МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ГРАЖДАНСКОЙ АВИАЦИИ

А.А. Егорова,
Н.Л. Рогожникова

АЛГОРИТМЫ ДИСКРЕТНОЙ МАТЕМАТИКИ

Учебное пособие

Москва
2019

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ (МГТУ ГА)»

Кафедра прикладной математики

А.А. Егорова, Н.Л. Рогожникова

АЛГОРИТМЫ ДИСКРЕТНОЙ МАТЕМАТИКИ

Учебное пособие

Утверждено Редакционно-издательским советом МГТУ ГА в качестве учебного пособия

Москва
2019

УДК
ББК 517.8
Е-30

Печатается по решению редакционно-издательского совета
Московского государственного технического университета ГА

Рецензенты:
Коновалов В.М. (МГТУ ГА) – канд. техн. наук, доцент;
Акчурин М.Р. (ООО «Технологический центр Дойче Банка») – канд. техн. наук

Егорова А.А.
Е-30 Алгоритмы дискретной математики: учебное пособие. /А.А. Егорова, Н.Л. Рогожникова. — Воронеж: ООО «МИР», 2019. — 100 с.

ISBN

Учебное пособие издается в соответствии с рабочей программой учебной дисциплины «Алгоритмы дискретной математики» по учебному плану для студентов I курса направления 01.03.04 очной формы обучения.

Данное учебное пособие разбито на ряд взаимосвязанных и взаимоувязанных тем или разделов, представляющих собой логически законченные информационные «блоки», которые содержат наиболее важные алгоритмические задачи дискретной математики, прежде всего, при построении и анализе компьютерных алгоритмов.

Учебное пособие охватывает следующие темы: понятие алгоритма и алгоритмизации, методы стандартизации алгоритма (машина Тьюринга, нормальные алгоритмы Маркова), понятие рекурсии и примитивно-рекурсивные функции, анализ алгоритмов и основы вычислительной сложности, теоретико-числовые алгоритмы, в том числе рекурсивный алгоритм Евклида и алгоритмы проверки простоты числа, жадные алгоритмы и задача о выборе процесса, а также комбинаторные алгоритмы и основные задачи на комбинаторных объектах: исчерпывающий поиск, подсчет и оценивание методом рекуррентных соотношений и производящих функций, классы сложности. В отдельном разделе представлены упражнения, структурированные по разделам, позволяющие студентам закрепить освоенный материал.

Рассмотрено и одобрено на заседании кафедры 18.06.2019 г. и методического совета 18.06.2019 г.

В авторской редакции.

ББК 517.8
Св. тем. план 2019 г.
поз. 31

ЕГОРОВА Алла Альбертовна, РОГОЖНИКОВА Наталья Львовна

АЛГОРИТМЫ ДИСКРЕТНОЙ МАТЕМАТИКИ

Учебное пособие

Подписано в печать 08.07.2019 г.
Формат 60x80/16 Печ. л. 3 Усл. печ. л. 2,79
Заказ 510/ Тираж 35 экз.

Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д.20
Отпечатано ООО «МИР»
394033, г. Воронеж, Ленинский пр-т 119А, лит. Я, оф. 215

ISBN

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящее учебное пособие предназначено для студентов специальности 01.03.04, изучающих дисциплину «Алгоритмы дискретной математики». Дисциплина относится к обязательным дисциплинам базовой части образовательной программы направления подготовки 01.03.04, квалификация бакалавр.

Алгоритмы дискретной математики - дисциплина, изучающая вычислительные алгоритмы над комбинаторными объектами.

Для изучения данной учебной дисциплины необходимы знания, умения и навыки, формируемые предшествующими дисциплинами: математический анализ, линейная алгебра и аналитическая геометрия, комбинаторика и теория графов, программирование для ЭВМ.

Данное учебное пособие имеет своей целью помочь студентам в освоении дисциплины «Алгоритмы дискретной математики»:

- усвоение теоретических основ построения алгоритмов дискретной математики, составляющих фундамент ряда математических дисциплин прикладного характера;
- усвоение различных алгоритмов и понятий, используемых при изучении теоретико-программистских дисциплин;
- овладение алгоритмами дискретной математики, необходимыми при решении практических задач.

Учебное пособие разбито на ряд взаимосвязанных и взаимоувязанных тем или разделов, представляющих собой логически законченные информационные “блоки”, которые содержат наиболее важные алгоритмические задачи дискретной математики, прежде всего, при построении и анализе компьютерных алгоритмов.

1. РЕКУРСИВНЫЕ И ИТЕРАТИВНЫЕ АЛГОРИТМЫ

1.1. Понятие алгоритма и алгоритмизации.

Теория алгоритмов - наука, изучающая общие свойства и закономерности алгоритмов и разнообразные формальные модели их представления. К задачам теории алгоритмов относятся формальное доказательство алгоритмической неразрешимости задач, асимптотический анализ сложности алгоритмов, классификация алгоритмов в соответствии с классами сложности, разработка критериев сравнительной оценки качества алгоритмов и т. п.

Развитие теории алгоритмов начинается с доказательства К. Гёделем теорем о неполноте формальных систем, включающих арифметику, первая из которых была доказана в 1931 г. Уже 1934 году, совершая поездку в США (Принстонский университет), К. Гёдель прочитал курс лекций «О неразрешимых теоремах формальных математических систем». Возникшее в связи с этими теоремами предположение о невозможности алгоритмического разрешения многих математических проблем вызвало необходимость стандартизации понятия алгоритма. Первые стандартизованные варианты этого понятия были разработаны в 30-х годах XX века в работах А. Тьюринга, А. Чёрча и Э. Поста. Предложенные ими машина Тьюринга, машина Поста и лямбда-исчисление Чёрча оказались эквивалентными друг другу.

В настоящее время теория алгоритмов развивается, главным образом, по трем направлениям:

- Классическая теория алгоритмов изучает проблемы формулировки задач в терминах формальных языков, вводит понятие задачи разрешения, проводит классификацию задач по классам сложности.
- Теория асимптотического анализа алгоритмов рассматривает методы получения асимптотических оценок ресурсоемкости или времени выполнения алгоритмов, в частности, для рекурсивных алгоритмов. Асимптотический анализ позволяет оценить рост потребности алгоритма в ресурсах с увеличением объема входных данных.

– Теория практического анализа вычислительных алгоритмов решает задачи получения явных функции трудоёмкости, интервального анализа функций, поиска практических критериев качества алгоритмов, разработки методики выбора рациональных алгоритмов.

Вообще говоря, алгоритм – это некоторое формальное предписание, действуя согласно которому, можно получить решение задачи. Но алгоритмы решают не только частные задачи, но и целые классы задач. Подлежащие решению частные задачи, выделяемые по мере необходимости из рассматриваемого класса, определяются с помощью параметров. Параметры же являются своеобразными исходными данными для алгоритма.

Алгоритм (algorithm) – любая корректно определенная вычислительная процедура, на вход (input) которой подается некоторая величина или набор величин **(аргумент алгоритма)**, и результатом выполнения которой является выходная (**output**) величина или набор значений (**результат**).

Слово algotithm –произошло от аль-Хорезми – автора известного арабского учебника по математике.

Правильный (корректный) алгоритм - такой алгоритм, который после каждого ввода данных, своим результатом имеет корректный **вывод**. Корректный алгоритм решает данную вычислительную задачу.

Таким образом, мы можем выделить следующие основные особенности алгоритма:

- Определенность. Алгоритм разбивается на определенные шаги (этапы), каждый из которых должен быть простым и локальным.
- Ввод. Алгоритм имеет некоторое (быть может, равное нулю) число входных данных, т.е. величин, заданных ему до начала работы.
- Вывод. Алгоритм имеет одну или несколько выходных величин, т. е. величин, имеющих вполне определенное отношение к входным данным.
- Детерминированность. После выполнения очередного шага алгоритма однозначно определено, что делать на следующем шаге.

Размер входа алгоритма (input size):

1) число элементов на входе (Например, для сортировки или для преобразования Фурье);

2) общее число битов, необходимое для представления всех входных данных (например, перемножение двух целых чисел);

3) несколько числовых значений (например, число вершин и число ребер графа).

Каждый из наборов входных данных называется **экземпляром задачи,** которую решает алгоритм.

При анализе и разработке машинно-независимых алгоритмов используется **модель обобщенной однопроцессорной машины с произвольным доступом к памяти.** (Random - Access - Machine) или RAM-машиной. В этой модели команды процессора выполняются последовательно; одновременно выполняемые операции отсутствуют. В модели RAM есть целочисленный тип данных и тип чисел с плавающей точкой. Согласно этой модели, наш компьютер работает таким образом:

- для исполнения любой простой операции (+, *, -, =, if, call) требуется ровно один временной шаг;

- циклы и подпрограммы не считаются простыми операциями, а состоят из нескольких простых операций;

- каждое обращение к памяти занимает один временной шаг. Кроме этого, рассматриваемый компьютер обладает неограниченным объемом оперативной памяти. Кэш и диск в модели RAM не применяются.

Время работы (running time**) алгоритма** – число элементарных шагов, которые выполняет алгоритм. Время затрачивается также на **вызов** (call) процедуры и ее **исполнение** (execution).

Алгоритмика - дисциплина, изучающая алгоритмы, и их применение к решению задач. Алгоритмика отличается от теории алгоритмов тем, что не занимается поиском доказательства существования алгоритма, а занимается по-

иском оптимального (в основном за время выполнения), алгоритма, решающего данную задачу. Если такой алгоритм неизвестен, то пытается решить задачу хотя бы частично.

1.2. Стандартизация алгоритма: нормальный алгоритм Маркова и машина Тьюринга.

Одним из наиболее удачных стандартизованных вариантов алгоритма является разработанное десятью годами позже работ Тьюринга и Поста понятие нормального алгоритма. Оно было введено А. А. Марковым, многие годы читавшему курс математической логики на механико-математическом факультете МГУ, в работах по неразрешимости некоторых проблем теории ассоциативных вычислений. Традиционное написание и произношение слова «алгорифм» в этом термине также восходит к его автору.

Нормальный алгоритм Маркова (НАМ, также **марковский алгоритм**) — один из стандартных способов формального определения понятия алгоритма Нормальные алгоритмы являются вербальными, то есть представляют собой правила по переработке слов в некоторых алфавитах.

Алфавит - любое непустое множество. Элемент алфавита называется **буквой**. Последовательность букв данного алфавита называется **словом. Пустое слово** не имеет в своем составе ни одной буквы и обозначается **Λ.**

Пусть B – алфавит, а M_1 и M_2 – алгоритмы над алфавитом B. Говорят, что ***M_1* и *M_2* эквивалентны относительно *B***, если для любых слов P и Q в алфавите B выполняется два условия:

1) если M_1: $P \Rightarrow Q$, то M_2: $P \Rightarrow Q$,

2) если M_2: $P \Rightarrow Q$, то M_1: $P \Rightarrow Q$.

Марковской подстановкой называется операция над упорядоченной парой слов (P, Q), состоящая в следующем. В заданной слове R находят первое вхождение слова P (если оно есть), и не изменяя остальных частей слова С, заменяют в нем это вхождение словом Q. Полученное слово называется результатом применения марковской подстановки (P, Q) к слову R. Если же нет

вхождения слова P в слово R, то считается, что марковская подстановка (P, Q) не применима к слову R.

Частными случаями марковских подстановок являются подстановки с пустыми словами: (Λ, Q), (P, Λ), (Λ, Λ).

Определение всякого нормального алгоритма состоит из двух частей: определения алфавита алгоритма и определения его схемы.

Схемой нормального алгоритма называется конечный упорядоченный набор формул подстановки, каждая из которых может быть простой или заключительной.

Простыми формулами подстановки называются слова вида $P \to Q$, где P и Q — два произвольных слова в алфавите алгоритма (называемые, соответственно, **левой и правой частями формулы подстановки**).

Аналогично, **заключительными формулами подстановки** называются слова вида $P \to \cdot Q$, где P и Q — два произвольных слова в алфавите алгоритма.

Важно, что вспомогательные буквы $\to$ и $\to \cdot$ не принадлежат алфавиту алгоритма (в противном случае на исполняемую ими роль разделителя левой и правой частей следует избрать другие две буквы).

Пример 1.2.1. Пусть $A = \{k, m\}$ – алфавит. Рассмотрим следующую схему нормального алгоритма в A:

$$\begin{cases} k \to \cdot \Lambda \\ m \to m \end{cases}$$

Этот алгоритм всякое слово P в алфавите A, содержащее хотя бы одно вхождение буквы k, перерабатывает в слово, получающееся из P вычеркиванием самого первого (левого) вхождения буквы k. Пустое слово алгоритм перерабатывает в пустое.

Например, $kkmkmk \Rightarrow kmkm$, $km \Rightarrow m$, $mk \Rightarrow m$, $kk \Rightarrow k$, $mkm \Rightarrow mm$.

Алгоритм неприменим к словам, которые содержат только букву m.

Пример 1.2.2 Построим нормальный алгоритм для вычисления функции $f(x) = x + 1$ в десятичной системе.

Алфавит $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, а нормальный алгоритм будем строить в его расширении $B = A \cup \{a, b\}$.

Схема этого нормального алгоритма следующая:

$0b \to \cdot 1$	$a0 \to 0a$	$0a \to 0b$
$1b \to \cdot 2$	$a1 \to 1a$	$1a \to 1b$
$2b \to \cdot 3$	$a2 \to 2a$	$2a \to 2b$
$3b \to \cdot 4$	$a3 \to 3a$	$3a \to 3b$
$4b \to \cdot 5$	$a4 \to 4a$	$4a \to 4b$
$5b \to \cdot 6$	$a5 \to 5a$	$5a \to 5b$
$6b \to \cdot 7$	$a6 \to 6a$	$6a \to 6b$
$7b \to \cdot 8$	$a7 \to 7a$	$7a \to 7b$
$8b \to \cdot 9$	$a8 \to 8a$	$8a \to 8b$
$9b \to \cdot b0$	$a9 \to 9a$	$9a \to 9b$
$b \to \cdot 1$		$\Lambda \to a$

Если применить алгоритм к пустому слову, то на каждом шаге должна будет применяться самая последняя формула схемы. Получается бесконечный процесс: $\Lambda \to a \to aa \to aaa \ldots$.

Мы видим, что данный алгоритм не применим к пустому слову.

Если применить алгоритм к слову 74, то получим следующую последовательность: $74 \to a74 \to 7a4 \to 74a \to 74b \to 75$.

Сформулируем **принцип нормализации Маркова**: всякий алгоритм может быть реализован нормальным алгоритмом Маркова (или всякий алгоритм нормализуем).

Механизм нормальных алгоритмов настолько прост, что напоминает скорее детскую игру, чем математику. Но на самом деле это очень мощный механизм, поскольку через него можно выразить решение любой алгоритмически разрешимой задачи. Но из этого не следует, что любую задачу необходимо решать через подстановки. Это лишь означает, что любую алгоритмиче-

ски разрешимую задачу можно представить в виде такой системы подстановок. А если нельзя (и вы это смогли доказать), то такая задача вообще не имеет алгоритма решения.

Другим известным способом задания алгоритма является машина Тьюринга (МТ).

Машина Тьюринга состоит из:

1) управляющего устройства, которое может находиться в одном из состояний, образующих конечное множество $Q = \{q_1, \ldots, q_n\}$;

2) бесконечной ленты, разбитой на ячейки, в каждой из которых может быть задан один из символов конечного алфавита $A = \{a_1, \ldots, a_m\}$;

3) устройства обращения к ленте, т.е. считывающей и пишущей головки, которая в каждый момент времени обозревает ячейку ленты и в зависимости от символа в этой ячейке и состояния управляющего устройства:

а) записывает в ячейку символ (быть может, совпадающий с прежним или пустой);

б) сдвигается на ячейку влево или вправо, или остается на месте;

в) переходит в новое состояние (или остается в прежнем).

Среди состояний управляющего устройства выделяют начальное состояние q_1 и заключительное состояние q_z. В начальном состоянии машина находится перед началом работы, а попав в заключительное состояние, останавливается.

Полное состояние машины Тьюринга (конфигурация или **машинное слово)**, по которому можно однозначно определить ее дальнейшее поведение, определяется ее внутренним состоянием, состоянием ленты (словом, записанном на ленте) и положением головки на ленте и обозначается тройкой $\alpha_1 q_i \alpha_2$. При этом, q_i - текущее внутреннее состояние, α_1 - слово слева от головки, а α_2 - слово, образованное символом, обозреваемым головкой, и словом справа от него, причем слева от α_1 и справа от α_2 нет пустых символов.

Стандартной начальной конфигурацией называется конфигурация вида $q_1\alpha$, т.е. конфигурацию содержащую начальное состояние, в котором головка обозревает крайний левый символ слова, написанного на ленте.

Стандартной заключительной конфигурацией называется конфигурация вида $q_z\alpha$.

Данные МТ - это слова в алфавите ленты.

Память МТ - это конечное множество состояний (внутренняя память) и лента (внешняя память). Лента бесконечна в обе стороны. В начальный момент времени только конечное число ячеек ленты заполнено непустыми символами, остальные ячейки пусты, т. е содержат пустой символ λ (пробел).

Машина Тьюринга называется **детерминированной**, если каждой комбинации состояния и ленточного символа в таблице соответствует не более одного правила. Конкретная машина Тьюринга задаётся перечислением элементов множества букв алфавита *A*, множества состояний *Q* и набора правил, по которым работает машина.

Правила имеют вид: $q_i a_j \rightarrow q_i' a_j' d_k$.

Если головка находится в состоянии q_i, а в текущей ячейке записана буква a_j, то головка переходит в состояние q_1', в ячейку вместо a_j записывается a_j', головка делает движение d_k, которое имеет три варианта: на ячейку влево (L), на ячейку вправо (R), остаться на месте (E). Для каждой возможной конфигурации $<q_i, a_j>$ имеется ровно одно правило (для недетерминированной машины Тьюринга может быть большее количество правил.

Правил нет только для заключительного состояния, попав в которое машина останавливается.

Кроме того, необходимо указать конечное и начальное состояния, начальную конфигурацию на ленте и расположение головки.

Приведенное описание показывает **дискретность** машины Тьюринга.

Элементарные шаги МТ - это считывание и запись символов, сдвиг на ячейку влево или вправо, а также переход управляющего устройства в следующее состояние.

Результатом работы МТ является слово на ленте после остановки машины.

Возможность выбора в качестве начальной системы любого слова в алфавите ленты обеспечивает **массовость** машины Тьюринга.

Пример 1.2.3. Конфигурация с внутренним состоянием q_i, в которой на ленте записано *klmno*, а головка обозревает *n*, запишется как $klmq_ino$.

q_i

…	λ	λ	*k*	*l*	*m*	*n*	*o*	λ	λ	…

Пример 1.2.4. МТ с алфавитом $A = \{c, \lambda\}$, состояниями $\{ q_1, q_2 \}$ и системой команд $q_1c \rightarrow q_1cR$, $q_1\lambda \rightarrow q_1cR$ при любой начальной конфигурации будет работать бесконечно, заполняя символом *c* всю ленту вправо от начальной точки.

Число представимо в **единичном (унарном) коде**, когда для всех числовых функций существует алфавит $A = \{1, *, \lambda\}$ и число *x* представляется словом $1\ldots1 = 1^x$, состоящем из *x* единиц.

Пример 1.2.5. Сложить числа *f* и *m*, представленные в унарном коде. Другими словами, $1^f * 1^m = 1^{f+m}$, т.е. надо удалить разделитель * и сдвинуть одно из слагаемых к другому. Диаграмма переходов МТ для решения этой задачи представлена на рисунке 1.1.

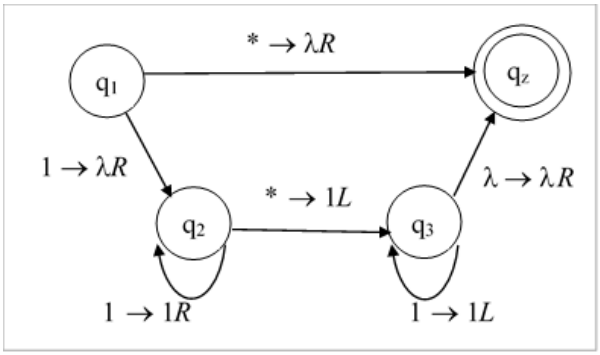


Рисунок 1.1 – Диаграмма переходов МТ для сложения чисел в унарном коде.

МТ для решения этой задачи имеет четыре состояния и следующую систему команд:

$q_1* \rightarrow q_z\lambda R$

$q_1 1 \rightarrow q_2\lambda R$

$q_2 1 \rightarrow q_2 1R$

$q_2* \rightarrow q_3 1L$

$q_3 1 \rightarrow q_3 1L$

$q_3\lambda \rightarrow q_z\lambda R$

Систему команд МТ можно интерпретировать и как описание работы конкретного механизма, и как программу, т.е. совокупность предписаний, однозначно приводящих к результату.

Возможно построение композиций машин Тьюринга. Например, на рисунке 1.2 представлена блок-схема композиций машин T_1 и T_2, которая обозначается $T_2(T_1)$.

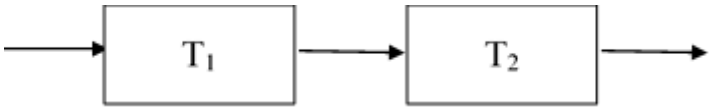


Рисунок 1.2 – Блок-схема композиции $T_2(T_1)$.

Эта блок-схема всегда предполагает, что исходными данными машины T_2 являются результаты T_1. Благодаря вычислимости композиции машин заключительная конфигурация T_2 превращается в стандартную начальную конфигурацию T_1.

Для всех конструктивных процедур можно строить реализующие их МТ. Сформулируем **тезис Тьюринга**: всякий алгоритм может быть реализован машиной Тьюринга.

Тезис Тьюринга доказать нельзя, поскольку само понятие алгоритма является неточным. Это не теорема и не постулат математической теории, а утверждение. Подтверждением тезиса Тьюринга служит, во-первых, математическая практика, а, во-вторых, то обстоятельство, что описание алгоритма в

терминах любой другой известной алгоритмической модели может быть сведено к его описанию в виде МТ.

1.3. Частично-рекурсивные функции и рекурсивность.

