М. А. БИРЮКОВ
Т. В. ЕРМАКОВА
А. И. ПОПОВ

АЛГОРИТМИЧЕСКИЕ ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

УЧЕБНОЕ ПОСОБИЕ

САНКТ-ПЕТЕРБУРГ
2026

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ,
СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ
им. проф. М. А. БОНЧ-БРУЕВИЧА»
(СПбГУТ)

**М. А. Бирюков, Т. В. Ермакова,
А. И. Попов**

АЛГОРИТМИЧЕСКИЕ ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

УЧЕБНОЕ ПОСОБИЕ

**САНКТ-ПЕТЕРБУРГ
2026**

УДК 004.42(075.8)
ББК 32.973я73
Б64

Рецензенты:
доктор технических наук, профессор,
профессор кафедры автоматизированных систем специального назначения
Военной академии связи имени Маршала Советского Союза С. М. Будённого
И. Б. Саенко,
кандидат технических наук, доцент,
доцент кафедры сетей связи и передачи данных СПбГУТ
О. И. Пантюхин

Утверждено редакционно-издательским советом СПбГУТ
в качестве учебного пособия
Протокол № 5 от 22.05.2026

Бирюков, М. А.

Б64 Алгоритмические основы программной инженерии : учебное пособие / М. А. Бирюков, Т. В. Ермакова, А. И. Попов ; СПбГУТ. – Санкт-Петербург, 2026. – 186 с.
ISBN 978-5-89160-390-5

Материал издания структурирован по разделам, которые последовательно раскрывают базовые понятия алгоритмов и программного обеспечения, теоретические основы вычислимости и сложности, а также современные подходы к построению алгоритмов и средств защиты информации. Теоретическая база сопровождается практическими примерами, контрольными вопросами и упражнениями.

Может быть использовано при изучении дисциплин, связанных с теорией алгоритмов, программной инженерией и математическими основами информатики. Кроме того, может служить дополнительным источником знаний при освоении курсов, посвященных языкам программирования, проектированию программного обеспечения, информационной безопасности и математической логике.

Предназначено для обучающихся по направлению 09.03.04 «Программная инженерия», а также магистрантов и аспирантов, интересующихся проблемами алгоритмизации, формальных методов в информатике и инженерных подходов к созданию программных систем.

УДК 004.42(075.8)
ББК 32.973я73

ISBN 978-5-89160-390-5
© Бирюков М. А., Ермакова Т. В., Попов А. И., 2026,
© Федеральное государственное бюджетное образовательное учреждение высшего образования «Санкт-Петербургский государственный университет телекоммуникаций им. проф. М. А. Бонч-Бруевича», 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5

1. ВВЕДЕНИЕ В ПРОГРАММНУЮ ИНЖЕНЕРИЮ 6
   1.1. Общее введение в программную инженерию 6
   1.2. Разработка программного обеспечения на основе инженерных принципов 10
   1.3. История развития средств вычислительной техники 14
   1.4. Исторические аспекты теории алгоритмов 19
   Контрольные вопросы 24

2. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ и АСУ (ПО) 25
   2.1. Основные понятия: программа, программное обеспечение, программные средства 25
   2.2. Инженерный подход к конструированию программного обеспечения 28
   2.3. Классификация программного обеспечения по назначению 31
   2.4. Системное программное обеспечение 36
   2.5. Операционные системы: функции, примеры 39
   2.6. Прикладное программное обеспечение: классификация, функции, примеры .. 43
   2.7. Инструментальное программное обеспечение 46
   2.8. Классификация языков программирования, примеры 49
   2.9. Этапы подготовки исполняемого кода 53
   2.10. Транслятор, компоновщик, отладчик 55
   2.11. Интегрированные среды разработки, состав 58
   2.12. Автоматическая генерация кода. CASE-средства 60
   2.13. Использование UML для генерации кода 62
   Контрольные вопросы 64

3. ОСНОВЫ ТЕОРИИ АЛГОРИТМОВ 65
   3.1. Алгоритм: виды и формы представления 65
   3.2. Характеристики и свойства алгоритмов 68
   3.3. Информация и алгоритмы ее обработки 71
   3.4. Определение алгоритма по Колмогорову и Маркову 75
   3.5. Алгоритм Евклида и великая теорема Ферма 77
   3.6. Алгоритмически неразрешимые задачи 81
   3.7. Теорема Геделя о неполноте 83
   3.8. Проблема останова машины Тьюринга 85
   3.9. Прочие нерешаемые задачи (совершенные числа, задача Поста и др.) 88
   3.10. Теория вычислимости и алгоритмическая сложность 92
   3.11. Случайные процессы и алгоритмы 94
   Контрольные вопросы 97

4. МАТЕМАТИЧЕСКИЕ ФОРМАЛИЗМЫ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ 98
   4.1. Основные парадигмы программирования 98
   4.2. Сравнительный анализ моделей 99
   4.3. Методы оценки вычислительных моделей 101
   Контрольные вопросы 103

5. СОВРЕМЕННЫЕ АЛГОРИТМЫ И ТЕОРИЯ СВЯЗИ 104
   5.1. Компьютерные сети и распределенные вычисления 104
   5.2. Оптимизация алгоритмов и программного обеспечения 106

5.3. Алгоритмы и теория простых чисел (RSA, Ферма, Евклид) 109
5.4. Вероятностный тест Миллера – Рабина 111
5.5. Алгоритм Рабина – Карпа 112
5.6. Генетические и муравьиные алгоритмы 114
5.7. Новые технологии и новые проблемы 116
Контрольные вопросы 117

6. МНОЖЕСТВА 118
6.1 Определение множеств 118
6.2 Теоретико-множественные операции 124
6.3. Комбинаторика и множества: примеры, упражнения 126
Контрольные вопросы 127

7. ФОРМАЛЬНЫЕ ТЕОРИИ И ЯЗЫКИ 128
7.1. Построение формальной системы 128
7.2. Формальные языки и объекты информатики 130
7.3. Алфавиты, слова, языки 132
7.4. Алгоритмические проблемы: принадлежность, выполнимость, оптимизация . 134
7.5. Сложность по Колмогорову и ее применение 135
Контрольные вопросы 137

8. КОНЕЧНЫЕ АВТОМАТЫ 138
8.1. Конечные автоматы как модель вычислений 138
8.2. Распознавание языков 141
Контрольные вопросы 143

9. МАШИНЫ ТЬЮРИНГА 145
9.1. Структура машины Тьюринга 145
9.2. Такт работы и пример программы 146
Контрольные вопросы 148

10. ТЕОРИЯ ВЫЧИСЛИМОСТИ 149
10.1. Метод диагонализации 149
10.2. Метод сводимости 151
Контрольные вопросы 153

11. ТЕОРИЯ СЛОЖНОСТИ 154
11.1. Классы сложности P, NP, NPC 154
11.2. Сложность по времени и памяти 159
11.3. Многоленточная машина Тьюринга 162
Контрольные вопросы 165

12. КОМПЬЮТЕРНЫЕ СЕТИ И БЕЗОПАСНОСТЬ 166
12.1. Основные понятия безопасности 166
12.2. Системный подход к защите 170
12.3. Базовые технологии безопасности 174
12.4. Алгоритмы шифрования и защита информации 179
Контрольные вопросы 183

СПИСОК РЕКОМЕНДУЕМЫХ ИСТОЧНИКОВ 184

ВВЕДЕНИЕ

Учебное пособие «Алгоритмические основы программной инженерии» посвящено фундаментальным принципам построения и анализа алгоритмов, а также их применению в программной инженерии. Оно объединяет теоретическую и практическую составляющие информатики, демонстрируя, каким образом строгие математические методы становятся основой для разработки надежного, масштабируемого и безопасного программного обеспечения.

Алгоритмы – это универсальный язык, на котором описываются все вычислительные процессы. Их исследование позволило сформулировать ключевые модели вычислений, определить границы алгоритмически разрешимого и заложить основу для построения современных вычислительных систем. Программная инженерия, в свою очередь, оформилась как дисциплина, обеспечивающая практическое воплощение алгоритмических идей в виде комплексных программных продуктов, соответствующих требованиям качества, безопасности и долговечности.

Учебное пособие построено так, чтобы шаг за шагом провести читателя от общих принципов разработки программного обеспечения к строгим моделям алгоритмов и вычислительных процессов. Вначале рассматриваются основы программной инженерии и программного обеспечения, далее излагаются ключевые понятия теории алгоритмов и вычислимости. Особое внимание уделено современным алгоритмам, применяемым в компьютерных сетях, криптографии и области искусственного интеллекта. Отдельные главы посвящены теории множеств, формальным языкам, конечным автоматам и машинам Тьюринга – фундаментальным моделям, без которых невозможно понимание природы вычислений.

Завершающие разделы раскрывают вопросы вычислительной сложности и компьютерной безопасности, показывая, как теоретические методы находят применение в практических задачах: анализе, защите информации и построении надежных программных систем.

Данное пособие формирует целостное представление о взаимосвязи алгоритмических основ и инженерных подходов, которые вместе составляют ядро современной цифровой цивилизации. Оно адресовано студентам, изучающим информатику и программную инженерию, а также всем, кто стремится понять фундаментальные механизмы, лежащие в основе программного обеспечения и вычислительных процессов.

1. ВВЕДЕНИЕ В ПРОГРАММНУЮ ИНЖЕНЕРИЮ

1.1. Общее введение в программную инженерию

Понятие программной инженерии

Программная инженерия (от англ. Software Engineering) – это область науки и практики, объединяющая методы разработки программных систем, принципы инженерного проектирования и организацию процессов, направленных на создание качественного программного обеспечения. В отличие от программирования, которое сосредоточено на написании кода, программная инженерия рассматривает весь жизненный цикл программного продукта – от анализа требований до сопровождения и утилизации.

Главное назначение программной инженерии – создание надежных, масштабируемых, безопасных и сопровождаемых программных решений, которые отвечают требованиям пользователей, учитывают ограничения среды эксплуатации и могут быть модернизированы.

Возникновение дисциплины было обусловлено ростом сложности программных систем. Если в начале развития вычислительной техники программы представляли собой десятки или сотни строк кода, то к концу 1960-х гг. появились проекты, включающие сотни тысяч операторов. Их сопровождение становилось невозможным без систематического подхода. В этот период было осознано, что программные системы должны проектироваться так же, как проектируются мосты, заводы или самолеты – с учетом архитектуры, рисков и требований безопасности.

Цели и задачи программной инженерии

Можно сформулировать основные цели дисциплины.

- Повышение качества программного обеспечения. Под качеством понимается корректность работы, устойчивость к ошибкам, удобство использования, совместимость с другими системами.
- Снижение стоимости и сроков разработки. Инженерный подход позволяет использовать повторно готовые решения, применять стандартизированные процессы и автоматизировать рутинные операции.
- Гарантия масштабируемости и сопровождения. Программные системы должны быть пригодны к развитию и адаптации под новые задачи.

Для достижения этих целей решается ряд конкретных задач:

– сбор, анализ и формализация требований пользователей;

– проектирование архитектуры программного продукта;

– определение используемых технологий, языков программирования, инструментов;

– организация командной работы разработчиков;

– построение системы контроля качества: тестирование, статический анализ, рецензирование кода;

– документирование продукта, обеспечение сопровождения и обновления;

– управление рисками (техническими, организационными, экономическими).

Отличие программной инженерии от программирования

Программирование – это процесс написания исходного кода. Оно решает локальные задачи, связанные с конкретной функциональностью.

Программная инженерия включает в себя гораздо более широкий спектр деятельности:

– анализ требований заказчика;
– разработку архитектурных решений;
– выбор методологии и инструментов;
– реализацию (программирование как часть процесса);
– тестирование и верификацию;
– внедрение и эксплуатацию;
– сопровождение и развитие продукта.

Пример

Программист может написать калькулятор для конкретного компьютера.

Инженер по ПО спроектирует систему расчетов, которая будет работать на разных платформах, поддерживать различные виды вычислений, хранить историю операций, иметь удобный интерфейс и документацию, а также будет протестирована на нагрузку и устойчивость к ошибкам.

Таким образом, программная инженерия – это управляемое и документированное проектирование программного продукта, тогда как программирование – лишь один из этапов этого процесса.

Исторические этапы развития программной инженерии

1. 1950–1960-е гг.

Появление первых языков высокого уровня (Fortran, COBOL). Программы начали использоваться в бизнесе, обороне, науке. Уже тогда стали очевидными трудности масштабирования и сопровождения кода.

2. 1970-е гг.

Развитие методов структурного программирования (Дейкстра, Хоар), акцент на модульности и читаемости кода. Формирование каскадной модели жизненного цикла программного обеспечения.

3. 1980-е гг.

Появление объектно-ориентированного подхода (Smalltalk, C++), акцент на повторное использование кода и моделирование реальных объектов.

4. 1990-е гг.

Распространение методов системного анализа и проектирования. Разработка спиральной модели жизненного цикла (Б. Боэм). Активное использование CASE-средств (Computer-Aided Software Engineering) для автоматизации анализа и проектирования.

5. 2000-е гг.

Популяризация гибких методологий разработки (Agile, Scrum, Kanban). Переход от строгих каскадных моделей к итеративным и инкрементальным процессам.

6. 2010-е гг. и современность.

Развитие DevOps, микросервисной архитектуры, контейнеризации (Docker, Kubernetes), массовое внедрение облачных технологий, методов машинного обучения и искусственного интеллекта.

Основные принципы программной инженерии

- Системный подход. Программное обеспечение – это система, включающая множество компонентов, взаимодействующих между собой.
- Модульность и абстракция. Сложные системы необходимо разделять на относительно независимые модули, что облегчает их разработку, тестирование и сопровождение.
- Повторное использование решений. Использование библиотек, шаблонов проектирования, готовых фреймворков и платформ.
- Эволюционное развитие. Программное обеспечение развивается итерациями. Новые версии должны улучшать существующую систему, сохраняя совместимость.
- Автоматизация. Использование средств контроля версий, систем сборки, CI/CD-пайплайнов, автоматизированного тестирования.
- Управляемость и измеримость. Любой процесс разработки должен поддаваться контролю: сроки, качество, ресурсы должны измеряться с помощью метрик.

Практическая ценность

Программная инженерия играет ключевую роль в создании современных программных систем, поскольку она ориентирована не только на разработку кода, но и на обеспечение качества и надежности конечного продукта. Применение инженерных подходов позволяет достигать предсказуемости сроков выполнения проектов и затрат на их реализацию, что имеет решающее значение для эффективного управления разработкой.

Кроме того, системное использование методов и стандартов программной инженерии способствует повышению качества программных решений, снижает вероятность возникновения критических ошибок и обеспечивает необходимый уровень безопасности при эксплуатации программных

систем. Еще одним важным преимуществом является возможность масштабирования программных продуктов и их долгосрочной поддержки, что делает разработанные системы устойчивыми к изменяющимся условиям и пригодными для развития в будущем.

Таким образом, программная инженерия формирует основу для создания надежных, безопасных и сопровождаемых программных систем, способных эффективно функционировать на протяжении всего жизненного цикла.

Примеры из практики

Применение методов программной инженерии особенно важно в тех областях, где от корректности работы программного обеспечения зависит безопасность и надежность критически важных процессов. Так, в медицине инженерные подходы используются при разработке программных комплексов для томографов, систем жизнеобеспечения и других медицинских устройств, где даже незначительная ошибка может привести к угрозе жизни пациента.

В авиации программная инженерия находит применение при создании систем управления полетами. Здесь используется строгая методология проектирования и формальные методы верификации, позволяющие гарантировать корректность функционирования программных модулей и исключить вероятность критических сбоев в условиях эксплуатации.

В банковской сфере инженерные подходы обеспечивают разработку распределенных систем обработки транзакций, где необходимы высокая производительность, защита от несанкционированного доступа и устойчивость к отказам. Благодаря этому удается поддерживать бесперебойную работу финансовых сервисов и обеспечивать доверие пользователей.

Практика показывает, что программная инженерия является необходимым условием для создания надежных систем в сферах, где цена ошибки чрезвычайно высока и требуется максимально высокий уровень качества и безопасности.

Пример применения

Примером, ярко демонстрирующим значение программной инженерии, может служить задача разработки мобильного приложения для доставки еды.

Если отказаться от инженерного подхода, процесс сведется к работе одного программиста, пишущего код, например, только для платформы Android. Такое приложение может функционировать на базовом уровне, однако при увеличении количества пользователей начнет давать сбои. Отсутствие продуманной архитектуры приведет к невозможности масштабирования, а недостаточно изученные вопросы безопасности – к уязвимости

платежных транзакций. Подобное решение окажется непригодным для долгосрочной эксплуатации и будет быстро вытеснено более надежными конкурентами.

Совершенно иным оказывается результат при использовании принципов программной инженерии. На стадии проектирования формируется архитектура с четким разделением на модули: регистрация пользователей, управление базой данных, обработка заказов, система оплаты. Продумывается интерфейс, ориентированный на удобство конечного пользователя. В ходе реализации выполняются нагрузочные и интеграционные тестирования, обеспечивается защита платежных операций, а также закладывается поддержка работы приложения на различных платформах. В результате создается надежный и масштабируемый продукт, готовый к промышленной эксплуатации и дальнейшему развитию.

Данный пример наглядно подтверждает, что программная инженерия – это не абстрактная теоретическая область, а фундаментальная основа современной цифровой цивилизации. Именно благодаря инженерным подходам стали возможны сложные операционные системы, устойчивые банковские сервисы, интернет-магазины, системы искусственного интеллекта и средства кибербезопасности. Без строгих методологий проектирования и организации разработки прогресс в области информационных технологий был бы невозможен.

1.2. Разработка программного обеспечения на основе инженерных принципов

Разработка программного обеспечения представляет собой комплексный процесс, в котором задействованы специалисты разных профилей: аналитики, архитекторы, программисты, тестировщики и инженеры по качеству. В отличие от ремесленного программирования, где основное внимание уделяется лишь написанию кода, инженерный подход предполагает формализацию, стандартизацию и управление всем жизненным циклом программного продукта. Целью разработки является не только создание функционирующей программы, но и обеспечение ее надежности, масштабируемости, сопровождаемости и безопасности.

Ключевым понятием инженерной парадигмы *является жизненный цикл программного обеспечения (ЖЦ)*. Под ним понимают совокупность этапов, через которые проходит программный продукт от первоначального замысла до утилизации. Классически в состав жизненного цикла включают анализ требований, проектирование архитектуры, реализацию (программирование), тестирование и отладку, внедрение и эксплуатацию, а также

сопровождение и модернизацию. На практике эти стадии редко выполняются строго последовательно: в современных условиях они часто пересекаются и повторяются, что отражает итеративный характер разработки.

В зависимости от особенностей проекта, состава команды и ограничений внешней среды применяются различные *модели жизненного цикла разработки программного обеспечения.*

Наиболее ранней и традиционной считается *каскадная модель (Waterfall)*. Она предполагает строго последовательное выполнение этапов и используется в тех случаях, когда требования заранее четко определены и не подлежат изменениям. Такой подход характерен для разработки военных и медицинских систем, где высока цена ошибок. Главным недостатком каскадной модели является ее низкая гибкость: внесение изменений на поздних стадиях сопряжено с большими затратами.

В противоположность этому, *итеративная модель* основана на поэтапной разработке программного продукта, разделенной на отдельные версии или итерации. Каждый цикл разработки позволяет вносить корректировки, учитывая изменения требований или условий эксплуатации. Эта модель широко применяется в коммерческих проектах, где важно быстро реагировать на запросы рынка.

Интересным развитием итеративного подхода стала *спиральная модель*, предложенная Б. Боэмом. Она сочетает элементы каскадного и итеративного подходов, уделяя особое внимание управлению рисками. Каждая итерация в спиральной модели включает этап анализа возможных рисков, что делает ее особенно подходящей для сложных и долгосрочных проектов.

Наконец, в современных условиях наибольшее распространение получили *гибкие (Agile) методы разработки*, такие как Scrum и Kanban. Они строятся на коротких итерациях (спринтах), постоянном взаимодействии с заказчиком и быстрой адаптации к изменениям требований. Agile-подходы получили широкое распространение в веб-разработке, создании мобильных приложений и стартапах, где ключевым фактором является скорость реакции на запросы пользователей и рынка (табл. 1.1).

