **56** | 33 | **53** | $\overline{42}$ | 42 |
| **56** | 17 | **53** | 33 | 33 | 33 |
| 28 | **53** | 17 | 28 | 28 | 28 |
| **53** | 28 | **28** | 17 | 17 | $\underline{17}$ |

Рис. 3. Пузырьковая сортировка

Мы рассмотрели алгоритм сортировки вставками и алгоритм пузырьковой сортировки, имеющие максимальное и среднее время работы порядка $n^2$. Хотя для небольших значений $n$ эти алгоритмы достаточно эффективны, тем не менее, их время работы как в среднем, так и в худшем случае не является лучшим возможным временем работы алгоритмов сортировки сравнением, которое согласно теоремам 11, 12 имеет порядок $n \lg n$. Также отметим, что при небольших значениях числа $n$ можно применять простой в реализации алгоритм сортировки Шелла, который является обобщением алгоритма сортировки вставками и имеет сложность $O(n^{1.5})$[30].

## 5. Алгоритм быстрой сортировки и оценки времени его работы

Основная причина медленной работы рассмотренных алгоритмов сортировки состоит в том, что все сравнения и обмены между ключами входной последовательности $x_1, \ldots, x_n$ происходили для пар соседних элементов. При таком подходе для постановки текущего ключа в нужную позицию сортируемой последовательности требуется относительно

[30] Описание алгоритма сортировки Шелла и его анализ см. в [3; 8, т. 3].

много операций. Естественно попытаться ускорить этот процесс, сравнивая далекие друг от друга ключи. C. A. R. Hoare предложил и весьма эффективно применил эту идею, сократив среднее время работы алгоритма до порядка $n \lg n$. Он назвал свой метод $QuickSort$ (*быстрая сортировка*), и это название вполне соответствует действительности, так как при его реализации на любом современном компьютере алгоритм оказывается очень быстрым. Причём сортировка входной последовательности осуществляется на месте.

В алгоритме $QuickSort$ входная последовательность ключей записывается в массив $x_1, \ldots, x_n$. Для сортировки элементов $x_1, \ldots, x_n$ среди них выбирается некоторый элемент $y$, пока не конкретизируем способ его выбора, в качестве так называемого *опорного элемента*. Далее элементы массива переставляются так, чтобы в полученном массиве для некоторой позиции $q$ все элементы $x_1, \ldots, x_{q-1}$ имели значения, меньшие, чем $y$, значение $x_q$ совпадало с $y$, а значения всех элементов $x_{q+1}, \ldots, x_n$ были бы не меньше $y$ относительно линейного порядка на ключах. Затем процедура быстрой сортировки рекурсивно применяется к двум полученным подмассивам $x_1, \ldots, x_{q-1}$ и $x_{q+1}, \ldots, x_n$ для их упорядочивания по отдельности. Поскольку подмассивы сортируются на месте, для их объединения никакие дополнительные действия не нужны. Весь массив $x_1, \ldots, x_n$ оказывается отсортированным, так как все значения ключей в первом подмассиве будут меньше значений ключей во втором подмассиве. Таким образом, мы приходим к следующей программной реализации алгоритма.

**Алгоритм $QuickSort(p, r)$ быстрой сортировки**

**Procedure** $QuickSort(p, r)$;
**if** p<r **then begin**
q:=$Partition(p, r)$;
$QuickSort(p, q-1)$;
$QuickSort(q+1, r)$
**end**.

Здесь $QuickSort(p, r)$ — рекурсивная процедура сортировки подмассива $x_p, x_{p+1}, \ldots, x_r$. Сортировка всего массива $x_1, \ldots, x_n$ осуществляется вызовом процедуры $QuickSort(1, n)$.

Ключевой частью рассматриваемого алгоритма является функция $Partition$, возвращающая значение $q$ позиции, которую должен занять опорный элемент $y$ после всей сортировки, и осуществляющая требуемую перестановку элементов массива. Для выбора опорного элемента и способа переупорядочивания разработаны различные модификации алгоритма быстрой сортировки. Мы будем следовать одной из них, когда в качестве опорного элемента в массиве выбирается самый правый элемент $x_r$, а процесс перестановки осуществляется следующим образом. Последовательно сравниваются элементы $x_j$, $j = p, p+1, \ldots, r-1$ с опорным элементом $x_r$. Если $x_j < x_r$, то элементы $x_i$, $x_j$ переставляются и значение переменной $i$ увеличивается на 1. Здесь $i$ — это номер текущей позиции для окончательного размещения элемента $x_r$, первоначально $i = p$. В результате этого процесса позиция $q$ определяется как итоговое значение переменной $i$.

**Function** $Partition(p, r)$;
$i := p$;
**for** $j = p$ **to** $r - 1$ **do**
**if** $x_j < x_r$ **then begin**
$Swap(x_i, x_j)$;
$i := i + 1$
**end**;
$Swap(x_i, x_r)$
**return** $i$.

**Пример 11.** Рассмотрим работу функции $Partition(1, 6)$ на входной последовательности $4, 7, 2, 3, 6, 5$, записанной в массив $x_1, \ldots, x_6$. На рис. 4 приведено состояние массива после итерации for-цикла для указанного значения счётчика итераций $j$. При выполнении этой итерации

выделенные элементы переставляются, текущая позиция $i$ опорного элемента $x_6$ подчёркнута. Отсортированная последовательность записана в последней строке.

| $j$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $i$ |
|---|---|---|---|---|---|---|---|
| | $\underline{4}$ | 7 | 2 | 3 | 6 | 5 | 1 |
| 1 | **4** | $\underline{7}$ | 2 | 3 | 6 | 5 | 2 |
| 2 | 4 | $\underline{7}$ | 2 | 3 | 6 | 5 | 2 |
| 3 | 4 | **2** | $\underline{\mathbf{7}}$ | 3 | 6 | 5 | 3 |
| 4 | 4 | 2 | **3** | $\underline{\mathbf{7}}$ | 6 | 5 | 4 |
| 5 | 4 | 2 | 3 | $\underline{7}$ | 6 | 5 | 4 |
| | 4 | 2 | 3 | **5** | 6 | **7** | |

Рис. 4. Работа функции $Partition(1, 6)$ на входной последовательности $4, 7, 2, 3, 6, 5$

Докажем корректность функции *Partition*. Индукцией по $j$ нетрудно показать, что после итерации for-цикла при $j = p, p + 1, \ldots, r - 1$ выполняются следующие свойства:

- $\forall k \ (p \le k < i \Rightarrow x_k < x_r)$;
- $\forall k \ (i \le k \le j \Rightarrow x_k \ge x_r)$;
- $x_r$ не перемещается;
- $i \le j + 1$ (здесь не учитывается изменение значения счётчика $j$).