Исторически первой алгоритмической моделью были рекурсивные функции. Этот подход к формализации понятия алгоритма предложили в 1936 году Гёдель и Клини.

Пусть N обозначает множество натуральных чисел $\{0, 1, 2, \ldots\}$. Мы будем рассматривать функции с областью определения $D_f \subseteq N^k$ (k - целое положительное число) и с областью значений $R_f \subseteq N$. Такие функции будем называть ***k* - местными частичными**. Слово «частичная» говорит о том, что функция определена на подмножестве (в частном случае при функция будет всюду определенной). Назовем k - местную частичную функцию f **эффективно вычислимой** (или просто **вычислимой**), если существует алгоритм, который вычисляет f. Этот алгоритм должен удовлетворять следующим критериям:

1) Если на вход алгоритма поступил вектор $\boldsymbol{x} = \langle x_1, x_2, \ldots, x_k \rangle$ из D_f, то вычисление должно закончится после конечного числа шагов и выдать $f(x)$.
2) Если на вход алгоритма поступил вектор $\boldsymbol{x}$, не принадлежащий области определения D_f, то алгоритм никогда не заканчивается.

Основная идея в подходе к понятию алгоритма через частично-рекурсивные функции заключается в том, чтобы получить все вычислимые функции из существенно ограниченного множества базисных функций с помощью простейших алгоритмических средств.

Множество исходных функций:

- постоянная функция $0(\boldsymbol{x}) = 0$;
- одноместная функция следования $s(x) = x + 1$;
- функция проекции pr_i, $1 \le i \le k$, $pr_i(\boldsymbol{x}) = x_j$.

Нетривиальные вычислительные функции можно получить с помощью композиции (суперпозиции) уже имеющихся функций. Этот способ явно алгоритмический.

Определим **примитивную рекурсию**. При $n \geq 0$ из n–местной функции f и $(n+2)$–местной функции g строится $(n+1)$–местная функция h по следующей схеме:

$h(x_1, x_2, \ldots, x_n, 0) = f(x_1, x_2, \ldots, x_n)$,

$h(x_1, x_2, \ldots, x_n, y+1) = g(x_1, x_2, \ldots, x_n, y, h(x_1, x_2, \ldots, x_n, y))$.

При $n = 0$ получаем(a-константа):

$f(0) = a$;

$f(y+1) = g(y, f(y))$

Этот способ позволяет задать только всюду определенные функции. Частично-определенные функции порождаются с помощью третьего Гёделева механизма – **оператора минимизации**. Эта операция ставит в соответствие частичной функции частичную функцию, которая определяется так:

1) область определения

$D_h = \{<x_1, x_2, \ldots, x_k> \mid \exists\, x_{k+1} \geq 0,\ f(x_1, x_2, \ldots, x_{k+1}) = 0$ и

$<x_1, x_2, \ldots, x_k, y> \in D_f\ \forall\, y \leq x_{k+1}\}$;

2) $h(x_1, x_2, \ldots, x_k)$ = минимальному значению y, при котором

$f(x_1, x_2, \ldots, x_k, y) = 0$.

Оператор минимизации обозначается как

$h(x_1, x_2, \ldots, x_k) = \mu y[f(x_1, x_2, \ldots, x_k, y) = 0]$.

Все функции, которые можно получить из базисных функций за конечное число шагов с помощью композиции, примитивной рекурсии и оператора минимизации, называются **частично-рекурсивными**. Если функция получена без механизма минимизации, то она называется **примитивно-рекурсивной**. Если функция получается всюду определенная, то она называется **общерекурсивной.**

Рассмотрим примеры частично-рекурсивных функций.

Пример 1.3.1. Сложение двух чисел.

Sum: $\langle x, y\rangle \to x + y$.

Эта функция является общерекурсивной в силу примитивной рекурсии

$Sum(x, 0) = pr_1(x) = x$,

$Sum(x, y+1) = s\,(Sum(x, y)) = Sum(x, y) + 1$

Пример 1.3.2. Умножение двух чисел.

$Prod$: $\langle x, y\rangle \to x \cdot y$.

Будем использовать примитивную рекурсию

$Prod(x, 0) = 0(\boldsymbol{x}) = 0$,

$Prod(x, y+1) = Sum(Prod(x, y), x)$.

Пример 1.3.3. Факториал.

Действительно,

$0! = 1$,

$(y+1)! = Prod(y!, y+1)$

Используя функции, для которых уже установлено, что они являются частично-рекурсивными, можно получить все новые и новые частично-рекурсивные функции.

Вообще, используя μ-оператор можно получить частично-определенные функции из всюду определенных функций. А используя минимизацию с суперпозицией и рекурсией, можно построить больше функций, чем только с помощью суперпозиции и рекурсии.

Приведем пример общерекурсивной функции, для построения которой нельзя обойтись без минимизации - **функции Аккермана**:

$f(0, y) = y+1$,

$f(x+1, 0) = f(x, 1)$,

$f(x+1, y+1) = f(x, f(x+1, y))$.

1.4. Итеративные и рекурсивные алгоритмы

Рассмотрим один из алгоритмов сортировки элементов - алгоритм сортировки вставками, решающий задачу сортировки, формулируемую следующим образом:

Вход: последовательность $\langle a_1, a_2, \ldots, a_n \rangle$.

Выход: перестановки (изменение порядка) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ входной последовательности таким образом, что для ее членов выполняется соотношение $\langle a'_1 \le a'_2 \le \ldots \le a'_n \rangle$.

Рассматриваемый алгоритм будет являться итерационным, так как он содержит итерационные циклы. Отметим, что в итерационных алгоритмах необходимо обеспечить обязательное достижение условия выхода из цикла (сходимость итерационного процесса). В противном случае произойдет зацикливание алгоритма, т.е. не будет выполняться основное свойство алгоритма - **результативность.**

Для представления алгоритма будем использовать псевдокод. Псевдокод занимает промежуточное место между естественным и формальным языками. С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи

Псевдокод сортировки методом вставок INSERTION_SORT представлен ниже:

```
1 for j ← 2 to length [A]
2     do key ← A[j]
3       \ Вставка элемента A[j] в отсортированную
        \ последовательность A[1 .. j - 1]
4       i ← j – 1
5       while i > 0 and  A[i] > key
```

6 **do** A[i + 1] ← A[i]

7 i ← i - 1

8 A[i + 1] ← key

Свойство элементов массива «ИНВАРИАНТ ЦИКЛА» (loop invariant) для элементов массива, находящихся на позициях ***A*[1 .. j-1]:**

В начале каждой итерации цикла **FOR** из строк 1-8 подмассив A[1 .. j-1] содержит те же элементы, которые были в нем с самого начала, но расположенные в отсортированном порядке.

Свойства инварианта цикла:

1. **Инициализация.** Они справедливы перед первой инициализации цикла.

2. **Сохранение.** Если они истинны перед очередной итерацией цикла, то остаются истинными и после нее.

3. **Завершение.** По завершении цикла инварианты позволяют убедиться в правильности алгоритма.

Если выполняются первые два свойства, то инварианты цикла остаются истинными перед очередной итерацией цикла. С помощью инвариантов цикла доказывается корректность работы итерационных алгоритмов.

Действительно, рассмотрим доказательство корректности алгоритма на примере алгоритма сортировки вставками:

1. **Инициализация.** Перед первой итерацией j= 2^1. Подмножество элементов A[1.. j-1] состоит только из одного элемента, сохраняющего исходное значение, так как он является уже отсортированным. Таким образом, инвариант цикла выполняется перед первой итерацией.

2. **Сохранение.** В строках 4-7 в теле внешнего цикла происходит сдвиг элементов A[j-1], A[j-2], … на одну позицию вправо до тех пора, пока не освободится подходящее место для элемента A[j], куда он и помещается.

3. **Завершение.** Внешний цикл **for** завершается при j = n +1. Тогда в подмножестве A[1..n] элементов находятся те же элементы, которые были в

нем до начала работы алгоритма, но расположенные в отсортированном порядке. Получаем, что весь массив отсортирован, что и подтверждает корректность алгоритма.

В алгоритме сортировки вставками применяется **инкрементный подход:** располагая отсортированными массивом $A[1..j-1]$, мы помещаем очередной элемент $A[j]$ туда, где он должен находиться.

Многие полезные алгоритмы имеют **рекурсивную структуру**: для решения данной задачи они рекурсивно вызывают сами себя один или несколько раз, чтобы решить вспомогательную задачу, имеющую непосредственное отношение к поставленной задаче. Сначала алгоритм применяется к первой, малой части задачи, а затем полученный результат используется для поиска решения большей части задачи.

В некоторых случаях «часть задачи» содержит лишь на один элемент меньше, чем задача в целом. Такие задачи иногда называют «сокращай и властвуй». В тех случаях, когда «часть задачи» включает в себя половину входных данных всей задачи, алгоритм называют «разделяй и властвуй».

Для примера рассмотрим рекурсивный алгоритм «разделяй и властвуй» на примере сортировки слиянием, работающий по **методу декомпозиции** или **разбиения**: сложная задача разбивается на несколько более простых, которые подобны исходной задаче, но имеют меньший объем; далее эти вспомогательные задачи решаются рекурсивным методом, после чего полученные решения комбинируются с целью получения решения исходной задачи.

Парадигма, лежащая в основе этого метода, на каждом этапе рекурсии включает в себя три этапа:

1) (**Разделение**) Сложная задача разбивается на несколько простых задач, которые подобны исходной задаче, но имеют меньший объем.

2) (**Покорение**) Вспомогательные задачи решаются рекурсивным методом.

3) (**Комбинирование**) Полученные решения комбинируются с целью получения решения исходной задачи.

Вход (instance**)** задачи сортировки – последовательность, подлежащая сортировке.

Этапы работы алгоритма сортировки слиянием можно описать следующим образом:

1) сортируемая последовательность, состоящая из ***n*** элементов, разбивается на две меньшие последовательности, каждая из которых содержит $n/2$ элементов.

2) **Покорение:** сортировка обеих вспомогательных последовательностей методом слияния.

3) **Комбинирование:** слияние двух отсортированных последовательностей для получения окончательного результата.

Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1, так как такую последовательность можно считать упорядоченной.

Основная операция, которая производится в процессе сортировке по методу слияния – это объединение двух отсортированных последовательностей в ходе комбинирования (последний этап). Это делается с помощью вспомогательной процедуры MERGE(A, p, q, r), где A - массив, а p, q, r - индексы, нумерующие элементы массива, такие, что $p \le q < r$. В этой процедуре предполагается, что элементы подмассивов $A[\,p..q]$ и $A[\,q+1..\,r\,]$ упорядочены. Она сливает эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы подмассива $A[\,p..r]$.

Время работы этой процедуры прямо пропорционально n - размеру подмассива. Это условие записывают $T(n) = \Theta(n)$, где $n = r - p + 1$ - количество подлежащих слиянию элементов. Алгоритм процедуры MERGE(A, p, q, r) имеет вид:

MERGE(A,p,q,r) (3-ий этап)

1 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 \ Создаем массив $L[\,1..n_1 + 1]$ и $R[1..\,n_2 + 1]$
4 **for** $i \leftarrow 1$ **to** n_1

```
5            do L[i] ← A[p + i - 1]
6        for j ← 1 to n2
7            do R[j] ← A[q + j]
8        L[n1 + 1] ← ∞
9        R[n2 + 1] ← ∞
10   i ← 1
11   j ← 1
12   for k ← p to r
13       do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

Процедура работает следующим образом. Представим, что у нас уже есть два отсортированных по возрастанию подмассива. Эти два подмассива нужно объединить в один, который также будет отсортирован по возрастанию. Основной шаг состоит в том, чтобы из двух наименьших элементов рассматриваемых подмассивов выбрать самый маленький и извлечь его из соответствующего подмассива (при этом в данном подмассиве наименьшим станет уже новый элемент). Этот шаг повторяется до тех пор, пока в одном из уже отсортированных подмассивов не закончатся элементы, после чего оставшиеся в другом подмассиве элементы просто перемещаются в формируемый подмассив.

С вычислительной точки зрения выполнение каждого основного шага занимает одинаковые промежутки времени, так как все сводится к сравнению наименьших элементов подмассивов. Поскольку необходимо выполнить по крайней мере n основных шагов, то и получаем, что время работы процедуры слияния равно $\Theta(n)$. В рассматриваемом алгоритме используется прием, позволяющий не тратить время на проверку, является ли подмассив пустым. Для этого в каждый из подмассивов добавляется, так называемый, сигнальный элемент, что позволяет упростить код программы. Не существует элемента больше, чем сигнальный. Процесс продолжается до тех пор, пока проверяемые

элементы в обоих подмассивах не окажутся сигнальными. Как только это произойдет, это будет обозначать, что все несигнальные элементы уже помещены в формируемый массив.

Опишем работу процедуры MERGE(*A, p, q, r*). В строке 1 вычисляется длина n_1 подмассива *A*[*p..q*], а в строке 2 - длина n_2 подмассива *A*[*q*+1.. *r*]. Далее в строке 3 создаются массивы *L*(левый) и *R*(правый), длины которых равны $n_1 + 1$ и $n_2 + 1$ соответственно. В цикле **for** в строках 4 и 5 подмассив *A*[*p..q*] копируется в массив *L*[*1..*n_1], а в цикле в строках 6 и 7 подмассив *A*[*q*+1.. *r*] копируется в массив *L*[*1..*n_2]. В строках 8 и 9 последним элементом массивов *L* и *R* присваиваются сигнальные значения.

На рисунке 1.3 показано, что в результате копирования и добавления сигнальных элементов получаем массив *L* с последовательностью чисел <2, 4, 5, ∞> и массив *R* с последовательностью чисел<1, 2, 3, ∞> . Светло-серые ячейки массива *A* содержат конечные значения, а светло-серые ячейки массивов *L* и *R* – значения, которые еще только должны быть скопированы обратно в массив *A*. В светло-серых ячейках содержатся исходные значения из подмассива *A*[9..14] вместе с двумя сигнальными элементами. В темно-серых ячейках массива *A* содержатся значения, которые будут заменены другими, а в темно-серых ячейках массивов *L* и *R* – значения, уже скопированные обратно в массив *A*. В частях рисунка *а)-е)* показано состояние массивов *A, L , R*, а также соответствующие индексы *k, i, j* перед каждой итерацией цикла в строках 12-17. В части ж) показано состояние массивов и индексов по завершении работы алгоритма.

В строках 10-17 алгоритма, выполняется *r-p*+1 основных шагов, в ходе каждого из которых производятся манипуляции с инвариантом цикла. Перед каждой итерацией цикла **for** в строках 10-17, подмассив *A*[*p..k*-1] содержит наименьших элементов массивов *L*[1..$n_1 + 1$] и *R*[1.. $n_2 + 1$] в отсортированном порядке. Кроме того, элементы *L*[*i*] и *R*[*j*] являются наименьшими элементами массивов *L* и *R*, которые еще не скопированы в массив *A*. Этот инвариант

цикла соблюдается перед первой итерацией рассматриваемого цикла **for** и каждая итерация цикла не нарушает его.

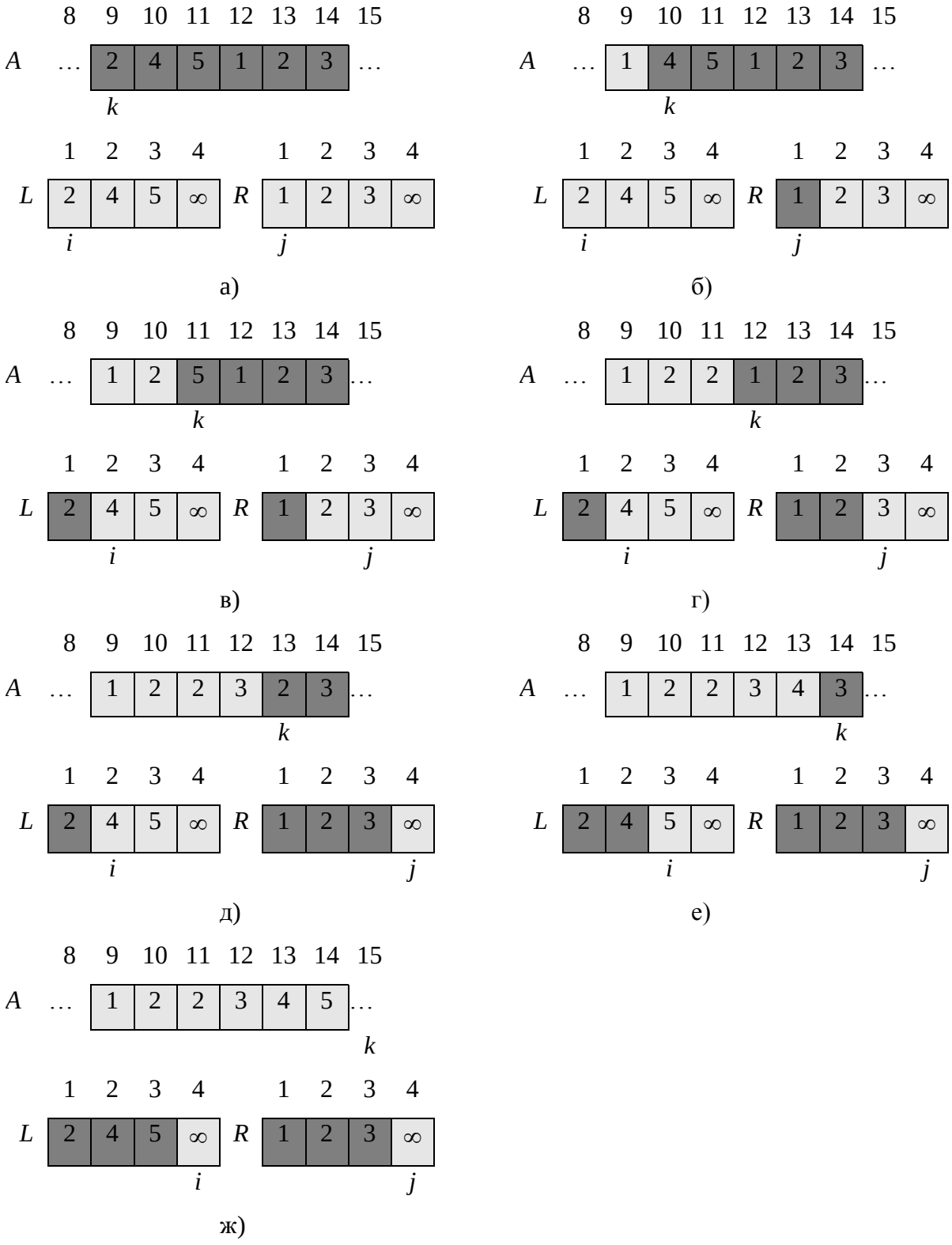


Рисунок 1.3 – Операции, выполняемые в строках 10-17 процедуры MERGE(*A*, *p*, *q*, *r*).

Покажем, что время процедуры MERGE(A, p, q, r) равно $\Theta(n)$, где $n = r - p + 1$. Заметим, что каждая из строк 1-3 и 8-11 выполняется в течение фиксированного времени; длительность циклов **for** в строках 4-7 равна $\Theta(n_1 + n_2) = \Theta(n)$, а в цикле **for** в строках 12-17 выполняется n итераций, на каждую из которых затрачивается фиксированное время.

Теперь процедуру MERGE можно использовать в качестве подпрограммы в алгоритме сортировки слиянием. Процедура MERGE_SORT(A, p, r), алгоритм которой представлен ниже, выполняет сортировку элементов в подмассиве $A[p..r]$. Если справедливо неравенство $p \geq r$, то в этом подмассиве содержится не более одного элемента, и, таким образом, он отсортирован. В противном случае производится разбиение, входе которого вычисляется индекс q, разделяющий массив $A[p..r]$ на два подмассива $A[p..q]$ с $\lceil n/2 \rceil$ элементами и $A[q..r]$ с $\lceil n/2 \rceil$ элементами. Чтобы отсортировать последовательность $A = \langle A[1], A[2], \ldots, A[n] \rangle$ вызывается процедура MERG_SORT(A,*1* , *length*[A]), где *length*[A] = n:

```
MERG_SORT(A,p,r)
    1  if  p < r
    2     then q ← ⌊ (p+r)/2⌋
    3  MERG_SORT(A, p, r)
    4  MERG_SORT(A, q + 1, r)
    5  MERGE (A, p, q, r)
```

На рисунке 1.4 проиллюстрирована работа этой процедуры в восходящем направлении. Если n – степень двойки, то в ходе работы алгоритма всегда происходит попарное объединение одноэлементных последовательностей в отсортированные последовательности длины 2, затем – попарное объединение двухэлементных последовательностей длины 4 и т.д., пока не будут получены последовательности, состоящие из $n/2$ элементов, которые объединяются в конечную отсортированную последовательность длины n.

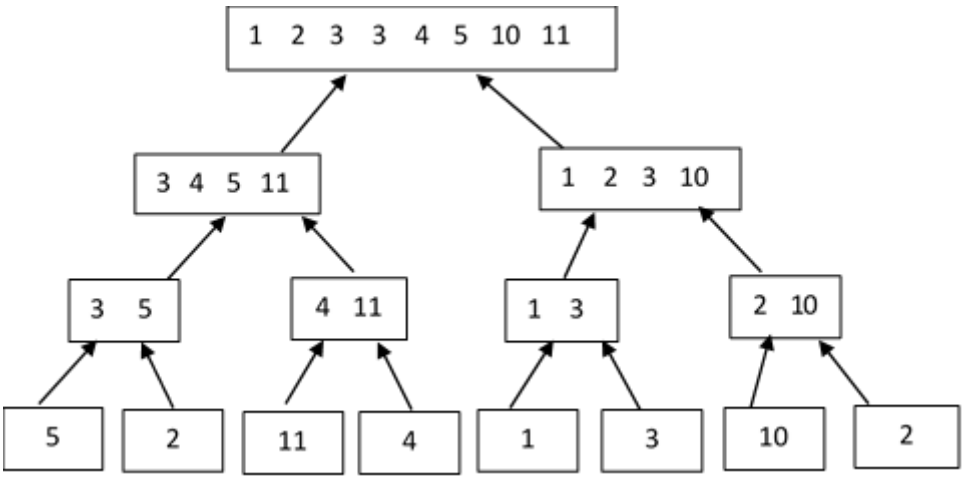


Рисунок 1.4 – Процесс сортировки массива с количеством элементов, являющимся степенью двойки, методом слияния.

2. АНАЛИЗ АЛГОРИТМОВ

Алгоритмы можно изучать и анализировать, не прибегая к использованию конкретного языка программирования или компьютерной платформы.

Цепь логических умозаключений об алгоритме невозможно построить без тщательного описания последовательности шагов, которые необходимо выполнить. Для этой цели наиболее часто употребляются, по отдельности или в совокупности, три формы преставления алгоритма: обычный язык, псевдокод и язык программирования. Выбор способа представления алгоритма зависит от личных предпочтений. Но никогда не надо забывать, что в основе любого алгоритма лежит идея. Если в описании алгоритма не просматривается явно ваша идея, значит, как писал Стивен Скиена, «вы используете для ее выражения нотацию низкого уровня».

2.1. Введение в анализ алгоритмов

Чтобы продемонстрировать правильность алгоритма, одного его описания недостаточно. Необходимо подробно описать задачу, которая подлежит решению. Постановка задачи состоит из двух частей: набора допустимых входных экземпляров и требований к выходу алгоритма. Невозможно доказать правильность алгоритма для нечетко поставленной задачи.

Каким образом можно показать, что алгоритм является неправильным? Самый лучший способ доказать неправильность алгоритма - найти **контрпример**, то есть такой экземпляр задачи, для которого алгоритм выдает неправильный ответ. Хороший контрпример должен обладать следующими свойствами:

- (**Проверяемость**) Чтобы продемонстрировать, что входной экземпляр является контрпримером, требуется вычислить ответ, который алгоритм выдаст для данного экземпляра, и предоставить лучший ответ, с тем, чтобы показать, что алгоритм не смог его найти;

- (**Простота**) Хороший контрпример не содержит ничего лишнего и ясно демонстрирует, почему именно данный алгоритм неправильный.

Развитие навыков поиска контрпримеров очень важен, при этом главную роль играет догадка, а не перебор всех вариантов. Искать контрпример можно, опираясь на следующие советы:

- Ищите мелкомасштабные решения.
- Рассмотрите все решения.
- Ищите слабое звено в алгоритме.
- Ищите ограничения.
- Рассматривайте крайние случаи.

Факт того, что для алгоритма не найден контрпример, вовсе не значит, что алгоритм правильный. Для этого требуется доказательство или демонстрация правильности. Для доказательства правильности алгоритма часто используется математическая индукция. В частности, метод математической индукции был применен нами ранее, для доказательства инварианта цикла в алгоритме сортировки вставками.

Пусть у нас имеется правильный алгоритм, то есть нам выдается правильный ответ. Однако, для нас не менее важны, такие параметры, как время работы алгоритма и объем требуемой памяти. Алгоритмы, разработанные для решения одной и той же задачи, часто очень сильно различаются по эффективности. Эти различия могут быть намного значительнее, чем те, что вызваны применением неодинакового аппаратного и программного обеспечения.

Например, на алгоритм сортировки вставками требует время, которое оценивается как c_1n^2, где n - количество сортируемых элементов, а c_1 – константа, не зависящая от n. Таким образом, время работы этого алгоритма пропорционально n^2. Для выполнения алгоритма сортировки слиянием требуется время, оцениваемое как c_2nlog_2n, где c_2 – некоторая другая константа, не зависящая от n. Обычно константа методом вставок меньше константы метода слияния $c_1 < c_2$. Для небольшого количества элементов на входе, сортировка включением работает быстрее, однако, если достаточно большое число, то становится заметным преимущество сортировки слиянием.

В рассматриваемой нами RAM – модели, мы можем подсчитать количество шагов, требуемых для исполнения конкретного экземпляра задачи. Но, чтобы понять насколько алгоритм хорош или плох для нас, нужно знать, как он работает со всеми экземплярами задач, то есть на всех входных данных.

Сложность алгоритма в наихудшем случае – это функция, определяемая максимальным количеством шагов, требуемых для обработки любого входного экземпляра размером n.

Сложность работы алгоритма в наилучшем случае – это функция, определяемая минимальным количеством шагов, требуемых для обработки любого входного экземпляра размером n.

Сложность работы алгоритма в среднем случае – это функция, определяемая количеством шагов, требуемых для обработки всех экземпляров размером n.

В практическом смысле наиболее важной является оценка сложности алгоритма в наихудшем случае.

2.2. Анализ итерационных и рекурсивных алгоритмов

Вернемся к рассмотрению итерационного алгоритма сортировки вставками. Время работы процедуры INSERTION_SORT зависит от набора входных значений. Напомним, что время работы алгоритма для того или иного ввода измеряется количеством элементарных операций, которые необходимо выполнить.

Будем полагать, что для выполнения каждой строки псевдокода требуется фиксированное время c_i. Обозначим через t_j количество проверок условия в цикле **while**. При нормальном завершении циклов **for** или **while** (то есть когда перестает выполняться условие, заданное в заголовке цикла) условие проверяется на один раз больше, чем выполняется тело цикла. Также очевидно, что комментарии не являются исполняемыми инструкциями, поэтому они не увеличивают время работы алгоритма.

Время работы алгоритма – это сумма промежутков времени, необходимых для выполнения каждой входящей в его состав исполняемой инструкции:

INSERTION_SORT(A)	время	количество раз
1 **for** $j \leftarrow 2$ **to** *length* [A]	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n - 1$
3 ⊿ Вставка элемента $A[j]$ в отсортированную последовательность $A[1 .. j - 1]$	0	$n - 1$
4 $i \leftarrow j - 1$	c_4	$n - 1$
5 **while** $i > 0$ **and** $A[i] >$ key	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i + 1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i + 1] \leftarrow key$	c_8	$n - 1$

Таким образом, получим:

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) +$

$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$

Время работы алгоритма в благоприятном случае (когда на вход подается отсортированный массив и $t_j = 1$) является линейной функцией:

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) =$

$= (c_1 + c_2 + c_4 + c_5 + c_8) - (c_2 + c_4 + c_5 + c_8).$

Время работы алгоритма в наихудшем случае (когда на вход подается массив в порядке, обратном требуемому, и $t_j = j$) является квадратичной функцией:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} +$$

$$+ c_8(n-1) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2}\right)n -$$

$$- (c_2 + c_4 + c_5 + c_8).$$

Для облегчения анализа процедуры INSERTION_SORT мы не учитывали не только фактическое время выполнения каждой инструкции, но и их абстрактные стоимости c_j. Это сделано потому, что нас интересует только главный член формулы, отражающий **скорость роста** или **порядок роста** функции. Наглядно относительный порядок функций представлен на рисунке 2.1.

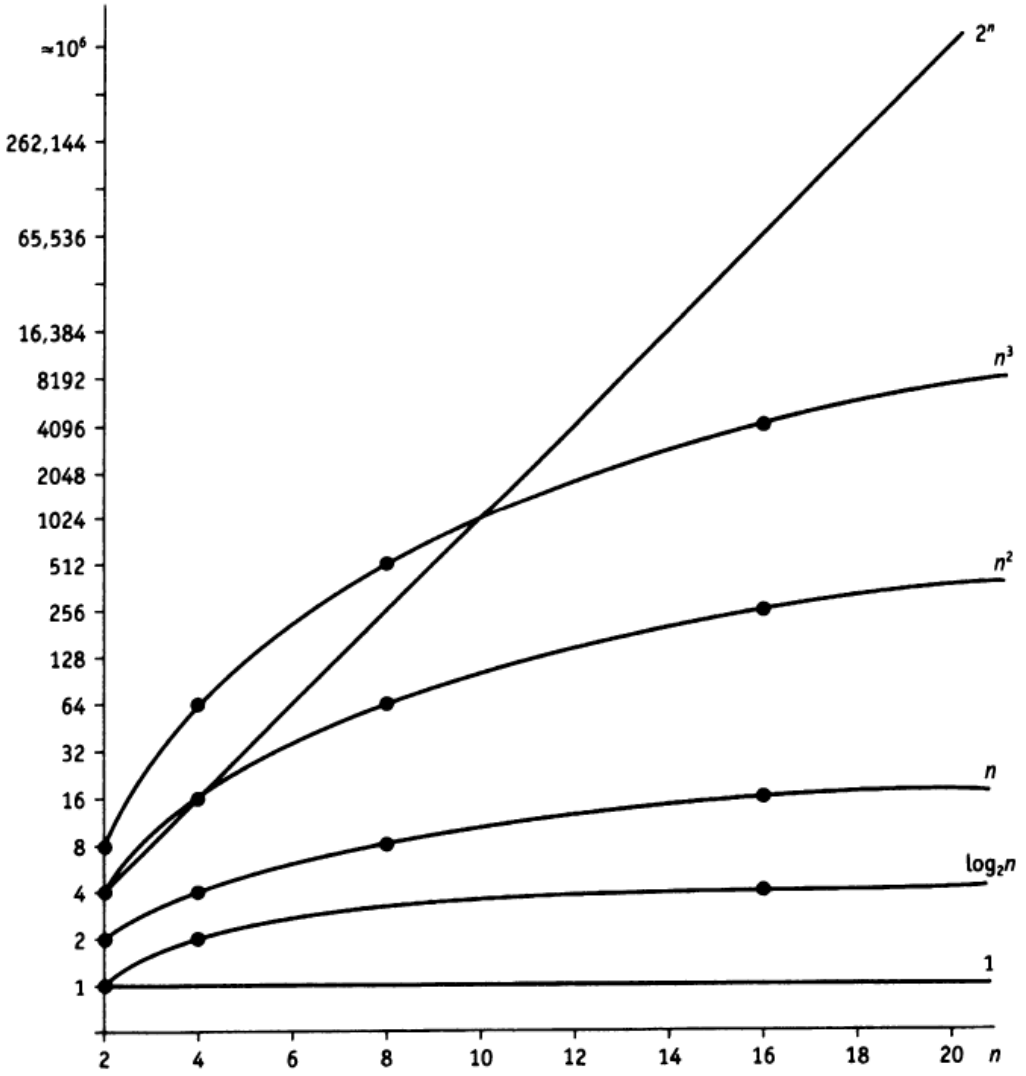


Рисунок 2.1 - Относительный порядок роста функций.

Постоянные множители при главном члене формулы нас также не интересуют, так как для оценки вычислительной сложности алгоритма при достаточно больших входных данных они также не важны. Таким образом, время работы алгоритма по методу вставок в наихудшем случае равно $\Theta(n^2)$.

Один алгоритм считается **эффективнее** другого, если время его работы в наихудшем случае имеет более низкий порядок роста.

Если алгоритм обращается к самому себе, время его работы может быть описано с помощью **рекуррентного уравнения** или **рекуррентного соотношения**, в котором полное время, требуемое для решения всей задачи с объемом ввода n, выражается через время решения вспомогательных задач. Затем, данное рекуррентное уравнение решается с помощью определенных математических методов, и устанавливаются границы производительности алгоритма.

Например, для алгоритма, основанного на методе «разделяй и властвуй», рекуррентное соотношение будет иметь вид:

$$T(n) = \begin{cases} \Theta(1), & \text{при } n \le c, \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{в противном случае} \end{cases} \qquad (2.1)$$

Если размер задачи достаточно мал ($n \le c$), где c – заранее известная константа, то задача решается непосредственно в течение определенного фиксированного времени, которое мы обозначаем через $\Theta(1)$. В случае, если наша задача делится на a подзадач, объем каждой из которых равен $1/b$ от объема исходной задачи и разбиение задачи на вспомогательные подзадачи происходит в течение времени $D(n)$, а объединение решений подзадач в решение исходной задачи - в течение времени $C(n)$, то мы получим рекуррентное уравнение, расположенное во второй строчке рекуррентного соотношения (2.1).

Для алгоритма сортировки по методу слияния рекуррентное соотношение (2.1) имеет вид:

$$T(n) = \begin{cases} \Theta(1), & \text{при } n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{при } n > 1. \end{cases}$$

2.3. Асимптотическая оценка рекуррентных соотношений

Рассматривая входные данные достаточно больших размеров для оценки только такой величины, как порядок роста времени работы алгоритма, мы фактически изучаем **асимптотическую** эффективность алгоритма.

Обозначения, используемые для асимптотического поведения времени работы алгоритма, используют функции, область определения которых - множество неотрицательных целых чисел N = {0, 1, 2, 3 …}.

Введем формальные определения, связанные с асимптотическими обозначениями.

Запись $T(n) = \Theta(n^2)$ обозначает, что

$$\exists\, c_1 ,\, c_2 > 0 \text{ и } \exists\, n_0 : c_1 \cdot n^2 \le T(n) \le c_2 \cdot n^2 \;\; \forall n \ge n_0$$

В общем случае, пусть $g(n)$ – некоторая функция. Тогда запись ***Θ(g(n))*** – обозначает множество функций $\Theta(g(n)) = f(n)$ таких что

$$\exists\, c_1 ,\, c_2 > 0 \text{ и } \exists\, n_0 : 0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \;\; \forall n \ge n_0$$

Функция *g(n)* называется ***асимптотически точной оценкой функции f(n).***

Запись $\Theta(g(n))$ читается как «тэта большое от жэ от эн».

Пусть $g(n)$ – некоторая функция, тогда запись ***O(g(n))*** –обозначает множество функций $O(g(n)) = f(n)$ таких что

$$\exists\, c > 0 \text{ и } \exists\, n_0 : \quad 0 \le f(n) \le c \cdot g(n) \;\; \forall n \ge n_0$$

Функция *g(n)* называется **асимптотической верхней границей функции *f(n)*.**

Запись $O(g(n))$ читается как «о большое от жэ от эн».

Пусть $g(n)$ – некоторая функция. Тогда запись ***Ω (g(n))*** –обозначает множество функций $\Omega(g(n)) = f(n)$ таких что

$$\exists\, \boldsymbol{c} > \mathbf{0} \text{ и } \exists\, \boldsymbol{n_0} : 0 \le c \cdot g(n) \le f(n) \;\; \forall n \ge n_0$$

Функция g(n) называется ***асимптотической нижней границей функции f(n).*** Запись $\Omega(g(n))$ читается как «омега большое от жэ от эн».

Отметим некоторые нюансы по асимптотическим обозначениям в уравнениях (аналогично и для неравенств):

1) Если $2n^2 + 3n + 1 = 2n^2 + \Theta(n) \;\Rightarrow\; 2n^2 + 3n + 1 = 2n^2 + f(n)$,

где $f(n) \in \Theta(n)$. В нашем случае $f(n) = 3n + 1$.

2) Если $2n^2 + \Theta(n) = \Theta(n^2)$, то при любом выборе безымянных функций, подставляемых вместо асимптотических обозначений в левую часть уравнения, можно выбрать и подставить в правую часть такие безымянные функции, что уравнение будет правильным. Таким образом,

$$\forall f(n) \in \Theta(n) \; \exists g(n) \in \Theta(n^2): 2n^2 + f(n) = g(n) \; \forall n.$$

3) Возможно также несколько соотношений объединить в цепочку:

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2).$$

Согласно первому уравнению,

$$\exists f(n) \in \Theta(n) \; \forall n : n^2 + 3n + 1 = 2n^2 + f(n).$$

Согласно второму уравнению,

$$\forall g(n) \in \Theta(n) \; \exists h(n) \in \Theta(n^2) \; \forall n: 2n^2 + g(n) = h(n).$$

Заметим, что в этом случае также $2n^2 + 3n + 1 = \Theta(n^2)$.

Как мы уже отмечали, рекуррентное соотношение – это уравнение или неравенство, описывающее функцию с использованием ее самой, но только с меньшими аргументами.

Рассмотрим три метода решения рекуррентных уравнений, то есть получения асимптотических Θ-оценок и *O*-оценок решения.

Первый метод - **метод подстановок -** заключается в том**,** что мы догадываемся, какой вид имеют граничные функции, а затем, с помощью метода математической индукции доказываем, что догадка была правильной.

Второй метод - **метод деревьев рекурсии** – рекуррентное соотношение преобразуется в дерево, узлы которого представляют время выполнения каждого уровня рекурсии; затем для решения соотношения используется метод оценок сумм.

Третий метод – **основной метод** – граничные оценки решений рекуррентных соотношений представляются в виде:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

где $a \geq 1$, $b > 1$, а функция $f(n)$ - это заданная функция.

Рассмотрим подробнее каждый из методов.

Метод подстановки, применяемый для решения рекуррентных уравнений, состоит из двух этапов:

1) делается догадка о виде решения;
2) с помощью метода математической индукции определяются константы и доказывается, что решение правильное.

Метод постановки можно применять для определения либо верхней, либо нижней границ рекуррентного соотношения. В качестве примера определим верхнюю границу рекуррентного соотношения

$$T(n) = 2T\left(\lfloor \frac{n}{2} \rfloor\right) + n.$$

Предположим, что наше решение имеет вид $T(n) = O(n\log_2 n)$. Метод заключается в том, что при подходящем выборе константы $c > 0$ выполняется неравенство $T(n) \le c \cdot n \cdot \log_2 n$. Предположим справедливость этого неравенства для величины $\lfloor n/2 \rfloor$, то есть что выполняется соотношение

$$T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \cdot \log_2(\lfloor n/2 \rfloor).$$

После подстановки данного выражения в рекуррентное соотношение получаем:

$$T(n) \le 2(c\lfloor n/2 \rfloor \cdot \log_2(\lfloor n/2 \rfloor)) + n \le cn \cdot \log_2(n/2) + n =$$

$$= cn \cdot \log_2 n - cn \cdot \log_2 2 + n = cn \cdot \log_2 n - cn + n \le cn \cdot \log_2 n,$$

где последнее неравенство выполняется при $c \ge 1$.

По методу математической индукции, при $n_0 = 2$ базой индукции будет $T(2)$ и $T(3)$. Далее, из нашего рекуррентного соотношения находим, что $T(2) = 4$, а $T(3) = 5$. Тогда $T(2) \le 2c \log_2 2$ и $T(3) \le 3c \log_2 3$. Решая неравенства, получаем, что достаточно выбрать $c \ge 2$.

К сожалению, не существует общего способа, позволяющего угадать правильное решение рекуррентного соотношения. Считается, что для этого требуется опыт, удача и творческое мышление. Поэтому сделать хорошую догадку зачастую довольно трудно.

Построение дерева рекурсии – наилучший путь, чтобы сделать догадку о виде решения, которая затем проверяется методом подстановок. Метод деревьев рекурсии широко применяется при рассмотрении рекуррентных соотношений, описывающих время работы алгоритмов, построенных по принципу «разделяй и властвуй».

Например, найдем с помощью метода деревьев рекурсии решение рекуррентного соотношения

$$T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \Theta(n^2).$$

Оценим решение сверху. Для этого заменим наше рекуррентное соотношение равносильным для нашего исследования:

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2, \text{ где } c > 0.$$

Для удобства предположим, что n – степень четверки.

На рисунке 2.2*а)* показана функция $T(n)$, которая на рисунках 2.2б) и 2.2в), расписывается в эквивалентное дерево рекурсии, представляющее анализируемое рекуррентное соотношение.

В корне дерева ставим выражение cn^2, являющееся временем верхнего уровня рекурсии, а три поддерева, берущих начало из корня, - это время выполнения подзадач размера $n/4$ (рисунок 2.2б).

На рисунке 2.3 показано дерево рекурсии, которое мы построим, уменьшая размер подзадач и дойдя до граничных условий.

Определим количество уровней, которое нам надо построить, чтобы достичь граничных условий. Размер вспомогательной задачи i –ом уровне глубины равен $n/4^i$. Таким образом размер задачи становится равным единице, когда $n/4^i = 1$, то есть $i = \log_4 n$. Получаем, что с учетом корня, в дереве всего $\log_4 n + 1$ уровней.

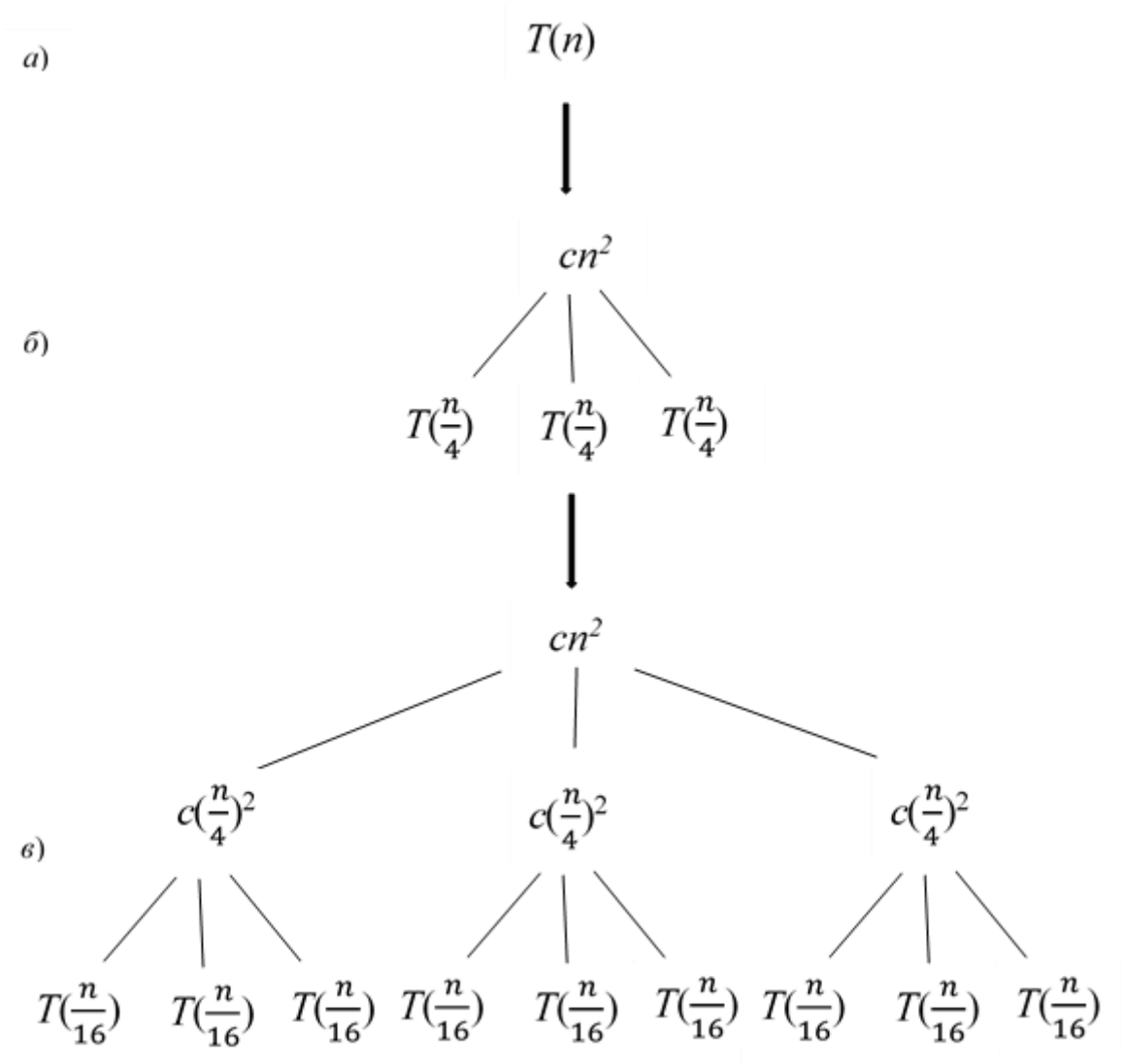


Рисунок 2.2 – Процесс построения дерева рекурсии для рекуррентного соотношения $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$.

Теперь определим время выполнения для каждого уровня дерева. На каждом уровне в три раза больше узлов, чем на предыдущем уровне, поэтому количество узлов на i –ом уровне равно 3^i. Поскольку размеры вспомогательных подзадач при спуске на один уровень уменьшаются в четыре раза, время выполнения каждого узла на i –ом уровне равно $c(n/4^i)^2$. Суммарное время выполнения вспомогательных подзадач на i –ом уровне равно

$$3^i\, c(n/4^i)^2 = (3/16)^i cn^2.$$

Последний уровень, находящийся на глубине $\log_4 n$, состоит из $3^{\log_4 n} = n^{\log_4 3}$ узлов, каждый из которых дает в общее время работы вклад, равный

$T(1)$. Поэтому время работы этого уровня равно величине $n^{\log_4 3}T(1)$, которая имеет оценку $\Theta(n^{\log_4 3})$.

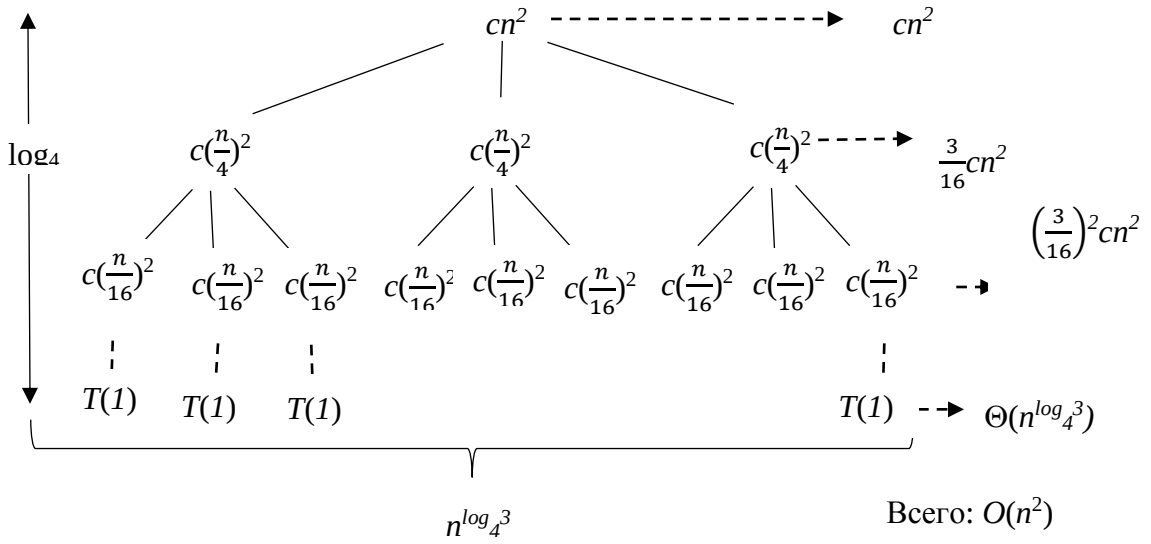


Рисунок 2.3 – Построение дерева рекурсии для рекуррентного соотношения $T(n) = 3T\left(\frac{n}{4}\right) + cn^2$.

Просуммируем времена работы всех уровней дерева и определим время работы алгоритма целиком:

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) =$$

$$= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) = \frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\frac{3}{16} - 1} cn^2 + \Theta(n^{\log_4 3})$$

Произведем асимптотическую оценку нашей формулы, используя в качестве верхней границы бесконечно убывающую геометрическую прогрессию:

$$\sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) =$$

$$= \frac{1}{1 - \frac{3}{16}} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) = O(n^2).$$

Из полученного выражения видно, что полное время работы дерева определяется временем работы его корня. Итак, получаем предположительно

решение $T(n) = O(n^2)$ –верхняя граница. Эта граница является асимптотической точной оценкой. Действительно, первый рекурсивный вызов дает вклад в общее время работы алгоритма, которое выражается как $\Theta(n^2)$, поэтому нижняя граница решения рекуррентного соотношения представляет собой $\Omega(n^2)$.

С помощью метода подстановок проверим корректность нашей догадки, то есть убедимся, что $T(n) = O(n^2)$ – верхняя граница рекуррентного соотношения $T(n) = 3T\left(\lfloor \frac{n}{4} \rfloor\right) + cn^2$. Для этого покажем, что для некоторой константы $d > 0$ выполняется неравенство $T(n) \le dn^2$. Для с > 0:

$T(n) \le 3T\left(\lfloor \frac{n}{4} \rfloor\right) + cn^2 \le 3d\lfloor n/4 \rfloor^2 + cn^2 \le 3d(n/4)^2 + cn^2 =$

$= (3/16)\, dn^2 + cn^2 \le dn^2.$

Последнее неравенство выполняется при $d \ge (16/13)c$.

Основной метод базируется на приведенной ниже теореме:

Теорема (Основная теорема).

Пусть $a \ge 1$ и $b > 1$ - константы, а $f(n)$ - произвольная функция, а $T(n)$-функция, определенная на множестве неотрицательных целых чисел с помощью рекуррентного соотношения

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

где выражение $\frac{n}{b}$ является целым и интерпретируется либо как верхняя границы отношения $\lceil \frac{n}{4} \rceil$, либо как нижняя граница отношения $\lfloor \frac{n}{4} \rfloor$. Тогда асимптотическое поведение функции можно выразить следующим образом:

1. Если $f(n) = O(n^{\log_b a - \varepsilon})$ для некоторой константы $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$.
2. Если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a}\log_2 n)$.
3. Если $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для некоторой константы $\varepsilon > 0$, и если $a \cdot f(n/b) \le c \cdot f(n)$ для некоторой константы $c < 1$ и всех достаточно больших n, то $T(n) = \Theta(f(n))$.

Обратим внимание, что в первом пункте функция $f(n)$ должна быть не просто меньше функции $n^{\log_b a}$, а полиномиально меньше. Аналогично, в третьем пункте теоремы функция $f(n)$ должна быть не просто больше функции $n^{\log_b a}$, а полиномиально больше.

Пример 2.3.1. Найти асимптотическую оценку для рекуррентного соотношения $T(n) = 9T\left(\frac{n}{3}\right) + n$.

В этом случае $a = 9$, $b = 3$, $f(n) = n$, так что $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$.

Поскольку $f(n) = O(n^{\log_3 9 - \varepsilon})$, где $\varepsilon = 1$, то можно применить первый пункт основной теоремы и сделать вывод, что решение - $T(n) = \Theta(n^2)$.

Пример 2.3.2. Найти асимптотическую оценку для рекуррентного соотношения $T(n) = T\left(\frac{2n}{3}\right) + 1$.

В этом случае $a = 1$, $b = 3/2$, $f(n) = 1$, так что $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

Поскольку $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, то можно применить второй пункт основной теоремы и сделать вывод, что решение - $T(n) = \Theta(\log_2 n)$.

Пример 2.3.3. Найти асимптотическую оценку для рекуррентного соотношения $T(n) = 3T\left(\frac{n}{4}\right) + n\log_2 n$.

В этом случае $a = 3$, $b = 4$, $f(n) = n\log_2 n$, поэтому $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$.

Так как Если $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$, где $\varepsilon \approx 0{,}2$, то применяем третий пункт основной теоремы. Для этого докажем условие регулярности для функции $f(n)$:

$a \cdot f(n/b) = 3(n/4)\log_2(n/4) \leq (3/4)\, n\log_2 n = c \cdot f(n)$, при $c = ¾$ и достаточно больших n. Следовательно, согласно пункту 3, решение этого рекуррентного соотношения - $T(n) = \Theta(n\log_2 n)$.

Пример 2.3.4. Найти асимптотическую оценку для рекуррентного соотношения $T(n) = 2T\left(\frac{n}{2}\right) + n\log_2 n$.

В этом случае основной метод неприменим.

У нас $a = 2$, $b = 2$, а $f(n) = n\log_2 n$, и $n^{\log_b a} = n$. Пункт 3 теоремы мы не можем применить, так как функция $f(n) = n\log_2 n$ не полиномиально больше, чем $n^{\log_b a}$. Отношение $f(n)/ n^{\log_b a} = (n\log_2 n)/ n = \log_2 n$ асимптотически меньше функции n^{ε} для любой константы $\varepsilon > 0$. Следовательно рассматриваемое нами рекуррентное соотношение находится «в промежуточном состоянии» между пунктами 2 и 3 основной теоремы.

Объясним три случая, рассматриваемых в основной теореме с помощью дерева рекурсии. Рассмотрим представленное на рисунке 2.4 рекурсивное дерево для алгоритма «разделяй и властвуй», выраженного рекуррентным соотношением $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

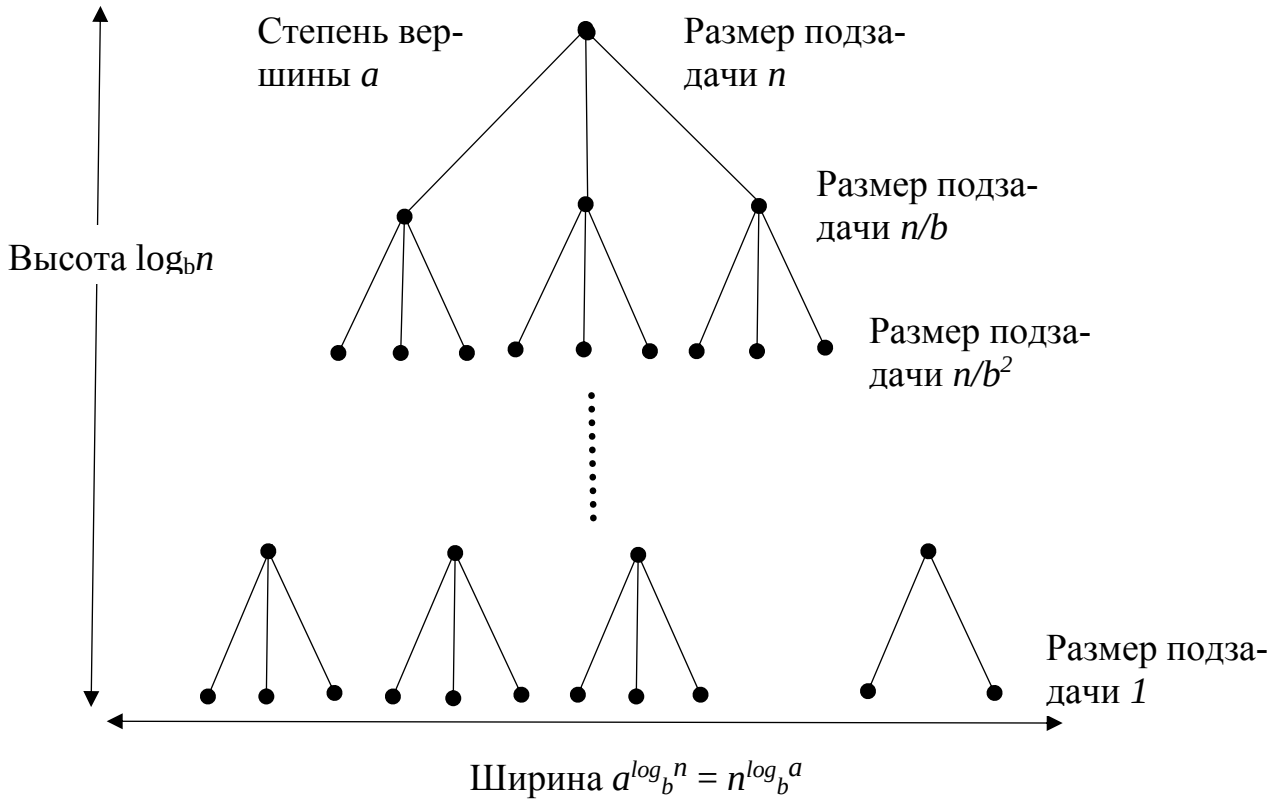


Рисунок 2.4 – Рекурсивное дерево, соответствующее рекуррентному соотношению $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

Как мы уже знаем, задача из n элементов разделяется на a подзадач размером n/b элементов. Каждая подзадача размером в k элементов выполняется за время $O(f(k))$. Общее время будет равно сумме временных затрат на выполнение подзадач, сложенной с временными расходами на создание рекурсивного дерева. Высота этого дерева равна $h = \log_b a$, а количество листьев равно $a^h = a^{\log_b n} = n^{\log_b a}$.

Тогда три случая основной теоремы соответствуют трем разным ситуациям, которые могут возникнуть в зависимости от величин a, b, и $f(n)$:

- Слишком много листьев. Если количество листьев превышает сумму затрат на внутреннюю обработку, то общее время будет равно $\Theta(n^{\log_b a})$.

- Одинаковый объем работы на каждом уровне. По мере прохождения вниз по дереву размер каждой задачи уменьшается, но количество задач, подлежащих решению увеличивается. Если сумма затрат на внутреннюю обработку одинакова для каждого уровня, то общее время исполнения вычисляется умножением затрат на каждом уровне $n^{\log_b n}$ на количество уровней $\log_b n$,и будет равно $\Theta(n^{\log_b a}\log_2 n)$.

- Слишком большое время обработки корня. Если с возрастанием размеров задачи затраты на внутреннюю обработку возрастают быстрыми темпами, то будут доминировать затраты на обработку корня. В таком случае время исполнения алгоритма будет равно $\Theta(f(n))$.

3. ТЕОРЕТИКО – ЧИСЛОВЫЕ АЛГОРИТМЫ

Раньше теория чисел рассматривалась как элегантная, но почти бесполезная область чистой математики. В наши дни теоретико-числовые алгоритмы нашли широкое применение. В определенной степени это произошло благодаря изобретению криптографических схем, основанных на больших простых числах.

3.1. Алгоритм Евклида.

Одним из самых старых и концептуальных алгоритмов теории чисел является алгоритм Евклида, предназначенный для вычисления наибольшего общего делителя двух целых чисел. Этот алгоритм впервые встречается в трактате «Начала» греческого математика Евклида, опубликованном в 300 году до н. э., в седьмой книге в теоремах 1 и 2.

Дадим основные определения теории чисел, касающиеся делимости.

Пусть N = {0, 1, 2, 3 …} – множество натуральных чисел. Пусть $a \in Z$ и $d \in Z$. Тогда $d \mid a$ обозначает, что " d делит a" или " d делится на a" т.е

$\exists k \in Z : a = kd$. a - **кратное** (multiple) d.

Если a не делится на d, то пишут

$$d \nmid a.$$

Число 0 делится на все целые числа.

Если $a > 0$ и $d \mid a$, то $|d| \le |a|$. Если $d \mid a$ и $d \ge 0$, то говорят, что d – **делитель** (devisor) числа a. 1 и a - **тривиальные делители** (trivial devisor). Остальные делители числа a - **нетривиальные делители, множители** (factors).

***a* - простое** (prime) **число,** если $a \in Z$ и $a > 0$, и единственными делителями для a являются тривиальные делители 1 и a.

***a* – составное** (composite) **число**, если $a \in Z$ и $a > 0$, и a не является простым числом.

Целое число **1** называется **единицей** (оно не является ни простым, ни составным).

Пример простых чисел:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, \ldots .$$

Наибольший общий делитель (**g**reatest **c**ommon **d**ivisor) двух целых чисел ***a*** и ***b***, отличных от нуля – это самый большой из общих делителей чисел ***a*** и ***b***. Обозначается, **НОД**(***a***, ***b***) или **gcd**(***a***, ***b***).

Доопределим, что **gcd**(***0***, ***0***) = 0.

Если $\boldsymbol{a} \neq 0$ и $\boldsymbol{b} \neq 0$, то **gcd**($\boldsymbol{a}$, $\boldsymbol{b}$) = min(1, ($|\boldsymbol{a}|, |\boldsymbol{b}|$)).

gcd*(a,b)* = gcd*(a,0)* = *a*.

Числа a и b называются взаимно простыми (relatively prime), если gcd(*a*,*b*) = 1.

Числа n_1, n_2, n_3,… n_k *называются* **попарно взаимно простыми** (pairwise relatively prime), если при $i \neq j$ выполняется соотношение

$$\gcd(n_i, n_j) = 1.$$

Если $p \mid ab$ *для* $\forall p$ – простого числа *и* $\forall a,b \in Z$, то $p \mid a$ или $p \mid b$.

Теорема (рекурсивная теорема о НОД).

Для $\forall$ ($a \geq 0$ и $a \in$ Z) и $\forall$ ($b > 0$ и $b \in$ Z): gcd(*a*, *b*) = gcd(*a*, *a* mod *b*).

На рекурсивной теореме о НОД основан рекурсивный алгоритм Евклида:

```
Euclid (a,b)
1 if b = 0
2   then return a
3   else return Euclid(b,a mod b)
```

В качестве входных данных в алгоритме выступают любые неотрицательные целые числа.

Пример работы процедуры Euclid для вычисления gcd (30,21):

Euclid(30,21) = Euclid(21,9) = Euclid(9,3) = Euclid(3,0) = 3.

Корректность процедуры Euclid следует из теоремы рекурсивной теоремы о НОД. Так как во второй строке алгоритма возвращается значение *a*, то *b* = 0, а из элементарного свойства НОД следует, что gcd*(a,b)* = gcd*(a,0)* = *a*.

Время работы алгоритма Euclid при размере входных данных *a* и *b* можно оценить, используя теорему Ламе.

Теорема (Теорема Ламе). Если для произвольного числа $k \geq 1$ выполняются условия $a > b \geq 0$ и $b < F_{k+1}$, то в вызове процедуры Euclid (*a*,*b*) производится менее *k* рекурсивных вызовов.

Поскольку число равно $\varphi^k/\sqrt{5}$, где – золотое сечение $(1+\sqrt{5})/2$, то количество рекурсивных вызовов в процедуре Euclid (a,b) равно $O(\log_2 b)$.

Таким образом, если с помощью алгоритма Евклида обрабатывается два β – битовых числа, то в нем производится $O(\beta)$ арифметических операций и $O(\beta^3)$ битовых операций (в предположении, что при умножении и делении β – битовых чисел выполняется $O(\beta^2)$ битовых операций).

Рассмотрим расширенный алгоритм Евклида.

Пусть выполняется $d = gcd(a,b) = ax + by$.

Построим алгоритм для вычисления НОД и нахождения целых коэффициентов *x и y:*

Extended-Euclid(a,b)

1 **if** b = 0

2 **then return** (a,1,0)

3 $(d',x',y') \leftarrow$ Extended-Euclid $(b, a \bmod b)$

4 $(d,x,y) \leftarrow (d',x',y' - \lfloor a/b \rfloor y')$

5 **return** (d,x,y)

В таблице представлена работа расширенного алгоритма Евклида:

Работа алгоритма Extended-Euclid(a,b) с входными числами 99 и 78					
a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	-	3	1	0

В каждой строке таблицы показан один уровень рекурсии: входные величины a и b, вычисленная величина $\lfloor a/b \rfloor$, а также возвращаемые величины d,

x и y. Возвращаемая тройка значений (d,x,y) становится тройкой, которая используется в ходе вычислений на следующем, более высоком уровне рекурсии.

Процедура Extended-Euclid(a,b) представляет собой разновидность процедуры Euclid(a,b). Первая строка в ней эквивалентна проверке равенства значения b нулю в первой строке процедуры Euclid. Если $b = 0$, то процедура Extended-Euclid во второй строке возвращает не только значение $d = a$, но и коэффициенты $x = 1$ и $y = 0$, так что $a = ax + by$. Если $b \neq 0$, то процедура Extended-Euclid сначала вычисляет набор величин (d',x',y'), таких что

$$d' = \gcd(b, a \bmod b) \text{ и}$$

$$d' = bx' + (a \bmod b)y'. \qquad (3.1)$$

Как и в процедуре Euclid, в этом случае имеем

$$d = \gcd(a, b) = d' = \gcd(b, a \bmod b).$$

Чтобы получить значения x и y, для которых выполняется равенство $d = gcd(a,b) = ax + by$, сначала перепишем уравнение (3.1) с использованием равенств $d = d'$ и $\gcd(a, b) = \gcd(|a|, |b|)$:

$$d = bx' + (a - \lfloor a/b \rfloor b)y' = ay' + b(x' - \lfloor a/b \rfloor y').$$

Таким образом, при выборе величин $x = y'$ и $y' = x' - \lfloor a/b \rfloor y'$ удовлетворяется уравнение $d = ax + by$, что доказывает корректность алгоритма в процедуре Extended-Euclid.

Поскольку количество вызовов в процедуре Euclid равно количеству вызовов в процедуре Extended-Euclid, то время работы процедуры Euclid с точностью до постоянного множителя равно времени работы процедуры Extended-Euclid. Таким образом, $a > b > 0$ количество рекурсивных вызовов равно $O(\log_2 b)$.

3.2 Алгоритмы проверки простоты числа.

Рассмотрим задачу поиска больших простых чисел. Данная задача возникает по многих приложениях, например, в криптографии, когда необходимо найти «случайное» простое число.

В начале обсудим плотность распределения простых чисел на числовой оси, после чего перейдем к исследованию нескольких алгоритмов, осуществляющих проверку простоты числа.

Функция распределения простых чисел $\pi(n)$ - количество простых чисел, не превышающих числа n.

Пример 3.2.1. $\pi(11) = 5$. Это числа 2, 3, 5, 7, 11.

Теорема (теорема о простых числах):

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1.$$

То есть приближенная оценка $n/\ln n$ дает достаточно точную оценку функции $\pi(n)$.

Пример 3.2.2. При $n = 10^9$, когда $\pi(n) = 50\,847\,534$, а $n/\ln n \approx 48\,254\,942$, отклонение не превышает 6%.

Вероятность того, что случайным образом выбранное число n окажется простым можно оценить как $1/\ln n$. То есть, чтобы найти простое число, длина которого совпадает с длиной числа n, понадобится проверить $\ln n$ целых чисел, выбрав их случайным образом в окрестности числа n.

Пример 3.2.3. Чтобы найти простое 512 – битовое число, потребуется перебрать $\ln 2^{512} \approx 355$ случайным образом выбранных 512-битовых чисел, проверяя их простоту.

В задаче проверки числа на простоту понятие «большой вход» обозначает ввод, содержащий «большие целые числа», а не ввод, содержащий «боль-

шое» количество целых чисел. Таким образом, размер входных данных определяется количеством битов, необходимых для представления этих данных, а не просто количеством чисел в них.

Время работы алгоритма, на вход которого подаются целые числа a_1, a_2, a_3, …, a_k является **полиномиальным**, если он выполняется за время, которое выражается полиномиальной функцией от величин $\log_2(a_1)$, $\log_2(a_2)$, … $\log_2(a_k)$, то есть если это время является полиномиальным по длине входных величин в бинарном представлении.

Одним из элементарных подходов к задаче проверки на простоту является **пробное деление.** Этот подход заключается в следующем: число *n* делим нацело на все целые числа 2, 3, … $\lfloor\sqrt{n}\rfloor$. (Пропускаем также все четные числа большие 2). Число *n* простое тогда и только тогда, когда оно не делится ни на один из пробных множителей.

Если для обработки каждого пробного делителя требуется фиксированное время, то в худшем случае время решения задачи при таком подходе будет равно $\Theta(\sqrt{n})$. Если бинарное представление числа *n* имеет длину β, то $\beta = \lceil\log_2(n + 1)\rceil$, поэтому $\sqrt{n} = \Theta(2^{\beta/2})$. Таким образом, пробное деление хорошо работает только при условии, что число очень мало, или окажется, что у него есть маленький простой делитель. Преимущество этого метода заключается в том, что он не только позволяет определить, является ли число просты, но и находит один из его простых делителей, если оно составное.

Введем еще несколько понятий, опираясь на уже известное нам из курса математической логики понятие группы.

Если группа $(S, \oplus)$ обладает свойством коммутативности, то такая группа называется **абелевой**, то есть

$$a \oplus b = b \oplus a \quad для \quad \forall\, a,b \in S.$$

Если группа $(S, \oplus)$ удовлетворяет условию $|S| < \infty$, то она называется **конечной.**

Операции сложении, умножения, вычитания **по модулю n**:

$[a]_n +_n [b]_n = [a + b]_n$

$[a]_n \cdot_n [b]_n = [a \cdot b]_n$

$[a]_n - {}_n[b]_n = [a - b]_n$

$(Z_n, +_n)$ - **аддитивная группа по модулю n**.

$|\ Z_n\ | = n$.

$(Z_n^*, \cdot_n)$ - **мультипликативная группа по модулю n:**

$Z_n^* = \{[a]_n \in Z_n : \gcd(a,n) = 1\}$.

Так как для $0 < a \le n$ *и* $\forall k \in Z$: $a \equiv (a + kn) \pmod n$, *то из* $\gcd(a,n) = 1$ следует, что $\gcd(a + kn, n) = 1$.

Пример 3.2.4. $Z_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

Пример 3.2.5. $8 \cdot 11 \equiv 13 \pmod{15}$

Запись $ax \equiv b \pmod n$ эквивалентна $[a]_n \cdot_n [x]_n = [b]_n$

Элемент, **обратный** (относительно умножения) к элементу a, обозначим как $(a^{-1} \bmod n)$.

Операция деления в Z_n^*: $a/b \equiv ab^{-1} \bmod n$.

Пример 3.2.6. В множестве Z_{15} $7^{-1} = 13 \pmod{15}$, т.к. $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, поэтому $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

Рассмотрим метод проверки простых чисел, который вполне пригоден для многих практических приложений – **проверку чисел на псевдопростоту**.

Обозначим через Z_n^+ ненулевые элементы множества Z_n :

$$Z_n^+ = \{1, 2, \ldots, n - 1\}.$$

Если число простое, то $Z_n^+ = Z_n^*$.

Назовем число n псевдопростым (base-a pseudoprime) по основанию a, если оно составное и выполняется утверждение малой теоремы Ферма:

$$a^{n-1} \equiv 1 \pmod n. \qquad (3)$$

Теорема (малая теорема Ферма): Если p – простое число, то для всех $a \in Z_n^*$: $a^{p-1} \equiv 1 \pmod p$.

Таким образом, если удается найти какой-нибудь элемент $a \in Z_n^+$, при котором n не удовлетворяет уравнению (3) , то n - определенно составное. Почти всегда справедливо и обратное утверждение, поэтому это утверждение является почти идеальным тестом на простоту. Мы проверяем, удовлетворяет ли число n уравнению (3) при $a = 2$. Если это не так, то делается вывод, что оно составное. В противном случае можно выдвинуть гипотезу, что n - простое.

Ниже приведен псевдокод процедуры Pseudoprime, проверяющей число n на простоту рассматриваемым способом:

```
Pseudoprime(n)
1 if Modular_Exponentiation(2, n -1, n) ≠ 1 (mod n)
2     then return Составное   \ определенно
3     else return  Простое  \ предположительно
```

В процедуре Pseudoprime применяется процедура Modular_Exponentiation, которая возводит число a в степень c по модулю n **методом повторного возведения в квадрат**:

```
Modular_Exponentiation(a, b, n)
1 c ← 0
2 d ← 1
3 Пусть ⟨b_k, b_{k-1},…, b_0⟩ - бинарное представление числа b
4 for i ← k downto 0
5       do c ← 2c
6          d ← (d·d) mod n
7          if b_i = 1
8             then c ← c + 1
9                  d ← (d·a) od n
10  return d
```

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Рисунок 3.1 – Работа процедуры Modular_Exponentiation.

Работа процедуры Modular_Exponentiation(*a*, *b*, *n*) при *a* = 7, *b* = 560 = ⟨1000110000⟩ и *n* = 561 представлена на рисунке 3.1., где показаны значения переменных после очередного исполнения цикла **for**. Процедура возвращает ответ 1.

Если процедура Pseudoprime(*n*) выдает нам на выходе, что число *n* – составное, то это всегда верно. Если же она утверждает, что число *n* - простое, то это заключение ошибочно только тогда, когда *n* - псевдопростое по основанию 2.

Например, существует всего 22 таких числа, меньших, чем 10 000, когда тест на псевдопростоту выдает ошибочный ответ.

К сожалению, не удается полностью исключить все ошибки, возникающие из-за псевдопростых чисел по другим основаниям, например, при *a* = 3, поскольку существуют составные числа *n*, удовлетворяющие уравнению (3) при всех $a \in Z_n^*$. Эти числа известны как **числа Кармайкла**. Первые три числа Кармайкла - 561, 1105 и 1129. Числа Кармайкла встречаются крайне редко, например, 255 таких чисел меньше, чем 100 000 000.

4. ЖАДНЫЕ АЛГОРИТМЫ

Алгоритмы**,** предназначенные для решения задач оптимизации, обычно представляют собой последовательность шагов, на каждом из которых предоставляется некоторое множество выборов. Определение наилучшего выбора, руководствуясь принципами динамического программирования, во многих за-

дачах оптимизации не является целесообразным: для этих задач подходят более простые и эффективные алгоритмы. **В жадном алгоритме** всегда делается выбор, который кажется самым лучшим в данный момент – т.е. производится локально оптимальный выбор в надежде, что он приведет к оптимальному решению глобальной задачи. Таким образом, свойство жадного выбора заключается в том, что глобальное оптимальное решение можно получить, делая локальный оптимальный (жадный) выбор.

Жадные алгоритмы не всегда приводят к оптимальному решению, но во многих задачах они дают нужный результат.

4.1 Задача о выборе процессов.

Рассмотрим простую, но нетривиальную задачу о выборе процессов, эффективное решение которой можно найти с помощью жадного алгоритма. Чтобы прийти к жадному алгоритму, сначала рассмотрим решение, основанное на парадигме динамического программирования, после чего покажем, что оптимальное решение можно получить исходя из принципов жадных алгоритмов.

Задачу о выборе процессов сформулируем как задачу о составлении расписания для нескольких конкурирующих процессов, каждый из которых безраздельно использует общий ресурс. Цель этой задачи – выбор набора взаимно совместимых процессов, образующих множество максимального размера.

Предположим, что имеется множество $S = \{ a_1, a_2 \ldots a_n \}$, которое состоит из n процессов. Каждый процесс a_i характеризуется начальным моментом s_i и конечным моментом f_i, где $0 \le s_i < f_i < \infty$. **Процессы** a_i и a_j **совместимы**, если интервалы $[s_i, f_i)$ и $[s_j, f_j)$ не перекрываются: $f_j \le s_i$ или $f_i \le s_j$.

Задача о выборе процессов (activity-selection problem) заключается в том, чтобы выбрать подмножество взаимно совместимых процессов, образующих множество максимального размера.

Для примера рассмотрим описанное ниже множество процессов, отсортированных в порядке возрастания моментов окончания:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

Из взаимно совместимых процессов можно составить, например, подмножество $\{a_3, a_9, a_{11}\}$. Но оно не является максимальным, т.к. существуют еще подмножества $\{a_1, a_4, a_8, a_{11}\}$ и $\{a_2, a_4, a_9, a_{11}\}$, состоящие из большего количества элементов.

Разобьем решение этой задачи на несколько этапов. Для начала сформулируем решение рассматриваемой задачи, основанное на принципах динамического программирования. Это оптимальное решение исходной задачи получается путем комбинирования оптимальных решений подзадач. Рассмотрим несколько вариантов выбора, которые делаются в процессе определения подзадачи, использующейся в оптимальном решении. Впоследствии станет понятно, что заслуживает внимания лишь один выбор – жадный - и что, когда делается этот выбор, одна из подзадач гарантированно полчаса пустой. Остается лишь одна непустая подзадача. Исходя из этих наблюдений, мы разработаем рекурсивный жадный алгоритм, предназначенный для решения задачи о составлении расписания процессов. Процесс разработки жадного алгоритма будет завершен его преобразованием из рекурсивного в итерационный. Описанные этапы сложнее, чем те, что обычно имеют место при разработке жадных алгоритмов, однако они иллюстрируют взаимоотношение между жадными алгоритмами и динамическим программированием.

Итак, разработаем решение для задачи о выборе процессов по методу динамического программирования – найдем оптимальную подструктуру и с ее помощью построим оптимальное решение задачи, пользуясь оптимальным решением подзадач. Пусть S_{ij} – подмножество процессов из множества S, которые можно успеть выполнить в промежутке времени между завершением процесса a_i и началом процесса a_j.

$S_{ij} = \{ a_k \in S : f_i \le s_k < f_k \le s_k \}$.

Добавим фиктивные процессы a_0 и a_{n+1} таких, что $f_0 = 0$ и $f_{n+1} = \infty$.

Тогда $S = S_{0,n+1}$ и $0 \le i,\ j \le n + 1$.

Предположим, что процессы отсортированы в порядке возрастания соответствующих им конечных моментов времени:

$$f_0 \le f_1 \le \ldots \le f_n \le f_{n+1}$$

Тогда $S_{ij} = \varnothing$, если $j \le i$.

Действительно, пусть $\exists\ a_k \in S_{ij}$ для некоторых $j \le i$. Из определения S_{ij} следует, что $f_i \le s_k < f_k \le s_j < f_j$. Отсюда $f_i < f_j$. Что противоречит предположению о том, что процесс a_i следует за процессом a_j.

Если процессы отсортированы в порядке монотонного возрастания времени их окончания, то пространство подзадач образуется путем выбора максимального подмножества взаимно совместимых процессов из множества S_{ij} для $0 \le i < j \le n + 1$ (как мы доказали, все прочие S_{ij} пустые).

Рассмотрим некоторую непустую подзадачу S_{ij} и предположим, что ее решение включает некоторый процесс $a_k : f_i \le s_k < f_k \le s_j$. С помощью процесса a_k генерируется две подзадачи S_{ik} и S_{kj}.

Теорема (теорема об оптимальных решениях). Пусть оптимальное решение A_{ij} задачи S_{ij} включает в себя процесс a_k. Тогда оптимальные решения A_{ik} задачи S_{ik} и A_{kj} задачи S_{kj} в рамках оптимального решения тоже должны быть оптимальными.

В любое решение непустой задачи S_{ij} входит некоторый процесс a_k и любое оптимальное решение задачи содержит в себе подзадачи S_{ik} и S_{kj}.

Таким образом, максимальное подмножество взаимно совместимых процессов множества S_{ij} можно составить путем разбиения задачи на две подзадачи – нахождения максимальных подмножеств A_{ik} и A_{kj} взаимно совместимых процессов, и составления подмножеств максимального размера, включающих в себя взаимно совместимые задачи:

$$A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj} . \qquad (4.1)$$

Определим значение, соответствующее оптимальному рекурсивному решению. Пусть $c[i, j]$ – количество процессов в подмножестве максимального размера, состоящем из взаимно совместимых процессов в задаче S_{ij}.

Эта величина равна нулю, при $S_{ij} = \varnothing$; частности $c[i, j] = 0$ при $j \le i$.

Из (4.1): $c[i, j] = c[i, k] + c[k, j] + 1$, где $k = i + 1, \ldots, j - 1$.

Поскольку в подмножестве максимального размера для задачи S_{ij} должно использоваться одни из этих значений k, то нужно проверить, какое из них подходит лучше других. Таким образом, полное рекурсивное определение величины $c[i, j]$ принимает вид:

$$c[i, j] = \begin{cases} 0 & \text{при } S_{ij} = 0 \\ max\{c[i, j] + c[k, j] + 1\} & \text{при } S_{ij} \neq 0 \\ \quad i < k < j \text{ и } a_k \in S_{ij} & \end{cases}$$

Преобразуем решение динамического программирования в жадное решение. Рассмотрим произвольную непустую задачу S_{ij} и пусть a_m - процесс, который оканчивается раньше других, то есть $f_m = \min \{ f_k : a_k \in S_{ij} \}$. Тогда справедливы утверждения:

1) Процесс a_m используется в некотором подмножестве максимального размера, состоящем из взаимно совместимых процессов задачи S_{ij}.

2) Подзадача S_{im} пустая, поэтому в результате выбора процесса a_m непустой остается только подзадача S_{mj}.

В решение, основанном на принципе динамического программирования в оптимально решении используется две подзадач, а также до $j - i - 1$ вариантов выбора для подзадачи S_{ij}. Благодаря утверждениям, сформулированным выше, эти величины уменьшаются: в оптимальном решении используется лишь одна подзадача (вторая подзадача гарантированно пустая), а в процессе решения подзадачи S_{ij} достаточно рассмотреть только один выбор - тот процесс, который оканчивается раньше других. Кроме уменьшения количества подзадач и количества выборов появляется возможность решать каждую

подзадачу в нисходящем направлении, а не в восходящем, как это обычно происходит в динамическом программировании.

Ниже приведено простое рекурсивное решение, основанное на жадной нисходящей манере и реализованное в виде процедуры Recursive_Activity_Selector:

```
Recursive_Activity_Selector(s, f, i, n)
1 m ← i +1
2 while  m ≤ n  и  s_m < f_i   \ Поиск первого процесса в S_{i, n+1}
3          do m ← m + 1
4 if  m ≤ n
5      then  return{ a_m }∪ RECURSIV_ACTIVITY_SELECTOR(s, f, i, n)
6      else return ∅
```

На ее вход подаются значения начальных и конечных моментов процессов, представленные в виде массивов s и f, а также индексы i и n, определяющие подзадачу $S_{i, n+1}$, которую требуется решить. Параметр n идентифицирует последний реальный процесс a_n в подзадаче, а не фиктивный процесс a_{n+1}, который тоже входит в эту подзадачу. Процедура возвращает множество максимального размера, состоящее из взаимно совместных процессов задачи $S_{i, n+1}$. Так как мы предполагали, что все процессы располагаются в порядке монотонного возрастания времени их окончания (если это не так, то мы уже знаем, что можем их отсортировать за время $O(\log_2 n)$), то начальный вызов этой процедуры имеет вид Recursive_Activity_Selector(s, f, 0, n).

Если процессы отсортированы в порядке, заданном временем их окончания, то время, которое затрачивается на вызов процедуры Recursive_Activity_Selector(s, f, 0, n) равно $\Theta(n)$, так как сколько бы не было рекурсивных вызов, каждый процесс проверяется в цикле **while** в строке 2 ровно по одному разу. В частности процесс a_k проверяется в последнем вызове, когда $i < k$.

Представленную рекурсивную структуру легко преобразовать в итеративную. Процедура Recursive_Activity_Selector является почти **оконечной рекурсией**, так как она оканчивается рекурсивным вызовом самой себя, после чего выполняется операции объединения. Преобразование процедуры, построенной по принципу оконечной рекурсии, в итерационную-обычно простая задача, которую некоторые компиляторы языком программирования выполняют автоматически.

Итеративная процедура Greedy_Activity_Selector выполняет ту же задачу, что и рекурсивная процедура Recursive_Activity_Selector:

Greedy_ Activity_Selector(*s*, *f*)

1 $n \leftarrow$ length [*s*]

2 $A \leftarrow \{ a_1 \}$

3 $i \leftarrow 1$

4 **for** $m \leftarrow 2$ to n

5 **do if** $s_m \geq f_i$

6 **then** $A \leftarrow A \cup \{ a_m \}$

7 $i \leftarrow m$

8 **return** *A*

В ней также предполагается, что входные процессы расположены в порядке монотонного возрастания времен окончания. Выбранные процессы объединяются в этой процедура в множество *A*, которое и возвращается процедурой после ее окончания.

Процедура работает следующим образом. Переменная индексирует самое последнее добавление к множеству *A*, соответствующее процессу a_i в рекурсивной версии. Поскольку процессы рассматриваются в порядке монотонного возрастания моментов их окончания, f_i - всегда максимальное время окончания всех процессов, принадлежащих множеству *A*:

$$f_i = \max\{f_k : a_k \in A\}.$$

В строках 2 – 3 выбирается процесс a_1, инициализируется множество *A*, содержащее только этот процесс, а переменной i присваивается индекс этого

процесса. В цикле **for** в строках 4 – 7 происходит поиск процесса задачи $S_{i, n+1}$, оканчивающегося раньше других. В этом цикле по очереди рассматриваются каждый процесс a_m, который добавляется в множество A, если он совместим со всеми ранее выбранными процессами; этот процесс раньше других оканчивается в задаче $S_{i, n+1}$. В строке 5 осуществляется проверка, является ли процесс a_m совместимым с процессами, уже содержащимися в множестве A. Если процесс удовлетворяет условию, то в строках 6 – 7 он добавляется в множество A и переменной присваивается значение m.

Процедура Greedy_ Activity_Selector, как и ее рекурсивная версия, составляет расписание для n – элементного множества в течение времени $\Theta(n)$.

5. КОМБИНАТОРНЫЕ АЛГОРИТМЫ

В комбинаторных алгоритмах, занимающихся подсчетом элементов конечных множеств, часто необходимо порождать и исследовать все элементы некоторого класса комбинаторных объектов. Наиболее общие методы решения таких задач основываются на поиске с возвращением, однако во многих случаях объекты, а именно перестановки, подмножества, сочетания, композиции и разбиения целых чисел, настолько просты, что целесообразнее применять специализированные методы.

5.1. Комбинаторные объекты.

Рассматривая решение задачи с абстрактной точки зрения, как правило, избегают каких бы то ни было предположений относительно того, как мы намерены автоматизировать ее решение. Любой заданный класс абстрактных объектов может иметь несколько возможных представлений, и выбор наилучшего из них решающим образом зависит от того, каким образом объект будет использован, а также от типа производимых над ним операций. В данном курсе мы будем рассматривать смежные представления размещений абстрактных объектов.

В алгоритмах на дискретных структурах часто приходится встречаться с представлением конечных последовательностей и операциями с ними. С вычислительной точки зрения простейшим представлением конечной последовательности $s_1, s_2, \ldots, s_n$ является точный список ее членов, расположенных по порядку в смежных ячейках памяти. В языках высокого уровня — это одномерные, двухмерные и т. д. массивы данных. Наряду с очевидными преимуществами последовательное представление имеет и некоторые значительные недостатки. Смежное представление становится неудобным, если требуется изменить последовательность путем включения новых и исключения имеющихся там элементов. Включение между s_i и s_{i+1} нового элемента требует сдвига $s_i, s_{i+1}, \ldots, s_n$ вправо на одну позицию; аналогично исключение требует сдвига тех же элементов на одну позицию влево:

1. $n = n + 1$; {Включить элемент z на i-ое место}
2. **for** $j = n - 1$ **to** i **by** -1 **do** $s_j+1 = s_j$;
3. $s_i = z$;
4. {Исключить элемент с i-ого места }
5. **For** $j = 1$ **to** $n - 1$ **do** $s_j = s_{j+1}$;
6. $n = n - 1$.

В обоих случаях включение или удаление элементов при смежном представлении требует перемещения многих элементов. С точки зрения времени обработки такое перемещение элементов может оказаться дорогостоящим из-за сложности операций включения и удаления $O(n)$.

Важной разновидностью смежного размещения является случай, когда такому представлению подвергается подпоследовательность $s_{k_1}, s_{k_2}, \ldots, s_{k_m}$ некоторой основной последовательности $s_1, s_2, \ldots, s_n$. В этом случае подпоследовательность можно представить более удобно, используя характеристический вектор - последовательность из нулей и единиц, где i-и разряд равен единице, если s_j принадлежит рассматриваемой подпоследовательности. Например, для последовательности (1,2,3,4,5,6,7,8,9) характеристический вектор

подпоследовательности чисел, кратных 3, имеет вид (0,0,1,0,0,1,0,0,1). Характеристические векторы полезны в том случае, когда формирование нужной подпоследовательности выполняется путем последовательного удаления из основной последовательности элементов, которые не входят в подпоследовательность. Главное неудобство характеристических векторов состоит в том, что они не экономичны.

Существуют два основных подхода к представлению множеств в памяти.

1. При первом подходе хранят описание каждого элемента, действительно присутствующего в множестве, как это делается, когда выписываются все элементы множества и заключаются в фигурные скобки.

2. При втором подходе изначально определяются все потенциально возможные элементы множества, а затем для любого подмножества этого универсального множества для каждого возможного члена указывается, принадлежит ли он на самом деле данному подмножеству или нет.

При первом подходе представления множеств используют как смежное, так и связанное размещение его элементов в памяти. Как и для последовательностей, наилучший метод представления множеств существенно зависит от операций, которые мы собираемся выполнять над ними. Типичные операции над множествами: выяснить, имеется ли конкретный элемент в данном множестве; добавить в множество новые элементы; удалить элементы из множества; выполнить обычные теоретико-множественные операции, такие как объединение или пересечение двух множеств. Как правило, для представления множеств применяют связанную память.

При втором методе множество представляется в виде вектора на смежной памяти. Пусть U — универсальное множество (т. е. все рассматриваемые множества являются его подмножествами), состоящее из п элементов. Любое подмножество $S \subseteq U$ представляется в виде характеристического вектора из n элементов:

1	1	0	1	1	1	0	0	0	0	1	1	0	0	0

Элемент i в этом векторе равен 1 тогда и только тогда, когда i-ый элемент множества U принадлежит S, в противном случае он устанавливается равным 0. Представление в виде характеристического вектора удобнее тем, что можно определять принадлежность i-го элемента множеству за время, не зависящее от его размера. Основные операции над множествами, такие как объединение и пересечение, можно осуществлять как операции $\vee$ и $\wedge$ над двоичными векторами. Недостаток этого представления заключается в том, что операции объединение и пересечение занимают время, пропорциональное мощности универсального множества U, а не рассматриваемого множества S. Данное представление требует дополнительной памяти для хранения характеристического вектора, что для больших n (размер универсального множества U) бывает практически невыполнимо.

Задачи, требующие генерации комбинаторных объектов, возникают при вычислении комбинаторных формул. Например, часто приходится вычислять суммы, имеющие вид $\Sigma f(x_1, x_2, \ldots, x_n)$, где суммирование выполняется по всем последовательностям $x_1, x_2, \ldots, x_n$, удовлетворяющим некоторым ограничениям.

Алгоритм систематического порождения состоит из трех компонент: выбор начальной конфигурации, трансформация одного объекта в следующий и условия окончания, говорящего о том, когда остановиться.

Общая форма такого алгоритма имеет вид:

1. Выбрать начальную конфигурацию
2. **while not** условие окончания **do**
3. Вывести объект и преобразовать в следующий объект

В рамках такой схемы выбор начальной конфигурации должен учитывать тот факт, что алгоритм предпринимает еще одно преобразование после того, как распечатан последний объект. Поскольку это постороннее преобразование не должно приводить к ошибке в программе, некоторые из операций

выбора начальной конфигурации могут присваивать значения на вид посторонним переменным, например, нулевому или (n + 1)-му элементу массива, в то время как массив обычно имеет элементы только с номерами от 1 до n. Обращение к этой посторонней переменной или изменение ее значения часто можно использовать как условие окончания.

Процедуру, последовательно производящую объекты путем последовательных обращений, можно записать по данным операции выбора начальной конфигурации, трансформации и условию окончания:

Procedure Geberate (объект, flag)

1. **if** flag **then**
2. выбрать начальную конфигурацию
3. flag ← false
4. **else**
5. преобразовать в следующий объект
6. **if** условие окончания **then**
7. flag ← true
8. **return**

В этой процедуре используется как сигнал вновь начать порождение и как сигнал порождения последнего объекта.

В алгоритме порождения всех элементов множества нас прежде всего интересует общее количество времени, требующегося для порождения всего множества. В частности, в некоторых алгоритмах можно порождать все множество за время, пропорциональное его мощности. Такие алгоритмы, называемые линейным, наиболее благоприятны для использования, так как их эффективность-асимптотически наилучшая из возможных. Также при анализе комбинаторных алгоритмов представляет интерес количество изменений, которые происходят при переходе к последующим объектам. Желательно, чтобы такого рода изменения были возможно малыми. Если такие изменения в алго-

ритме, действительно, малы, то алгоритм называется **алгоритмом с минимальными изменениями**. Точное определение минимального изменения зависит от рассматриваемых комбинаторных конфигураций.

Алгоритмы, требующие случайного порождения элементов в классе комбинаторных объектов, возникают при оценке сложности по методу Монте-Карло. Например, если мы не можем оценить поведение алгоритма в среднем, для суждений об эффективности, возможно придется испытать его на большом числе случайно выбранных входных конфигураций. При этом случайное порождение часто затруднено тем, что мы хотим равные вероятности всех возможных конфигураций.

Один из способов подхода к систематическому и случайному порождению состоит в задании определенного соответствия между целыми числами 0,1, …, n - 1 и n объектами. Систематическое порождение тогда осуществляется перечислением целых чисел от 0 до n - 1 и обращением каждого числа в объект, в то время как случайное порождение осуществляется случайным порождением целого числа от 0 до n - 1 и обращением его в объект.

Рассмотрим перестановки различных элементов. При составлении размещений без повторений из n по k мы получали расстановки, отличающиеся друг от друга либо составом, либо порядком элементов. Но если брать расстановки, которые включают все n элементов, то они могут отличаться друг от друга лишь порядком входящих в них элементов. Такие расстановки называются перестановками из n элементов, а их число обозначается $P_n = n!$. Без ограничения общности мы полагаем, что элементами множества являются целые числа от 1 до n, то есть рассматриваются перестановки только целых чисел от 1 до n. Такое упрощение будет возможно также провести для других комбинаторных объектов.

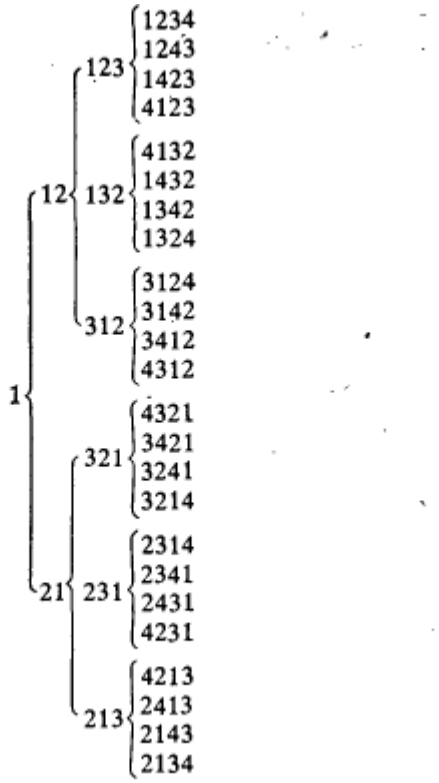
Последовательность перестановок на множестве {1, 2, …, n} представлена в **лексикографическом порядке**, если она записана в порядке возрастания получающихся чисел.

Например, лексикографическая последовательность перестановок трех элементов имеет вид 123, 132, 213, 231, 312, 321.

В общем случае, если $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ и $\tau = (\tau_1, \tau_2, \ldots, \tau_n)$ – перестановки, то говорят, что σ лексикографически меньше τ, если и только если для некоторого $k \geq 1$ мы имеем $\sigma_j = \tau_j$ для всех $j < k$ и $\sigma_k < \tau_k$.

В дальнейшем, говоря об упорядочении элементов, мы будем предполагать именно лексикографический порядок.

Последовательность $n!$ перестановок на множестве $\{1, 2, ..., n\}$, в которой соседние перестановки различаются так мало, как только возможно, — лучшее, на что можно надеяться с точки зрения минимизации объема работы, необходимого для порождения перестановок. Для того чтобы такое различие было минимально возможным, любая перестановка в нашей последовательности должна отличаться от предшествующей ей транспозицией двух соседних элементов. Такую последовательность перестановок легко построить рекурсивно. Для $n = 1$ единственная перестановка $\{1\}$ удовлетворяет нашим требованиям. Предположим, мы имеем последовательность перестановок π_1 ; π_2; π_3 , … на множестве $\{1, 2, ..., n\}$, в которой последовательные перестановки различаются только транспозицией смежных элементов. Расширим каждую из этих $(n - 1)!$ перестановок, вставляя элемент n на каждое из n возможных мест. Порядок порождаемых таким образом перестановок будет следующим:



Данную последовательность перестановок можно порождать итеративно, получая каждую перестановку из предшествующей ей и небольшого количества добавочной информации. Это делается с помощью трех векторов: текущей перестановки $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, обратной к ней перестановки $\rho = (\rho_1, \rho_2, \ldots, \rho_n)$ и записи направления d_i, в котором сдвигается каждый элемент i (-1, если он сдвигается влево; +1, если вправо; и 0, если не сдвигается).

Элемент сдвигается до тех пор, пока не достигнет элемента, большего, чем он сам; в этом случае сдвиг прекращается. В этот момент направление сдвига данного элемента изменяется на противоположное и передвигается следующий меньший его элемент, который можно сдвинуть. Поскольку хранится перестановка, обратная к π, то в π легко найти место следующего меньшего элемента. Следующий алгоритм представляет собой реализацию рассмотренного метода:

1. **for** i =1 **to** n **do** begin
2. $\pi_i = \rho_i = i$;
3. $d_i = -1$; **end**;
4. $d_1 = 0$;

5. $\pi_0 = \pi_{n+1} = m = n + 1$; \ Метка границы
6. **while** $m \neq 1$ **do begin**
7. **print**($\pi_1, \pi_2, \ldots, \pi_n$);
8. $m = n$
9. **while** $\pi_{\rho m + d m} > m$ **do begin**
10. $d_m = -d_m$;
11. $m = m - 1$;
12. **end**;
13. $\pi_{\rho m} \leftrightarrow \pi_{\rho m + d m}$; \ Изменить π
14. $\rho_{\pi_{\rho m}} \leftrightarrow \rho_m$;\ Изменить $\rho = \pi^{-1}$, $\pi_{\rho m + d m} = m$
15. **end.**

Корректность алгоритма доказывается индукцией по n. Алгоритм порождения всех $n!$ перестановок линеен, сложность его определяется как $n! + o(n!)$. Этот алгоритм - один из наиболее эффективных алгоритмов для порождения перестановок. Рабочая программа на языке Pascal реализации эффективного метода генерации перестановок приводится ниже:

```
program start_effect; {Эффективная генерация перестановок}
const
  max_n = 20; {Количество перестановок}
type
  vector = array[0..max_n + 1] of integer;
procedure effect(var z: vector; n: integer);
var
  p, d: vector;
  pm, dm, zpm: integer;
  i, m, w, k: integer;
begin
  k := 0;
  for i := 1 to n do
```

```
begin
  z[i] := i; p[i] := i; d[i] := -1;
end;
d[1] := 0;
m := n + 1;
z[0] := m;
z[n + 1] := m;
while m <> 1 do
begin
  k := k + 1;
  write( k, ') '); {Печать перестановок}
  for i := 1 to n do
    write( z[i], ' ');
  writeln;
  m := n;
  while z[p[m] + d[m]] > m do
  begin
    d[m] := -d[m];
    m := m - 1;
  end;
  pm := p[m];
  dm := pm + d[m];
  w := z[pm];
  z[pm] := z[dm];
  z[dm] := w;
  zpm := z[pm];
  w := p[zpm];
  p[zpm] := pm;
  p[m] := w;
end;
```

```
end;
var{main}
  z: vector;
  n: integer; {Длина перестановок}
begin
  readln(n);
  effect(z, n);
end.
```

Рассмотрим задачу порождения подмножеств множества $\{a_1, a_2, a_3, \ldots, a_n\}$. Эта задача эквивалентна порождению n – разрядных двоичных наборов a_i, принадлежащих подмножеству, если и только если i-й разряд равен единице. Таким образом, задача порождения всех подмножеств множества сводится к задаче порождения всех возможных двоичных последовательностей длины n. Очевидно, что наиболее прямым способом порождения всех двоичных наборов длины n является счет в системе счисления с основанием 2. Алгоритм решения задачи порождения всех n - разрядных наборов с помощью счета в системе счисления с основанием 2 имеет вид:

1. **for** $i = 0$ **to** n **do** $b_i = 0$;
2. **while** $b_n \neq 1$ **do begin**
3. print($b_{n-1}, b_{n-2}, \ldots, b_0$);
4. $i = 0$;
5. **while** $b_i = 1$ **do begin**
6. $b_i = 0$;
7. $i = i + 1$;
8. **end;**
9. $b_i = 1$;
10. **end.**

Для порождения подмножеств множества с помощью счета в двоичной системе счисления добавим фиктивный элемент a_{n+1}:

1. $S = \varnothing$;

2. **while** $a_{n+1} \notin S$ **do begin**
3. print(S);
4. $i = 0$;
5. **while** $a_i \in S$ **do begin**
6. $S = S - \{ a_i \}$;
7. $i = i + 1$;
8. **end;**
9. $S = S \cup \{ a_i \}$;
10. **end**.

Данный алгоритм является линейным, так как проверка условия осуществляется только один рад для каждого двоичного набора, еще раз для каждого второго набора и т.д., или $2^n + 2^{n-1} + \ldots + 2 + 1 = 2^{n+1}-1$ раз. Однако алгоритм не является алгоритмом минимального изменения, и порядок, в котором порождаются подмножества, не имеет каких –либо специальных свойств, которые говорят в его пользу. (естественный лексикографический порядок получается применение алгоритма возвращения для порождения подмножеств).

Рассмотрим задачу «Счастливый билет», решаемую на основе данного алгоритма порождения двоичных наборов. Дано n ($n > 2$) произвольных цифр: $a_1, a_2, a_3, \ldots, a_n$, где $a_i \in \{1, 2, ..., 9\}$, и произвольное целое число m. Написать программу, которая расставляла бы между каждой парой цифр: $a_1, a_2, a_3, \ldots, a_n$, записанных именно в таком порядке, знаки «+», «-» так, чтобы значением получившегося выражения было число m. Например, если $a_1, a_2, a_3, \ldots, a_n$ соответственно равны 1, 2, ... ,9 и $m = 5$, то подойдет следующая расстановка знаков: 1 - 2 + 3 – 4 + 5 – 6 +7 – 8 + 9.

Если требуемая расстановка невозможна, то сообщить об этом.

Исходные данные вводятся из командной строки и имеют следующий формат:

Первая строка — целое число n.

Вторая строка — целое число m.

Третья строка — цифры $a_1, a_2, a_3, \ldots, a_n$ через пробелы.

Результаты расчетов Выводятся на экран.

Пример исходных данных:

14

71

65884752345789

Пример выходных данных:

6+5+8+8+4+7+5+2+3+4-5+7+8+9 = 71

6+5+8+8+4+7+5-2-3+4+5+7+8+9 = 71

6+5+8+8+4+7-5+2+3+4+5+7+8+9 = 71

6-5+8+8+4+7+5+2+3+4+5+7+8+9 = 71

Время счета = 0.60 с.

Ясно, что для решения данной задачи достаточно выполнить полный перебор всех возможных вариантов расстановки знаков «±» между каждой парой цифр a_1, a_2, a_3, …, a_n и выбрать те расстановки знаков «±», которые удовлетворяют условию равенства суммы величине m. Всего позиций для расстановки «±» равно n - 1, а значит для полного перебора необходимо проверить 2^{n-1} двоичных наборов, если «+» будет соответствовать 1, а «-» будет соответствовать 0.

Программа решения задачи «счастливый билет», реализованная на языке Pascal, представлена ниже:

```
1   program lucky_ticket;{Вычисление счастливой суммы}
2   const
3     max_n = 20;
4   type
5     vector = array [1..max_n] of longint;
6   procedure summa(var a, b: vector; n, m: integer);
7   var
8     s: longint;
9     i: integer;
10  begin
```

```
11              s := a[1];
12              for i := 1 to n - 1 do
13                s := s + (2 * b[i] - 1) * a[i + 1];
14              if s = m then
15              begin
16                write(a[1]);
17                for i := 1 to n - 1 do
18                begin
19                  if b[i] = 1 then write('+') else write('-');
20                  write(a[i + 1]);
21                end;
22                writeln(' = ', s:4);
23              end;
24            end;
25            procedure subset(var a: vector; n, m: integer);
26            var
27              b: vector;
28              i: integer;
29            begin
30              for i := 1 to n do b[i] := 0;
31              while b[n] <> 1 do
32              begin
33                summa(a, b, n, m);
34                i := 1;
35                while b[i] = 1  do
36                begin
37                  b[i] := 0;
38                  i := i + 1;
39                end;
40                b[i] := 1;
```