Таблица 1.1

Сравнение моделей разработки ПО

Модель	*Особенности*	*Преимущества*	*Недостатки*	*Примеры применения*
Каскадная	Последовательные этапы	Простота, предсказуемость	Низкая гибкость, дорого исправлять ошибки	Системы с жесткими требованиями
Итеративная	Разработка частями	Гибкость, контроль качества	Сложнее управлять	Коммерческие проекты

Окончание табл. 1.1

Модель	*Особенности*	*Преимущества*	*Недостатки*	*Примеры применения*
Спиральная	Итерации + управление рисками	Хорошо подходит для крупных проектов	Высокие затраты	Банковские и государственные системы
Agile (Scrum)	Короткие спринты, адаптивность	Быстрая реакция на изменения	Требует дисциплины команды	Веб-приложения, стартапы

Таким образом, разработка программного обеспечения в инженерной парадигме основывается на четком понимании жизненного цикла, выборе соответствующей модели и строгом следовании принципам проектирования и управления. Это позволяет создавать не просто работающие программы, но надежные, масштабируемые и долговечные системы, соответствующие современным требованиям качества.

Инженерные принципы разработки программного обеспечения

Инженерный подход к созданию программного обеспечения опирается на ряд фундаментальных принципов, которые отличают его от ремесленного программирования и позволяют создавать масштабируемые и надежные системы.

Прежде всего, важна *системность*: программный продукт рассматривается как целостная система, состоящая из взаимосвязанных компонентов, а не как набор отдельных функций. Такой взгляд обеспечивает согласованность решений на всех уровнях разработки.

Не менее значимым является принцип *архитектурности*, предполагающий предварительное проектирование структуры программного продукта, определение модулей, слоев и способов их взаимодействия. Архитектурный подход позволяет предвидеть возможные узкие места и обеспечивает масштабируемость системы.

Ключевую роль играет *стандартизация*. Использование международных стандартов качества, таких как ISO/IEC 25010 или ISO/IEC 12207, позволяет унифицировать процессы, задавать четкие критерии оценки программного обеспечения и обеспечивать сопоставимость результатов в различных проектах.

Важным элементом инженерного подхода является *документированность*. Она включает создание спецификаций, диаграмм, инструкций и руководств, которые позволяют новым членам команды быстро включаться в работу и обеспечивают поддержку продукта на протяжении всего жизненного цикла.

Особое внимание уделяется *повторному использованию решений*. Применение готовых библиотек, API и шаблонов проектирования позволяет

сократить затраты времени и ресурсов, а также повысить надежность, опираясь на проверенные практикой компоненты.

Принцип *верификации и валидации* гарантирует соответствие продукта заявленным требованиям и корректность реализации. Эти процедуры позволяют минимизировать риск критических ошибок, которые могут проявиться уже в процессе эксплуатации.

Наконец, современный инженерный подход невозможен без *автоматизации процессов*. Сюда входят непрерывная интеграция и доставка (CI/CD), автоматизированное тестирование, использование систем контроля версий. Автоматизация снижает вероятность человеческих ошибок и ускоряет цикл разработки.

Применение перечисленных принципов можно проиллюстрировать на примере создания системы бронирования авиабилетов. Если ограничиться только написанием кода интерфейса и работы с базой данных, система может функционировать, но при высоких нагрузках будет давать сбои, а данные начнут теряться. Применение же инженерного подхода предполагает проведение анализа требований (включая нагрузку, безопасность и интеграцию с внешними сервисами), проектирование архитектуры (с выделением модулей пользователей, билетов, платежей и интеграции с API авиакомпаний), выбор технологического стека, настройку CI/CD для автоматической сборки и тестирования, проведение нагрузочных испытаний и написание документации для администраторов и пользователей. В результате создается масштабируемая система, готовая к эксплуатации миллионами пользователей по всему миру.

Практическая значимость инженерных принципов заключается в том, что они позволяют существенно снизить возможность провала проекта за счет управления рисками и четкой архитектуры, обеспечить надежность работы систем в критических областях (таких как авиация или банковская сфера), сократить затраты благодаря повторному использованию компонентов и автоматизации, а также гарантировать долговечность программных решений, которые могут развиваться и поддерживаться десятилетиями.

Таким образом, инженерные принципы являются фундаментом современной разработки программного обеспечения, гарантируя ее качество, устойчивость и соответствие требованиям быстро меняющегося цифрового мира (табл. 1.2).

Таблица 1.2

Отличие «ремесленного» и инженерного подхода

Критерий	*Ремесленный подход*	*Инженерный подход*
Основная цель	Быстро написать код	Создать качественный продукт
Документирование	Отсутствует или минимальное	Полная спецификация, диаграммы

Окончание табл. 1.2

Критерий	*Ремесленный подход*	*Инженерный подход*
Масштабируемость	Отсутствует	Закладывается на стадии архитектуры
Тестирование	Минимальное	Системное, многоуровневое
Использование стандартов	Нет	Применяются международные стандарты
Жизненный цикл	Неопределенный	Управляемый и измеримый

Разработка программного обеспечения на основе инженерных принципов – это управляемый процесс, основанный на стандартах, архитектурных решениях, методологиях и автоматизации. Такой подход позволяет создавать ПО, способное работать в условиях высокой нагрузки, интегрироваться с другими системами и сопровождаться десятилетиями.

1.3. История развития средств вычислительной техники

История вычислительной техники представляет собой процесс постепенной эволюции средств автоматизации вычислений от механических устройств до современных суперкомпьютеров, облачных и распределенных систем. Этот процесс обусловлен развитием естественных наук, инженерной мысли, микроэлектроники и потребностями общества в обработке все возрастающих объемов информации.

Развитие вычислительной техники неразрывно связано с формированием программной инженерии: новые поколения вычислительных средств требовали новых методов организации программирования, а повышение сложности программных систем стимулировало совершенствование аппаратуры (табл. 1.1).

Ранний этап развития

1. *Механические устройства для вычислений.*

Первые счетные инструменты появились в глубокой древности (абак, счеты). В XVII в. были созданы первые механические калькуляторы: «Паскалево колесо» (1642 г., Блез Паскаль) – устройство для сложения и вычитания; арифмометр Лейбница (1673 г., Готфрид Вильгельм Лейбниц) – позволял выполнять умножение и деление.

В XIX в. Чарльз Бэббидж разработал проект аналитической машины – концепцию универсального вычислителя, включавшего память, арифметическое устройство и устройство управления. Идеи Бэббиджа предвосхитили

принципы современных компьютеров. Ада Лавлейс составила для этой машины алгоритмы, фактически став автором первой программы.

2. *Электромеханический этап.*

Конец XIX в. ознаменовался появлением перфокарточных машин Германна Холлерита, примененных для обработки переписи населения в США (1890 г.). Эти устройства заложили основы автоматизированной обработки данных и дали начало компании IBM. В первой половине XX в. релейные вычислительные устройства использовались для инженерных расчетов и криптоанализа.

3. *Этап электронных вычислительных машин.*

С середины XX в. развитие вычислительной техники принято классифицировать по поколениям, каждое из которых определяется элементной базой (табл. 1.3).

Таблица 1.3

Поколения ЭВМ

Поколение	*Время*	*Элементная база*	*Характерные особенности*	*Примеры*
I	1940–1950-е	Электронные лампы	Низкая надежность, большие размеры, программирование на машинных кодах	ENIAC (США), МЭСМ (СССР)
II	1950–1960-е	Транзисторы	Повышение надежности и скорости, появление ассемблеров и первых языков высокого уровня	IBM 7090, БЭСМ-6
III	1960–1970-е	Интегральные схемы	Массовое распространение ЭВМ, многозадачность, операционные системы	IBM System/360
IV	1970–1990-е	Большие и сверхбольшие интегральные схемы (БИС, СБИС)	Появление персональных компьютеров, графических интерфейсов, сетевых технологий	IBM PC, Apple II
V	1990-е – наст. время	СБИС, многоядерные процессоры, специализированные ускорители	Суперкомпьютеры, облачные вычисления, мобильные устройства, искусственный интеллект	Cray, дата-центры Google, iPhone

Архитектурные решения

Развитие вычислительной техники сопровождалось формированием различных архитектурных решений, определяющих организацию вычислительного процесса и взаимодействие его основных компонентов. Каждая

архитектура отражает определенный этап развития информатики и отвечает требованиям своего времени, сохраняя при этом актуальность в тех или иных областях применения.

Одним из наиболее значимых достижений XX в. стала *архитектура фон Неймана*, предложенная в 1945 г. Ее принципиальной особенностью является хранение программ и данных в одной памяти. Такое решение сделало вычислительные системы универсальными: одна и та же машина могла выполнять различные задачи, загружая в память новые программы. Именно универсальность и гибкость определили долгую жизнь этой архитектуры, которая до настоящего времени сохраняет статус основной модели для большинства персональных компьютеров и серверов.

Альтернативой этому подходу стала *гарвардская архитектура*, в которой память для команд и память для данных разделены. Это позволяет ускорить выполнение программ, так как процессор может одновременно считывать команду и данные, избегая узких мест, присущих архитектуре фон Неймана. Гарвардская архитектура широко применяется в микроконтроллерах и встроенных системах, где критически важны высокая производительность при ограниченных ресурсах и надежность работы.

С ростом потребности в вычислительной мощности возникли *параллельные и многопроцессорные архитектуры*. Уже в 1970–1980-е гг. появились векторные вычислительные машины и суперкомпьютеры, такие как Cray-1, которые могли обрабатывать большие массивы данных за счет параллельного выполнения операций. В начале XXI в. параллелизм стал неотъемлемым свойством массовых вычислительных устройств: многоядерные процессоры вошли в стандартную комплектацию персональных компьютеров и серверов, обеспечивая одновременное выполнение множества потоков и программ.

Современный этап развития связан с переходом от отдельных машин к сетям вычислителей. *Распределенные системы и облачные технологии* позволяют объединять сотни и тысячи узлов в единую инфраструктуру, обеспечивающую доступ к вычислительным ресурсам миллионов пользователей одновременно. Центры обработки данных, лежащие в основе облачных сервисов, стали новой формой организации вычислений, ориентированной на масштабируемость, гибкость и доступность.

Таким образом, архитектурные решения – от модели фон Неймана до облачных вычислений – отражают эволюцию способов организации вычислительных процессов. Эта эволюция продиктована как ростом сложности задач, так и изменением требований к скорости, масштабируемости и эффективности работы вычислительных систем.

Эволюция вычислительных средств

Механические калькуляторы

↓

Электромеханические устройства (перфокарты, реле)

↓

ЭВМ I поколения (лампы)

↓

ЭВМ II поколения (транзисторы)

↓

ЭВМ III поколения (интегральные схемы)

↓

ЭВМ IV поколения (микропроцессоры, персональные компьютеры)

↓

ЭВМ V поколения (суперкомпьютеры, облака, искусственный интеллект)

Развитие программного обеспечения в контексте вычислительной техники

- *I поколение.*

Программирование велось непосредственно в машинных кодах и при помощи кабельных соединений.

- *II поколение.*

Появились ассемблеры, первые компиляторы, языки высокого уровня (Fortran, COBOL).

- *III поколение.*

Развитие операционных систем, многозадачности, системного программного обеспечения.

- *IV поколение.*

Распространение графических интерфейсов, офисных пакетов, систем управления базами данных.

- *V поколение.*

Появление глобальной сети Интернет, мобильных приложений, распределенных систем, технологий искусственного интеллекта.

Ключевые вехи

- *ENIAC (1946, США).*

Первая электронная цифровая машина. Занимала более 150 м^2, содержала 18 тысяч ламп и выполняла около 5000 операций в секунду.

- *МЭСМ (1951, СССР).*

Первая советская ЭВМ, созданная под руководством С. А. Лебедева.

- *IBM System/360 (1964).*

Универсальная серия машин, обеспечивавшая совместимость программ и периферии.

- *БЭСМ-6 (1967, СССР).*

Одна из самых производительных машин Европы того времени.

- *IBM PC (1981).* Персональный компьютер, определивший дальнейшее направление развития отрасли.
- *Cray-1 (1976).*

Суперкомпьютер, предназначенный для научных расчетов и моделирования.

Современный этап

Современные вычислительные системы можно разделить на несколько направлений:

– *суперкомпьютеры* для научных исследований, обработки климатических моделей, моделирования ядерных процессов;

– *облачные центры обработки данных*, предоставляющие вычислительные ресурсы «как услугу»;

– *мобильные устройства*, обеспечивающие повсеместный доступ к вычислительным ресурсам;

– *интернет вещей (IoT)*, встраивающий вычисления в повседневные предметы;

– *квантовые компьютеры*, находящиеся на стадии активных исследований.

Значение для общества

Развитие вычислительной техники оказало фундаментальное влияние:

– на *экономику* – автоматизация банковских операций, электронная коммерция, цифровые валюты;

– *науку* – моделирование процессов в физике, биологии, медицине;

– *образование* – развитие электронных библиотек, дистанционного обучения;

– *социальные коммуникации* – глобальная сеть, социальные медиа, системы мгновенного обмена информацией.

Ошибки в работе вычислительных систем приобрели критическую значимость: сбои в энергетике, транспорте или финансовых системах могут приводить к катастрофическим последствиям.

История развития вычислительной техники – это путь от механических калькуляторов к интеллектуальным распределенным системам. Каждый новый этап эволюции определял не только технический прогресс, но и развитие методов программирования, что в конечном счете привело к формированию программной инженерии как научной дисциплины.

1.4. Исторические аспекты теории алгоритмов

Алгоритмы являются фундаментальной основой информатики и программной инженерии. Каждая программа представляет собой реализацию алгоритма, а эффективность работы вычислительной системы во многом определяется качеством выбранных алгоритмических решений.

Развитие теории алгоритмов проходило несколько этапов: от практических правил арифметики в древности до формализации вычислений в XX в. и современных алгоритмических методов, применяемых в криптографии, искусственном интеллекте и анализе больших данных.

Особенность этой дисциплины заключается в том, что она не только предоставляет методы решения задач, но и определяет границы того, что принципиально возможно или невозможно вычислить.

Ранние этапы: от практических правил к алгоритмам

Древний Восток

Вавилонские таблички (около 1800 г. до н. э.) содержат правила для извлечения квадратного корня, основанные на приближенных вычислениях. По сути, это первый известный численный алгоритм.

Египетские папирусы описывают метод умножения, основанный на последовательном удвоении одного числа и сложении нужных компонентов – аналог бинарного умножения.

Пример

Умножение 13 на 12 в «египетской системе» выглядело так:

$1 \times 12 = 12$

$2 \times 12 = 24$

$4 \times 12 = 48$

$8 \times 12 = 96$

Так как $13 = 8 + 4 + 1$, складывались соответствующие строки: $96 + 48 + 12 = 156$.

Античность

Наиболее значительным вкладом в развитие алгоритмов стал *алгоритм Евклида* (около 300 г. до н. э.) для нахождения наибольшего общего делителя (НОД) двух чисел.

Пример работы алгоритма Евклида

Найти НОД (252, 105).

1. $252 \div 105 = 2$, остаток 42.
2. $105 \div 42 = 2$, остаток 21.
3. $42 \div 21 = 2$, остаток 0.

$\Rightarrow$ НОД = 21.

Алгоритм Евклида до сих пор применяется в криптографии, например, при вычислении обратных элементов в алгоритме RSA.

Средневековье

В IX в. персидский математик *аль-Хорезми* написал труд по арифметике, где описал систематизированные правила для выполнения операций. Его имя стало основой термина «алгоритм».

Новое время

В XVII–XVIII вв. развитие математики привело к созданию универсальных методов решения классов задач.

Рене Декарт ввел аналитическую геометрию. Решение геометрической задачи сводилось к системе алгебраических уравнений, что является примером алгоритмического подхода.

Исаак Ньютон разработал метод касательных (метод Ньютона) для приближенного нахождения корней уравнений. Этот алгоритм применяется и сегодня в численных методах.

Пример

Метод Ньютона для вычисления $\sqrt{2}$.

Начальное приближение: $x_0 = 1{,}5$.

Формула: $x_{n+1} = (x_n + 2/x_n) / 2$.

1. $x_1 = (1{,}5 + 2/1{,}5)/2 = 1{,}4166\ldots$

2. $x_2 = (1{,}4166 + 2/1{,}4166)/2 \approx 1{,}4142$ (что уже близко к истинному значению).

XX век

XX в. стал переломным в понимании алгоритмов. Именно тогда появились строгие математические модели вычислений.

Проблема Гильберта

В 1900 г. Давид Гильберт сформулировал список из 23 фундаментальных задач математики. Среди них была десятая проблема Гильберта: требуется найти алгоритм, который по любому диофантову уравнению определял бы, имеет ли оно решение в целых числах.

Диофантовы уравнения – это уравнения, решениями которых должны быть целые числа.

Пример: $2x + 3y = 7$ (решение: $x = 2$, $y = 1$).

Пример сложного уравнения: $x^2 + y^2 = z^2$ (решения называются пифагоровыми тройками, например $3^2+4^2=5^2$).

В 1970 г. Ю. В. Матиясевич доказал, что общего алгоритма для решения всех диофантовых уравнений не существует.

*Машина Тьюринг*а

В 1936 г. Алан Тьюринг предложил абстрактную модель вычислителя – *машину Тьюринга*. Она представляет собой:

– бесконечную ленту, разделенную на ячейки;

– считывающую и записывающую головку;

– таблицу правил перехода.

Машина Тьюринга позволила формализовать понятие алгоритма и вычислимости.

Пример

Машина Тьюринга может сложить два числа, записанных в унарной системе (111+11=11111).

Главный результат Тьюринга – доказательство существования проблемы остановки: не существует алгоритма, который для любой программы сможет определить, завершится она или будет выполняться бесконечно.

λ-исчисление Черча

Формальные модели и современные алгоритмы

Одним из фундаментальных направлений в развитии теории алгоритмов стало создание универсальных моделей вычислений. К числу первых и наиболее значимых относится λ-исчисление, разработанное Алонзо Черчем в 1930-е гг. Основная идея этой системы заключалась в том, что любой процесс вычисления может быть представлен в виде функций, которые принимают аргументы и возвращают результаты. С помощью символа λ задается абстрактное отображение: так, выражение $\lambda x.(x + 1)$ обозначает функцию, прибавляющую единицу к аргументу. Применение функции к конкретному значению сводится к подстановке, а возможность передачи функций в качестве аргументов другим функциям обеспечивает универсальность модели. Именно эта универсальность позволила Черчу показать, что λ-исчисление обладает той же выразительной силой, что и другие формализмы вычислений, в частности машины Тьюринга.

В советской математической школе развитие аналогичных идей связано с именем А. А. Маркова, предложившего систему нормальных алгоритмов. В отличие от λ-исчисления, акцент в ней был сделан на переписывании строк по заданным правилам. Каждое правило представляло собой подстановку одного фрагмента символов другим, а вычислительный процесс описывался как последовательное применение этих подстановок. Подобная модель оказалась удобной для формализации алгоритмов, связанных с обработкой символьных данных, и послужила важным вкладом в развитие общей теории алгоритмов.

Значительный шаг в понимании природы алгоритмов сделал А. Н. Колмогоров, предложивший концепцию алгоритмической сложности. Согласно

этой теории, сложность объекта можно определить как длину минимальной программы, способной его породить. Например, регулярная строка вида «101010…» легко описывается короткой инструкцией «повторить 01 n раз» и потому имеет низкую алгоритмическую сложность. В противоположность этому случайная последовательность большой длины не допускает значимого сжатия, и ее минимальное описание будет практически совпадать с самой строкой. Эта идея позволила по-новому взглянуть на понятие случайности и оказала влияние на теорию информации, методы сжатия и криптографию.

Наряду с построением универсальных моделей вычислений, теория алгоритмов выявила существование задач, не имеющих алгоритмического решения. Одним из наиболее известных результатов в этом направлении является проблема останова, сформулированная А. Тьюрингом. Она утверждает, что невозможно создать алгоритм, который для произвольной программы и ее входных данных мог бы определить, завершится ли вычисление или будет продолжаться бесконечно. Другим примером является десятая проблема Гильберта, окончательно решенная А. Матиясевичем: доказано, что не существует универсального метода решения всех диофантовых уравнений. Аналогичные результаты были получены и Э. Постом, показавшим, что для некоторых систем переписывания строк невозможно алгоритмически определить достижимость одной строки из другой. Эти открытия очертили принципиальные границы вычислимого и продемонстрировали, что область алгоритмов не всесильна.