По завершении работы for-цикла при $j = r - 1$ элементы массива $x_p, \ldots, x_{r-1}$ разбиты на два множества: значения всех элементов $x_p, x_{p+1}, \ldots, x_{i-1}$ меньше $x_r$, а значения остальных больше или равны $x_r$. После выполнения for-цикла опорный элемент $x_r$ меняется местами с $x_i$. Таким образом, полученное упорядочивание элементов массива удовлетворяет всем требуемым свойствам.

Корректность алгоритма быстрой сортировки *QuickSort* очевидным образом доказывается индукцией по числу сортируемых элементов.

**Пример 12.** Рассмотрим работу алгоритма быстрой сортировки $QuickSort(1,6)$ на входной последовательности $4,7,2,3,6,5$ (рис. 5). Сначала функция $Partition(1,6)$ возвращает позицию $i = 4$ опорного элемента 5 и переупорядочивает массив. Далее выполняется алгоритм $QuickSort(1,3)$. В результате работы функции $Partition(1,3)$ получаем упорядочивание $2,3,4$ с опорным элементом 3. Алгоритмы $QuickSort(1,1)$ и $QuickSort(3,3)$ останавливают свою работу, так как не выполняется условие оператора if. В итоге процедура $QuickSort(1,3)$ завершает работу и на первых трёх местах имеется упорядочивание $2,3,4$. Затем аналогично выполняется алгоритм $QuickSort(5,6)$, и получается упорядочивание $6,7$ на последних двух местах.

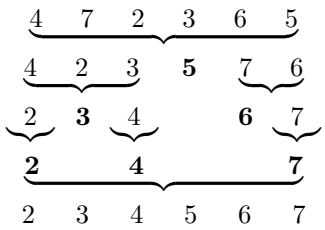



Рис. 5. Быстрая сортировка

Оценим максимальное $T_{\max}(n)$, минимальное $T_{\min}(n)$ и среднее $T_{\text{ave}}(n)$ время работы алгоритма быстрой сортировки $QuickSort$ на входных данных размерности $n$ [31]. Пусть $T'_{\max}(n)$, $T'_{\min}(n)$ — соответственно максимальное и минимальное время работы алгоритма $Partition$ на входных данных размерности $n$. Непосредственно из алгоритма находятся положительные константы $c_{min}$, $c_{max}$ такие, что для любого $n \geq 2$ выполняются неравенства

$$T'_{\min}(n) \geq c_{min}\, n, \;\; T'_{\max}(n) \leq c_{max}\, n - 1.$$

[31] $n = r - p + 1$ и $n = 0$ соответствует условию $r < p$.

**Лемма 13.** Максимальное время $T_{\max}(n)$ работы алгоритма быстрой сортировки *QuickSort* есть $\Theta(n^2)$.

**Доказательство.** Пусть $c$ — фиксированная константа такая, что $c \geq \max\{\, c_{max},\, T_{\max}(0),\, T_{\max}(1)\,\}$. Индукцией по $n$ докажем неравенство $T_{\max}(n) \leq c(n^2+1)$ для любого $n \geq 0$. Базис индукции при $n = 0, 1$ очевиден. Пусть $n \geq 2$. Из алгоритма *QuickSort* и индукционного предположения получаем оценки

$$\begin{aligned} T_{\max}(n) &\leq \max_{q\in\{1,\ldots,n\}} \{\, T_{\max}(q-1) + T_{\max}(n-q)\,\} + T'_{\max}(n) + 1 \leq \\ &\leq \max_{1\leq q\leq n} \{\, c(q-1)^2 + c(n-q)^2\,\} + 2\,c + cn = \\ &= c(n-1)^2 + 2c + cn \leq c(n^2+1), \end{aligned}$$

здесь мы воспользовались равенством

$$\max_{1\leq q\leq n} f(q) = f(1) = (n-1)^2$$

для параболы $f(q)$, заданной уравнением $f(q) = (q-1)^2 + (n-q)^2$. Таким образом, $T_{\max}(n) = O(n^2)$.

Теперь покажем, что время порядка $n^2$ достигается при работе алгоритма быстрой сортировки. Пусть $T(n)$ — время работы алгоритма *QuickSort* на уже отсортированной входной последовательности $x_1 < \ldots < x_n$ длины $n$. При работе алгоритма на таких входных последовательностях после каждого вызова функции *Partition* процедура *QuickSort* применяется к двум подмассивам, один из которых будет пустой, а другой — также отсортированный, меньшей длины. Поэтому $T(n) = T(n-1) + T(0) + \Theta(n)$. Далее, как и выше, индукцией по $n$ нетрудно доказать, что $T(n) = \Omega(n^2)$. Таким образом, $T_{\max}(n)=\Theta(n^2)$. Лемма 13 доказана.

**Лемма 14.** Минимальное время $T_{\min}(n)$ работы алгоритма быстрой сортировки *QuickSort* есть $\Theta(n \lg n)$.

**Доказательство.** Пусть $c$ — произвольная константа такая, что $0 < 2c \le c_{min}$. Индукцией по $n$ докажем, что $T_{\min}(n) \ge cn\lg n$ для любого $n \ge 1$. Действительно, базис индукции при $n = 1$ очевиден. Пусть $n \ge 2$. Из алгоритма *QuickSort* получаем

$$\begin{aligned} T_{\min}(n) &\ge \min_{q\in\{1,\ldots,n\}} \{\, T_{\min}(q-1) + T_{\min}(n-q) \,\} + T'_{\min}(n) \ge \\ &\ge \min_{q\in\{1,\ldots,n\}} \{\, T_{\min}(q-1) + T_{\min}(n-q) \,\} + 2c\,n. \end{aligned}$$