```
41              end;
42            end;
43            var{main}
44              a: vector;
45              n, m, i: integer;
46              Hour,Minute,Second,Sec100: word;
47              rHour,rMinute,rSecond,rSec100: word;
48             delta: longint;
49            begin{main}
50              read(n); {Ввод количества чисел}
51              read(m);{Ввод счастливой суммы}
52              for i := 1 to n do read(a[i]);
53              GetTime(Hour,Minute,Second,Sec100);
54              subset(a, n, m);
55             GetTime(rHour,rMinute,rSecond,rSec100);
56             delta :=rHoure-Houre;
57             delta:=delta*60+rMinute-Minute;
58             delta:=delta+60+rSecond-Second;
59             delta=delta*100+rSec100-Sec100;
60            Writeln('Время счета=',delta div 100, '.',delta mod 100, 'c');
61          end.
```

Процедура SubSet (подмножество) этой программы реализует рассмотренный выше алгоритм счета в системе счисления с основанием 2 для порождения всех (*n* -1)-разрядных наборов. Порожденные двоичные наборы используются в процедуре Summa (сумма) для формирования суммы, соответствующей данному набору знаков «±», где «+» соответствует 1, а «-» - 0.

Наименьшим возможным изменением при переходе от одного подмножества к другому является добавление или удаление одного элемента множества. В терминах двоичных наборов это означает, что последовательные

наборы должны различаться в одном разряде. Такие последовательности наборов известны как **коды Грея**: ***n* – разрядный код Грея** есть упорядоченная циклическая последовательность 2^n *n*- разрядных наборов {кодовых слов}, такая что последовательные кодовые слова различаются в одном разряде. Коды Грея можно удобно представить их последовательностью переходов, то есть упорядоченным списком номеров разрядов (пронумерованных справа налево), которые меняются при переходе от одного кодового слова к другому. Обычно, начальным кодовым словом является (00 … 0), хотя на самом деле это несущественно, поскольку код циклический. Подробнее о кодах Грея и применении их в комбинаторных алгоритмах можно почитать в книге Д. Кнута «Искусство программирования. Том 4, А. Комбинаторные алгоритмы. Часть 1.»[7, с. 329].

Рассмотрим задачу генерации размещений с повторениями. Порождение множества всех размещений с повторениями длины k из элементов $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$ эквивалентно генерации множества k-разрядных чисел в системе счисления с основанием n: на r-м месте в размещении будет располагаться элемент a_i, если цифра в r-м разряде соответствующего числа равна i. Всего размещений с повторениями n^k. Например, для $k = 2$ и $n = 3$ все наборы длины два в системе счисления с основанием три можно записать: 00, 01, 02, 10, 11, 12, 20, 21, 22. Тогда эквивалентные размещения примут вид (a_0a_0), (a_0a_1), (a_0a_2), (a_1a_0), (a_1a_0), (a_1a_1), (a_1a_2), (a_2a_0), (a_2a_1), (a_2a_2).

Алгоритм порождения всех k-разрядных наборов с помощью счета в системе счисления с основание n использует фиктивный элемент b_k в системе счисления с основанием n, где $b_i \in \{0, 1, \ldots, n - 1\}$, $i = 0, 1, 2, \ldots, k$:

1. **for** $i = 0$ **to** k **do** $b_i = 0$;
2. **while** $b_k \neq 1$ **do begin**
3. print($b_{k-1}, b_{k-2}, \ldots, b_0$);
4. $i = 0$;
5. **while** $b_i = n - 1$ **do begin**
6. $b_i = 0$;

7. $i = i + 1$;

8. **end;**

9. $b_i = b_i + 1$;

10. **end.**

В задачах порождения сочетаний обычно требуются не все подмножества множества $\{a_1, a_2, a_3, \ldots, a_n\}$, а только те, которые удовлетворяют некоторым ограничениям. Особый интерес представляют подмножества фиксированной длины k, C_n^k сочетаний из n предметов по k штук. Как обычно, предполагаем, что основным множеством является множество натуральных чисел {1, 2, ..., n}; таким образом, будем порождать все сочетания длины k из целых чисел {1, 2, ..., n}. Так, например, $C_5^4 = 5$ сочетаний из пяти предметов по четыре (т.е. четырехэлементные подмножества множества {1,2,3,4,5}) записываются в лексикографическом порядке следующим образом: 1234 1345 2345 1245 1345. Сочетания в лексикографическом порядке можно порождать последовательно простым способом. Начиная с сочетания (1, 2, ..., k} следующее сочетание находится просмотром текущего сочетания справа налево с тем, чтобы определить место самого правого элемента, который еще не достиг своего максимального значения. Этот элемент увеличивается на единицу, и всем элементам справа от него присваиваются новые возможные наименьшие значения:

1. $c_0 = -1$;

2. **for** $i = 0$ **to** k **do** $c_i = i$;

3. **while** $j \neq 1$ **do begin**

4. print($c_1, c_2, \ldots, c_k$);

5. $j = k$;

6. **while** $c_j = n - k + j$ **do** $j = j - 1$

7. $c_j = c_j + 1$;

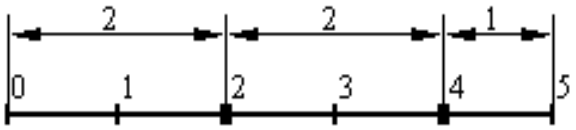
8. **for** $i = j + 1$ **to** k **do** $c_i = c_{i-1} - 1$;

9. **end.**

Данный алгоритма порождения всех C_n^k сочетаний линеен, его сложность $C_n^k + O(C_n^k)$.

Пусть стоит задача порождения разбиения положительного числа n в последовательность неотрицательных целых чисел $\{p_1,p_2,\ldots,p_k\}$, так что $p_1+p_2+\ldots+p_k = n$ причем на p_i могут накладываться различные ограничения. Если порядок чисел p_i важен, то $(p_1, p_2, \ldots, p_k)$ называется **композицией *n***. Поиск композиций ведется с ограничением $p_i > 0$.

Удобное представление композиций получается из рассмотрения целого числа п как отрезка прямой, состоящего из отрезков единичной длины. Каждая композиция n соответствует способу выбора подмножества из $(n - 1)$ точек разделения отрезка длины n на единичные отрезки. Элементами композиции являются расстояния мжеду смежными точками:



Следовательно, алгоритм порождения композиций n сводится к генерации подмножеств $(n-1)$-элементного множества и обращению их в композиции.

Если k фиксировано, то такие композиции называются **композициями *n* из *k* частей**. При их поиске ограничение $p_i > 0$ может сниматься, т.е. разрешается $p_i = 0$.

Если порядок p_i не важен и $p_i > 0$, то $\{p_1,p_2,\ldots,p_k\}$ является мультимножеством и называется **разбиением *n***.

Пример 5.1.1: Пусть n =3. Тогда

композиции: (3), (1,2), (2,1), (1,1,1),

композиции из двух частей ($p_i > 0$): (1,2), (2,1),

композиции из двух частей ($p_i = 0$): (0,3), (1,2), (2,1), (3,0),

разбиения: {3}, {1,2}, {1,1,1}.

Разбиение можно интерпретировать как мультимножество $\{m_1 \cdot p_1, m_2 \cdot p_2, \ldots, m_k \cdot p_k\}$.

Пример 5.1.2: Для n = 7 все возможные разбиения представлены в таблице:

№	Разбиение	№	Разбиение
1	{7·1} = {1,1,1,1,1,1,1}	9	{1·4,3·1} = {4,1,1,1}
2	{1·2,5·1} = {2,1,1,1,1,1}	10	{1·4,1·2,1·1} = {4,2,1}
3	{2·2,3·1} = {2,2,1,1,1}	11	{1·4,1·3} = {4,3}
4	{3·2,1·1} = {2,2,2,1}	12	{1·5,2·1} = {5,1,1}
5	{1·3,4·1} = {3,1,1,1,1}	13	{1·5,1·2} = {5,2}
6	{1·3,1·2,2·1} = {3,2,1,1}	14	{1·6,1·1} = {6,1}
7	{1·3,2·2} = {3,2,2}	15	{1·7} = {7}
8	{2·3,1·1} = {3,3,1}		

Данные разбиения представлены в порядке, где каждое разбиение удовлетворяет условию $p_1 > p_2 > \ldots > p_l$.

Идея приведенного списка разбиений состоит в том, чтобы переходить от одного разбиения к следующему, рассматривая самый правый элемент $m_k \bullet z_k$ разбиения. Если $m_k \bullet z_k$ достаточно велико ($m_k > 1$), можно исключить два z_k для того, чтобы добавить еще одно $z_k + 1$ (или включить одно $z_k + 1$, если в текущий момент его нет). Если $m_k = 1$, то $m_{k+1} \bullet z_{k+1} + m_k \bullet z_k$ достаточно велико для того, чтобы добавить $z_{k-1} + 1$. Все, что остается, превращается в соответствующее число единиц и формируется новое разбиение. Алгоритм рассмотренного процесса разбиения представлен ниже:

1. $k = 1$;
2. $z_{-1} = 0$;
3. $m_{-1} = 0$;
4. $z_0 = n + 1$;
5. $m_0 = 0$;
6. $z_1 = 1$;
7. $m_1 = n$;
8. **while** k ≠ 0 **do begin**
9. print($m_1 \cdot z_1$, $m_2 \cdot z_2$, …, $m_k \cdot z_k$);
10. Summa = $m_k \cdot z_k$;
11. **if** $m_k = 1$ **then begin**

12. $k = k - 1$;
13. Summa = Summa + $m_k \cdot z_k$;
14. **end**;
15. **if** $z_{k-1} = z_k + 1$ **then begin**
16. $k = k - 1$;
17. $m_k = m_k + 1$;
18. **end**
19. **else begin**
20. $z_k = z_k + 1$;
21. $m_k = 1$;
22. **end**;
23. **if** Summa > z_k **then begin**
24. $z_{k+1} = 1$;
25. m_{k+1} = Summa - z_k;
26. $k = k + 1$;
27. **end**;
28. **end**.

Представленный алгоритм имеет линейную сложность, так как число операций, необходимых для перехода от одного разбиения к другому, ограничено константой, не зависящей от n и k.

5.2. Исчерпывающий поиск.

Задача исчерпывающего поиска является фундаментальной в алгоритмах на дискретных структурах. Удивительно то, что, накладывая незначительные ограничения на структуру исходных данных, можно получить множество разнообразных стратегий поиска различной степени эффективности.

При **последовательном поиске** подразумевается исследование элементов множества $a_1, a_2, \ldots, a_n$ в том порядке, в котором они встречаются. «Начни сначала и продвигайся, пока не найдешь нужный элемент; тогда остановись».

Такая последовательная процедура является очевидным способом поиска. Алгоритм выполняет последовательный поиск элемента z в множестве $a_1, a_2, \ldots, a_n$:

1. $c = 0$
2. **for** $i = 1$ **to** n **do begin**
3. **if** $z = a_i$ then **do begin**
4. $c = 1$
5. **break**;
6. **end**;
7. **end**;
8. **if** $c = 1$ then Элемент найден
9. **else** Элемент не найден

Несмотря на свою простоту, последовательный поиск содержит ряд очень интересных идей.

Оценим среднюю сложность поиска элементов множества $a_1, a_2, \ldots, a_n$. Для нахождения i-го элемента a_i, требуется I сравнений. Для вычисления же среднего времени поиска необходимо задать информацию о частоте обращения к каждому элементу множества. Будем предполагать, что частота обращения распределена равномерно, т.е. что ко всем элементам обращаются одинаково часто. Тогда средняя сложность поиска элемента множества является линейной: $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2} = O(n)$.

Рассмотрим распределение частот обращения к элементам в общем случае. Пусть p, обозначает частоту (распределение вероятностей) обращения к элементу a_i, где $p_i > 0$ и $\sum_{i=1}^{n} p_i = 1$. В этом случае средняя сложность (математическое ожидание) поиска элемента будет равна $\sum_{i=1}^{n} ip_i$. Хорошим приближением распределения частот к действительности является закон Зипфа:
$p_i = c/i$ для $i = 1, 2, \ldots, n$. (Дж. К. Зипф заметил, что n-е наиболее употребительное в тексте на естественном языке слово встречается с частотой, приблизительно обратно пропорциональной n.) Нормирующая константа c выбирается

так, что $\sum_{i=1}^{n} p_i = 1$. Пусть элементы множества a_1, a_2, …, a_n упорядочены согласно указанным частотам.

Тогда $c = 1/(\sum_{i=1}^{n} 1/i) = 1/H_n \approx 1/\ln(n)$ и среднее время успешного поиска составит $\sum_{i=1}^{n} ip_i = \sum_{i=1}^{n} c = nc = n/H_n \approx n/\ln(n)$, что много меньше (n + 1)/2.

Последний пример показывает, что даже простой последовательный поиск требует выбора разумной структуры данных множества, который бы повышал эффективность работы алгоритма. Более того, это общепринятая стратегия, время от времени переупорядочивать данные, для большинства последовательных файлов во внешней памяти, когда последовательная природа файла диктуется техническими характеристиками носителя информации. Алгоритм последовательного поиска данных одинаково эффективно выполняется при размещении множества a_1, a_2, …, a_n на смежной или связанной памяти.

Логарифмический (бинарный или метод деления пополам) **поиск данных** применим к сортированному множеству элементов $a_1 < a_2 < \ldots < a_n$ размещение которого выполнено на смежной памяти. Для большей эффективности поиска элементов надо, чтобы пути доступа к ним стали более короткими, чем просто последовательный перебор. Наиболее очевидный метод: начать поиск со среднего элемента, т.е. выполнить сравнение с элементом $a_{\lfloor \frac{1+n}{2} \rfloor}$. Результат сравнения позволит определить, в какой половине последовательности a_1, a_2, $a_{\lfloor \frac{1+n}{2} \rfloor}$. …, a_n продолжить поиск, применяя к ней ту же процедуру, и т.д.

Основная идея бинарного поиска довольно проста. Чтобы досконально разобраться в алгоритме, лучше всего представить данные $a_1 < a_2 < \ldots < a_n$ в виде двоичного дерева сравнений, которое отвечает бинарному поиску. Двоичное дерево называется **деревом сравнений**, если для любой его вершины (корня дерева или корня поддерева) выполняется условие:

{Вершины левого поддерева} <{Вершина корня} <{Вершины правого поддерева}.

Пусть на очередном шаге деления пополам оказалось, что необходимо выполнить поиск среди элементов $a_i < a_{i+1} < \ldots < a_j$. В качестве корня принимается элемент $a_{\lfloor \frac{i+j}{2} \rfloor}$, i+j, , где $\lfloor \frac{i+j}{2} \rfloor$ — наибольшее целое, меньшее или равное $\frac{i+j}{2}$. Левое поддерево располагается в векторе a_i, a_{i+1}, …, $a_{\lfloor \frac{i+j}{2} \rfloor - 1}$, а правое поддерево – в векторе $a_{\lfloor \frac{i+j}{2} \rfloor + 1}$, …, a_n.

Алгоритм логарифмического поиска элемента z в $a_1 < a_2 < \ldots < a_n$ в псевдокоде представлен ниже:

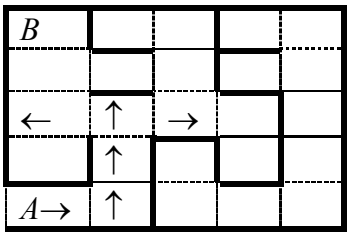
1. *find* = 0;{Признак поиска элемента}
2. *i* = 1; {Левая граница поддерева}
3. *j* = n; {Правая граница поддерева}
4. **while** *i* ≤ *j* **do begin**
5. *m* = $\lfloor \frac{i+j}{2} \rfloor$;{Корень текущего поддерева}
6. **if** *z* = a_m **then begin**
7. *find* = 1; {Элемент найден}
8. **break**;
9. **end**;
10. **else if** *z* > a_m **then** *i* = *m* + 1; {Новая левая граница}
11. **else** *j* = *m* – 1; {Новая правая граница}
12. **end**;
13. **if** *find* = 1 **then** Элемент найден;
14. **else** Элемент не найден.

Средняя сложность бинарного поиска среди элементов $a_1 < a_2 < \ldots < a_n$ сравнима с высотой двоичного дерева. В худшем случае искомый элемент может оказаться либо на последнем уровне, либо вообще не будет найден. Ранее мы уже установили, что количество уровней в дереве равно $\lceil \log_2(n+1) \rceil$. Значит сложность поиска является логарифмической $O(\log_2(n))$. Для реализации алгоритма перечисления перестановок часто используют метод поиска с возвращением, осуществляющего исчерпывающий поиск множества решений.

Метод поиска с возвращением постоянно пытается расширить частичное решение. Если расширение текущего частичного решения невозможно, то возвращаются к более короткому частичному решению и пытаются снова его продолжить. Идею поиска с возвращением легче всего понять в связи с задачей прохода через лабиринт: цель — попасть из некоторого заданного квадрата *A* в другой заданный квадрат *B* путем последовательного перемещения по квадратам. Трудность состоит в том, что существующие преграды запрещают некоторые перемещения. Один из способов прохода через лабиринт — это двигаться из начального квадрата в соответствии с двумя правилами:

• в каждом квадрате выбирать еще не исследованный путь;

• если из исследуемого в данный момент квадрата не ведут неисследованные пути, то нужно вернуться на один квадрат назад по последнему пройденному пути, по которому пришли в данный квадрат:



Первое правило говорит о том, как расширить исследуемый путь, если это возможно, а второе правило — о том, как выходить из тупика. В этом и состоит сущность поиска с возвращением: продолжать расширение исследуемого решения до тех пор, пока это возможно, и когда решение нельзя расширить, возвращаться по нему и пытаться сделать другой выбор на самом близком шаге, где имеется такая возможность.

Приведем описание общего алгоритма поиска с возвращением.

В самом общем случае полагаем, что решение задачи состоит из вектора (a_1, a_2, a_3, …) конечной, но неопределенной длины, удовлетворяющего некоторым ограничениям. Каждое a_i является элементом линейно упорядоченного

множества A_i. Таким образом, при исчерпывающем поиске должны рассматриваться элементы множества $A_1 \times A_2 \times A_3 \times \ldots \times A_i$, где $i = 0, 1, 2, ...$ в качестве возможных решений. В качестве исходного частичного решения примем пустой вектор () и на основе имеющихся ограничений выясним, какие элементы из A_1 являются кандидатами в a_1. Обозначим это подмножество кандидатов через S_1. В качестве a_1 выбираем наименьший элемент из S_1. В результате имеем частичное решение (a_1). В общем случае различные ограничения, описывающие решения, говорят о том, из какого подмножества S_i множества A_i выбираются кандидаты для расширения частичного решения от (a_1, a_2, a_3 , …, a_{k-1}) до (a_1, a_2, a_3, …, a_{k-1}, a_k). Если частичное решение (a_1, a_2, a_3 , …, a_{k-1}) не представляет возможности для выбора элемента a_k, то $S_k = \varnothing$; мы возвращаемся и выбираем новый элемент a_{k-1}. Если новый элемент a_{k-1} выбрать нельзя, возвращаемся еще дальше и выбираем новый элемент a_{k-2} и т.д.

Общий алгоритм поиска с возвращением в псевдокоде представлен ниже:

1. $S_1 = A_1$; \Выделить кандидатов
2. $k = 1$; \длина частичного решения
3. **while** $k > 0$ **do begin**
4. **while** $S_k \neq \varnothing$ **do begin**
5. $a_k \in S_k$ \ Расширить частичное решение
6. $S_k = S_k - \{a_k\}$;\ Удалить выбранного кандидата
7. **If** (a_1, a_2, a_3, …, a_{k-1}, a_k) – решение **then** Сохранить решение;
8. $k = k + 1$;\ Расширить частичное решение
9. $S_1 \subseteq A_1$; \Выделить кандидатов
10. **end**;
11. $k = k - 1$;\ Вернуться, уменьшить частичное решение
12. **end**.

Поиск с возвращением приводит к алгоритмам экспоненциальной сложности, так как из предположения, что все решения имеют длину не более n,

исследованию подлежат приблизительно $\prod_{k=1}^{n}|A_k|$ элементов. В предположении, что все $|A_k|$ = C— константа, получаем экспоненциальную сложность $\prod_{k=1}^{n}|A_k| = C^n$. Нужно помнить, что поиск с возвращением представляет собой только общий метод. Непосредственное его применение обычно ведет к алгоритмам, время работы которых недопустимо велико. Поэтому, чтобы метод был полезен, к нему нужно относиться как к схеме, с которой следует подходить к задаче. Схема должна быть хорошо приспособлена (часто это требует большой изобретательности) к конкретной задаче, так чтобы в результате алгоритм годился для практического использования.

Алгоритм поиска с возвращением можно использовать для порождения перестановок. Алгоритм решения этой задачи представлен ниже:

1. s_1 = 1; {Первый кандидат}
2. k = 1; {Длина частичного решения}
3. **while** k > 0 do begin
4. **while** s_k< n **do begin**
5. $a_k = s_k$; {Расширить частичное решение }
6. $s_k = s_k + 1$; {Удалить выбранного кандидата}
7. **while** $s_k < n$ **and not** flag(s_k) do $s_k = s_k + 1$;
8. **if** k = n **then** Перестановка (a_1, a_2, a_3, …, a_{k-1}, a_k) - решение;
9. **else begin**
10. $k = k + 1$; {Расширить частичное решение}
11. s_k = 1;
12. **while** $s_k < n$ **and not** flag(s_k) **do** $s_k = s_k + 1$;
13. **end**;{else}
14. **end**;
15. $k = k - 1$; {Вернуться, уменьшить частичное решение}
16. **end**;
17. **function** flag(s_k) {Поиск элемента s_k в перестановке (a_1, a_2, a_3, …, a_{k-1})}
18. flag= TRUE;
19. i=1;

20. **while** i < *k* **and** flag **do begin**

21. **if** $a = s_k$ **then** flag = FALSE;

22. $i = i + 1$;

23. **end**.

При адаптации общего алгоритма поиска с возвращением к задаче порождения перестановок мы не вычисляли и не хранили явно множество S_k. Для решения нашей задачи достаточно было хранить только наименьшее значение из S_k, то есть s_k, и следующее значение вычислять по мере необходимости. Проверка условия $S_k \neq \varnothing$ соответствует условию $s_k \leq n$; поскольку алгоритм устроен так, что перебор значений элемента s_k выполняется в порядке их возрастания. Поэтому неравенство $s_k > n$ соответствует пустому множеству кандидатов $S_k = \varnothing$.

Программа на языке Pascal реализации рассмотренного метода генерации перестановок приводится ниже:

```
1     program per1; {Порождение перестановок}
2     const
3       max_n = 20;
4     type
5       vector = array[1..max_n] of longint;
6     var
7       a: vector;
8       n: integer;
9       function flag(var a: vector; sk: longint; k: integer): boolean;
10    {Поиск элемента в перестановке a[1], a[2], …, a[k-1] }
11    var
12      i: integer;
13      yes: boolean;
14    begin
15      yes := true;
16      i := 1;
```

```
17            while (i < k) and yes do
18              begin
19              if a[i] = sk then
20                yes := false;
21              i := i + 1;
22            end;
23            flag := yes;
24          end;
25          procedure backtrack( var a: vector; n: integer);
26          {Генерация перестановок a[1], a[2], …, a[n] }
27          const
28           m: longint = 0;{Количество перестановок}
29          var
30            s: vector;p, i, k: integer;
31          begin
32            p := 0;
33            for i := 1 to n do
34              s[i] := 0;
35            s[1] := 1;
36            k := 1;
37            while k > 0 do
38            begin
39              while s[k] <= n do
40              begin
41                a[k] := s[k];
42                repeat{Поиск следующего кандидата на место a[k] }
43                  s[k] := s[k] + 1;
44                until(s[k] > n) or flag(a, s[k], k);
45                if k = n then begin {Перестановка найдена }
46                p := p + 1;
```