Несмотря на наличие неразрешимых задач, развитие теории алгоритмов привело к созданию методов, имеющих огромное прикладное значение. В криптографии ключевую роль играет алгоритм RSA, основанный на сложности разложения больших чисел на множители, а также алгоритм Евклида, используемый для нахождения обратных элементов при работе с арифметикой по модулю. В компьютерных сетях базовым инструментом стала разработка алгоритмов маршрутизации, среди которых особое место занимает алгоритм Дейкстры, применяемый для нахождения кратчайших путей в графах и лежащий в основе протоколов OSPF. В области обработки текстовой информации существенный вклад внес алгоритм Рабина – Карпа, использующий хеширование для эффективного поиска подстрок.

Современные достижения связаны с развитием методов искусственного интеллекта и машинного обучения. Такие алгоритмы, как градиентный спуск, деревья решений и нейронные сети, представляют собой новое поколение вычислительных методов, которые во многом выходят за рамки классической парадигмы детерминированных алгоритмов. Они демонстрируют способность работать с неструктурированными данными, обучаться

на примерах и находить решения задач, ранее считавшихся слишком сложными для формального алгоритмического подхода (табл. 1.4).

Таблица 1.4

Вехи развития теории алгоритмов

Этап	*Ученые*	*Примеры алгоритмов*	*Значение*
Античность	Евклид	Алгоритм НОД	Первое строгое описание алгоритма
Средневековье	Аль-Хорезми	Арифметические правила	Систематизация вычислений
XVII–XVIII вв.	Декарт, Ньютон	Метод Ньютона	Универсальные методы
XIX в.	Буль	Алгебра логики	Формализация логики
XX в.	Тьюринг	Машина Тьюринга	Теория вычислимости
XX в.	Черч	λ-исчисление	Формальная модель функций
XX в.	Марков	Нормальные алгоритмы	Теория строковых преобразований
XX в.	Колмогоров	Алгоритмическая сложность	Связь алгоритмов и информации

Таким образом, история развития формальных моделей вычислений и современных алгоритмов показывает, как от абстрактных построений Черча, Маркова и Колмогорова теория пришла к практическим методам, обеспечивающим работу криптографических систем, компьютерных сетей и технологий искусственного интеллекта. Она демонстрирует единство фундаментальной математики и прикладных задач, подчеркивая, что именно строгие формальные основы делают возможным создание современных вычислительных систем.

Эволюция представлений об алгоритме

Практические правила → Алгоритм Евклида → Работы аль-Хорезми

↓

Математическая формализация (Декарт, Ньютон, Буль)

↓

Формальные модели (Тьюринг, Черч, Марков)

↓

Теория сложности (Колмогоров)

↓

Современные алгоритмы (криптография, сети, ИИ)

История теории алгоритмов демонстрирует переход от интуитивных правил к строгим математическим моделям вычислений. Сегодня

алгоритмы лежат в основе всех сфер человеческой деятельности: от медицины и экономики до кибербезопасности и искусственного интеллекта.

Знание исторических аспектов позволяет понимать не только текущее состояние информатики, но и ее фундаментальные ограничения: существуют задачи, которые алгоритмически неразрешимы.

Контрольные вопросы

1. Дайте определение понятию программная инженерия. В чем заключается ее отличие от обычного программирования?

2. Какие цели и задачи ставит перед собой программная инженерия как научная дисциплина?

3. Перечислите основные этапы жизненного цикла программного обеспечения. Как они взаимосвязаны?

4. Опишите каскадную модель разработки ПО. В каких случаях она эффективна, а в каких – ограничена?

5. Чем отличается итеративная и спиральная модели разработки ПО? Приведите примеры их применения.

6. Как развитие вычислительной техники отразилось на методах программирования? Назовите основные поколения ЭВМ и их особенности.

7. Какие исторические открытия Евклида, аль-Хорезми, Тьюринга и Колмогорова повлияли на становление теории алгоритмов?

8. Что такое «проблема останова» и «десятая проблема Гильберта»? Почему они важны для понимания ограничений алгоритмов?

9. Назовите основные принципы программной инженерии (системность, модульность, повторное использование и др.). Приведите примеры их применения.

10. В чем практическая значимость программной инженерии для современного общества? Приведите примеры из реальной жизни (например, банковские системы, авиация, мобильные приложения).

2. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ и АСУ (ПО)

2.1. Основные понятия: программа, программное обеспечение, программные средства

Современные вычислительные системы представляют собой не просто совокупность аппаратных компонентов, но и тесное взаимодействие аппаратуры с программным обеспечением. Аппаратная часть (hardware) без программ является «пассивной» – это лишь набор электронных устройств, не выполняющих полезных функций. Лишь с появлением программ и программного обеспечения (software) компьютер превращается в универсальное средство обработки информации.

В информатике и программной инженерии важно строго разграничивать базовые понятия: «программа», «программное обеспечение» и «программные средства». Эти термины нередко используются как синонимы, однако они имеют различное значение и уровень обобщения.

Понятие программы

Программа – это упорядоченная последовательность инструкций (команд), предназначенных для исполнения вычислительной системой с целью получения определенного результата.

Основные характеристики программы:

– *алгоритмичность* – программа всегда реализует алгоритм;

– *детерминированность* – при одинаковых входных данных программа должна давать один и тот же результат (если иное не оговорено, например, при случайных процессах);

– *машинная интерпретируемость* – команды программы могут быть исполнены аппаратурой или ее виртуальной моделью;

– *целевая направленность* – любая программа предназначена для решения конкретной задачи.

Программы могут быть:

– *прикладными* (Word, Excel, браузер);

– *системными* (драйверы, загрузчики);

– *служебными* (архиваторы, антивирусы).

Исторический пример

Первые программы писались непосредственно в машинных кодах (наборы нулей и единиц). Так, ЭНИАК (1946) программировался путем ручного соединения кабелей. Позднее появились ассемблеры и языки высокого уровня (Fortran, 1957; COBOL, 1959), что сделало программы более удобными для людей.

Понятие программного обеспечения

Программное обеспечение (ПО, от англ. Software) – это совокупность всех программ, данных и документации, необходимых для функционирования вычислительной системы в процессе ее эксплуатации.

Ключевые особенности ПО:

1. Широта охвата.

ПО включает не только исполняемые программы, но и вспомогательные файлы, конфигурации, базы данных, справочные материалы, руководство пользователя.

2. Независимость от аппаратуры.

Одно и то же устройство может выполнять разные функции в зависимости от установленного ПО.

3. Иерархическая организация.

ПО делится на уровни: системное, инструментальное, прикладное.

4. Разнообразие форм распространения.

ПО может быть коммерческим, свободным, встроенным, облачным.

Пример

Операционная система Windows является программным обеспечением, так как включает ядро ОС, системные библиотеки, драйверы устройств, графическую оболочку и служебные утилиты.

Программные средства

Программные средства – это конкретные программы или комплексы программ, ориентированные на решение определенных задач. В отличие от понятия «ПО», которое является собирательным, термин «программные средства» используется для указания на конкретные инструменты.

Программные средства делятся на три группы (табл. 2.1).

1. Системные программные средства обеспечивают работу аппаратуры:
   - операционные системы (Windows, Linux, macOS);
   - драйверы устройств;
   - утилиты обслуживания.
2. Прикладные программные средства ориентированы на конечного пользователя:
   - офисные пакеты (MS Office, LibreOffice);
   - графические редакторы (Photoshop, GIMP);
   - инженерные системы (MATLAB, AutoCAD);
   - бухгалтерские пакеты (1С).
3. Инструментальные программные средства предназначены для программистов:
   - компиляторы и интерпретаторы;

– отладчики;
– интегрированные среды разработки (Visual Studio, PyCharm, Eclipse).

Таблица 2.1

Сравнение понятий

Понятие	*Определение*	*Сфера применения*	*Примеры*
Программа	Последовательность команд для ЭВМ	Решение конкретной задачи	Калькулятор, архиватор, драйвер принтера
Программное обеспечение	Совокупность программ, данных и документации	Поддержка работы всей вычислительной системы	Windows, Linux, macOS
Программные средства	Конкретные программы, решающие задачи	Системные, прикладные или инструментальные	Word, AutoCAD, Visual Studio

Роль ПО в автоматизированных системах управления (АСУ)

Автоматизированная система управления (АСУ) – это комплекс аппаратных, программных и организационных средств, обеспечивающий управление объектами или процессами.

ПО в АСУ играет ключевую роль:

– реализует алгоритмы управления (например, стабилизация параметров производства);
– обеспечивает взаимодействие с аппаратурой (драйверы датчиков, контроллеров);
– поддерживает базы данных и информационные системы;
– гарантирует надежность, отказоустойчивость и безопасность.

Пример

В промышленной АСУ программа управления может собирать данные с датчиков температуры, обрабатывать их и автоматически регулировать мощность нагрева.

Взаимодействие аппаратного и программного обеспечения

Пользователь
↓
Прикладные программы (Word, Excel, CAD)
↓
Системное ПО (ОС, драйверы, библиотеки)
↓
Аппаратное обеспечение (процессор, память, устройства ввода-вывода)

Значение правильного понимания терминов

Четкое различие между «программой», «ПО» и «программными средствами» необходимо:

– для корректного ведения технической документации;

– при проектировании автоматизированных систем;

– для стандартизации (например, в ГОСТ и международных стандартах ISO/IEC).

Ошибочное отождествление этих понятий может привести к недопониманию между разработчиками, инженерами и пользователями, а также к проблемам при эксплуатации сложных систем.

Программа – это конкретная последовательность инструкций для ЭВМ.

Программное обеспечение – это вся совокупность программ, данных и документации, необходимых для функционирования вычислительной системы.

Программные средства – это конкретные реализации ПО для решения определенных задач (системные, прикладные, инструментальные).

Знание этих различий является основой для изучения классификации программного обеспечения, принципов его разработки и внедрения в автоматизированные системы управления.

2.2. Инженерный подход к конструированию программного обеспечения

Программное обеспечение в современных вычислительных системах рассматривается не как набор отдельных прикладных решений, а как сложный объект проектирования, требующий системного анализа, планомерной организации и строгого соблюдения стандартов качества. Развитие информационных технологий привело к тому, что программные системы внедрены во все сферы деятельности общества: от автоматизированных производств и транспортных систем до финансовых сервисов, образования и здравоохранения. В этих условиях даже незначительная ошибка в программном продукте может привести к крупным материальным убыткам, угрозе безопасности или нарушению функционирования критически важных инфраструктур.

Именно поэтому разработка программного обеспечения должна рассматриваться в качестве самостоятельного инженерного процесса. Инженерный подход к конструированию программного обеспечения является практическим воплощением программной инженерии как научной дисциплины. Он опирается на методы системного анализа, стандартизацию

процессов, использование формальных моделей и ориентацию на обеспечение качества на протяжении всего жизненного цикла программного продукта.

Сущность инженерного подхода

Инженерный подход предполагает, что создание программного продукта рассматривается аналогично созданию любого технического сооружения или машины. Как в строительстве невозможно возводить здание без предварительного проекта, так и в программной инженерии недопустимо писать код без анализа требований и архитектурного проектирования.

Ключевая идея инженерного подхода состоит в том, что программное обеспечение должно проектироваться, разрабатываться и сопровождаться с применением научно обоснованных методов, а не на основе интуиции или индивидуальных практик отдельных разработчиков. При этом каждый этап разработки фиксируется документально, обеспечивая преемственность и возможность воспроизводимости результатов.

Характерные особенности инженерного подхода

Системность. Программное обеспечение рассматривается как совокупность взаимосвязанных элементов, объединенных общей архитектурой и подчиненных целям системы в целом.

Формализация. Требования и проектные решения описываются в формализованных моделях, позволяющих исключить неоднозначность толкований. Для этого применяются языки моделирования (UML, SysML), стандарты ISO/IEC, ГОСТ.

Модульность. Сложная система делится на отдельные подсистемы и модули, каждый из которых имеет четко определенные интерфейсы и функции. Это облегчает разработку, тестирование и модернизацию.

Автоматизация. Инженерный подход предполагает широкое использование инструментальных средств: систем контроля версий (Git), систем автоматической сборки, средств статического анализа кода, инструментов автоматизированного тестирования и развертывания.

Качество и надежность. Качество не рассматривается как завершающий этап, а закладывается на протяжении всего процесса разработки. Важнейшими показателями качества являются корректность, производительность, отказоустойчивость, удобство использования и безопасность.

Документирование. Обязательное сопровождение всех этапов разработки документацией обеспечивает прозрачность проекта, делает его доступным для анализа и доработки другими специалистами, а также снижает зависимость от отдельных исполнителей.

Отличие инженерного подхода от ремесленного

Исторически программирование начиналось как ремесленная деятельность. Один разработчик создавал программы, исходя из собственного

понимания задачи и опираясь на личный опыт. Такой подход позволял получать быстрые результаты для небольших задач, но он оказался несостоятельным в условиях роста сложности программных систем.

Ремесленный подход характеризуется отсутствием системного планирования, слабой формализацией требований, минимальной документацией и низкой предсказуемостью конечного результата. Стоимость исправления ошибок в таких проектах возрастает экспоненциально на поздних стадиях разработки.

Инженерный подход противопоставлен этому. Он основан на планомерности, управляемости и контроле качества. Все решения принимаются осознанно, проектирование и реализация сопровождаются документами, а тестирование проводится системно на каждом уровне. Такой подход позволяет создавать долговечные и масштабируемые программные продукты.

Этапы инженерного конструирования программного обеспечения

1. *Анализ требований.* На этом этапе осуществляется сбор, систематизация и формализация требований заказчика и пользователей. Итогом является техническое задание, содержащее функциональные, нефункциональные и эксплуатационные требования.

2. *Проектирование архитектуры.* Разрабатывается архитектурная модель программного продукта, определяются основные компоненты, их взаимодействие, используемые базы данных, интерфейсы и протоколы.

3. *Реализация.* На основе проектной документации осуществляется написание исходного кода. Используются стандарты кодирования и инструменты коллективной разработки.

4. *Тестирование и верификация.* Проверяется соответствие разработанного программного продукта требованиям. Проводится модульное, интеграционное, системное и нагрузочное тестирование.

5. *Внедрение.* Программный продукт устанавливается в целевую среду, выполняется настройка, обучение пользователей, интеграция с другими системами.

6. *Сопровождение и развитие.* Обеспечивается исправление ошибок, выпуск обновлений, добавление новых функций, адаптация к изменяющимся условиям эксплуатации.

Методологии инженерной разработки

В инженерном подходе используются различные методологии, которые задают последовательность и организацию этапов жизненного цикла программного обеспечения.

Каскадная модель (Waterfall) предполагает строгое последовательное прохождение этапов, что делает ее подходящей для проектов с фиксированными требованиями и высокой ценой ошибок.

Итеративная модель позволяет получать рабочий продукт уже на ранних стадиях и постепенно дорабатывать его в нескольких циклах.

Спиральная модель совмещает итеративный подход с управлением рисками, что особенно важно для дорогостоящих проектов.

Agile-методологии ориентированы на гибкость и тесное взаимодействие с заказчиком, обеспечивая быструю реакцию на изменения требований.

Пример реализации инженерного подхода

При создании системы электронного документооборота для крупной организации инженерный подход предполагает последовательное выполнение этапов: от анализа требований и построения архитектурной схемы до тестирования, внедрения и сопровождения. Такая система проектируется с учетом нагрузки, безопасности, интеграции с существующими службами, наличия резервного копирования и возможностей масштабирования. В результате организация получает надежный и устойчивый программный комплекс, рассчитанный на долгосрочную эксплуатацию.

Значение инженерного подхода

Инженерный подход к конструированию программного обеспечения позволяет достигать предсказуемых результатов в разработке, снижает вероятность ошибок, повышает надежность и безопасность систем. Он обеспечивает экономическую эффективность за счет снижения стоимости исправления ошибок и уменьшения эксплуатационных рисков. Кроме того, он способствует созданию долговечных систем, которые могут сопровождаться и модернизироваться в течение многих лет.

Инженерный подход является необходимым условием успешной разработки современных программных систем. Он основан на применении методов системной инженерии, формальных моделей, стандартов качества и автоматизированных средств поддержки разработки. Такой подход обеспечивает высокое качество программного продукта, его надежность, управляемость и устойчивость к изменениям внешней среды.

2.3. Классификация программного обеспечения по назначению

Программное обеспечение (ПО) представляет собой важнейший компонент любой вычислительной системы. Именно оно превращает аппаратное обеспечение – совокупность микропроцессоров, запоминающих устройств и периферии – в универсальное средство обработки информации. Компьютер без программного обеспечения является лишь набором электронных схем, не способных выполнять целенаправленные действия.

Для понимания роли и функций различных видов ПО применяется классификация. Одной из базовых является классификация по назначению, которая позволяет разделить все программное обеспечение на несколько основных классов в зависимости от выполняемых функций и решаемых задач.

Общая схема классификации

Согласно международным стандартам и отечественной практике, выделяют три ключевых класса программного обеспечения:

1. *Системное программное обеспечение.* Обеспечивает функционирование аппаратной части и создает условия для работы других программ.

2. *Прикладное программное обеспечение.* Ориентировано на решение конкретных задач пользователей.

3. *Инструментальное программное обеспечение.* Используется для создания и сопровождения других программных продуктов.

Классы программного обеспечения по назначению

Программное обеспечение

- Системное ПО
- Прикладное ПО
- Инструментальное ПО

Системное программное обеспечение

Системное программное обеспечение представляет собой совокупность программ, которые управляют аппаратными средствами компьютера и обеспечивают взаимодействие между пользователем, прикладными программами и оборудованием.

- *Операционные системы (ОС).*

Это ядро и сопутствующие модули, управляющие процессами, памятью, файлами и устройствами ввода-вывода. Примеры: Windows, Linux, macOS, Astra Linux.

- *Драйверы устройств.*

Специализированные программы, обеспечивающие корректную работу конкретного оборудования (например, видеокарт, принтеров, сетевых адаптеров).

- *Системные утилиты.*

Набор вспомогательных программ, выполняющих диагностику, резервное копирование, архивацию и мониторинг состояния системы.

- *Базовое ПО встроенных систем.*

BIOS и UEFI, прошивки микроконтроллеров, управляющие низкоуровневыми функциями оборудования.

Функции системного ПО включают:
– управление аппаратными ресурсами;
– обеспечение стабильной работы прикладных программ;
– реализацию пользовательского интерфейса;
– организацию сетевого взаимодействия.

Пример

Смартфон с операционной системой Android способен запускать множество прикладных программ благодаря наличию ядра Linux, драйверов и библиотек, обеспечивающих универсальность платформы.

Прикладное программное обеспечение

Прикладное ПО ориентировано на решение прикладных задач пользователей. Этот класс является наиболее разнообразным и развивается быстрее всего.

Основные подвиды прикладного ПО.

- *Офисные приложения.*

Текстовые редакторы, электронные таблицы, презентационные пакеты. Примеры: Microsoft Office, LibreOffice.

- *Графические и мультимедийные системы.*

Photoshop, CorelDraw, AutoCAD, видеоредакторы (Adobe Premiere).

- *Инженерные и научные приложения.*

MATLAB, Mathcad, SolidWorks, ANSYS. Используются для моделирования, анализа и расчетов.

- *Деловые системы.*

ERP-системы (SAP, Oracle ERP), бухгалтерские комплексы (1С: Предприятие), CRM-системы.

- *Образовательные программы.*

Системы дистанционного обучения (Moodle), интерактивные тренажеры.

- *Специализированные системы.*

ПО для медицины (например, программы обработки данных томографа), авиации (системы навигации), финансов (банковские приложения) (табл. 2.2).

Таблица 2.2

Примеры прикладного ПО по назначению

Подкласс ПО	*Примеры*	*Основное назначение*
Офисные приложения	MS Word, Excel, PowerPoint	Работа с документами, таблицами, презентациями
Графические редакторы	Photoshop, CorelDraw, GIMP	Создание и обработка изображений

Окончание табл. 2.2

Подкласс ПО	*Примеры*	*Основное назначение*
Инженерные системы	AutoCAD, SolidWorks, MATLAB	Моделирование и проектирование
Деловые системы	1C, SAP, Oracle ERP	Управление финансами и бизнес-процессами
Образовательное ПО	Moodle, Duolingo	Обучение и тестирование
Специализированные	ПО МРТ, авиационные тренажеры	Решение узкопрофильных задач

Функции прикладного ПО:

– автоматизация деятельности пользователя;
– повышение производительности труда;
– поддержка принятия решений;
– обработка больших массивов данных.

Инструментальное программное обеспечение

Инструментальное ПО применяется программистами и инженерами для создания, отладки, тестирования и сопровождения других программных продуктов.