Случай 1. Минимум указанного выражения достигается при $q = 1$ или $q = n$. Рассмотрим функцию $g(x) = x\lg x$ на интервале $(0,\infty)$. Так как вторая производная $g''(x) > 0$ положительна, функция $g(x)$ является выпуклой вниз [32]. Поэтому

$$g(n) - g(n-1) \le g'(n) = \lg(en) \le n + 1.$$

Используя индукционное предположение, получаем

$$\begin{aligned} T_{\min}(n) &\ge T_{\min}(0) + T_{\min}(n-1) + 2c\,n \ge \\ &\ge cg(n-1) + 2c\,n \ge \\ &\ge cg(n) = cn\lg n. \end{aligned}$$

Случай 2. Пусть не выполняется случай 1. В доказательстве леммы 12 показано, что минимум функции $f_k(x) = x\lg x + (k-x)\lg(k-x)$ на интервале $(0,k)$ есть значение $f_k(k/2) = g(k) - k$. Используя индукционное предположение, получаем

$$\begin{aligned} T_{\min}(n) &\ge \min_{1<q<n} \{\, c(q-1)\lg(q-1) + c(n-q)\lg(n-q) \,\} + 2c\,n = \\ &= c\,\min\{\, f_{n-1}(x) \mid 0 < x < n-1 \,\} + 2c\,n = \\ &= c\,(\, g(n-1) - (n-1) + 2n) \ge cg(n) = cn\lg n. \end{aligned}$$

Таким образом, $T_{\min}(n) = \Omega(n\lg n)$.

[32] Функция $g(x)$ называется *выпуклой вниз* на интервале $(a,b)$, если для любого $x_0 \in (a,b)$ касательная $l(x) = g(x_0) + g'(x_0)(x - x_0)$ лежит ниже графика функции, т. е. $l(x) \le g(x)$ для всех $x \in (a,b)$.

Теперь покажем, что время порядка $n \lg n$ достигается при работе алгоритма быстрой сортировки[33]. Сначала для любого $n \geq 2$ определим входную последовательность $x_1, \ldots, x_n$ такую, что при работе алгоритма быстрой сортировки после каждого вызова функции $Partition(p, r)$ процедура $QuickSort$ применяется к двум подмассивам, один из которых $x_p, \ldots, x_{q-1}$ имеет размер $\lfloor (r-p+1)/2 \rfloor$, а другой $x_{q+1}, \ldots, x_r$ — размер $\lceil (r-p+1)/2 \rceil - 1$. Последовательность $x_1, \ldots, x_n$ с таким свойством будем называть *сбалансированной*.

Индукцией по $n$ докажем, что любую последовательность из $n$ различных элементов можно упорядочить так, что получится сбалансированная последовательность. Пусть $z_1 < \ldots < z_n$ — упорядочивание по возрастанию произвольной последовательности $y_1, \ldots, y_n$ различных элементов. Если $n = 2$, последовательность $z_1, z_2$ очевидно является сбалансированной. Пусть $n > 2$. Переставим элемент $z_{\lfloor n/2 \rfloor + 1}$ на последнее место, а последовательности $z_1, \ldots, z_{\lfloor n/2 \rfloor}$ и $z_{\lfloor n/2 \rfloor + 2}, \ldots, z_n$ соответственно длины $\lfloor n/2 \rfloor$ и $\lceil n/2 \rceil - 1$ упорядочим требуемым образом согласно индукционному предположению. Далее во второй полученной последовательности длины $\lceil n/2 \rceil - 1$ её последний элемент переместим на её первое место. Нетрудно понять, что построенная последовательность является сбалансированной.

Обозначим через $T(n)$ максимальное время работы алгоритма быстрой сортировки на сбалансированных входных последовательностях размерности $n$. Индукцией по $n$ докажем справедливость неравенства

$$T(n) \leq c\, n \lg n$$

для любого $n \geq 2$, где $c$ — фиксированная константа такая, что $c \geq \max\{\, c_{max},\, T(2),\, T(3),\, T(4) \,\}$. Базис индукции при $n = 2, 3, 4$ очевиден. Пусть $n \geq 5$. Тогда $\lfloor n/2 \rfloor \geq \lceil n/2 \rceil - 1 \geq 2$. Из алгоритма $QuickSort$ и

[33] Ввиду полученной в лемме 15 верхней оценки среднего времени $T_{\text{ave}}(n) = O(n \lg n)$ этого, вообще говоря, не требуется. Здесь мы строим пример входных данных, на которых достигается асимптотический порядок минимального времени.

индукционного предположения получаем

$$\begin{aligned} T(n) &\leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil - 1) + T'_{\max}(n) + 1 \leq \\ &\leq c\lfloor n/2 \rfloor \lg \lfloor n/2 \rfloor + c(\lceil n/2 \rceil - 1) \lg(\lceil n/2 \rceil - 1) + cn \leq \\ &\leq 2\, c\, \frac{n}{2}\, \lg \frac{n}{2} + cn = cn \lg n. \end{aligned}$$

Следовательно, $T(n) = O(n \lg n)$. Поэтому алгоритм быстрой сортировки на любой сбалансированной входной последовательности требует времени порядка $n \lg n$. Таким образом, минимальное время $T_{\min}(n)$ есть $\Theta(n \lg n)$. Лемма 14 доказана.

**Лемма 15.** Среднее время $T_{\text{ave}}(n)$ работы алгоритма быстрой сортировки *QuickSort* есть $\Theta(n \lg n)$.

**Доказательство.** При вызове процедуры *QuickSort* сначала происходит обращение к функции *Partition*, для которой потребуется не более, чем $c_{max}\, n$ операций. Определяется позиция опорного элемента, и далее снова с помощью алгоритма *QuickSort* сортируются две подпоследовательности, длины которых равны соответственно $q - 1$ и $n - q$, где $1 \leq q \leq n$. Предполагая, что все входные данные равновероятны, и оценивая ожидаемое время работы алгоритма *QuickSort*, нетрудно получить следующее неравенство:

$$T_{\text{ave}}(n) \leq c_{max}\, n + \frac{1}{n} \sum_{q=1}^{n} ( T_{\text{ave}}(q-1) + T_{\text{ave}}(n-q) ).$$

Так как для любой функции $h(q)$ справедливы равенства

$$\sum_{q=1}^{n} h(q-1) = \sum_{q=1}^{n} h(n-q) = \sum_{q=0}^{n-1} h(q),$$

имеем следующую оценку среднего времени работы:

$$T_{\text{ave}}(n) \leq c_{max}\, n + \frac{2}{n} \sum_{q=0}^{n-1} T_{\text{ave}}(q).$$