```
47              write( p, ')');
48              for i := 1 to n do
49                write( a[i], ' '); writeln;
50              end {if}
51            else
52            begin{ }
53              k := k + 1;
54              s[k] := 1;
55              while (s[k] <= n) and not flag(a, s[k], k) do
56                s[k] := s[k] + 1;
57            end{else}
58          end;{while}
59          k := k - 1;
60        end;{while}
61      end;{backtrack}
62      begin
63      {Основная программа}
64      readln(n);
65      backtrack(a, n);
66      end.
```

Метод решета, заключается в том, что рассматривается конечное множество и исключаются все элементы этого множества, не представляющие интереса. Этот метод является логическим дополнением к методу поиска с возвращением, который перечисляет все элементы множества, представляющие интерес.

Методы решена полезны прежде всего в теоретико-числовых вычислениях. Например, одним из наиболее известных методов отыскания простых числе является «решето Эратосфена». Это решето перечисляет составные (не простые) числа между N и N^2 для некоторого N.

Для случая $N = 6$ процесс имеет вид:

Шаг 0 (исходный):

6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

Шаг 1 (исключаются кратные 2):

~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~ 31 ~~32~~ 33 ~~34~~ 35 ~~36~~

Шаг 2 (Исключаются кратные 3):

~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ 35 ~~36~~

Шаг 0 (Исключаются кратные 5):

~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29 ~~30~~ 31 ~~32~~ ~~33~~ ~~34~~ ~~35~~ ~~36~~

Процесс прекращается после просеивания для наибольшего простого числа, меньшего N.

Методы решета могут быть полезны, если в множестве возможных решений элементы удобно занумерованы натуральными числами и хранятся в виде характеристического вектора. В этом векторе i–й разряд равен нулю, если i–й элемент не является решением, и равен единице в противном случае. Таким образом, на множествах, состоящих буквально из миллионов элементов, возможен поиск без явного порождения и исследования каждого элемента множества. Кроме того, в большинстве вычислительных устройств булевы операции можно производить параллельно над многими разрядами, обеспечивая тем самым значительную экономию времени.

5.3. Подсчет и оценивание.

Вся область комбинаторных алгоритмов насыщена задачами, которые требуют подсчета или оценивания числа элементов в конечном множестве или перечисления без повторений всех этих элементов в некотором специальном порядке. Следовательно, стандартные процедуры подсчета и оценивания являются необходимыми инструментами для каждого, кто имеет дело с комбинаторными алгоритмами.

Рассмотрим процедуры подсчета, которые применимы, когда исследуемое множество (или семейство множеств) имеет хорошую структуру. При ис-

пользовании такой структуры эти процедуры часто позволяют применять аналитические методы в противоположность вычислительному подходу поиска с возвращением.

Существует два вида задач подсчета. В более простом случае задается конкретное множество и требуется определить точно число элементов в нем. В этом случае можно применить методики исчерпывающего поиска. Более общим, однако, является случай, когда имеется семейство множеств, заданное некоторым параметром, и нас интересует мощность множеств как функция параметра. В этом случае редко требуются точные значения. Например, если известно, что мощность множества растет по некоторому параметру экспоненциально, то этого может оказаться достаточным, чтобы вообще отказаться от предложенного подхода к изучению проблемы, не занимаясь различными деталями. Процедуры асимптотического разложения, рекуррентных соотношений и производящих функций применяются ко второму типу задач. Рассмотрим только основные идеи этого подхода.

Известные нам комбинаторные формулы подсчета означают вычисление или определение свойств некоторой последовательности чисел, соответствующие той или иной задаче. Мы рассмотрим полезный инструмент для работы с последовательностями. Идея состоит в том, чтобы каждой числовой последовательности сопоставляется функция действительного или комплексного переменного, с тем, чтобы обычные операции над последовательностями соответствовали простым операциям над соответствующими функциями. Аналитические методы работы с функциями оказываются проще и эффективнее, чем непосредственные комбинаторные методы работы с последовательностями.

Пусть a_0, a_1, a_2, a_3, … - произвольная последовательность. Сопоставим последовательности функцию действительного или комплексного переменного:

$$A(x) = \sum_{k=0}^{\infty} a_k x^k \qquad (5.1)$$

Функция $A(x)$ называется производящей функцией последовательности $a_0, a_1, a_2, a_3, \ldots$. Как правило, поиск функции $A(x)$ по формуле (5.1) прямыми методами является сложной задачей. Однако заметим, что последовательность $\{a_k\}$ может быть восстановлена по $A(x)$. Выражение (5.1) является разложением $A(x)$ в ряд Тейлора в окрестности точки $x = 0$. Воспользуемся этим замечанием и приведем некоторые наиболее распространенные производящие функции и соответствующие им последовательности:

Производящая функция $A(x)$	Последовательность $\{a_k\}$
$\frac{1}{1-x} = \sum_{k=0}^{\infty} 1 \cdot x^k$	$a_k = 1, k \geq 0$ (5.2)
$\frac{1}{(1-x)^2} = \sum_{k=0}^{\infty} (k+1) \cdot x^k$	$a_k = k + 1, k \geq 0$
$ln \frac{1}{1-x} = \sum_{k=0}^{\infty} \frac{1}{k} \cdot x^k$	$a_0 = 0, a_k = \frac{1}{k}, k \geq 0$
$\ln (1+x) = \sum_{k=0}^{\infty} \frac{(-1)^{k+1}}{k} \cdot x^k$	$a_0 = 0, a_k = \frac{(-1)^{k+1}}{k}, k \geq 0$
$(1+x)^r = \sum_{k=0}^{\infty} \binom{r}{k} x^k$	$a_k = \binom{r}{k}, k \geq 0, r$ – любое
$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots,$	$a_k = \frac{1}{k!}, k \geq 0$
$e^{rx} = \sum_{k=0}^{\infty} \frac{(r)^k}{k!} \cdot x^k$	$a_k = \frac{1}{k!}, k \geq 0, r$ – любое
$1/(1\text{-}x)^r = \sum_{k=0}^{\infty} \binom{r+k-1}{k}$	$a_k = \binom{r+k-1}{k}, k \geq 0, r$ – любое

Простейшие производящие функции, представленные в таблице, используются как кирпичики для получения производящих функций более сложных последовательностей.

Пример 5.3.1. Показать что $\sum_{i=0}^{k} C_{r+1}^{i} = C_{r+k+1}^{k}$.

Заметим, что $\frac{1}{(1+x)^{r+1}} = \sum_{k=0}^{\infty} C_{r+k}^{k} x^k$ является производящей функцией последовательности $a_k = C_{r+k}^{k}$. Следовательно, искомая сумма будет равна

$\sum_{i=0}^{k} C_{r+1}^{i} = \sum_{i=0}^{k} a_i$.

Рассмотрим последовательность $b_k = \sum_{i=0}^{k} a_i$. Так как b_k и a_i связаны частичными суммами, то $B(x) = \frac{A(x)}{1-x} = \frac{1}{(1-x)^{r+2}}$. Последнее разложение из таблицы производящих функций позволяет записать последнее выражение в виде:

$B(x) = \frac{1}{(1-x)^{r+2}} = \sum_{i=0}^{k} C_{r+k+1}^{i} x^k$.

Отсюда $b_k = C_{r+k+1}^{k} = \sum_{i=0}^{k} a_i = \sum_{i=0}^{k} C_{k+i}^{i}$.

Рассмотрим последовательность $\{u_n\}$, n = 0, 1, 2, … . Говорят, что задано однородное линейное рекуррентное соотношение с постоянными коэффициентами порядка r, если для членов последовательности $\{u_n\}$ выполняется равенство:

$$u_{n+r} = c_1 u_{n+r-1} + c_2 u_{n+r-2} + \ldots + c_r u_n, \qquad (5.3)$$

где $c_0, c_1, \ldots, c_r$ - постоянные величины. Это выражение позволяет вычислить очередной член последовательности по предыдущим r членам. Ясно, что, задав начальные значения $u_1, u_2, \ldots, u_{r-1}$, можно последовательно определить все члены последовательности. Мы рассмотрим общий метод решения (т.е. поиска u_n как функции от n) рекуррентного соотношения. Для решения задачи достаточно найти производящую функцию

$$U(x) = \sum_{k=0}^{\infty} u_k x^k$$

последовательности $\{u_n\}$. Введем обозначение для полинома

$$K(x) = 1 - c_1 x - c_2 x^2 - \ldots - c_r x^r$$

и рассмотрим произведение $U(x)\,K(x) = C(x)$. Непосредственным умножением можно убедиться, что $C(x)$ - это полином, степень которого не превышает r -1,

так как коэффициенты при x^{n+r} (n = 0, 1, ...) в $U(x)K(x)$, согласно уравнению (5.3), удовлетворяют соотношению равны

$$u_{r+m} - (c_1u_{n+r-1} + c_2u_{n+r-2} + \ldots + c_ru_n) = 0.$$

Характеристическим полиномом соотношения (5.2) называется

$F(x) = (x - \alpha_1)^{e1}(x - \alpha_2)^{e2} \ldots (x - \alpha_r)^{er}$, где $e_1 + e_2 + \ldots + e_r = r$.

Сравнивая $K(x)$ и $F(x)$, получаем $K(x) = x^r F(1/x)$.

Отсюда $K(x) = x^r (1/x - \alpha_1)^{e1}(1/x - \alpha_2)^{e2} \ldots (1/x - \alpha_r)^{er} =$

$= (1 - \alpha_1x)^{e1}(1 - \alpha_2x)^{e2} \ldots (1 - \alpha_rx)^{er}$, $e_1 + e_2 + \ldots + e_r = r$.

Разложение на множители используем для представления

$$U(x) = \frac{C(x)}{K(x)}$$

в виде суммы простых дробей:

$$U(x) = \frac{C(x)}{K(x)} = \sum_{i=1}^{r} \sum_{n=1}^{e_i} \frac{\beta_{in}}{(1-\alpha_i x)^n}. \qquad (5.4)$$

Таким образом, $U(x)$ является суммой функций вида

$$\frac{\beta}{(1-\alpha x)^n} = \beta \sum_{k=0}^{\infty} \binom{n+k-1}{k} \alpha^k x^k.$$

Тогда выражение (5.4) принимает вид

$$U(x) = \frac{C(x)}{K(x)} = \sum_{i=1}^{r} \sum_{n=1}^{e_i} \beta_{in} \binom{n+k-1}{k} \alpha_i^k x^k. \qquad (5.5)$$

Данное уравнение является производящей функцией $U(x) = \sum_{n=0}^{\infty} u_n x^n$ последовательности$\{u_n\}$. Для определения необходимо найти коэффициент при в разложении (5.5).

Пример 5.3.2. Найти последовательность $\{u_n\}$, удовлетворяющую рекуррентному соотношению $u_{n+2} = 5u_{n+1} - 6u_n$, $u_0 = u_1 = 1$.

Решение.

$K(x) = 1 – 5x + 6x^2$, $U(x)\,K(x) = C(x)$, то есть

$(1 – 5x + 6x^2)(u_0 + u_1x + u_2x^2 + \ldots) = u_0 + (u_1 - 5u_0)x + (u_2 - 5u_1 + 6u_0)\,x^2 + \ldots =$

$= u_0 + (u_1 - 5u_0)x = 1 - 4x$. Таким образом, $C(x) = 1 - 4x$.

Характеристический полином $f(x) = x^2 – 5x + 6 = (x – 2)(x - 3)$. Отсюда,

$$U(x) = \frac{C(x)}{K(x)} = \frac{1-4x}{(1-2x)(1-3x)} = \frac{A}{1-2x} + \frac{B}{1-3x}.$$

Воспользовавшись методом неопределенных коэффициентов, имеем

$A = 2$, $B = -1$.

Тогда, принимая во внимание выражение (6.2), получаем:

$$U(x) = \frac{C(x)}{K(x)} = \frac{2}{1-2x} + \frac{-1}{1-3x} = 2\sum_{k=0}^{\infty} 2^k x^k - \sum_{k=0}^{\infty} 3^k x^k = \sum_{k=0}^{\infty}(2^{k+1} - 3^k)x^k.$$

С другой стороны, $U(x) = \sum_{k=0}^{\infty} u_k x^k$. Сравнивая коэффициенты при одинаковых степенях x^k, заключаем, что $u_k = 2^{k+1} - 3^k$.

6. КЛАССЫ СЛОЖНОСТИ.

Почти все изученные нами алгоритмы имеют **полиномиальное время работы** (polynomial-time algorithms): для входных данных размера n их время работы в наихудшем случае равно $O(n^k)$, где k – некоторая константа. Все ли задачи можно решить в течение полиномиального времени? Ответ отрицательный. Вообще говоря, о задачах, разрешимых с помощью алгоритмов с полиномиальным временем работы, говорят, как о легко разрешимых или простых, а о задачах, время работы которых превосходит полиномиальное, - как о трудно разрешимых или сложных.

Аргументы в пользу задач, разрешаемых алгоритмами за полиномиальное время:

1. На практике крайне редко встречаются задачи, решения которых выражаются полиномом высокой степени. Опыт показывает, что если для задачи становится известен алгоритм с полиномиальным временем работы, то зачастую впоследствии разрабатывается и более эффективный алгоритм.

2. Для многих приемлемых вычислительных моделей задача, решаемая за полиномиальное время, может быть решена в течение полиномиального времени и в другой модели. Например, задачи, разрешаемые за полиномиальное время с помощью последовательных машин с произвольной выборкой, являются разрешимыми за полиномиальное время и на абстрактных машинах

Тьюринга, и на параллельных компьютерах, если зависимость количества процессоров от объема входных данных описывается полиномиальной функцией.

3. Класс задач, решаемых за полиномиальное время, обладает свойством замкнутости, поскольку множество полиномов замкнуто относительно операций сложения, умножения и композиции. Например, если выход одного алгоритма с полиномиальным временем работы соединить со входом другой такой задачи, то получим полиномиальный составной алгоритм. Если же в другом алгоритме с полиномиальным временем работы фиксированное количество раз вызывается подпрограмма с полиномиальным временем работы, то время работы такого составного алгоритма также является полиномиальным.

Класс P состоит из задач, разрешимых в течение полиномиального времени работы. Точнее говоря- это задачи, которые можно решить за время $O(n^k)$, где k – некоторая константа, а n – размер входных данных задачи.

Класс NP состоит из задач, которые поддаются проверке в течение полиномиального времени. Имеется в виду, что если мы каким-то образом получаем «сертификат» решения, то в течение времени, полиномиальным образом зависящего от размера входных данных задачи, можно проверить корректность такого решения. Например, в задаче о гамильтоновом цикле с заданным ориентированным графом $G = (V, E)$ сертификат бы имел вид последовательности $(v_1, v_2, \ldots v_{|V|})$ из $|V|$ вершин. В течение полиномиального времени легко проверить, что $(v_i, v_{i+1}) \in E$.

Любая задача класса P принадлежит классу NP, поскольку принадлежность задачи классу P означает, что ее решение можно получить в течение полиномиального времени, даже не располагая сертификатом.

Задача принадлежит классу **NPC** (такие задачи называются NP-полными), если она принадлежит классу NP и является такой же «сложной», как и любая задача из класса NP.

Приведем три метода доказательства NP – полноты.

1. **Задачи принятия решения и задачи оптимизации**. Каждому допустимому решению задачи оптимизации сопоставляется некоторое

значение и находится допустимое решение с лучшим значением. Отметим, что NP –полнота применяется непосредственно не к задачам оптимизации, а к задачам принятия решения, в которых ответ может быть положительным или отрицательным. Между задачами оптимизации и задачами принятия решения существует связь. Наложив ограничение на оптимизируемое значение, поставленную задачу оптимизации можно свести к соответствующей задачи принятия решения. Например, для задачи оптимизации нахождения в ориентированном графе G пути из вершины v в вершину u с наименьшим количеством ребер, задача принятия решения формулируется следующим образом: существует ли для заданных исходных данных, в число которых входит направленный граф G, вершины v и u, и целое число k, путь из вершины v к вершине u, состоящий не более чем из k ребер.

2. **Приведение.**

1) Заданный экземпляр α задачи A с помощью алгоритма приведения с полиномиальным временем преобразуется в экземпляр β задачи B.

2) Запускается алгоритм, решающий экземпляр β задачи принятия решения B в течение полиномиального времени.

3) Ответ для экземпляра β используется в качестве ответа для экземпляра α.

3. **Первая NP-полная задача.** Поскольку метод приведения базируется на том, что для какой-то задачи заранее известна ее NP-полнота, то для доказательства NP-полноты различных задач необходима «первая» NP-полная задача. В качестве такой задачи часто используют задачу, в которой задана булева комбинационная схема, состоящая из элементов И, ИЛИ и НЕ. В задаче спрашивается, существует ли для этой схемы такой набор входных булевых величин, для которых будет выдано значение 1. Можно доказать, что рассматриваемая «первая» задача, действительно, является NP-полной.

7. УПРАЖНЕНИЯ.

Упражнения к разделу 1.

1) Какими еще параметрами, кроме занимаемой памяти, можно характеризовать алгоритм на практике?

2) Постройте диаграмму переходов машины Тьюринга для T_+ для сложения двух заданных натуральных чисел, $A = (1, \lambda)$.

3) Дан алфавит $A = (1, 0)$ и состояние $Q = \{q_1\}$. Построить машину Тьюринга, удаляющую 0.

4) Дан алфавит $A = (1)$. Построить машину Тьюринга для получения следующего натурального числа.

5) Построить нормальный алгоритм Маркова для перевода из четверичной системы счисления в двоичную систему счисления.

6) Построить нормальный алгоритм Маркова для вычисления функции $f(x) = x - 1$ в десятичной системе счисления.

7) Напишите конфигурацию МТ с внутренним состоянием q_4, в которой на ленте записано *klwmnso*, а головка обозревает *s*?

8) Проиллюстрируйте процесс работы алгоритма сортировки методом слияния для массива $A = \langle 4, 53, 64, 45, 13, 3, 85, 249 \rangle$.

9) Перепишите процедуру Merge так, чтобы в ней не использовался сигнальный элемент. Сигналом к остановке должен служить тот факт, что все элементы массива *L* или массива *R* скопированы обратно в массив *A*, после чего в этот массив копируются элементы, оставшиеся в непустом массиве.

Упражнения к разделу 2.

1) Методом математической индукции докажите, что если n равно степени двойки, то решением рекуррентного соотношения

$$T(n) = \begin{cases} 2, & \text{при } n = 2, \\ 2T\left(\frac{n}{2}\right) + n & \text{при } n = 2^k, k > 1. \end{cases}$$

является $T(n) = n\log_2 n$.

2) Пусть $f(n)$ и $g(n)$ – асимптотически неотрицательные функции. Докажите с помощью базового определения Θ–обозначений, что

$$max(f(n), g(n)) = \Theta(f(n) + g(n)).$$

3) Объясните почему выражение «время работы алгоритма А равно, как минимум, $O(n^2)$» не имеет смысла.

4) Докажите, что время работы алгоритма равно $\Theta(g(n))$ тогда и только тогда, когда время работы алгоритма в наихудшем случае равно $O(g(n))$, а время работы в наилучшем случае равно $\Omega(g(n))$.

5) Расположите приведенные ниже функции по скорости их асимптотического роста:

$\log_2 n$	n^2	$n!$
$\log_2 (\log_2 n)$	n	$(\log_2 n)!$
$2^{\sqrt{\log 2n}}$	$n \log_2 n$	$n^{\log_2 log_2 n}$
$(n + 1)!$	2^{2^n}	$\sqrt{log_2 n}$
$\ln n$	$n2^n$	1

6) С помощью основной теоремы найдите точные асимптотические границы следующих рекуррентных соотношений:

1 $T(1) = 1, T(n) = 2T(n/2) + 6n - 1, \forall n \geq 2.$

2. $T(1) = 4, T(n) = 2T(n/2) + 3n + 2, \forall n \geq 2.$

3. $T(1) = 1, T(n) = 6T(n/6) + 2n + 3, \forall n \geq 2.$

7) Дайте точную асимптотическую оценку для рекуррентного соотношения $T(1) = 2, T(n) = 4T(n/3) + 3n - 5, \forall n \geq 2.$.

8) Разработайте два алгоритма возведения числа в целую неотрицательную степень a^n , $n \in Z^+$ различающиеся по сложности. Обратите внимание, что вычисление a^{15} может потребовать только шесть операций умножения, а для вычисления a^{100} может потребоваться всего 14 операций умножения. Для разработанных алгоритмов определите вычислительную сложность и сравнительный анализ.

Упражнения к разделу 3.

1) Докажите, что если p - простое число и $0 < k < p$, то $\gcd(k, p) = 1$.

2) Разработайте эффективный алгоритм для операций деления β–битового числа на более короткое целое число и вычисления остатка от деления β-битового целого числа на более короткое целое число . Время работы алгоритма должно быть равным $O(\beta^2)$.

3) Вычислите величины (d, x, y), которые возвращаются при вызове процедуры Extended_Euclid(899,493).

4) Какие значения возвращает процедура Extended_Euclid(F_{k+1},F_k)? Докажите верность вашего ответа.

5) Докажите, что если x – нетривиальный квадратный корень из единицы по модулю n, то и gcd(x-1, n), и gcd(x+1, n) являются нетривиальными делителями.

6) Разработайте алгоритм нахождения наибольшего общего делителя двух целых чисел, используя итеративный алгоритм Евклида.

Упражнения к разделу 4.

1) Предположим, что в задаче о выборе процессов вместо того, чтобы выбирать процесс, который оканчивается раньше других, выбирается процесс, который начинается позже других и совместим со всеми ранее выбранными процессами. Опишите этот подход как жадный алгоритм и докажите, что он позволяет получить оптимальное решение.

2) Предположим, что имеется множество процессов, для которых нужно составить расписание при наличии большого количества ресурсов. Цель – включить в расписание все процессы, использовав при этом как можно меньше ресурсов. Сформулируйте эффективный жадный алгоритм, позволяющий определить, какой ресурс должен использоваться тем или иным процессом. (Эта задача также известна как задача раскрашивания интервального графа. Можно составить интервальный граф, вершины которого сопоставляются заданным процессам, а ребра соединяют несовместимые процессы. Минимальное количество цветов, необходимых для раскрашивания всех вершин таким образом, чтобы никакие две соединенные вершины не имели один и тот

же цвет, будет равно минимальному количеству ресурсов, необходимых для работы всех заданных процессов.)

3) Докажите, что дискретная задача о рюкзаке не обладает свойством жадного выбора: У нас имеется n предметов. Предмет под номером i имеет стоимость v_i руб. и вес w_i кг, где w_i и v_i - целые числа. Нужно унести вещи, суммарная стоимость которых была бы как можно большей, однако грузоподъемность рюкзака ограничивается W килограммами, где W - целая величина. Какие предметы следует взять с собой?

Упражнения к разделу 5.

1) Найти число k-мерных граней в n– мерном кубе.

2) Решить рекуррентное соотношение

$u_{n+2} - 4u_n + 1 + 3u_n = 0$, $u_0 = 8$, $u_1 = 10$.

3) Разработайте и реализуйте алгоритм, который генерирует все разбиения числа n в виде количества частей $c_1 \ldots c_n$, где $\alpha_1 \geq \alpha_2 \geq \ldots$ и $\alpha_1 + \alpha_2 + \ldots = n$ и $c_1 + 2c_2 + \ldots + nc_n = n$? Сгенерируйте их в солексном порядке, то есть лексикографическом порядке $c_n \ldots c_1$, который эквивалентен лексикографическому порядку соответствующих разбиений $\alpha_1\alpha_2 \ldots$.

4) Вычислите количество способов, которыми можно выбрать k точек из m точек на окружности так, чтобы среди них не было двух последовательных (соседних) точек.

5) Разработать и реализовать алгоритм разбиений целого числа n в обратном лексикографическом порядке: $\alpha_1 \geq \alpha_2 \geq \ldots \geq \alpha_m \geq 1$ с $\alpha_1 + \alpha_2 + \ldots + \alpha_m = n$ и $1 \leq m \leq n$ для $n \geq 1$? Значение α_0 устанавливается равным нулю.

8. СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

№	Автор	Наименование, издательство, год издания

1.	Матрос Д.Ш., Поднебесова Г.Б.	Теория алгоритмов: учебник. - М.: БИНОМ. Лаборатория знаний, 2008. – 202 с.
2.	Абрамов С.А.	Лекции о сложности алгоритмов. - М.:МЦНМО, 2009. – 256 с.
3.	Зюзьков В.М., Шелупанов А.А.	Математическая логика и теория алгоритмов: Учебное пособие для вузов.- 2-е изд. - М.: Горячая линия-Телеком, 2007. – 176 с.
4.	Стивен С. Скиена	Алгоритмы. Руководство по разработке. – 2-е изд.: Пер. с англ.- СПб.: БХВ-Петербург, 2011.-720 с.
5.	Кормен Т., Лейзерстон Ч., Ривест Р., Штайн К.	Алгоритмы: построение и анализ. 2-е изд.: Пер. с англ.-М.: Издательский дом «Вильямс», 2012. – 1296 с.
6.	Захарова Л.Е.	Алгоритмы дискретной математики. – М.: Московский государственный институт электроники и математики, 2010.
7.	Кнут Д.Э.	Искусство программирования. Том 4, А. Комбинаторные алгоритмы. Часть 1. – Издательство: Вильямс, 2012, с. 960.
8.	Кнут Д.Э.	Искусство программирования. Том 1, Основные алгоритмы. Издательство: Вильямс, 2012, с. 720.
9.	Новиков Ф.А.	Алгоритмы дискретной математики для программистов. – СПб.: Питер, 2001.
10.	Виленкин Н.Я., Виленкин А.Н., Виленкин П.А.	Комбинаторика. – М.: ФИМА, МЦНМО, 2006.

11.	Макконнелл Дж.	Анализ алгоритмов. Активный обучающий подход. - 3-е дополнительное изд.: Пер. с англ.- М.: ТЕХНОСФЕРА, 2013. – 415 с.