В этот класс входят различные продукты.

- *Компиляторы и интерпретаторы.*

Программы, преобразующие исходный код в машинные инструкции (gcc, javac, Python-интерпретатор).

- *Отладчики.*

Средства пошагового выполнения программ, выявления ошибок (gdb, WinDbg).

- *Интегрированные среды разработки (IDE).*

Комплексные средства, включающие редактор кода, отладчик и средства тестирования (Visual Studio, PyCharm, Eclipse).

- *CASE-средства.*

Автоматизированные системы проектирования и генерации кода (Enterprise Architect, Rational Rose).

- *Системы контроля версий.*

Системы контроля версий для отслеживания изменений в файлах проекта, управления различными версиями кода и организации совместной работы разработчиков под общей кодовой базой (Git, Subversion).

- *Средства CI/CD.*

Средства CI/CD для автоматизации процессов сборки, тестирования и развертывания ПО (Jenkins, GitLab CI, Azure DevOps).

Инструментальное ПО обеспечивает стандартизацию и автоматизацию разработки, повышает ее эффективность и позволяет создавать более надежные и масштабируемые продукты.

Роль инструментального ПО

Разработчик

↓

Инструментальное ПО (IDE, компиляторы, тестовые среды)

↓

Создание новых программ

↓

Системное и прикладное ПО

Классификация программного обеспечения по уровню пользователей

ПО можно разделить и по степени сложности и уровню подготовки пользователей.

- *Массовое ПО.*

Ориентировано на широкий круг пользователей, не требующих специальной подготовки. Примеры: браузеры, медиаплееры, офисные приложения.

- *Профессиональное ПО.*

Требует высокой квалификации, используется специалистами в конкретных областях. Примеры: MATLAB, системы автоматизированного проектирования, программное обеспечение медицинской диагностики.

Дополнительные критерии классификации

Хотя классификация по назначению является основной, существуют и дополнительные критерии:

– *по способу распространения:* коммерческое, свободное (open source), условно-бесплатное (shareware);

– *по масштабу применения:* индивидуальное, корпоративное, государственное;

– *по типу платформы:* настольное ПО, мобильные приложения, облачные сервисы, встроенные системы.

Классификация программного обеспечения по назначению отражает его функциональную структуру и позволяет систематизировать огромное разнообразие существующих программных продуктов.

Системное ПО обеспечивает взаимодействие аппаратуры и программ.

Прикладное ПО решает задачи пользователей в разных сферах деятельности.

Инструментальное ПО создает условия для разработки новых программных продуктов.

Все три класса взаимосвязаны: без системного ПО невозможна работа прикладных программ, без инструментального ПО невозможно создание новых приложений, а прикладное ПО формирует основную ценность для конечного пользователя.

2.4. Системное программное обеспечение

Системное программное обеспечение (СПО) является фундаментальной частью любой вычислительной системы. Оно формирует основу, без которой невозможно функционирование прикладных программ и взаимодействие пользователя с аппаратной частью. СПО можно рассматривать как «связующее звено» между железом (hardware) и приложениями, обеспечивающее управление ресурсами, выполнение программ, организацию файловых систем и сетевое взаимодействие.

Развитие системного программного обеспечения сопровождало всю историю вычислительной техники. От простейших загрузочных программ первых компьютеров до современных многоуровневых операционных систем, поддерживающих многозадачность, распределенные вычисления и виртуализацию, прошло несколько поколений развития.

Системное программное обеспечение (СПО) – это совокупность программных средств, предназначенных для управления аппаратными ресурсами вычислительной системы, организации взаимодействия пользователя и приложений с оборудованием, а также обеспечения функционирования прикладного и инструментального ПО.

Главные характеристики системного ПО:

– независимость от конкретных задач пользователя;

– универсальность (одна и та же ОС может поддерживать множество приложений);

– работа в фоновом режиме;

– ориентация на стабильность, надежность и безопасность.

Классификация системного ПО (табл. 2.3)

Таблица 2.3

Классификация системного ПО

Класс ПО	*Основное назначение*	*Примеры*
Операционные системы	Управление ресурсами, интерфейс для приложений	Windows, Linux, macOS, Android
Драйверы устройств	Обеспечение работы оборудования	NVIDIA Driver, Canon Printer Driver
Системные утилиты	Диагностика, настройка, защита	Архиваторы, антивирусы, дефрагментаторы

Окончание табл. 2.3

Класс ПО	*Основное назначение*	*Примеры*
Базовое ПО (BIOS/UEFI)	Низкоуровневое управление аппаратурой	BIOS AMI, UEFI Phoenix
Сервисные оболочки	Средства взаимодействия с системой	Bash, PowerShell

Системное программное обеспечение можно разделить на несколько основных групп.

1. *Операционные системы (ОС).*

Центральный элемент СПО, управляющий процессами, памятью, устройствами ввода-вывода и файлами.

Обеспечивает интерфейс для пользователя и приложений.

Примеры: Windows, Linux, macOS, Astra Linux, Android, iOS.

2. *Драйверы устройств.*

Программы, обеспечивающие взаимодействие между ОС и конкретным оборудованием.

Примеры: драйверы видеокарт (NVIDIA, AMD), драйверы принтеров, драйверы звуковых карт.

3. *Системные утилиты.*

Вспомогательные программы для обслуживания системы.

Примеры: программы дефрагментации диска, антивирусы, архиваторы, средства резервного копирования, диспетчеры задач.

4. *Базовое ПО встроенных систем.*

BIOS, UEFI, микропрограммы (firmware), управляющие аппаратурой на низком уровне.

Используются до загрузки ОС, обеспечивают тестирование оборудования и начальную инициализацию.

5. *Сервисные оболочки и средства администрирования.*

Интерфейсы, облегчающие работу с ОС (графические оболочки, консольные интерпретаторы команд).

Примеры: Bash (Linux), PowerShell (Windows).

Использование системного ПО

Функции системного ПО выходят далеко за рамки запуска приложений. Оно необходимо для решения следующих задач.

1. *Управление аппаратными ресурсами.*

ОС распределяет процессорное время между задачами, управляет памятью, дисковыми ресурсами и сетевыми соединениями.

2. *Обеспечение интерфейса.*

Пользователь взаимодействует с компьютером через графический или консольный интерфейс, реализуемый системным ПО.

3. *Поддержка многозадачности и многопользовательского режима.*

Современные ОС позволяют одновременно работать нескольким пользователям и запускать десятки процессов.

4. *Сетевая функциональность.*

Системное ПО обеспечивает работу протоколов TCP/IP, организацию доступа в интернет, настройку локальных сетей.

5. *Безопасность.*

Контроль доступа, аутентификация пользователей, шифрование данных, антивирусная защита – все это реализуется средствами системного ПО.

6. *Сервисные функции.*

Резервное копирование, восстановление после сбоев, обновление драйверов и программных компонентов.

Роль системного ПО в вычислительной системе

Пользователь

↓

Прикладное ПО

↓

Системное ПО (ОС, драйверы, утилиты)

↓

Аппаратное обеспечение (CPU, RAM, устройства ввода-вывода)

Примеры использования

В *офисных приложениях* системное ПО обеспечивает доступ к файловой системе и принтерам.

В *игровых приложениях* драйверы видеокарт и аудиоустройств обеспечивают высокую производительность графики и звука.

В *банковских системах* ОС и утилиты резервного копирования гарантируют сохранность и защиту данных.

Во *встроенных системах* (например, микроконтроллерах автомобилей) прошивки управляют датчиками и исполнительными механизмами.

Историческая справка

Первое системное ПО появилось в середине XX в. в виде загрузочных программ, позволявших вводить задачи с перфокарт. В 1960-е гг. появились первые операционные системы, поддерживавшие пакетную обработку заданий. В 1970-е гг. ОС научились работать в многозадачном режиме. В 1980-е гг. персональные компьютеры получили MS-DOS, а затем Windows с графическим интерфейсом. С конца 1990-х гг. началась эпоха сетевых ОС и виртуализации. Сегодня системное ПО развивается в сторону облачных технологий, мобильных платформ и искусственного интеллекта.

Системное программное обеспечение играет ключевую роль в вычислительных системах, предоставляя функционирование аппаратных средств,

выполнение прикладных задач и безопасность информации. Оно включает операционные системы, драйверы, утилиты, базовое ПО и сервисные оболочки. Без системного ПО невозможно ни использование компьютеров, ни развитие информационных технологий в целом.

2.5. Операционные системы: функции, примеры

Операционная система является центральным элементом системного программного обеспечения, без которого невозможна эксплуатация современных вычислительных систем. Она представляет собой совокупность взаимосвязанных программных модулей, управляющих аппаратными ресурсами компьютера и создающих среду для выполнения прикладных и инструментальных программ.

С момента появления первых электронных вычислительных машин операционные системы прошли длительный путь развития: от простейших загрузчиков, запускавших одну программу за раз, до многоуровневых, многопользовательских и распределенных комплексов, способных управлять тысячами процессов одновременно. Современные ОС обеспечивают функционирование не только персональных компьютеров, но и мобильных устройств, промышленных комплексов, сетевых сервисов и встроенных систем реального времени.

Определение и сущность

Операционная система (ОС) – это системный программный комплекс, предназначенный для управления вычислительными ресурсами, организации выполнения программ и обеспечения взаимодействия пользователя с аппаратурой. Она играет двойную роль: с одной стороны, скрывает сложность аппаратного обеспечения и предоставляет пользователю удобный интерфейс, а с другой – создает стандартизированную платформу для разработчиков прикладных программ.

Основные функции операционных систем

1. *Управление процессами.*

ОС распределяет процессорное время между задачами, обеспечивает их запуск, приостановку и завершение. Она реализует планирование процессов по приоритетам и времени, обеспечивает межпроцессное взаимодействие и синхронизацию.

2. *Управление памятью.*

ОС выделяет память для процессов, следит за ее использованием, предотвращает конфликты и обеспечивает защиту данных. Важной функцией является поддержка виртуальной памяти, которая позволяет работать

с объемами данных, превышающими физический объем оперативной памяти.

3. *Управление файловыми системами.*

ОС организует хранение информации на долговременных носителях. Она поддерживает иерархическую структуру каталогов, обеспечивает доступ к файлам, их защиту, шифрование и совместный доступ. Существуют различные файловые системы: FAT32, NTFS, ext4, APFS, каждая из которых оптимизирована под определенные условия эксплуатации.

4. *Управление устройствами ввода-вывода.*

ОС обеспечивает работу периферийных устройств через драйверы. Она формирует единый интерфейс для работы с клавиатурой, мышью, дисплеем, принтером, сетевыми картами и внешними накопителями, скрывая от пользователя особенности конкретного оборудования.

5. *Обеспечение пользовательского интерфейса.*

ОС предоставляет командную строку (CLI) или графическую оболочку (GUI), позволяя пользователю управлять системой. Интерфейсы развиваются от примитивных текстовых команд (MS-DOS) до современных многозадачных графических окружений с поддержкой сенсорных экранов и голосового ввода.

6. *Сетевая поддержка.*

Современные ОС интегрированы в глобальные и локальные сети. Они поддерживают протоколы TCP/IP, обеспечивают работу веб-сервисов, обмен файлами, удаленное администрирование и защиту сетевых соединений.

7. *Обеспечение безопасности.*

ОС контролирует доступ к ресурсам, реализует аутентификацию пользователей, разграничение прав, ведение журналов событий. Встроенные механизмы безопасности предотвращают несанкционированный доступ, поддерживают антивирусную защиту и шифрование.

8. *Сервисные функции.*

ОС включает средства резервного копирования, обновления компонентов, диагностики оборудования и восстановления системы после сбоев (табл. 2.4).

Таблица 2.4

Функции операционной системы

Функция	*Основное содержание*
Управление процессами	Планирование задач, синхронизация, многозадачность
Управление памятью	Выделение и защита, виртуальная память
Управление файлами	Организация хранения, файловые системы, защита данных
Управление устройствами	Драйверы, единый интерфейс для оборудования

Окончание табл. 2.4

Функция	*Основное содержание*
Интерфейс с пользователем	Командная строка, графическая оболочка, мультимодальные интерфейсы
Сетевая поддержка	TCP/IP, доступ к интернету, распределенные вычисления
Безопасность	Аутентификация, контроль доступа, шифрование
Сервисные функции	Резервное копирование, обновления, диагностика

Примеры операционных систем

1. *Windows.*

Наиболее распространенное семейство ОС для персональных компьютеров. Отличается развитым графическим интерфейсом, широким набором прикладного ПО и совместимостью с разнообразным оборудованием. Используется также в серверных версиях (Windows Server).

2. *Linux.*

ОС с открытым исходным кодом, применяемая в самых разных сферах: от серверов и суперкомпьютеров до смартфонов и встраиваемых систем. Популярные дистрибутивы: Ubuntu, Debian, Red Hat, Fedora, Astra Linux.

3. *UNIX и производные.*

Класс операционных систем, отличающихся надежностью и многоуровневой архитектурой. На основе UNIX создана macOS, а также многие серверные ОС.

4. *macOS.*

Операционная система корпорации Apple для компьютеров Mac. Отличается тесной интеграцией с аппаратным обеспечением, удобным интерфейсом и высокой степенью защищенности.

5. *Мобильные ОС.*

Ориентированы на работу с сенсорными устройствами и поддерживают миллионы приложений. Android (на базе ядра Linux) и iOS (на основе технологий macOS).

6. *Серверные ОС.*

Предназначены для управления сетями, базами данных, виртуализацией и облачными сервисами. Примеры: Windows Server, Red Hat Enterprise Linux, FreeBSD.

7. *Встроенные ОС.*

Специализированные системы для управления микроконтроллерами и промышленными устройствами. Примеры: QNX, FreeRTOS, VxWorks. Отличаются компактностью, высокой скоростью работы и возможностью функционировать в режиме реального времени (табл. 2.5).

Таблица 2.5

Сравнительная характеристика ОС

ОС	*Основная область применения*	*Отличительные особенности*
Windows	Персональные ПК, офисные системы	Дружелюбный интерфейс, широкая совместимость
Linux	Серверы, суперкомпьютеры, встраиваемые системы	Открытость, надежность, гибкость
macOS	ПК Apple, мультимедиа	Интеграция с аппаратным обеспечением
Android	Смартфоны, планшеты	Популярность, разнообразие приложений
iOS	Устройства Apple	Безопасность, единая экосистема
QNX	Промышленные системы	Работа в реальном времени, устойчивость
Windows Server	Корпоративные сети, базы данных	Поддержка AD, удобное администрирование

Роль ОС в вычислительной системе

Пользователь
↓
Прикладные программы
↓
Операционная система
↓
Драйверы и системные службы
↓
Аппаратное обеспечение

Историческое развитие

Операционные системы появились в 1950–1960-е гг. как программы управления пакетной обработкой заданий на мэйнфреймах. В 1970-е гг. с развитием UNIX возникли концепции многозадачности и многопользовательского режима. В 1980-е гг. персональные компьютеры получили MS-DOS и первые версии Windows, что сделало вычислительные системы доступными массовому пользователю. В 1990-е гг. ОС стали ориентированы на графический интерфейс и сетевые функции. В XXI в. акцент сместился на мобильные платформы, облачные технологии, виртуализацию и информационную безопасность.

2.6. Прикладное программное обеспечение: классификация, функции, примеры

Прикладное программное обеспечение (ППО) является важнейшим компонентом информационной инфраструктуры современного общества. Оно обеспечивает решение практических задач пользователей в различных сферах деятельности: от офисного документооборота и бизнес-процессов до научных исследований, инженерных расчетов, образовательных технологий и медицины. Если системное программное обеспечение отвечает за общее функционирование вычислительной системы, то прикладное ПО определяет ее ценность и полезность для конкретного пользователя.

Развитие прикладного программного обеспечения напрямую связано с эволюцией вычислительной техники. На ранних этапах программы создавались для решения строго ограниченного круга задач: научных расчетов и бухгалтерского учета. С распространением персональных компьютеров и глобальных сетей количество направлений применения программного обеспечения значительно расширилось. Сегодня прикладные программы сопровождают человека практически во всех сферах жизни, а их разнообразие требует строгой классификации.

Прикладное программное обеспечение (ППО) – это совокупность программных средств, предназначенных для автоматизации прикладных задач, возникающих в профессиональной, учебной, научной, административной или бытовой деятельности пользователей.

От системного ПО прикладные программы отличаются тем, что не управляют аппаратными ресурсами, а используют их для решения поставленных задач. Прикладное ПО является конечным продуктом, который непосредственно взаимодействует с пользователем.

Основные функции прикладного ПО

1. *Автоматизация прикладной деятельности.*

Программы позволяют выполнять задачи быстрее и точнее по сравнению с ручными методами.

Пример: бухгалтерский пакет 1С автоматизирует учет финансовых операций.

2. *Обработка информации и поддержка принятия решений.*

Современные аналитические системы анализируют большие массивы данных (Big Data), предоставляют отчеты и прогнозы.

3. *Повышение производительности труда.*

Использование текстовых процессоров, электронных таблиц и графических редакторов значительно ускоряет работу специалистов.

4. *Визуализация информации.*

Научные и инженерные пакеты (например, MATLAB, AutoCAD) представляют данные в форме графиков, диаграмм и 3D-моделей.

5. *Обеспечение коммуникации и совместной работы.*

Средства коллективного доступа (Microsoft Teams, Slack, Trello) позволяют организовать совместное выполнение проектов.

6. *Образовательные и развивающие функции.*

Электронные учебники, языковые тренажеры и симуляторы создают условия для дистанционного обучения и повышения квалификации.

Классификация прикладного программного обеспечения (табл. 2.6).

Таблица 2.6

Классификация прикладного ПО по назначению

Категория ПО	*Примеры*	*Область применения*
Офисные приложения	MS Office, LibreOffice	Работа с документами и таблицами
СУБД	Oracle, MySQL, PostgreSQL	Хранение и обработка данных
Графические редакторы	Photoshop, CorelDraw, GIMP	Создание изображений
Инженерные пакеты	AutoCAD, MATLAB, ANSYS	Моделирование, расчеты
Деловые системы	1C, SAP, Salesforce	Управление бизнесом
Образовательные программы	Moodle, Duolingo	Обучение и тестирование
Специализированные системы	ПО МРТ, авиационные симуляторы	Профессиональные задачи

Прикладное ПО можно классифицировать по различным признакам: по назначению, по масштабу применения, по сфере использования и по способу распространения.

1. Классификация по назначению.

- *Офисные приложения.* Обеспечивают работу с текстами, таблицами, презентациями (MS Office, LibreOffice, Google Docs).
- *Системы управления базами данных (СУБД).* Предназначены для хранения и обработки больших объемов информации (Oracle, MySQL, PostgreSQL).
- *Графические и мультимедийные редакторы.* Используются для обработки изображений, звука и видео (Adobe Photoshop, CorelDraw, Sony Vegas).
- *Инженерные и научные системы.* Служат для моделирования, анализа и расчетов (AutoCAD, MATLAB, ANSYS, Mathcad).

- *Деловые и корпоративные системы.* Включают ERP- и CRM-системы для управления производством и бизнес-процессами (SAP, 1С: Предприятие, Salesforce).
- *Образовательные программы.* Ориентированы на обучение и тестирование (Moodle, Blackboard, Duolingo).
- *Специализированные системы.* Решают узкопрофильные задачи, например медицинские (системы обработки данных МРТ), авиационные (пилотажные тренажеры).

2. Классификация по масштабу применения

- *Индивидуальные программы.* Рассчитаны на одного пользователя (текстовый редактор, калькулятор).
- *Групповые системы.* Поддерживают коллективную работу (системы документооборота, совместные редакторы).
- *Корпоративные системы.* Автоматизируют деятельность крупных организаций, охватывают все уровни управления (ERP-системы).

3. Классификация по сфере использования:

- бизнес-приложения;
- научные приложения;
- образовательные приложения;
- развлекательные программы (игры, медиаплееры);
- программы для государственного управления.

4. Классификация по способу распространения:

- коммерческое ПО (лицензионные пакеты, распространяемые за плату);
- свободное ПО (open source), доступное с исходным кодом;
- условно-бесплатное (shareware);
- бесплатное (freeware).