Пусть $c$ — произвольная константа такая, что $c \geq \max\{\, 2\,c_{max},\, T_{\text{ave}}(0) + T_{\text{ave}}(1)\,\}$. Докажем справедливость неравенства $T_{\text{ave}}(n) \leq cn \lg n$ для любого $n \geq 2$. При $n = 2$ имеем

$$T_{\text{ave}}(2) \leq 2\,c_{max} + T_{\text{ave}}(0) + T_{\text{ave}}(1) \leq 2\,c \lg 2.$$

В примере 1 (viii) мы доказали неравенство

$$\sum_{q=2}^{n-1} q \ln q \leq \frac{n^2}{2} \ln n - \frac{n^2}{4} - \ln 4 + 1.$$

Используя индукционное предположение, при $n \geq 3$ получаем

$$\begin{aligned} T_{\text{ave}}(n) &\leq \frac{1}{2}\,cn + \frac{2}{n}\,(\,T_{\text{ave}}(0) + T_{\text{ave}}(1)\,) + \frac{2}{n}\,c \sum_{q=2}^{n-1} q \lg q \leq \\ &\leq \frac{1}{2}\,cn + \frac{2}{n}\,c + \frac{2}{n}\,c \lg e \left( \frac{n^2}{2} \ln n - \frac{n^2}{4} - 2 \ln 2 + 1 \right) = \\ &= c \left( \frac{1}{2}\,n - \frac{2}{n} - \frac{n}{2}\,\lg e + \frac{2}{n}\,\lg e \right) + cn \lg n \leq \\ &\leq cn \lg n. \end{aligned}$$

Таким образом, $T_{\text{ave}}(n) = O(n \lg n)$. Учитывая лемму 14, получаем $T_{\text{ave}}(n) = \Theta(n \lg n)$. Лемма 15 доказана.

Непосредственно из лемм 13–15 вытекает следующая теорема.

**Теорема 14.** Минимальное и среднее время работы алгоритма быстрой сортировки есть $\Theta(n \lg n)$, максимальное время равно $\Theta(n^2)$.

Таким образом, в классе алгоритмов сортировки сравнением алгоритм *QuickSort* является асимптотически оптимальным в среднем. С другой стороны, уже отсортированная входная последовательность неожиданно требует асимптотически максимального времени работы $\Theta(n^2)$. В отличие от других рассмотренных методов сортировки, алгоритм *QuickSort* быстрой сортировки "предпочитает" неупорядоченные входные данные. Несмотря на такую медленную работу в наихудшем случае, этот алгоритм часто оказывается предпочтительнее, особенно

для больших значений размерности данных, благодаря оптимальности его работы в среднем.

Дальнейшее улучшение алгоритма быстрой сортировки связано с более аккуратным выбором опорного элемента в функции $Partition$, например, случайным образом или как медиану трёх случайно определенных элементов массива. В этом случае получаем рандомизированную версию быстрой сортировки. Такой подход ускоряет время работы за счёт уменьшения константы в выражении $\Theta(n \lg n)$.

# 6. Алгоритм пирамидальной сортировки и оценки его трудоёмкости

При сортировке $n$-элементной входной последовательности *алгоритм пирамидальной сортировки*[34] требует времени работы в худшем и в среднем случае $\Theta(n \lg n)$ и, следовательно, является оптимальным в классе алгоритмов сортировки сравнением. При этом сортировка входной последовательности осуществляется на месте и не требует дополнительной памяти. Алгоритм использует структуру данных, называемую пирамидой. Познакомимся с этим понятием.

Рассмотрим произвольный массив $A = (a_1, \ldots, a_n)$ размерности $n$. Свяжем с массивом $A$ бинарное дерево $D_A$, определяемое следующим образом. Каждая вершина дерева $D_A$ соответствует элементу массива $A$. В корне дерева находится элемент $a_1$. Для любого $i = 1, 2, \ldots, \lfloor n/2 \rfloor$ вершина $a_i$ имеет двух сыновей $a_{2i}$ и $a_{2i+1}$, за исключением случая, когда $n$ чётно и $i = \lfloor n/2 \rfloor$. При этом $a_{2i}$ и $a_{2i+1}$ являются левым и правым сыном и располагаются под вершиной $a_i$ слева и справа соответственно. Если $n$ чётно, вершина $a_{n/2}$ имеет единственного сына $a_n$. Другие вершины дерева $D_A$ не имеют сыновей. Пример такого бинарного дерева $D_A$ для массива $A = (7, 4, 3, 9, 8, 6, 1, 10)$ представлен на рис. 6.

[34] Пирамидальная сортировка была открыта J. W. J. Williams. Эффективный подход к построению пирамиды предложил R. W. Floid.

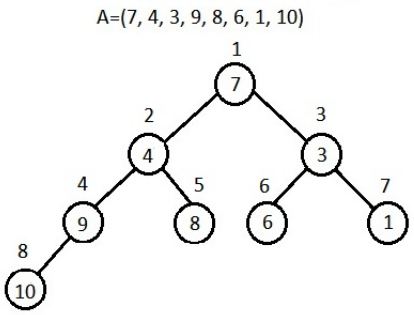



Рис. 6. Бинарное дерево $D_A$ для массива $A$

Мы получили представление произвольного массива $A$ бинарным деревом $D_A$. Заметим, что это дерево обладает следующими свойствами:

• на всех уровнях[35], кроме, быть может, последнего, дерево полностью заполнено. Иными словами, каждая вершина глубины меньшей, чем $h-1$ имеет ровно двух сыновей, где $h$ — высота дерева;

• родитель каждой вершины последнего уровня имеет ровно двух сыновей, за исключением, возможно, крайней правой вершины $v$. Родитель вершины $v$ обязательно имеет левого сына.

Массив $A$, элементы которого являются вершинами бинарного дерева $D_A$, обладающего указанными свойствами, однозначно восстанавливается по этому дереву $D_A$. Действительно, занумеруем вершины дерева $D_A$, переходя по его уровням сверху вниз, а вершины одного уровня нумеруем слева направо, последовательно переходя от левого сына к правому сыну. Такая нумерация соответствует нумерации элементов массива $A$ в порядке возрастания его индексов. Поэтому можно отождествлять массив $A$ и бинарное дерево $D_A$.

*Пирамида* — это структура данных, представляющая собой массив, который рассматривается как бинарное дерево, определённое выше и

[35] Вершины дерева, имеющие одинаковую глубину, образуют один *уровень*. Все уровни занумерованы в соответствии со значением глубины их вершин.

дополнительно обладающее следующим *свойством убывания*:

$$a_i \geq a_{2i},\ i = 1, \ldots, \lfloor n/2 \rfloor,$$

$$a_j \geq a_{2j+1},\ j = 1, \ldots, \lfloor (n-1)/2 \rfloor.$$

Другими словами, значение каждого из сыновей не превышает значения его родителя. Из этого свойства следует, что значение корня пирамиды является наибольшим среди значений всех её вершин. Для такой структуры данных также используется название *куча*, и ввиду свойства убывания такие пирамиды называют *убывающими пирамидами*.

Не каждый массив является пирамидой. Например, для массива $A = (7, 4, 3, 9, 8, 6, 1, 10)$ нарушается условие $a_5 \leq a_2$ (см. рис. 1). Однако каждый массив $A = (a_1, \ldots, a_n)$ можно преобразовать в массив, состоящий из тех же элементов и являющийся пирамидой[36]. Действительно, в дереве $D_A$ элементы $a_{\lfloor n/2 \rfloor + 1}, \ldots, a_n$ — концевые вершины, а все вершины $a_1, \ldots, a_{\lfloor n/2 \rfloor}$ имеют сыновей. Перестроение массива $A$ проведём снизу вверх. Начиная с последней родительской вершины $a_{\lfloor n/2 \rfloor}$ и заканчивая корнем $a_1$, проверяем, выполняется ли для текущей рассматриваемой вершины с индексом $i$ свойство убывания: $a_i \geq a_{2i}$ и $a_i \geq a_{2i+1}$ (последнее неравенство нужно проверять, когда $2i + 1 \leq n$). Если это свойство не выполнено, элемент $a_i$ следует поменять с бóльшим из её сыновей $a_{2i}, a_{2i+1}$. Для этого в переменную $largest$ записываем индекс наибольшего из элементов $a_i, a_{2i}, a_{2i+1}$. Если $largest = i$, то $a_i$ уже *погрузился* до нужного места, иначе меняем местами элементы $a_i$ и $a_{largest}$. Далее начинаем проверку свойства убывания для элемента $a_{largest}$, находящегося на следующем уровне, и так до тех пор, пока элемент $a_i$ не погрузится до нужного места в дереве $D_A$. Погружение вершины с индексом $i$ оформим процедурой $Pushdown$ с параметрами $pushingnumber$ — индекс элемента массива, погружаемого вниз до правильной позиции, и $heapsize$ — размерность рассматриваемого массива

[36] Далее предполагаем $n \geq 2$, так как одноэлементный массив является пирамидой.

$A$ (в дальнейшем мы будем уменьшать размерность пирамиды). Сам алгоритм построения пирамиды оформим процедурой $BuildHeap$.

**Procedure** $Pushdown(pushingnumber, heapsize)$;
$i := pushingnumber$;
**while** $i \le \lfloor heapsize/2 \rfloor$ **do**
**begin**
  $l := 2i$;
  $r := 2i + 1$;
  **if** $a_l > a_i$ **then** $largest := l$
    **else** $largest := i$;
  **if** $(r \le heapsize)$ **and** $(a_r > a_{largest})$ **then** $largest := r$;
  **if** $largest = i$ **then** $i := heapsize$
    **else begin**
      $Swap(a_i, a_{largest})$;
      $i := largest$
    **end**
**end**.

**Procedure** $BuildHeap(heapsize)$;
**for** $i = \lfloor heapsize/2 \rfloor$ **downto** 1 **do** $Pushdown(i, heapsize)$.

**Пример 13.** На рис. 7 показано построение пирамиды процедурой $BuildHeap$ для массива $A$ с бинарным деревом $D_A$, изображённым на рис. 6. Здесь приведено дерево $D_A$ для текущего состояния массива $A$ после выполнения процедуры $Pushdown(i, 8)$, $i = 4, 3, 2, 1$.

**Лемма 16.** Пусть $T(n)$ — время работы алгоритма $Pushdown(i, n)$ и $k$ — высота вершины $a_i$ в дереве $D_A$. Тогда существует такая положительная константа $c$, не зависящая от $i, n$ и входного массива $A$ размерности $n$, что $T(n) \le ck \le c \lg n$ для любого $n \ge 2$.

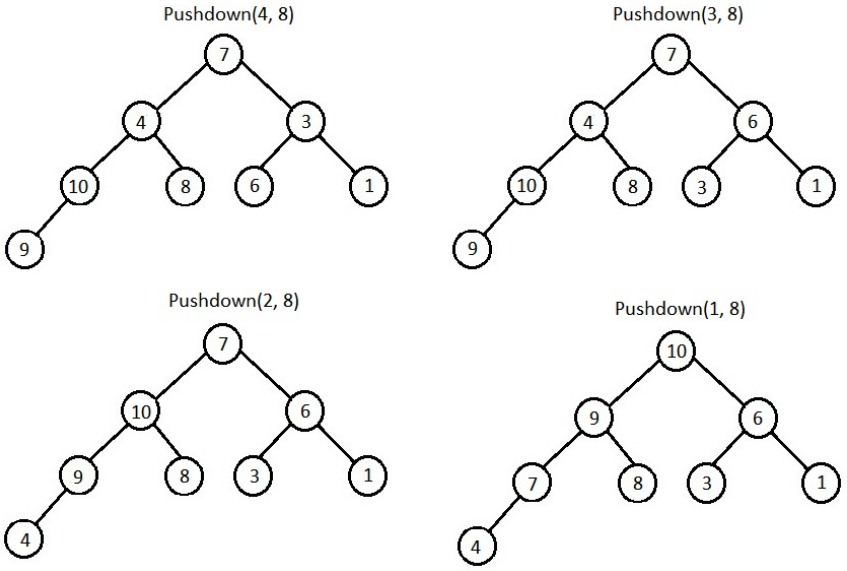



Рис. 7. Построение пирамиды

**Доказательство.** Пусть $h$ — высота бинарного дерева $D_A$. Так как каждая вершина дерева $D_A$ глубины меньшей, чем $h-1$ имеет ровно два сына, $h = \lfloor \lg n \rfloor$ по лемме 11. Поэтому $k \leq h \leq \lg n$. Осталось заметить, что каждая итерация while-цикла означает спуск по дереву $D_A$ на один уровень. Лемма 16 доказана.