Примеры использования

1. *Офисная сфера.* Создание отчетов, деловой переписки, финансовых таблиц.
2. *Инженерия.* Проектирование зданий и машин в AutoCAD и SolidWorks.
3. *Бизнес.* Автоматизация производственных процессов в ERP-системах.
4. *Наука.* Анализ данных в MATLAB и SPSS.
5. *Образование.* Проведение дистанционных лекций и тестов через Moodle.
6. *Медицина.* Обработка изображений компьютерной томографии.

Место прикладного ПО в вычислительной системе

Пользователь
↓
Прикладное программное обеспечение
↓
Системное программное обеспечение (ОС, драйверы)
↓
Аппаратное обеспечение

Значение прикладного ПО

Прикладное программное обеспечение является ключевым фактором цифровизации современного общества. Оно обеспечивает:

- повышение эффективности труда;
- автоматизацию процессов;
- возможность обработки больших объемов информации;
- поддержку принятия управленческих решений;
- интеграцию в глобальное информационное пространство.

Развитие прикладного ПО напрямую связано с прогрессом в области искусственного интеллекта, облачных технологий и мобильных устройств. Оно формирует основу для внедрения новых цифровых сервисов, которые определяют конкурентоспособность организаций и государств.

2.7. Инструментальное программное обеспечение

Инструментальное программное обеспечение представляет собой особый класс программ, предназначенных не для конечного пользователя, а для специалистов-разработчиков. Его основная задача заключается в создании, отладке, сопровождении и оптимизации других программных продуктов.

Если прикладное ПО обеспечивает решение прикладных задач, а системное ПО управляет вычислительными ресурсами, то инструментальное ПО образует своего рода «метауровень» – оно служит средствами производства нового программного обеспечения.

Инструментальное программное обеспечение (ИПО) – это совокупность программных средств, используемых для разработки, тестирования, отладки и сопровождения прикладных и системных программ.

В отличие от прикладных программ, которые решают конечные задачи пользователя, инструментальное ПО адресовано программистам, инженерам и администраторам и обеспечивает технологическую базу разработки.

Основные функции инструментального ПО

1. *Редактирование и управление исходным кодом.*

Средства написания и структурирования программных текстов (текстовые редакторы, IDE).

2. *Компиляция и интерпретация.*

Преобразование исходного кода в машинные инструкции или его выполнение «на лету».

3. *Отладка и профилирование.*

Пошаговое выполнение программ, анализ переменных, выявление ошибок, измерение производительности.

4. *Автоматизация сборки и интеграции.*

Поддержка систем контроля версий, инструментов CI/CD, автоматических билдов.

5. *Тестирование.*

Проведение модульного, интеграционного и нагрузочного тестирования.

6. *Проектирование и моделирование.*

CASE-средства, позволяющие создавать диаграммы UML, автоматизировать архитектурное проектирование и документирование.

7. *Администрирование и сопровождение.*

Средства мониторинга, отладки и анализа производительности работающих систем.

Классификация инструментального ПО (табл. 2.7).

Таблица 2.7

Примеры инструментального ПО

Категория	*Примеры*	*Назначение*
IDE	Visual Studio, PyCharm, Eclipse	Редактирование, сборка и отладка программ
Компиляторы	gcc, javac	Преобразование исходного кода в машинный
Интерпретаторы	Python Interpreter, Node. js	Выполнение кода без компиляции
Отладчики	gdb, WinDbg, Valgrind	Выявление и устранение ошибок
Системы контроля версий	Git, Subversion	Управление изменениями кода
CI/CD	Jenkins, GitLab CI/CD	Автоматизация сборки и тестирования
CASE-средства	Enterprise Architect, ArgoUML	Проектирование архитектуры
Тестовые фреймворки	JUnit, PyTest	Автоматизированное тестирование

1. *Редакторы исходного кода и интегрированные среды разработки (IDE).*

Visual Studio, Eclipse, PyCharm, IntelliJ IDEA.

2. *Компиляторы и интерпретаторы.*

gcc (C/C++), javac (Java), Pyhon Interpreter, Node. js.

3. *Отладчики.*

gdb, WinDbg, Valgrind.

4. *Системы контроля версий.*

Git, Subversion, Mercurial

5. *Инструменты непрерывной интеграции и доставки (CI/CD).*

Jenkins, GitLab CI/CD, GitHub Actions.

6. *CASE-средства.*

Rational Rose, Enterprise Architect, ArgoUML.

7. *Тестовые фреймворки.*

JUnit (Java), PyTest (Python), NUnit (. NET).

8. *Системы управления проектами.*

Jira, Redmine, Trello (в совокупности с другими инструментами).

Схема применения инструментального ПО

Разработчик
↓
Редактор кода / IDE
↓
Компилятор / Интерпретатор
↓
Отладчик / Тестовый фреймворк
↓
Готовая программа

Примеры использования

- Разработка веб-приложений осуществляется в среде PyCharm или Visual Studio Code с использованием систем контроля версий Git.
- Инженеры используют CASE-средства для моделирования архитектуры программного комплекса и автоматической генерации кода.
- В крупных компаниях Jenkins обеспечивает автоматическую сборку проектов, запуск тестов и развертывание в рабочей среде.
- Для обеспечения качества программ применяется комбинация статического анализа кода, модульного тестирования и профилирования производительности.

Значение инструментального ПО

Инструментальное программное обеспечение обеспечивает технологическую основу всей индустрии программной инженерии. Оно позволяет:

- стандартизировать процесс разработки;
- ускорить выпуск программных продуктов;
- повысить их качество и безопасность;
- обеспечить коллективную разработку и контроль версий;
- снизить стоимость сопровождения и модернизации.

Без развитого инструментального ПО невозможно существование современных методов разработки, таких как Agile, DevOps, непрерывная интеграция и доставка.

2.8. Классификация языков программирования, примеры

Языки программирования являются фундаментальными средствами взаимодействия человека с вычислительными машинами. Именно с их помощью идеи, алгоритмы и абстрактные модели преобразуются в последовательность команд, выполняемых компьютером. Современная информатика оперирует десятками тысяч языков программирования, часть из которых используется крайне узко, а часть имеет глобальное распространение. Развитие языков отражает как эволюцию аппаратной базы, так и смену методологических парадигм в программной инженерии.

Классификация языков программирования необходима для систематизации знаний, выбора оптимальных инструментов для решения конкретных задач и понимания исторической логики их развития (табл. 2.8).

Таблица 2.8

Систематизация языков программирования

Классификация	*Примеры языков*	*Особенности*
Уровень: низкий	Assembler	Прямое управление оборудованием
Уровень: средний	C	Баланс эффективности и абстракции
Уровень: высокий	Java, Python, C#	Удобство, переносимость
Парадигма: императивные	C, Pascal	Последовательность команд
Парадигма: ООП	C++, Java, C#	Объекты, классы, наследование
Парадигма: функциональные	Lisp, Haskell, F#	Композиция функций
Парадигма: логические	Prolog	Решение через правила
Назначение: системные	C, Rust	ОС, драйверы
Назначение: прикладные	Java, Python, C#	Бизнес- и пользовательские приложения

Окончание табл. 2.8

Классификация	*Примеры языков*	*Особенности*
Назначение: научные	Fortran, MATLAB	Численные расчеты
Назначение: базы данных	SQL	Управление данными
Назначение: веб-разработка	JavaScript, PHP, Ruby	Клиент и сервер для веб-сервисов

Классификация по уровню близости к аппаратуре

1. *Языки низкого уровня.*

- Отличаются минимальной степенью абстракции.
- Программист работает напрямую с регистрами процессора и командами машинного кода.
- Основное преимущество – полный контроль над оборудованием и высокая эффективность.
- Основной недостаток – сложность разработки, низкая переносимость кода.
- Пример: Assembler (x86, ARM).

2. *Языки среднего уровня.*

- Сохраняют возможность прямого доступа к памяти, но обеспечивают удобство и структурированность кода.
- Баланс между производительностью и читаемостью.
- Широко используются для разработки системного ПО.
- Пример: C.

3. *Языки высокого уровня.*

- Абстрагированы от аппаратной реализации, ориентированы на удобство программиста.
- Поддерживают автоматическое управление памятью, развитые библиотеки и высокоуровневые конструкции.
- Позволяют создавать сложные проекты при минимальных затратах времени.
- Примеры: Python, Java, C#, Pascal.

Классификация по парадигмам программирования

Парадигма определяет методологию построения программ.

1. *Императивные языки.*

- Основаны на пошаговом изменении состояния программы.
- Характеризуются конструкциями «оператор – результат».
- Примеры: C, Pascal.

2. *Объектно-ориентированные языки.*

- Базируются на концепциях классов и объектов.

- Поддерживают инкапсуляцию, наследование, полиморфизм.
- Предназначены для построения масштабируемых программных систем.
- Примеры: Java, C++, C#, Python.

3. *Функциональные языки.*

- Основаны на математической теории функций.
- Основное средство – применение и композиция функций, а не последовательность команд.
- Способствуют созданию кода, свободного от побочных эффектов.
- Примеры: Lisp, Haskell, F#.

4. *Логические языки.*

- Используют формальную логику для описания фактов и правил.
- Выполнение программы заключается в поиске решений, удовлетворяющих заданным условиям.
- Пример: Prolog.

5. *Сценарные (скриптовые) языки.*

- Предназначены для автоматизации рутинных операций и управления приложениями.
- Чаще всего интерпретируемые.
- Примеры: JavaScript, Bash, PowerShell.

Классификация по назначению

1. *Системные языки.*

Применяются для разработки операционных систем, драйверов, компиляторов.

Примеры: C, Rust.

2. *Прикладные языки.*

Универсальные языки, используемые для создания бизнес-приложений, сервисов, мобильных систем.

Примеры: Java, Python, C#.

3. *Языки для научных расчетов.*

Ориентированы на численные вычисления и обработку массивов данных.

Примеры: Fortran, MATLAB.

4. *Языки для работы с базами данных.*

Специализированы для запросов и управления данными.

Пример: SQL.

5. *Языки для веб-разработки.*

Применяются для создания веб-приложений и сайтов.

Клиентская сторона: JavaScript.

Серверная сторона: PHP, Ruby, Node. js.

Классификация языков программирования

Языки программирования

```
│
├── По уровню
│   ├── Низкого уровня (Assembler)
│   ├── Среднего уровня (C)
│   └── Высокого уровня (Java, Python, C#)
│
├── По парадигме
│   ├── Императивные
│   ├── Объектно-ориентированные
│   ├── Функциональные
│   ├── Логические
│   └── Сценарные
│
└── По назначению
    ├── Системные
    ├── Прикладные
    ├── Научные
    ├── Базы данных
    └── Веб-языки
```

Примеры практического применения

Assembler используется при создании микропрограмм, драйверов устройств и встроенных систем.

C применяется для разработки ядра Linux и многих других операционных систем.

Java широко используется для корпоративных приложений и Android-разработки.

Python применяется в машинном обучении, обработке данных, веб-разработке.

SQL является универсальным инструментом работы с реляционными базами данных.

JavaScript используется для создания интерактивных веб-страниц и клиентских приложений.

Значение классификации

Классификация языков программирования играет важную роль в обучении, научных исследованиях и практике разработки. Она позволяет:

– систематизировать разнообразие языков;

– облегчить выбор инструментов для конкретной задачи;

– выявить сильные и слабые стороны различных подходов;

– понимать историческую и методологическую эволюцию программирования.

Современные тенденции показывают, что языки программирования все чаще сочетают в себе несколько парадигм (например, Python и Scala), что расширяет их функциональность и делает их универсальными инструментами в руках специалистов.

2.9. Этапы подготовки исполняемого кода

Когда программист пишет программу на языке программирования, компьютер не может сразу ее понять. Машина «понимает» только наборы нулей и единиц – машинный код. Поэтому между написанием программы и ее запуском проходит несколько этапов, на которых исходный текст переводится в форму, удобную для компьютера. Эти этапы называются подготовкой исполняемого кода.

Общая схема

Исходный код (написанный программистом)
↓
Компиляция или интерпретация
↓
Объектный код (части машинных команд)
↓
Компоновка (сборка вместе с библиотеками)
↓
Исполняемый файл (. exe,. out)
↓
Загрузка в память и запуск

1. Исходный код

Программист пишет текст программы на языке высокого уровня (например, C, Python, Java). Этот текст легко читать человеку, но компьютер его не понимает.

Пример:

```
#include <stdio. h>
int main() {
printf("Hello, world!\n");
return 0;
}
```

2. Компиляция

Компиляция – это перевод программы с языка программирования в «понятный» для компьютера вид.

Программа-компилятор проверяет синтаксис (правильность написания).

Если ошибок нет, исходный код переводится в объектный.

На этом этапе также выполняется *оптимизация* – компилятор старается сделать код быстрее и эффективнее.

Примеры компиляторов:

для C и C++ – gcc, clang;

для Java – javac (создает не машинный код, а байткод).

3. Объектный код

После компиляции получается объектный файл (. o или. obj). Это уже набор машинных команд, но программа пока еще не готова к запуску. В ней могут отсутствовать некоторые части – например, функции из библиотек.

4. Компоновка (линковка)

Компоновщик соединяет все объектные файлы и нужные библиотеки в одну цельную программу.

Если программа использует стандартные функции (например, printf), компоновщик добавляет их из библиотек.

В результате получается исполняемый файл (. exe в Windows, . out в Linux).

5. Интерпретация (альтернативный вариант)

Не все языки требуют компиляции.

В интерпретируемых языках (Python, JavaScript) программа выполняется построчно специальной программой – *интерпретатором*.

Это проще, но работает медленнее, чем скомпилированный код.

Пример: в Python достаточно написать файл script. py и запустить его командой:

```
python script. py
```

6. Байткод и виртуальные машины

Некоторые языки используют промежуточный вариант.

В Java и C# исходный код сначала переводится в *байткод*.

Байткод не зависит от операционной системы и процессора.

Для выполнения байткод запускается на виртуальной машине (JVM для Java, CLR для C#).

Преимущество: один и тот же байткод можно запустить на разных компьютерах (Windows, Linux, macOS).

7. Загрузка в память и запуск

Когда программа готова, операционная система загружает ее в оперативную память.

Загрузчик проверяет файл.

В память помещаются код и данные.

Управление передается на первую функцию (например, main). С этого момента программа начинает работать.

Практический пример (табл. 2.9)

1. Программист написал программу на C.
2. Компилятор gcc перевел ее в объектный код.
3. Компоновщик связал код с библиотекой libc.
4. Получился исполняемый файл program. exe.

Операционная система загрузила его в память и запустила.

Таблица 2.9

Сравнение способов подготовки кода

Подход	*Примеры языков*	*Особенности*
Компиляция	C, C++, Fortran	Быстро работает, но не всегда переносимо
Интерпретация	Python, JavaScript	Легко запускать, но медленнее
Байткод + VM	Java, C#	Хорошая переносимость, баланс скорости и удобства

Зачем знать эти этапы

Чтобы понимать, почему иногда программа не запускается (ошибки компиляции, отсутствующие библиотеки).

Чтобы оптимизировать работу программы.

Чтобы лучше разбираться в устройстве языков программирования.

Чтобы уметь работать с разными системами (Windows, Linux, macOS).

Таким образом, подготовка исполняемого кода – это «мост» между идеями программиста и машиной, позволяющий превратить текст программы в работающий инструмент.

2.10. Транслятор, компоновщик, отладчик

При подготовке программного обеспечения к исполнению используются специальные программные средства, которые помогают перевести исходный код в машинные инструкции, объединить отдельные части программы и выявить ошибки. К таким средствам относятся *транслятор*, *компоновщик* и *отладчик*.

Транслятор

Транслятор – это программа, которая переводит исходный код, написанный на языке программирования, в форму, понятную компьютеру.

Существует множество разновидностей трансляторов.

1. *Компилятор* – переводит весь исходный код сразу в машинные инструкции или промежуточный код.

Пример: компилятор gcc для C и C++, javac для Java.

2. *Интерпретатор* – выполняет программу построчно, не создавая отдельного исполняемого файла.

Пример: интерпретатор Python, JavaScript-движки (V8).

3. *Гибридные трансляторы* – используют оба подхода: сначала создают промежуточный код, а затем исполняют его на виртуальной машине.

Пример: Java (байткод + JVM), C# (байткод + CLR).

Функции транслятора:

– проверка синтаксиса и семантики кода;
– диагностика ошибок;
– генерация объектного или исполняемого кода;
– оптимизация программы.

Компоновщик

Компоновщик (linker) – программа, которая объединяет несколько объектных файлов и библиотек в единый исполняемый файл.

Компоновщик решает следующие задачи:

– связывает функции и переменные, определенные в разных модулях;
– подключает стандартные библиотеки (например, функции printf, sqrt и т. д.);
– формирует итоговый исполняемый файл (. exe в Windows, . out в Linux).

Пример

при компиляции программы на C компилятор создает объектные файлы (.o), а компоновщик объединяет их с библиотеками (libc) и выдает исполняемый файл.

Схема работы компоновщика

Отладчик

Отладчик (debugger) – программа, которая помогает находить и исправлять ошибки в коде. Он позволяет программисту выполнять программу пошагово, наблюдать за изменением переменных и анализировать причину сбоев.

Функции отладчика:

– запуск программы в «пошаговом режиме»;

– установка *точек останова* (breakpoints), чтобы приостановить выполнение;

– просмотр значений переменных и состояния памяти;

– анализ вызовов функций (стек вызовов);

– диагностика причин аварийного завершения.

Примеры отладчиков:

- gdb (GNU Debugger) – для C/C++ в Linux;
- встроенный отладчик в Visual Studio;
- pdb (Python Debugger) – для Python.

Пример работы отладчика

```
int main() {
int a = 5, b = 0;
int c = a / b; // Ошибка: деление на ноль
return 0;
}
```

При запуске отладчик укажет строку с ошибкой.

Программист сможет пошагово проследить выполнение и увидеть, что переменная $b = 0$.

Роль инструментов в процессе разработки

Транслятор обеспечивает преобразование исходного кода в исполняемую форму.

Компоновщик собирает части программы и подключает библиотеки.

Отладчик помогает исправлять ошибки и контролировать работу программы.

Эти средства являются неотъемлемой частью инструментального программного обеспечения и используются на каждом этапе жизненного цикла разработки (табл. 2.10).

Таблица 2.10

Сравнение транслятора, компоновщика и отладчика

Средство	*Назначение*	*Примеры*
Транслятор	Переводит исходный код в объектный	gcc, javac, Python Interpreter
Компоновщик	Объединяет объектные файлы и библиотеки	ld (Linux), link. exe (Windows)
Отладчик	Находит и исправляет ошибки	gdb, Visual Studio Debugger, pdb

2.11. Интегрированные среды разработки, состав

Современная разработка программ невозможна без применения комплексных инструментальных систем, которые объединяют в себе средства редактирования, компиляции, отладки и тестирования. Такие системы получили название *интегрированные среды разработки* (Integrated Development Environment, IDE).

Интегрированная среда разработки (IDE) – это программный комплекс, объединяющий в одном интерфейсе инструменты, необходимые для создания, тестирования и сопровождения программного обеспечения.

Главная цель IDE заключается в том, чтобы облегчить работу программиста, уменьшить количество рутинных действий и повысить эффективность разработки.

Состав интегрированной среды разработки

IDE включает в себя спостоянный состав компонент.

1. *Редактор исходного кода.*

Поддерживает подсветку синтаксиса, автодополнение кода, форматирование.

Часто включает встроенные шаблоны и фрагменты кода.

2. *Компилятор и/или интерпретатор.*

Позволяет напрямую запускать и проверять программу без выхода из среды.

3. *Отладчик.*

Поддерживает пошаговое выполнение программы, установку точек останова, просмотр переменных и состояния памяти.

4. *Система управления проектами.*

Организация исходных файлов в виде проекта.

Управление зависимостями и настройками сборки.

5. *Интеграция с системами контроля версий.*

Поддержка Git, Subversion и других средств командной работы.

6. *Инструменты тестирования.*

Автоматизированный запуск модульных тестов.

Генерация отчетов о покрытии кода тестами.

7. *Графический интерфейс пользователя (GUI-редактор).*

В некоторых IDE есть конструкторы интерфейсов для приложений.

8. *Документация и справочные материалы.*

Встроенные подсказки по библиотекам и функциям (табл. 2.11).

Таблица 2.11

Основные компоненты IDE

Компонент	*Назначение*
Редактор кода	Написание и редактирование исходного текста
Компилятор / интерпретатор	Преобразование исходного кода в исполняемую форму
Отладчик	Поиск и устранение ошибок
Система управления проектом	Организация исходных файлов и зависимостей
Интеграция с VCS	Совместная разработка и контроль версий
Средства тестирования	Автоматизация проверки качества кода
GUI-редактор	Создание интерфейсов
Справочные материалы	Поддержка и документация

Примеры интегрированных сред разработки

Visual Studio (C++, C#,. NET) – мощная IDE от Microsoft, используется для создания приложений под Windows и веб-сервисов.