**Лемма 17.** Минимальное, среднее и максимальное время работы алгоритма $BuildHeap(n)$ есть $\Theta(n)$.

**Доказательство.** Пусть $h$ — высота бинарного дерева $D_A$, соответствующего массиву $A$ размерности $n$. Время работы процедуры $Pushdown(i, n)$ зависит от высоты $k$ вершины с индексом $i$ и не превосходит $ck$ операций, где $c$ — константа, определённая в лемме 16. Причём $k \leq h$ и число вершин высоты $k$ в бинарном дереве $D_A$ не превосходит $2^{h-k}$ по лемме 10(i). Кроме того, $n \geq 2^h$ в силу леммы 11. Таким об-

разом, для времени $T(n)$ работы алгоритма $BuildHeap(n)$ при $n \geq 2$ выполняются следующие оценки

$$\begin{aligned} T(n) &\leq \sum_{k=0}^{h} 2^{h-k}\, ck = c2^h \sum_{k=0}^{h} \frac{k}{2^k} \leq \\ &\leq cn \sum_{k=0}^{\infty} \frac{k}{2^k} = 2cn = O(n), \end{aligned}$$

здесь при $q = 1/2$ мы воспользовались равенством[37]

$$\sum_{k=0}^{\infty} kq^k = \frac{q}{(1-q)^2}\,.$$

В алгоритме $BuildHeap(n)$ число вызовов процедуры $Pushdown(i, n)$ равно $\lfloor n/2 \rfloor$. Поэтому $T(n) = \Omega(n)$. Лемма 17 доказана.

Рассмотрим алгоритм пирамидальной сортировки. Требуется отсортировать элементы массива $A = (a_1, \ldots, a_n)$ в порядке возрастания. С помощью процедуры $BuildHeap(n)$ массив $A$ преобразуем в пирамиду. Ввиду свойства убывания пирамиды наибольший элемент массива будет находиться в корне пирамиды $a_1$. Обменяем местами элементы $a_1$ и $a_n$. В результате наибольший элемент массива $A$ займёт требуемую позицию. Перейдём к массиву меньшей размерности $a_1, \ldots, a_{n-1}$, который также преобразуем в пирамиду. Заметим, что свойство убывания сохранится для всех вершин с индексом $i$ при $i = 2, \ldots, n-1$ и может нарушиться только для нового корневого элемента $a_1$. Поэтому для получения пирамиды достаточно лишь погрузить элемент $a_1$ до нужного места с помощью процедуры Pushdown$(1, n-1)$. После этого массив $A$ на первых $n-1$ позициях превратится в пирамиду. Снова обменяем элементы $a_1$, $a_{n-1}$ и перейдём к массиву размерности $n-2$. Продолжаем описанный процесс до тех пор, пока не дойдём до одноэлементного массива $a_1$. Программная реализация этого алгоритма приведена в следующем псевдокоде.

[37] Это равенство получается дифференцированием суммы геометрической прогрессии $\sum\limits_{k=0}^{\infty} q^k = \frac{1}{(1-q)}$ и умножением на $q$ при условии $-1 < q < 1$.

**Алгоритм $HeapSort(n)$ пирамидальной сортировки**

$BuildHeap(n)$;
**for** $i = n$ **downto** 2 **do**
**begin**
  $Swap(a_1, a_i)$;
  $Pushdown(1, i-1)$
**end**.

**Пример 14.** На рис. 8 показан пример работы алгоритма пирамидальной сортировки массива $A = (7, 4, 3, 9, 8, 6, 1, 10)$ с бинарным деревом $D_A$, приведённым на рис. 6.

**Теорема 15.** Максимальное и среднее время работы алгоритма пирамидальной сортировки есть $\Theta(n \lg n)$. Минимальное время работы, когда сортируются различные элементы, равно $\Theta(n \lg n)$ и есть $\Theta(n)$, если допускаются равные элементы сортируемой последовательности.

**Доказательство.** Оценим минимальное $T_{\min}(n)$, среднее $T_{\text{ave}}(n)$ и максимальное $T_{\max}(n)$ время работы алгоритма $HeapSort(n)$. В силу лемм 16, 17 получаем $T_{\min}(n) = \Omega(n)$ и $T_{\max}(n) = O(n) + (n-1)O(\lg n) = O(n \lg n)$. По теореме 12 имеем $T_{\text{ave}}(n) = \Omega(n \lg n)$. Следовательно, $T_{\text{ave}}(n) = \Theta(n \lg n)$ и $T_{\max}(n) = \Theta(n \lg n)$.

Рассмотрим случай, когда допускаются равные элементы сортируемой последовательности. На входной последовательности, состоящей из равных элементов, в алгоритме $HeapSort(n)$ время работы каждой из вызываемых процедур $Pushdown(1, i-1)$, $i = n, n-1, \ldots, 2$ ограничено одной константой, не зависящей от $i$ и $n$ (поскольку выполняется не более одной итерации while-цикла). Поэтому $T_{\min}(n) = \Theta(n)$.

Теперь предположим, что все элементы сортируемого массива $A$ размерности $n$ различны. После выполнения процедуры $BuildHeap(n)$ массив $A$ является $n$-вершинной пирамидой. Пусть $h$ — высота дерева $D_A$.