IntelliJ IDEA (Java, Kotlin) – популярная IDE для разработки на JVM-языках.

PyCharm (Python) – специализированная среда для Python, поддерживает Django и научные библиотеки.

Eclipse (Java, C, C++) – свободная и кроссплатформенная IDE.

NetBeans (Java, PHP) – открытая среда, активно используемая в образовании.

Xcode (Swift, Objective-C) – основная IDE для разработки приложений под macOS и iOS.

Роль IDE в процессе разработки

Программист
↓
Редактор кода → Компилятор/Интерпретатор → Отладчик → Тестирование
↓
Система контроля версий и документация
↓
Готовое программное обеспечение

Значение IDE

Интегрированные среды разработки позволяют:

- сократить время написания и тестирования программ;
- уменьшить количество ошибок;

- автоматизировать рутинные процессы (сборка, тестирование, публикация);
- облегчить совместную работу программистов;
- обеспечить единый стандарт разработки в команде.

Современные IDE фактически стали стандартом в программной инженерии, поскольку они объединяют в себе все этапы жизненного цикла программного продукта – от написания кода до его сопровождения.

2.12. Автоматическая генерация кода. CASE-средства

Автоматизация процессов программной инженерии является важным направлением развития информационных технологий. По мере усложнения программных систем потребность в ускорении разработки и снижении числа ошибок привела к созданию инструментов, позволяющих генерировать код автоматически. Такие инструменты опираются на концепцию *CASE-средств* (Computer-Aided Software Engineering), которые обеспечивают поддержку жизненного цикла программного обеспечения.

Автоматическая генерация кода

Автоматическая генерация кода – это процесс преобразования моделей, диаграмм или спецификаций в программный код с помощью специализированных инструментов. Вместо того чтобы писать код вручную, разработчик строит модель системы, а программа автоматически формирует соответствующие классы, функции и структуры.

Основные цели автоматической генерации кода:

- сокращение времени разработки;
- снижение числа синтаксических ошибок;
- обеспечение единообразия стиля программирования;
- упрощение сопровождения и модификации программ.

Примеры применения:

- генерация HTML-страниц из шаблонов;
- создание SQL-запросов по описанию структуры базы данных;
- автоматическое создание кода классов на основе UML-диаграмм;
- генерация API-методов из спецификаций OpenAPI/Swagger.

Преимущества автоматической генерации кода:

- скорость разработки (быстрее, чем вручную);
- снижение количества ошибок;
- стандартизация программного продукта.

Недостатки:

- сгенерированный код может быть избыточным;
- требуется дополнительная настройка и оптимизация;
- ограниченность в случае сложной бизнес-логики.

CASE-средства (табл. 2.12)

Таблица 2.12

Примеры CASE-средств

CASE-средство	*Основные возможности*
Rational Rose (IBM)	Моделирование UML, генерация кода на C++, Java
Enterprise Architect	Поддержка полного жизненного цикла ПО
Visual Paradigm	UML, BPMN, интеграция с IDE
ArgoUML	Свободное средство моделирования UML
MagicDraw	Моделирование систем, реверс–инжиниринг

CASE-средства (Computer-Aided Software Engineering) – это программные комплексы, предназначенные для автоматизации процессов анализа, проектирования, разработки и сопровождения программного обеспечения.

Функции CASE-средств:

1. *Моделирование систем.* Создание UML-диаграмм (классов, вариантов использования, последовательностей).

2. *Генерация кода.* Преобразование диаграмм в каркас программных модулей.

3. *Реверс-инжиниринг.* Построение диаграмм на основе уже существующего кода.

4. *Управление требованиями.* Хранение и отслеживание требований к проекту.

5. *Документирование.* Автоматическая генерация технической документации.

Схема применения CASE-средств

Анализ требований
↓
Моделирование (диаграммы, спецификации)
↓
Автоматическая генерация кода
↓
Редактирование и отладка в IDE
↓
Готовое программное обеспечение

Значение автоматической генерации кода и CASE-средств

- позволяют ускорить процесс разработки программных систем;
- уменьшают зависимость от человеческого фактора;
- обеспечивают стандартизацию архитектурных решений;
- поддерживают полный жизненный цикл ПО: от постановки требований до сопровождения;
- активно используются в крупных проектах, где важны согласованность действий команды и высокая надежность продукта.

2.13. Использование UML для генерации кода

Разработка сложных программных систем требует формализованного описания архитектуры, структуры и поведения компонентов. Одним из наиболее распространенных средств моделирования является *UML (Unified Modeling Language – унифицированный язык моделирования)*. UML применяется не только для документирования систем, но и для автоматической генерации кода на основе созданных моделей.

UML как средство моделирования

UML – это универсальный графический язык для визуализации, спецификации, конструирования и документирования программных систем. Он основан на объектно-ориентированном подходе и поддерживает такие ключевые элементы, как классы, объекты, наследование, ассоциации, диаграммы взаимодействия.

Основные типы UML-диаграмм:

- *диаграммы структуры* (классов, компонентов, развертывания);
- *диаграммы поведения* (вариантов использования, активности, состояний);
- *диаграммы взаимодействия* (последовательности, коммуникации).

Принцип генерации кода из UML

Использование UML для генерации кода основано на том, что модель, описывающая классы и их связи, может быть преобразована в каркас программного кода.

Алгоритм процесса.

1. Построение UML-диаграмм (например, диаграммы классов).
2. Описание атрибутов и методов классов.
3. Указание связей (наследование, агрегация, композиция).
4. Применение CASE-средства для генерации исходного кода.
5. Доработка и детализация кода вручную программистом.

Пример: UML-диаграмма классов

Класс: Student

– name: string
– age: int
+ getInfo(): string

Автоматическая генерация кода (на языке Java):

```
public class Student {
private String name;
private int age;

public String getInfo() {
return name + " (" + age + ")";
}
}
```

Преимущества генерации кода из UML (табл. 2.13)

- *Сокращение времени разработки.* Часть кода создается автоматически.
- *Унификация структуры.* Все программисты команды получают единый каркас кода.
- *Снижение числа ошибок.* Исключаются ошибки при ручном написании шаблонов классов.
- *Автоматическая документация.* UML-модель остается синхронизированной с кодом.

Таблица 2.13

Инструменты для генерации кода из UML

CASE-средство	*Поддерживаемые языки*	*Особенности*
Enterprise Architect	Java, C++, C#, PHP	Поддержка полного цикла ПО
Rational Rose (IBM)	Java, C++	Классическое средство UML-моделирования
Visual Paradigm	Java, C#, PHP, Python	Двусторонняя синхронизация код ↔ диаграммы
ArgoUML	Java	Бесплатное средство с открытым кодом
MagicDraw	Java, C#, C++	Интеграция с IDE, реверс-инжиниринг

Ограничения

- Автоматическая генерация формирует только *каркас программы*, бизнес-логика должна быть написана вручную.
- При изменении модели требуется повторная синхронизация с кодом.
- Возможны избыточность и сложность сгенерированного кода.

Использование UML для генерации кода

UML-модель (диаграмма классов, вариантов использования)
↓
CASE-средство
↓
Автоматическая генерация кода
↓
Доработка и тестирование в IDE
↓
Готовое программное обеспечение

Значение UML для программной инженерии

Использование UML для генерации кода повышает уровень формализации проектирования и позволяет интегрировать этап моделирования в общий процесс разработки. Такой подход является частью методологий *Model-Driven Development (MDD)* и *Model-Driven Architecture (MDA)*, которые рассматривают модель как основной артефакт разработки.

Таким образом, UML обеспечивает «мост» между концептуальным проектированием и практической реализацией, ускоряя процесс разработки и улучшая качество программных продуктов.

Контрольные вопросы

1. Дайте определение понятий *программа* и *программное обеспечение*. Чем они отличаются?
2. В чем заключается инженерный подход к конструированию программного обеспечения?
3. Как классифицируется программное обеспечение по назначению? Приведите примеры каждой категории.
4. Что относится к системному программному обеспечению и какие основные функции оно выполняет?
5. Перечислите ключевые функции операционной системы и объясните их значение.
6. Какие классы прикладного программного обеспечения существуют? Приведите примеры их использования в разных сферах.
7. Что такое инструментальное программное обеспечение? Какие средства в него входят и какова их роль в разработке ПО?
8. Объясните классификацию языков программирования по уровням абстракции, парадигмам и назначению.
9. Каковы основные этапы подготовки исполняемого кода? Опишите роль транслятора, компоновщика и отладчика.
10. Что такое CASE-средства и UML? Как они используются для автоматической генерации кода и чем полезны в программной инженерии?

3. ОСНОВЫ ТЕОРИИ АЛГОРИТМОВ

3.1. Алгоритм: виды и формы представления

Понятие алгоритма

Алгоритм является одним из центральных понятий информатики и программной инженерии. В самом общем виде *алгоритм* – это конечная упорядоченная последовательность инструкций, выполнение которых приводит к решению задачи или получению требуемого результата.

Иными словами, алгоритм можно рассматривать как «рецепт решения задачи», где каждая инструкция должна быть четко определена, а результат выполнения предсказуем и воспроизводим.

Слово «алгоритм» происходит от латинизированного имени персидского математика аль-Хорезми (лат. *Algorithmi*), который в IX в. описал методы вычислений с использованием арабских цифр. Его работы положили начало формированию математической дисциплины, посвященной систематическому описанию вычислений.

Основные свойства алгоритма

Классическое определение алгоритма включает ряд свойств, без которых последовательность действий не может считаться алгоритмом:

1. *Конечность* – выполнение алгоритма должно завершаться за конечное число шагов. Алгоритм не может быть бесконечным процессом.

2. *Определенность (детерминированность)* – каждое действие должно быть ясно описано и не допускать двусмысленности. Исполнитель алгоритма всегда понимает, что нужно делать.

3. *Результативность* – выполнение алгоритма должно приводить к получению результата, который удовлетворяет исходной задаче.

4. *Массовость* – алгоритм должен решать не только одну задачу, но и целый класс задач с аналогичной структурой.

5. *Дискретность* – алгоритм состоит из отдельных элементарных шагов, которые выполняются последовательно.

Пример

Алгоритм сложения двух чисел всегда состоит из конечного числа шагов, имеет понятные инструкции («взять число А», «взять число В», «сложить их»), гарантирует результат и применим к любым парам чисел.

Виды алгоритмов

Классификация алгоритмов помогает систематизировать их и выбрать наиболее подходящий метод решения задачи.

1. *По структуре управления.*

- *Линейные* – шаги выполняются строго последовательно.

Пример: вычисление площади круга по формуле $S = \pi R^2$.

- *Разветвляющиеся* – выбор дальнейших действий зависит от условия.

Пример

Проверка четности числа (если $n \bmod 2 = 0$, то число четное, иначе нечетное).

- *Циклические* – содержат повторяющиеся действия.

Пример: поиск максимального элемента в массиве путем последовательного перебора всех значений.

2. *По типу обрабатываемых данных:*

- *Численные алгоритмы* – выполняют арифметические операции.
- *Логические алгоритмы* – работают с истиной и ложью (булевой алгеброй).
- *Текстовые алгоритмы* – предназначены для обработки строк и символов (например, алгоритм поиска подстроки Кнута – Морриса – Пратта).

3. *По степени детерминированности:*

- *Детерминированные* – при одинаковых входных данных всегда дают одинаковый результат.
- *Недетерминированные* – могут приводить к разным результатам (например, алгоритмы на основе случайного выбора).

4. *По уровню универсальности:*

- *Специализированные* – решают конкретную задачу (например, сортировка пузырьком).
- *Универсальные* – применимы к широкому классу задач (например, алгоритм Евклида для нахождения наибольшего общего делителя).

5. *По степени автоматизации:*

- *Формализованные* – все шаги строго описаны и не требуют вмешательства человека.
- *Эвристические* – содержат элементы выбора, которые зависят от опыта или интуиции.

Пример: эвристики в алгоритмах поиска решений NP-полных задач.

Формы представления алгоритмов

Алгоритмы могут быть представлены разными способами. Выбор формы зависит от аудитории (человека или машины) и цели (объяснение, проектирование, программирование).

1. *Словесное описание.*

Используется для объяснения сути алгоритма на естественном языке.

Пример

«Возьми два числа, сравни их. Если первое больше, выведи его, иначе выведи второе».

2. *Псевдокод.*

Универсальное средство описания алгоритмов, близкое к языкам программирования, но независимое от конкретного синтаксиса.

Пример:

Ввод a, b
Если a > b то
Вывести a
Иначе
Вывести b
Конец

3. *Блок-схема.*

Графическое представление алгоритма с использованием стандартных фигур (прямоугольник – действие, ромб – условие, овал – начало/конец).

4. *Табличная форма.*

Удобна для логических функций и конечных автоматов (табл. 3.1).

Таблица 3.1

Логическая функция И (AND)

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

5. *Формулы и уравнения.*

Используются в математических алгоритмах (например, формула Герона для вычисления площади треугольника).

Практическое значение

- В *обучении и проектировании* словесные описания и блок-схемы помогают студентам и разработчикам понять логику работы алгоритма.
- В *программировании* чаще всего используется псевдокод, так как он легко переводится на любой язык.
- В *математике* алгоритмы часто задаются через формулы.
- В *прикладных областях* (экономика, биоинформатика, робототехника) выбор формы зависит от задач и исполнителей (человек или машина).

Таким образом, алгоритм является универсальной формой описания процесса, которая лежит в основе всей вычислительной техники и программной инженерии.

3.2. Характеристики и свойства алгоритмов

Понятие алгоритма имеет фундаментальное значение для информатики и программной инженерии. Однако сам факт наличия последовательности действий еще не делает ее алгоритмом в строгом смысле. Чтобы процедура могла считаться алгоритмом, она должна обладать рядом характеристик и свойств, которые обеспечивают корректность выполнения, воспроизводимость результата и применимость к широкому классу задач.

Эти свойства были систематизированы в XX в. в рамках становления *теории алгоритмов* (работы Черча, Тьюринга, Маркова, Колмогорова). Именно в этот период стало возможным строгое математическое описание вычислений, что позволило отделить корректные и вычислимые процедуры от неформализованных рассуждений.

Основные свойства алгоритмов

1. *Конечность.*

Алгоритм обязан завершаться после выполнения конечного числа шагов.

- Если процесс бесконечен, он перестает быть алгоритмом и превращается в «правило поведения».
- В программировании нарушение конечности часто проявляется как *зацикливание* (например, бесконечный цикл while (true) без условия выхода).

Пример

- Алгоритм Евклида для нахождения наибольшего общего делителя всегда завершится, так как числа на каждом шаге уменьшаются.
- Алгоритм поиска простых чисел полным перебором делителей работает бесконечно, если не задать верхнюю границу.

2. *Определенность (детерминированность).*

Каждая инструкция алгоритма должна быть строго и однозначно сформулирована.

- В естественном языке часто встречаются двусмысленности («возьмите немного», «повторите несколько раз»), что неприемлемо для алгоритма.
- В программировании это выражается в строгом синтаксисе команд.

Пример

- «Возьми число N, если оно четное – раздели пополам, иначе умножь на 3 и прибавь 1» – определенный алгоритм.
- «Возьми число и сделай с ним что-нибудь» – не является алгоритмом.

3. *Результативность.*

Алгоритм должен гарантировать получение результата, если заданы допустимые входные данные.

- Это свойство отличает рабочий алгоритм от неполного описания процесса.
- В информатике результатом может быть не только число, но и структурированный ответ, текст, изображение или даже сообщение об ошибке.

Пример

- Алгоритм сортировки массива всегда выдает упорядоченный массив.
- Если алгоритм может бесконечно «думать» и не выдать результат, он нарушает это свойство.

4. *Массовость (универсальность).*

Алгоритм должен быть применим не к одному конкретному примеру, а ко всему множеству задач данного типа.

- Это свойство делает алгоритмы ценными, так как они становятся универсальными решениями.

Пример

- Алгоритм нахождения наибольшего числа в массиве работает для любого массива, а не только для {2, 5, 7}.
- «Посчитать площадь треугольника со сторонами 3, 4 и 5» – не алгоритм, а единичная инструкция.

5. *Дискретность.*

Алгоритм должен состоять из последовательности отдельных шагов, каждый из которых выполняется за конечное время.

- Это свойство отражает пошаговый характер вычислений.
- В вычислительной технике шаг соответствует машинной инструкции или группе инструкций.

Пример

- «Вычислить сумму чисел массива» означает пошаговое сложение: взять первое число, прибавить второе, затем третье и так далее.

6. *Эффективность (практическая реализуемость).*

Алгоритм должен быть не только корректным, но и исполнимым за разумное время и с допустимыми затратами ресурсов.

- В реальной практике разработка программ невозможна без учета ограничений по времени выполнения и объему памяти.
- Некоторые алгоритмы, например, полный перебор комбинаций, формально корректны, но на практике неэффективны.

Дополнительные характеристики алгоритмов

Помимо основных свойств, в информатике рассматриваются дополнительные характеристики, позволяющие оценить качество алгоритмов (табл. 3.2).

Таблица 3.2

Характеристики алгоритмов

Характеристика	*Суть*	*Пример*
Конечность	Алгоритм всегда завершается	Алгоритм Евклида
Определенность	Нет двусмысленных шагов	Проверка четности
Результативность	Всегда выдает результат	Сортировка массива
Массовость	Применим к классу задач	Поиск максимума
Дискретность	Выполняется пошагово	Цикл «for»
Эффективность	Решение выполнимо за разумное время	Бинарный поиск
Временная сложность	Количество операций	$O(n^2)$, $O(n \log n)$
Пространственная	Использование памяти	Сортировка слиянием
Надежность	Корректность при разных данных	Алгоритм Евклида
Адаптивность	Изменение стратегии	Быстрая сортировка
Простота	Легкость понимания и реализации	Сортировка выбором

1. *Временная сложность.*

Количество шагов, необходимых для выполнения алгоритма, выражается через асимптотические оценки ($O(n)$, $O(n^2)$, $O(\log n)$).

Пример

Алгоритм сортировки пузырьком имеет сложность $O(n^2)$, а быстрая сортировка – $O(n \log n)$.

2. *Пространственная сложность (по памяти).*

Оценка объема памяти, необходимого для работы алгоритма.

Пример

Сортировка вставками требует минимальной дополнительной памяти, а сортировка слиянием использует дополнительный массив.

3. *Надежность.*

Алгоритм должен правильно работать при любых допустимых входных данных.

Если алгоритм «падает» на особых случаях (например, деление на ноль), он ненадежен.

4. *Адаптивность.*

Возможность изменять стратегию в зависимости от входных данных.

Пример

Некоторые алгоритмы сортировки работают быстрее на «почти отсортированных» массивах.

5. *Простота понимания и реализации.*

Чем проще алгоритм для понимания и реализации, тем выше его практическая ценность.

Иногда простой алгоритм (например, сортировка выбором) используется чаще, чем более эффективный, но сложный.

Исторический аспект

В античности свойства алгоритмов еще не выделялись, но уже существовали алгоритмические процедуры: вычисления по алгоритму Евклида (III в. до н. э.), правила построения таблиц вавилонскими астрономами.

В XX в. понятие алгоритма приобрело строгую математическую основу благодаря трудам Черча, Тьюринга, Поста, Маркова.

С развитием вычислительной техники стало ясно, что одних «корректных» алгоритмов недостаточно – нужны эффективные, работающие за разумное время. Так возникла теория сложности.

Практическое значение

Знание характеристик алгоритмов имеет огромное значение в программной инженерии:

- позволяет оценивать, насколько алгоритм применим в реальной задаче;
- помогает прогнозировать скорость работы и потребление ресурсов;
- позволяет сравнивать разные алгоритмы для выбора оптимального;
- гарантирует надежность программного обеспечения при работе с различными входными данными.

Таким образом, характеристики алгоритмов являются фундаментом, связывающим теорию вычислений с практикой программной инженерии.

3.3. Информация и алгоритмы ее обработки

Понятие информации

Термин *информация* (от лат. *informatio* – разъяснение, изложение, представление) является одним из наиболее универсальных в современном научном знании. Он используется в философии, кибернетике, физике, биологии, экономике, социологии. В каждой области определение информации имеет особенности, но в информатике под ней понимаются *сведения об объектах и процессах окружающего мира, которые могут быть восприняты, переданы, сохранены и обработаны*.