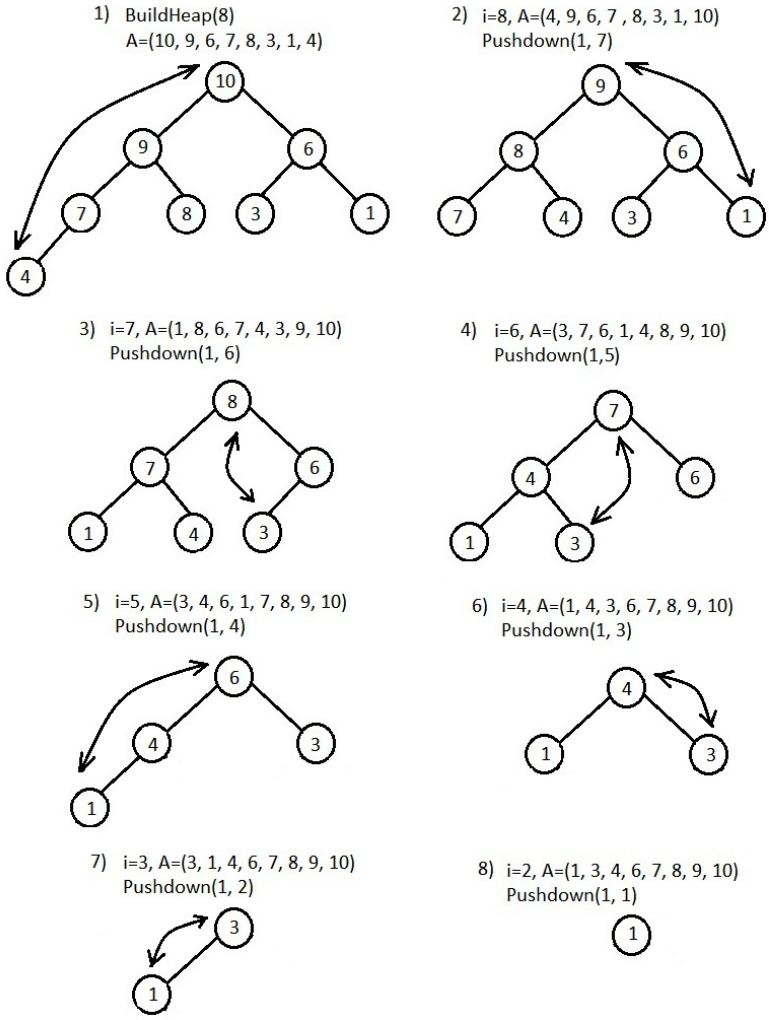



Рис. 8. Пирамидальная сортировка массива $A = (7, 4, 3, 9, 8, 6, 1, 10)$

По лемме 11 имеем $h = \lfloor \lg n \rfloor$. Рассмотрим такое наибольшее положительное целое число $h'$, что вершины дерева $D_A$, имеющие глубину, не превосходящую $h'$, образуют полное бинарное дерево. Тогда $h'$ — высота этого дерева и $h - 1 \le h' \le h$. Рассмотрим число $i_0 \ge 0$ итераций for-цикла, после которых элементы массива $A$ на первых $n - i_0$ местах образуют полное бинарное дерево высоты $h'$. Это дерево является пирамидой. Обозначим её через $D'$.

Упорядочим вершины дерева $D'$ в порядке возрастания их значений. Последние $2^{h'}$ вершин относительно этого порядка будем называть *тяжёлыми вершинами*. Оценим число $m$ тяжёлых вершин глубины, меньшей $h'$ в дереве $D'$. В силу свойства убывания пирамиды $D'$ предки произвольной тяжёлой вершины также являются тяжёлыми. Следовательно, тяжёлые вершины бинарного дерева $D'$ образуют бинарное $2^{h'}$-вершинное дерево $D''$ с тем же корнем. Используя лемму 10 (v), получаем $2^{h'} = m + k \ge 2k - 1$, где $k$ — число концевых вершин глубины $h'$ в дереве $D''$. Поэтому выполняются неравенства

$$2^{h'-1} \le m \le 2^{h'}.$$

По лемме 10 (i),(iii) число концевых вершин дерева $D'$ равно $2^{h'}$, т. е. числу тяжёлых вершин. Поэтому после $2^{h'}$ итераций for-цикла, соответствующих значениям счётчика $i = i_0 + 1, i_0 + 2, \ldots, i_0 + 2^{h'}$, $m$ тяжёлых неконцевых вершин дерева $D'$ достигнут корня дерева в результате последовательных обменов со своими непосредственными предками. Следовательно, для времени $T(n)$ работы алгоритма $HeapSort(n)$ выполняется неравенство $T(n) \ge S(D'')$, где $S(D'')$ — сумма глубин всех $m$ вершин глубины, меньшей $h'$ в дереве $D''$. Используя лемму 12, получаем $S(D'') \ge \sum_{j=1}^{m} \lfloor \lg j \rfloor$, причём $m \ge 2^{h'-1} \ge \lceil n/8 \rceil$. Учитывая пример 1(ix), заключаем

$$T(n) \ge \sum_{j=1}^{\lceil n/8 \rceil} \lfloor \lg j \rfloor = \Omega(n \lg n).$$

Таким образом, $T_{\min}(n) = \Theta(n \lg n)$. Теорема 15 доказана.

# 7. Линейный алгоритм сортировки подсчётом

До сих пор мы рассматривали алгоритмы сортировки, основанные на сравнении входных элементов. Согласно теореме 11 время работы таких алгоритмов составляет $\Omega(n \lg n)$ в худшем случае. Познакомимся с линейным *алгоритмом сортировки подсчётом*, предложенным H. H. Seward. Разумеется, этот алгоритм улучшает оценку $\Omega(n \lg n)$ за счёт использования внутренней структуры сортируемых объектов.

В сортировке подсчётом предполагается, что все входные элементы $x_1, \ldots, x_n$ — целые неотрицательные числа, не превосходящие некоторой заранее известной целой константы $k$. При программной реализации этого алгоритма используется входной массив $A$ размерности $n$, в массив $B$ размерности $n$ записывается отсортированная входная последовательность, нам потребуется также вспомогательный массив $C$ размерности $k + 1$.

Основную идею сортировки подсчётом легко понять в случае, когда все входные элементы различны. Предварительно для каждого элемента $x_i$ входного массива $A$ подсчитываем количество $C(x_i)$ элементов входной последовательности, которые меньше $x_i$. Далее элемент $x_i$ помещаем в выходной массив $B$ на позицию с номером $C(x_i) + 1$. Например, если элементов, меньших $x_i$ ровно 3, то в выходном массиве элемент $x_i$ размещаем в $B(4)$. Если во входной сортируемой последовательности присутствуют равные числа $x_{i_1} = \ldots = x_{i_m} = x$, $i_1 < \ldots < i_m$, то эту схему требуется модифицировать так, чтобы не записывать равные элементы на одно место выходного массива. Для этого предварительно подсчитываем количество $C(x)$ элементов входной последовательности, не превосходящих $x$, и размещаем элементы $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ в выходном массиве $B$ на места с номерами $C(x) - (m-1), C(x) - (m-2), \ldots, C(x)$ соответственно. В результате равные элементы входной последовательности будут находиться в выходном массиве $B$ в том же порядке, что и во входном и на правильных местах.