В контексте вычислительной техники информация связывается с понятием *данных*, т. е. формализованных сведений, представленных в форме, пригодной для хранения в памяти компьютера и обработки алгоритмами.

Информация обладает следующими свойствами:

- *объективность* – существует независимо от субъекта, но становится значимой только в процессе восприятия или использования;
- *достоверность* – отражает действительное состояние объекта или явления;
- *полезность* – имеет ценность для принятия решений или совершения действий;
- *актуальность* – значима в определенный момент времени, но может устареть;
- *структурированность* – может быть преобразована в систему данных, пригодных для алгоритмической обработки.

Информация в теории алгоритмов

В программной инженерии информация рассматривается как объект, над которым работают алгоритмы. Алгоритм – это механизм, позволяющий преобразовать входную информацию в выходную, тем самым создавая новое знание.

Формально процесс обработки информации можно представить как функцию:

$$f: X \to Y,$$

где X – множество входных данных;

Y – множество выходных данных;

f – алгоритм (правило преобразования).

Виды информации и ее обработка

1. *Числовая информация.*

Наиболее изученная и традиционная форма информации.

Обрабатывается арифметическими алгоритмами: сложение, умножение, вычисление корней, интегрирование.

Пример: алгоритм быстрого возведения в степень O(log n).

2. *Текстовая информация.*

Представляется как последовательность символов.

Алгоритмы обработки включают поиск подстроки, проверку орфографии, шифрование.

Пример: алгоритм Кнута – Морриса – Пратта (КМП) для поиска слова в тексте.

3. *Графическая и аудиовизуальная информация.*

Представляется в виде массивов пикселей или звуковых сигналов.

Обрабатывается с помощью алгоритмов сжатия (JPEG, MP3), фильтрации (медианный фильтр), распознавания (алгоритмы компьютерного зрения).

4. *Логическая информация.*

Отражает истинность или ложность утверждений.

Обрабатывается с помощью логических операций И, ИЛИ, НЕ, таблиц истинности, булевой алгебры.

5. *Мультимедийная и комплексная информация.*

Включает текст, графику, звук, видео.

Алгоритмы должны учитывать синхронизацию различных типов данных (например, потоковое вещание видео в сети) (табл. 3.3).

Таблица 3.3

Типы информации и примеры алгоритмов ее обработки

Тип информации	*Способы представления*	*Примеры алгоритмов*
Числовая	Целые, вещественные числа	Алгоритм Евклида, методы интегрирования
Текстовая	Последовательность символов	Поиск подстроки КМП, LZW-сжатие
Графическая	Массивы пикселей	JPEG, распознавание образов
Логическая	Булевы переменные	Таблицы истинности, SAT-решатели
Мультимедийная	Комбинация разнородных данных	Алгоритмы потокового кодирования (MPEG-4)

Алгоритмы обработки информации

Ключевая задача программной инженерии – разработка эффективных алгоритмов обработки информации.

1. *Алгоритмы сортировки* (пузырек, быстрая сортировка, сортировка слиянием).

Цель: упорядочивание данных для ускорения последующей обработки.

2. *Алгоритмы поиска* (линейный поиск, бинарный поиск, индексные структуры – B-деревья).

Цель: быстрое нахождение нужных элементов.

3. *Алгоритмы сжатия* (Хаффман-кодирование, ZIP, JPEG).

Цель: уменьшение объема данных.

4. *Алгоритмы защиты и шифрования* (RSA, AES, эллиптические кривые).

Цель: сохранность и конфиденциальность информации.

5. *Алгоритмы анализа данных* (кластеризация, деревья решений, нейронные сети).

Цель: выявление закономерностей в больших объемах информации.

Теоретические подходы к информации

1. *Теория информации К. Шеннона* (1948 г.).

Определяет информацию через понятие *энтропии* – меры неопределенности случайной величины.

Основная идея: чем более непредсказуем источник, тем больше информации несет его сообщение.

Применение: кодирование данных, оптимизация каналов связи, криптография.

2. *Алгоритмическая теория информации А. Н. Колмогорова.*

Информация оценивается через длину минимального алгоритма, который способен воспроизвести данные.

Пример: строка «111111111111» имеет короткое описание («12 единиц подряд»), а строка «101011010011» не сжимается и считается более «сложной».

3. *Семиотический подход.*

Рассматривает информацию как систему знаков и символов.

Применяется в лингвистике, когнитивных науках, интерфейсных технологиях.

Общий процесс обработки информации

Входные данные (X)
↓
Алгоритм (f)
↓
Выходные данные (Y)

Ограничения алгоритмической обработки информации

- *Алгоритмическая неразрешимость.*

Существуют задачи, для которых не существует алгоритма (например, проблема останова Тьюринга, десятая проблема Гильберта).

- *Сложностные ограничения.*

Некоторые задачи формально решаемы, но требуют экспоненциального времени, что делает их практическое решение невозможным (например, задача коммивояжера в общем случае).

- *Погрешности и неопределенность.*

В реальных системах данные могут быть неполными или зашумленными, что требует использования вероятностных и эвристических алгоритмов.

Современные тенденции обработки информации

- *Big Data* – алгоритмы обработки огромных массивов данных (MapReduce, Spark).
- *Машинное обучение и нейронные сети* – адаптивные алгоритмы, которые учатся на данных и улучшают результаты со временем.

- *Квантовые алгоритмы* (Шора, Гровера) – новые подходы к обработке информации с использованием принципов квантовой механики.
- *Интеллектуальные системы* – алгоритмы обработки естественного языка (NLP), компьютерного зрения, систем рекомендаций.

Практическое значение

- В программной инженерии правильный выбор алгоритма обработки информации определяет скорость работы, надежность и конкурентоспособность программного продукта.
- В повседневной жизни алгоритмы обработки информации незаметно сопровождают человека: от поиска в Google и рекомендаций в YouTube до систем медицинской диагностики и финансовых транзакций.
- В перспективе развитие алгоритмов обработки информации определяет эволюцию цифрового общества: от «умных городов» до искусственного интеллекта.

3.4. Определение алгоритма по Колмогорову и Маркову

Проблема формального определения алгоритма

На протяжении веков алгоритмы рассматривались в прикладном смысле – как правила для вычислений или решения задач. Однако до XX в. отсутствовало строгое математическое определение этого понятия. С развитием вычислительной техники и формальной логики возникла необходимость строго описать, *что такое алгоритм*, и установить границы того, что вообще может быть вычислено.

В 1930–1950-е гг. несколько ученых предложили независимые, но эквивалентные модели, позволяющие формализовать понятие алгоритма. Среди них ключевую роль сыграли работы А. Н. Колмогорова и А. А. Маркова, предложивших собственные определения и подходы.

Подход А. Н. Колмогорова

Академик Андрей Николаевич Колмогоров рассматривал алгоритм с точки зрения его описания и «сложности».

Основная идея:

- алгоритм можно определить через минимальную длину программы, которая описывает данные или процесс;
- чем короче программа (в битах), тем меньше «алгоритмическая сложность» объекта.

Эта концепция получила название *алгоритмическая теория информации*.

Пример

Строка 1111111111111111 (16 единиц) может быть описана как «16 раз подряд написать 1». Алгоритм здесь очень короткий.

Строка 1010011100101110 не имеет простого описания и требует «длинной» программы для генерации.

Таким образом, Колмогоров предложил измерять «информацию» и «сложность» объектов через алгоритмы, которые их порождают.

Значение подхода Колмогорова:

- связал понятия алгоритма и информации;
- стал основой для современной теории сжатия данных, оценки случайности и сложности систем;
- дал формальное определение «Колмогоровской сложности».

Подход А. А. Маркова

Советский математик Андрей Андреевич Марков предложил формализацию алгоритма через систему переписывания строк.

Нормальные алгоритмы Маркова – это упорядоченные наборы правил вида:

$$X \to Y$$

где X – подстрока, которую нужно заменить;

Y – подстрока, на которую происходит замена.

Правила применяются последовательно, пока возможно преобразование. Таким образом, вычисление сводится к последовательным преобразованиям символов.

Пример

Задача: увеличить число, записанное в унарной системе (например, 111 означает число 3).

Правило:

1$ → 11

(где $ – конец строки).

Применяя его, строка 111$ превращается в 1111$, что соответствует увеличению числа на 1.

Значение подхода Маркова:

- универсальность: любые вычислимые функции можно выразить с помощью нормальных алгоритмов;
- связь с языками и автоматами: система Маркова является одной из базовых моделей вычислений, наряду с машинами Тьюринга.

Сравнение подходов Колмогорова и Маркова приведено в таблице ниже (табл. 3.4).

Таблица 3.4

Сравнение определений алгоритма

Ученый	*Суть определения*	*Ключевые идеи*
А. Н. Колмогоров	Алгоритм как минимальная программа, описывающая объект	Алгоритмическая сложность, связь информации и алгоритмов
А. А. Марков	Алгоритм как система правил переписывания строк	Нормальные алгоритмы, универсальность вычислений

Общие черты

Оба подхода стремились дать строгое, формализованное определение алгоритма.

В обоих случаях алгоритм трактуется как *универсальное средство вычислений*, независимое от конкретной техники.

Подходы Колмогорова и Маркова эквивалентны другим моделям (λ-исчислению Черча, машинам Тьюринга) и подтверждают гипотезу Черча–Тьюринга о границах вычислимости.

Практическое значение

Подход Колмогорова лежит в основе методов оценки случайности, теории кодирования, анализа сложности данных.

Подход Маркова применяется в теории формальных языков, компиляторостроении и доказательстве вычислимости функций.

Совокупно эти определения позволяют не только строго описывать алгоритмы, но и определять пределы возможного для вычислительных систем.

Определения алгоритма по Колмогорову и Маркову отражают разные стороны одного и того же явления: Колмогоров акцентировал внимание на «измерении сложности алгоритмов», а Марков – на их «структурной реализации через правила». Оба подхода стали фундаментом современной теории алгоритмов и информационных технологий.

3.5. Алгоритм Евклида и великая теорема Ферма

Алгоритм Евклида

Алгоритм Евклида является одним из древнейших и важнейших алгоритмов в истории математики и информатики. Он был описан в трактате «Начала» греческого математика Евклида (III в. до н. э.) и предназначен для нахождения наибольшего общего делителя (НОД) двух чисел.

НОД (a, b) – это наибольшее целое число, которое делит оба числа a и b без остатка.

Идея алгоритма

Если число a делится на b с остатком r, то НОД (a, b) равен НОД (b, r). Процесс продолжается до тех пор, пока остаток не станет равным нулю.

Формальное описание:

Вход: два числа a и b ($a > b > 0$).

Пока $b \neq 0$:

```
r = a mod b
a = b
b = r
```

Выход: a (это НОД)

Пример работы:

Найдем НОД (252, 105):

252 ÷ 105 = 2, остаток 42 → НОД (252, 105) = НОД (105, 42)

105 ÷ 42 = 2, остаток 21 → НОД (105, 42) = НОД (42, 21)

42 ÷ 21 = 2, остаток 0 → НОД (42, 21) = 21.

Результат: НОД (252, 105) = 21.

Свойства алгоритма Евклида

1. Работает за конечное число шагов (свойство конечности).

2. Эффективен: количество шагов пропорционально длине чисел (логарифму от их величины).

3. Универсален: применим к любым целым числам.

4. Используется как базовый шаг во многих современных алгоритмах (RSA, криптография, теории чисел).

Великая теорема Ферма

Великая теорема Ферма утверждает:

Уравнение $x^n + y^n = z^n$ не имеет решений в целых положительных числах x, y, z при $n > 2$.

Эта гипотеза была впервые сформулирована французским математиком Пьером Ферма в XVII в. Он оставил запись на полях книги, что «нашел поистине чудесное доказательство, но поля книги слишком малы, чтобы его поместить».

На протяжении более чем 350 лет теорема оставалась недоказанной и стала одной из самых известных нерешенных задач математики. Только в 1994 г. английский математик Эндрю Уайлс представил полное доказательство, опирающееся на методы современной алгебраической геометрии и теорию эллиптических кривых.

Связь алгоритма Евклида и великой теоремы Ферма (табл. 3.5)

Таблица 3.5

Сравнение алгоритма Евклида и теоремы Ферма

Характеристика	*Алгоритм Евклида*	*Великая теорема Ферма*
Время появления	III в. до н. э.	XVII век
Автор	Евклид	Пьер Ферма
Суть	НОД(a, b)	Нет решений у $x^n + y^n = z^n$ при $n > 2$
Характер	Конструктивный алгоритм	Теорема (долгое время без доказательства)
Применение	Криптография, теории чисел	Теоретическая математика
Современное значение	Используется в RSA	Символ нерешенных задач

На первый взгляд эти два объекта относятся к разным областям: алгоритм Евклида – практический метод вычислений, а великая теорема Ферма – глубокое теоретическое утверждение. Однако они тесно связаны общей областью – теорией чисел.

1. *Алгоритм Евклида* используется при работе с диофантовыми уравнениями (уравнения с целыми решениями).

Например, в задаче нахождения целых решений уравнения вида:

$$ax + by = d,$$

где d = НОД (a, b).

2. *Диофантовы уравнения* лежат в основе многих математических задач, включая обобщенные уравнения, подобные уравнению Ферма.

3. *Великая теорема Ферма* – это частный случай более общей проблемы существования решений для диофантовых уравнений.

Таким образом, алгоритм Евклида можно рассматривать как «строительный кирпич» для более сложных исследований, в том числе тех, что связаны с уравнением Ферма.

Практическое значение

Алгоритм Евклида лежит в основе современной криптографии, алгоритмов шифрования (RSA), методов оптимизации и компьютерной алгебры.

Великая теорема Ферма не имеет прямого прикладного применения, но сыграла важную роль в развитии математики: ее доказательство потребовало создания новых направлений и методов, которые сегодня используются в информатике, криптографии и теории кодирования.

Алгоритм Евклида – пример практического и универсального алгоритма, дошедшего до нас из античности и используемого в современных

вычислениях. Великая теорема Ферма – пример «вечной» математической задачи, которая стимулировала развитие науки на протяжении веков. Их объединяет то, что оба объекта принадлежат к фундаментальной области – *теории чисел*, которая одновременно служит и источником алгоритмов, и полем для формулировки глубоких теоретических утверждений.

3.6. Алгоритмически неразрешимые задачи

Алгоритм – это точная и конечная последовательность действий, выполнение которой приводит к решению поставленной задачи. Однако исследования в области логики и математики XX в. показали, что существует особый класс задач, которые не могут быть решены никаким алгоритмом, т. е. не существует универсальной процедуры, гарантирующей получение ответа за конечное число шагов для всех возможных случаев.

Такие задачи получили название *алгоритмически неразрешимых*. Их существование оказалось фундаментальным открытием: математика и информатика столкнулись с принципиальными границами формализованных вычислений.

Исторический контекст

1. *Программа Гильберта (начало XX в.).* Немецкий математик Давид Гильберт поставил задачу: построить универсальную формализацию всей математики, при которой каждое утверждение можно было бы либо доказать, либо опровергнуть с помощью алгоритма.

2. *Теорема Геделя о неполноте (1931).* Курт Гедель показал, что в любой достаточно мощной формальной системе существуют утверждения, которые невозможно ни доказать, ни опровергнуть средствами этой системы. Это был первый удар по идее универсальной алгоритмизации математики.

3. *Работы Черча и Тьюринга (1936).* Алонзо Черч (λ-исчисление) и Алан Тьюринг (машины Тьюринга) независимо друг от друга доказали: существует целый класс задач, которые невозможно решить никаким алгоритмом. Так возникла современная теория вычислимости.

Классические примеры алгоритмически неразрешимых задач (табл. 3.6).

Таблица 3.6

Примеры алгоритмически неразрешимых задач

Задача	*Формулировка*	*Доказательство*
Проблема останова	Определить, завершится ли программа или нет	Тьюринг, 1936
Десятая проблема Гильберта	Проверить существование решений у диофантовых уравнений	Матиясевич, 1970
Проблема соответствия Поста	Проверить возможность составления одинаковых строк	Э. Пост, 1946
Проверка теорем в арифметике	Определить, выводимо ли произвольное утверждение	Гедель, 1931

1. *Проблема останова (Halting problem).*

Суть: можно ли составить программу, которая для любой другой программы и любого ее входа заранее определит, завершится ли она или будет выполняться бесконечно?

- Тьюринг доказал, что такой программы не существует.
- Это означает, что не существует универсального «предсказателя поведения» программ.

Пример

Если у нас есть программа, которая вычисляет факториал числа, мы можем проверить, что она всегда завершится. Но если программа сложная, с множеством ветвлений и циклов, универсально проверить ее завершение невозможно.

2. *Десятая проблема Гильберта.*

Постановка: существует ли алгоритм, который определяет, имеет ли произвольное диофантово уравнение (уравнение с целыми коэффициентами и целыми решениями) хотя бы одно решение?

Пример

- Уравнение $2x + 3y = 7$ имеет решение в целых числах.
- Уравнение $x^2 + y^2 = -1$ не имеет решений.

Для конкретных уравнений можно найти ответ, но общего метода для всех уравнений не существует. Ю. В. Матиясевич в 1970 г. окончательно доказал неразрешимость этой задачи.

3. *Проблема соответствия Поста (1946).* Формулировка: дано множество пар строк. Нужно выяснить, можно ли, выбирая строки в определенной последовательности, составить две одинаковые цепочки.

На первый взгляд задача кажется технической, но она важна в теории формальных языков. Доказано, что не существует алгоритма, решающего ее для всех случаев.

4. *Задача о выводимости в арифметике (1931).* Вопрос: существует ли универсальный алгоритм, который проверяет, является ли произвольное арифметическое выражение теоремой?

- Ответ: нет.
- Это напрямую связано с теоремой Геделя о неполноте.

Связь с теорией вычислимости

Алгоритмически неразрешимые задачи связаны с гипотезой Черча–Тьюринга: все, что можно вычислить алгоритмически, может быть выполнено на машине Тьюринга. Если задача неразрешима на машине Тьюринга, то она неразрешима вообще, независимо от того, какой язык программирования или вычислительная модель используется.

Таким образом, алгоритмическая неразрешимость отражает объективные границы вычислений.

Практическое значение

1. *Теория программирования.*

Проблема останова объясняет, почему невозможно создать универсальный анализатор, который всегда определяет корректность или завершение любой программы.

2. *Криптография.*

Современные методы защиты информации используют труднорешаемые задачи (например, факторизацию больших чисел). Они не всегда формально неразрешимы, но близки по сложности к этому классу.

3. *Искусственный интеллект.*

Алгоритмически неразрешимые задачи ставят пределы возможностям ИИ. Машина может учиться и обрабатывать данные, но не способна «в принципе» решить некоторые классы проблем.

4. *Философия и логика.*

Появление неразрешимых задач показало, что не всякая математическая истина может быть формализована и получена с помощью вычислений. Это изменило само понимание науки.

Классификация задач по разрешимости

Все задачи
- Разрешимые алгоритмически
  - Простые (полиномиальные: $O(n)$, $O(n \log n)$)
  - Трудные (NP-полные, экспоненциальные)
- Алгоритмически неразрешимые
- Проблема останова
- Десятая проблема Гильберта
- Проблема соответствия Поста
- Теорема Геделя (неполнота)

Алгоритмически неразрешимые задачи – это фундаментальное открытие XX в., показавшее границы формализованных методов решения.

- Они доказали, что не существует универсального алгоритма для всех проблем.
- Они задали границы вычислимости и легли в основу современной информатики, теории программирования, криптографии и искусственного интеллекта.
- Они имеют не только практическое, но и философское значение, определяя пределы формального знания.

3.7. Теорема Геделя о неполноте

Одной из важнейших задач математики в начале XX в. было стремление построить полную и непротиворечивую систему аксиом, которая охватывала бы все истины арифметики. Эту задачу особенно активно развивал немецкий математик Давид Гильберт, предложивший так называемую программу Гильберта. Согласно его замыслу:

1) любая математическая теория должна быть сведена к конечному набору аксиом;

2) каждое утверждение должно быть либо доказуемо, либо опровержимо;

3) система должна быть непротиворечивой, то есть из нее нельзя вывести одновременно утверждение и его отрицание.