**Алгоритм** $CountingSort(n)$ **сортировки подсчётом**

**for** $i = 0$ **to** $k$ **do** $C(i) := 0$;
**for** $i = 1$ **to** $n$ **do** $C(A(i)) := C(A(i)) + 1$;
**for** $i = 1$ **to** $k$ **do** $C(i) := C(i) + C(i-1)$;
**for** $i = n$ **downto** $1$ **do**
**begin**
$B(C(A(i))) := A(i)$;
$C(A(i)) := C(A(i)) - 1$
**end**.

**Пример 15.** Процесс работы алгоритма $CountingSort(4)$ на входной последовательности $A = (4, 2, 4, 3)$ при $k = 4$ представлен в виде состояний массивов $B$, $C$:

$B = (*, *, *, *)$, $C = (*, *, *, *, *)$;
$B = (*, *, *, *)$, $C = (0, 0, 0, 0, 0)$, $i = k = 4$;
$B = (*, *, *, *)$, $C = (0, 0, 1, 1, 2)$, $i = n = 4$;
$B = (*, *, *, *)$, $C = (0, 0, 1, 2, 4)$, $i = k = 4$;
$B = (*, 3, *, *)$, $C = (0, 0, 1, 1, 4)$, $i = n = 4$;
$B = (*, 3, *, 4)$, $C = (0, 0, 1, 1, 3)$, $i = 3$;
$B = (2, 3, *, 4)$, $C = (0, 0, 0, 1, 3)$, $i = 2$;
$B = (2, 3, 4, 4)$, $C = (0, 0, 0, 1, 2)$, $i = 1$.

Отсортированная последовательность $(2, 3, 4, 4)$ записана в массиве $B$.

**Теорема 16.** Алгоритм $CountingSort(n)$ корректно сортирует последовательность длины $n$ из целых неотрицательных чисел, не превосходящих некоторой целой константы $k$. Минимальное, среднее и максимальное время его работы есть $\Theta(n + k)$.

**Доказательство.** Во втором for-цикле выполняется проверка каждого входного элемента. Если его значение равно $x$, то к величине $C(x)$ прибавляется единица. Таким образом, после выполнения этого цикла

для каждого $i = 0, 1, \ldots, k$ в $C(i)$ будет записано количество элементов входного массива $A$, равных $i$. Используя эту информацию, в следующем for-цикле для каждого $i = 0, 1, \ldots, k$ в $C(i)$ перезаписывается число входных элементов, не превосходящих $i$. Наконец, в последнем for-цикле каждый элемент $A(i)$ входного массива помещается в надлежащую позицию выходного массива $B$. Действительно, если все $n$ входных элементов различны, то в выходном отсортированном массиве $B$ число $A(i)$ должно стоять на месте с номером $C(A(i))$, поскольку имеется $C(A(i))$ элементов, меньших $i$. Если в массиве $A$ встречаются повторения, то после каждой записи числа $A(i)$ в массив $B$ число $C(A(i))$ уменьшается на единицу, поэтому при следующей встрече с числом, равным $A(i)$, оно будет записано на одну позицию левее. Таким образом, алгоритм $CountingSort$ корректно сортирует входную последовательность элементов массива $A$.

Оценим время работы алгоритма $CountingSort(n)$ на произвольной входной последовательности. На выполнение первого и третьего for-циклов затрачивается время $\Theta(k)$, а на выполнение второго и четвёртого for-циклов — время $\Theta(n)$. Таким образом, полное время работы есть $\Theta(n + k)$. Теорема 16 доказана.

**Следствие 2.** Если $k = \Theta(n)$, то алгоритм $CountingSort(n)$ является линейным.

# Список литературы


1. *Алексеев В. Б.* Введение в теорию сложности алгоритмов. М.: МГУ, 2002.

2. *Андерсен Д. А.* Дискретная математика и комбинаторика. М.: Издат. дом "Вильямс", 2004.

3. *Ахо А. В., Хопкрофт Д. Э., Ульман Д. Д.* Структуры данных и алгоритмы. М.: Издат. дом "Вильямс", 2007.

4. *Виленкин Н. Я.* Комбинаторика. М.: Наука, 1969.

5. *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. М.: Мир, 1981.

6. *Емеличев В. А., Мельников О. И., Сарванов В. И., Тышкевич Р. И.* Лекции по теории графов. М.: Наука, 1990.

7. *Иванов Б. Н.* Дискретная математика. Алгоритмы и программы. М.: Физматлит, 2007.

8. *Кнут Д. Э.* Искусство программирования. М.: Издат. дом "Вильямс", 2007. Т. 1–4.

9. *Корман Т., Ривест Р., Лейзерсон Ч.* Алгоритмы: построение и анализ. М.: Издат. дом "Вильямс", 2007.

10. *Кузюрин Н. Н., Фомин С. А.* Эффективные алгоритмы и сложность вычислений. М.: МГУ, 2009.

11. *Липский В.* Комбинаторика для программистов. М.: Мир, 1988.

12. *Макконнелл Дж.* Основы современных алгоритмов. М.: Техносфера, 2004.

13. *Новиков Ф. А.* Дискретная математика для программистов. СПб.: Издат. дом "Питер", 2007.

14. *Пападимитриу Х., Стайглиц К.* Комбинаторная оптимизация. Алгоритмы и сложность. М.: Мир, 1985.

15. *Плотников А. Д.* Дискретная математика. М.: Новое знание, 2005.

16. *Рейнгольд Э., Нивергельт Ю., Део Н.* Комбинаторные алгоритмы, теория и практика. М.: Мир, 1980.

17. *Рыбников К. А.* Введение в комбинаторный анализ. М.: МГУ, 1985.

18. *Харари Ф.* Теория графов. М.: Мир, 1973.

19. *Холл М.* Комбинаторика. М.: Мир, 1970.

20. *Шень А.* Программирование: теоремы и задачи. М.: МЦНМО, 2004.

21. *Эндрюс Г.* Теория разбиений. М.: Наука, 1982.

Учебное издание

**Федоряева** *Татьяна Ивановна*

**Комбинаторные алгоритмы**
Учебное пособие

Редактор Е. П. Войтенко

Подписано в печать 22.12.2011 г.
Формат 60×84 1/16. Оффсетная печать.
Уч.-изд. л. 7,4. Усл.-печ. л. 7. Тираж 100 экз.
Заказ №
Редакционно-издательский центр НГУ.
630090, Новосибирск, 90, ул. Пирогова, 2.