Эти надежды казались достижимыми: логика Фреге, аксиомы Пеано для арифметики и система Цермело – Френкеля для теории множеств выглядели достаточно строгими. Однако в 1931 г. молодой австрийский математик Курт Гедель опубликовал работу *«О формально-неразрешимых положениях Principia Mathematica и родственных систем»*, которая доказала, что цель Гильберта в принципе недостижима.

Первая теорема Геделя о неполноте

Формулировка

В любой достаточно мощной и непротиворечивой формальной системе, способной выразить арифметику натуральных чисел, существует утверждение, которое не может быть ни доказано, ни опровергнуто в рамках этой системы.

Что значит «достаточно мощная система»?

Это система, содержащая:

- аксиомы натуральных чисел (арифметика Пеано);
- операции сложения и умножения;
- базовые правила логики (например, modus ponens).

Идея доказательства

Гедель предложил технику нумерации Геделя, при которой каждое математическое выражение можно закодировать числом. С помощью этого метода он построил высказывание, которое фактически утверждает: «Это утверждение недоказуемо в данной системе».

- Если бы оно оказалось доказуемым, возникло бы противоречие (система доказывает ложь).
- Если оно недоказуемо, то утверждение истинно, но остается недостижимым средствами системы.

Таким образом, система оказывается *неполной*: в ней всегда будут истины, недоступные доказательству.

Аналогия для понимания

Это напоминает парадокс лжеца: «Это утверждение ложно». Разница в том, что Гедель построил строгую формальную версию подобного парадокса внутри арифметики.

Вторая теорема Геделя о неполноте

Формулировка:

Если формальная система арифметики непротиворечива, то ее непротиворечивость не может быть доказана средствами самой этой системы.

Иными словами, система не может гарантировать свою непротиворечивость изнутри – для этого нужны более сильные средства или переход в метасистему.

Пример (на основе арифметики Пеано)

Арифметика Пеано – это система аксиом, описывающих натуральные числа. Она включает:

- аксиому существования числа 0;
- аксиому существования функции «следующее число» ($S(n)$);
- аксиомы равенства и индукции.

Система кажется полной и самодостаточной. Но теорема Геделя утверждает, что всегда найдутся утверждения о натуральных числах, которые не могут быть доказаны или опровергнуты в рамках этих аксиом (табл. 3.7).

Таблица 3.7

Две теоремы Геделя о неполноте

Теорема	*Суть*	*Последствие*
Первая теорема	Существуют утверждения, которые недоказуемы и неопровержимы	Система неполна
Вторая теорема	Система не может доказать свою непротиворечивость	Система не самодостаточна

Пример: утверждения о простых числах или сложных диофантовых уравнениях могут быть такими «недостижимыми истинами».

Значение и последствия

1. *Для математики*

Невозможно создать универсальную, полную и непротиворечивую систему для всей арифметики.

Всегда будут существовать утверждения, истинные, но недоказуемые средствами данной системы.

2. *Для логики и информатики*

Теорема Геделя тесно связана с понятием *алгоритмически неразрешимых задач* (см. раздел 3.6).

Она показывает, что не существует универсального алгоритма проверки истинности любых утверждений.

3. *Для философии*

Открытие Геделя стало ударом по логическому позитивизму и идее «абсолютного знания».

Истина не сводится к алгоритмическому доказательству.

4. *Для программирования и ИИ*

Невозможно создать программу, которая способна проверить истинность любого высказывания в арифметике.

Это ограничение распространяется и на искусственный интеллект: каким бы мощным он ни был, он не сможет обойти фундаментальные границы вычислимости.

Взаимосвязь с другими результатами

- Теоремы Геделя согласуются с результатами *Тьюринга* (проблема останова) и *Черча* (λ-исчисление), подтверждая гипотезу Черча – Тьюринга: существуют границы алгоритмизации.
- Они связаны с теоремой *Гильберта о несуществовании алгоритма для десятых проблем* и с алгоритмически неразрешимыми задачами.

Теоремы Геделя о неполноте показали:

1) невозможно построить формальную систему, которая была бы одновременно полной и непротиворечивой;

2) существует фундаментальный предел алгоритмизации: истина шире, чем доказуемость;

3) современная информатика, теория алгоритмов и искусственный интеллект развиваются с осознанием этих границ.

Таким образом, результаты Геделя определили границы математики и логики так же, как теория относительности ограничила физику, а эволюционная теория – биологию.

3.8. Проблема останова машины Тьюринга

Проблема останова – одна из фундаментальных задач теории алгоритмов и вычислимости. Она была сформулирована английским математиком Аланом Тьюрингом в 1936 г. в его знаменитой работе *«О вычислимых числах, с приложением к Entscheidungsproblem»*.

Эта проблема стала центральной в теории алгоритмов, так как показала: существует класс задач, которые невозможно решить никаким алгоритмом, независимо от вычислительной мощности компьютера или языка программирования.

Фактически проблема останова является вычислительным аналогом теоремы Геделя о неполноте: как в логике существуют утверждения, которые нельзя доказать или опровергнуть, так и в информатике существуют задачи, для которых нельзя составить универсальную программу-решатель.

Проблема останова

Существует ли алгоритм *H*, который для произвольной программы *P* и входных данных *X* определяет:

- *Да* – если программа *P*(*X*) остановится за конечное время;
- *Нет* – если программа *P*(*X*) будет выполняться бесконечно.

Тьюринг доказал, что универсального алгоритма *H* не существует.

Идея доказательства (парадоксальное построение)

Доказательство Тьюринга основано на методе от противного.

1. *Предположим, что алгоритм H существует.* Тогда мы можем написать программу:

$$H(P, X) \rightarrow \text{Да/Нет}$$

где *P* – программа;

X – входные данные.

2. *Создадим программу K(P)*, которая вызывает *H*(*P*, *P*) и работает по правилу:

- если *H* говорит «Да» (*P*(*P*) остановится), то *K* зацикливается;
- если *H* говорит «Нет» (*P*(*P*) не остановится), то *K* останавливается.

3. *Рассмотрим вызов K(K).*

Если *H*(*K*, *K*) = «Да», то *K* должна зациклиться, что противоречит ответу *H*.

Если *H*(*K*, *K*) = «Нет», то *K* должна остановиться, и снова возникает противоречие.

Следовательно, алгоритм *H* не может существовать.

Для простых программ человек легко определяет их поведение. Например:

```
def f(n):
while n > 0:
n = n – 1
```

- при *n* = 5 программа остановится через 5 шагов;
- при *n* = 0 программа завершится мгновенно.

Но если программа усложняется, содержит вложенные циклы, рекурсию, случайные ветвления или обращается к внешним данным (например, файлам или сети), невозможно заранее предсказать ее поведение универсальным способом.

Теоретическое значение

1. *Границы вычислимости.*

Проблема останова показала, что существуют задачи, которые неразрешимы в принципе.

2. *Связь с теоремами Геделя.*

Подобно тому как Гедель доказал неполноту арифметики, Тьюринг показал неполноту вычислений.

3. *Независимость от реализации.*

Проблема останова не зависит от языка программирования или типа компьютера. Если задача неразрешима для машины Тьюринга, она неразрешима и для любой другой вычислительной модели.

Практическое значение

Хотя проблема останова имеет теоретическую природу, ее последствия важны в прикладной информатике.

1. *Программирование.*

Нельзя написать программу, которая всегда определяет, завершится ли любая другая программа.

Поэтому компиляторы и отладчики могут находить ошибки только частично, используя приближения.

2. *Анализ кода.*

Современные инструменты статического анализа (например, линтеры) могут выявлять только ограниченные классы ошибок.

Универсальной проверки на бесконечные циклы или зависания не существует.

3. *Кибербезопасность.*

Невозможно создать универсальный антивирус, который заранее определяет поведение любой программы.

Антивирусы используют эвристики, базы сигнатур и машинное обучение, но они всегда ограничены.

4. *Искусственный интеллект.*

ИИ может прогнозировать поведение систем, но не способен обойти фундаментальные ограничения проблемы останова.

Это ставит предел возможностям автоматического программирования и верификации (табл. 3.8).

Таблица 3.8

Сравнение разрешимых и неразрешимых задач

Класс задач	*Примеры*	*Возможность решения*
Разрешимые	Сортировка массива, поиск максимума	Существуют алгоритмы
Трудные (NP-полные)	Задача коммивояжера, раскраска графа	Решаемы, но дорого по времени
Алгоритмически неразрешимые	Проблема останова, десятая проблема Гильберта	Алгоритмов не существует

Логика доказательства проблемы останова

Предположим, что существует $H(P, X)$
↓
Создаем программу $K(P)$, использующую H
↓
Рассматриваем $K(K)$
↓
Возникает противоречие
↓
H не существует

Проблема останова является фундаментальным результатом в теории алгоритмов. Она доказывает:

- не все задачи могут быть решены алгоритмами;
- существуют универсальные ограничения вычислительных систем;
- даже самый мощный компьютер или искусственный интеллект не смогут предсказать поведение любой программы.

Таким образом, проблема останова Тьюринга – это не только теоретический результат, но и философский ориентир, определяющий границы машинного разума и автоматизации знаний.

3.9. Прочие нерешаемые задачи (совершенные числа, задача Поста и др.)

До этого момента мы познакомились с фундаментальными результатами теории алгоритмов – теоремами Геделя, проблемой останова Тьюринга, десятой проблемой Гильберта. Все они показали, что границы алгоритмизации объективно существуют. Однако круг неразрешимых задач гораздо шире.

Многие нерешаемые или открытые задачи возникали в совершенно разных областях: теории чисел, алгебре, теории формальных языков, геометрии. Некоторые из них доказано неразрешимы, другие остаются гипотезами, которые могут быть либо решаемыми, либо неразрешимыми, но пока доказательств нет.

Именно совокупность этих задач позволяет студенту осознать, что «алгоритмизация» имеет строгие границы, а также увидеть, как идеи неразрешимости влияют на развитие прикладной математики, информатики и даже криптографии.

1. Совершенные числа

Совершенным называется положительное целое число, равное сумме всех своих собственных делителей (делителей, меньших самого числа).

Примеры

6 = 1 + 2 + 3.

28 = 1 + 2 + 4 + 7 + 14.

496 и 8128 также являются совершенными.

Совершенные числа известны еще со времен Евклида (III век до н. э.). Он доказал, что если число имеет вид

$$2^{p-1}(2^p - 1),$$

где $(2^p - 1)$ – простое число (так называемое число Мерсенна), то оно является совершенным.

Открытые проблемы

1. Существует ли бесконечно много совершенных чисел?

2. Существуют ли нечетные совершенные числа?

Несмотря на более чем 2000-летнюю историю, ответы неизвестны. До сих пор найдено лишь несколько десятков совершенных чисел, все они – четные.

Значение для алгоритмов

- Проверка числа на «совершенность» требует анализа его делителей, что неэффективно для больших чисел.
- Вопрос о нечетных совершенных числах может быть в принципе нерешаемым – то есть может не существовать алгоритма, который всегда даст ответ.

2. Проблема соответствия Поста

Формулировка (Эмиль Пост, 1946)

Дано конечное множество пар строк $(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)$. Нужно определить, существует ли последовательность индексов $i_1, i_2, \ldots, i_k$, такая что:

$$X_{i1}X_{i2}\ldots X_{ik} = Y_{i1}Y_{i2}\ldots Y_{ik}.$$

То есть, можно ли, «склеивая» строки из левой и правой частей пар, получить равенство.

Пример (разрешимый случай)

Пусть даны пары:

(a, ab);

(ba, a).

Выбираем последовательность (1, 2):
Левая часть: $a + ba = aba$;
Правая часть: $ab + a = aba$.
Равенство достигнуто, значит, решение существует.

Результат:
Пост доказал, что в общем случае задача алгоритмически неразрешима.
Практическое значение:
Проблема Поста связана с теорией формальных языков.

Она показывает, что не все задачи проверки эквивалентности слов и выражений можно автоматизировать.

3. Проблема слов в группах

В алгебре часто возникает задача: даны генераторы группы и соотношения между ними. Нужно определить, эквивалентны ли два слова, составленные из этих генераторов.

Пример

Пусть группа определяется соотношением $ab = ba$. Тогда слова aba и baa эквивалентны.

Результаты

Для некоторых классов групп (например, свободных абелевых) задача решаема.

В общем случае доказано, что задача неразрешима.

Значение:

Эта проблема имеет прямую связь с автоматической проверкой равенства выражений в системах символьной алгебры и компьютерной математике.

4. Проблема домино (тайлы Ванга)

Формулировка

Дано множество плиток домино с окрашенными краями. Нужно выяснить, можно ли замостить всю бесконечную плоскость этими плитками так, чтобы соседние плитки совпадали по цветам.

Результат:

Доказано, что задача неразрешима в общем случае.

Значение:

Эта задача неожиданно связана с компьютерными науками, так как она является моделью клеточных автоматов и динамических систем. Изучение проблемы Ванга послужило стимулом для развития теории автоматов и искусственной жизни.

5. Задачи логики предикатов

Логика первого порядка (кванторы $\forall$ и $\exists$, предикаты, отношения) кажется естественной системой для формализации математики. Однако было доказано, что:

- теория первого порядка в общем случае алгоритмически неразрешима;
- то есть не существует универсального метода, позволяющего для любого утверждения определить, истинно оно или ложно.

Это напрямую связано с теоремами Геделя о неполноте.

6. Другие примеры нерешаемых задач

1. *Задачи анализа.*

Некоторые вопросы о существовании решений дифференциальных уравнений не имеют алгоритмического метода.

2. *Задачи в топологии.*

Определение топологических инвариантов (например, гомеоморфности пространств) в ряде случаев неразрешимо.

3. *Задачи криптографии.*

Многие криптографические методы опираются на предположение, что некоторые задачи либо неразрешимы, либо крайне труднорешаемы (например, факторизация больших чисел) (табл. 3.9).

Таблица 3.9

Примеры нерешаемых и открытых задач

Задача	*Суть*	*Статус*
Совершенные числа	Существуют ли нечетные совершенные числа?	Неизвестно
Проблема соответствия Поста	Проверка равенства склеенных строк	Неразрешима
Проблема слов в группах	Проверка равенства слов	Неразрешима
Проблема домино	Замощение плоскости плитками	Неразрешима
Логика первого порядка	Проверка истинности всех утверждений	Неразрешима
Дифференциальные уравнения	Существование решений в общем виде	Неразрешимо
Нечетные совершенные числа	Вопрос о существовании	Открыто

Помимо фундаментальных задач Геделя и Тьюринга, существует множество других нерешаемых или открытых проблем. Они демонстрируют, что алгоритмические методы имеют границы не только в логике и программировании, но и в других разделах математики.

Эти задачи не бесполезны: напротив, их изучение стимулировало развитие новых направлений – теории формальных языков, алгебраической теории групп, клеточных автоматов, криптографии.

Таким образом, «прочие нерешаемые задачи» являются важнейшей частью общей картины вычислимости, подтверждающей: не все может быть сведено к алгоритму, и именно это открывает новые горизонты для науки.

3.10. Теория вычислимости и алгоритмическая сложность

Развитие теории алгоритмов в XX в. выявило два уровня анализа вычислений.

1. *Вычислимость* (от англ. *computability*): можно ли вообще решить задачу алгоритмическим методом, то есть существует ли алгоритм, который всегда приводит к правильному результату за конечное число шагов?

2. *Сложность* (от англ. *complexity*): если алгоритм существует, то какие ресурсы – время и память – требуются для его выполнения?

Эти два направления образовали самостоятельные дисциплины:

- теорию вычислимости, изучающую границы возможного;
- теорию алгоритмической сложности, исследующую эффективность решения задач.

Теория вычислимости

*Т*еория вычислимости изучает, какие задачи принципиально разрешимы с помощью алгоритмов и какие нет.

Основные идеи

1. *Гипотеза Черча – Тьюринга:* все, что можно вычислить алгоритмически, можно вычислить на машине Тьюринга или эквивалентной модели (λ-исчисление Черча, нормальные алгоритмы Маркова).

2. *Разрешимые задачи:* для них существует алгоритм. Примеры: сортировка массива, поиск кратчайшего пути в графе (алгоритм Дейкстры).

3. *Неразрешимые задачи:* для них не существует алгоритма. Примеры: проблема останова, проблема соответствия Поста.

Классификация задач по вычислимости

Все задачи
- Алгоритмически разрешимые
  - Простые (имеют эффективные алгоритмы)
  - Трудные (решаемы, но за очень долгое время)
- Алгоритмически неразрешимые
- Проблема останова
- Проблема Поста
- Десятая проблема Гильберта

Теория алгоритмической сложности

Даже если задача разрешима, возникает вопрос: насколько быстро можно ее решить?

*Т*еория алгоритмической сложности изучает ресурсы (время и память), необходимые алгоритмам для решения задач.

Основные понятия

1. *Временная сложность* – число шагов, которое требуется алгоритму для работы.

Обозначается асимптотикой $O(n)$, $O(n \log n)$, $O(n^2)$ и т. д.

2. *Пространственная сложность* – объем памяти, необходимый алгоритму.

3. *Существует несколько классов сложности* (табл. 3.10).

P – задачи, решаемые за полиномиальное время. Примеры: сортировка, поиск кратчайшего пути.

NP – задачи, решение которых можно проверить за полиномиальное время. Примеры: задача коммивояжера, раскраска графа.

NP-полные задачи – наиболее трудные в классе NP, к ним сводятся все остальные задачи этого класса.

EXP – задачи, решаемые за экспоненциальное время.

Таблица 3.10

Примеры классов сложности

Класс	*Характеристика*	*Примеры задач*
P	Решаемы быстро (полиномиально)	Сортировка, поиск пути в графе
NP	Проверка решения быстрая	Судоку, задача о рюкзаке
NP-полные	Самые трудные задачи в NP	Коммивояжер, SAT
EXP	Требуют экспоненциального времени	Полный перебор шахматных партий

Связь вычислимости и сложности

- Если задача неразрешима, то даже обсуждать ее сложность бессмысленно.
- Если задача разрешима, то встает вопрос: можно ли найти эффективный алгоритм?

Пример

Сортировка чисел разрешима и имеет алгоритмы $O(n \log n)$.

Задача коммивояжера разрешима, но известные алгоритмы работают за экспоненциальное время.

Проблема останова вообще неразрешима.

Практическое значение

1. *Программирование и разработка ПО.*

Теория вычислимости определяет, какие задачи вообще можно автоматизировать.

Теория сложности помогает выбрать оптимальный алгоритм.

2. *Криптография.*

Основана на предположении, что некоторые задачи (например, факторизация больших чисел) крайне трудны.

Сложность таких задач гарантирует безопасность криптосистем.

3. *Искусственный интеллект.*

Ограничен задачами вычислимости и сложности.

Невозможно создать ИИ, который решает алгоритмически неразрешимые задачи.

4. *Философский аспект.*

Теория вычислимости и сложности показывает, что границы возможностей компьютеров определяются не только их мощностью, но и природой самих задач.

Теория вычислимости определяет, какие задачи принципиально решаемы.

Теория сложности показывает, насколько эффективно можно решать разрешимые задачи.

И вместе они формируют фундамент современной информатики, объясняя, почему одни задачи легко автоматизировать, другие требуют огромных ресурсов, а третьи неразрешимы в принципе.

3.11. Случайные процессы и алгоритмы

Классическая теория алгоритмов основывается на *детерминизме*: выполнение алгоритма всегда приводит к одному и тому же результату для одинаковых входных данных. Однако в реальных вычислениях все чаще используются методы, основанные на случайности.

Случайные процессы – это такие процессы, в которых результат одного шага зависит не только от правил, но и от случайного выбора. Алгоритмы, использующие случайность, называют *вероятностными*, или *рандомизированными*.

Они применяются в ситуациях, когда:

- классические методы слишком медленны;
- задача имеет огромную вычислительную сложность;
- необходимо смоделировать поведение реальных случайных явлений.

Случайные процессы

Случайный процесс – это последовательность событий, каждое из которых зависит от вероятностного выбора.

Примеры

- подбрасывание монеты (дискретный случайный процесс);
- шум в канале связи;
- движение молекул в физике (броуновское движение).

Случайные процессы можно описывать с помощью:

- *цепей Маркова* – моделей, где вероятность следующего состояния зависит только от текущего;
- *случайных блужданий* – моделей, имитирующих хаотическое движение (например, частиц).

Рандомизированные алгоритмы

Рандомизированный алгоритм – это алгоритм, в ходе работы которого используются случайные числа для принятия решений.

Особенности

Один и тот же вход может приводить к разным траекториям вычислений.

Результат носит вероятностный характер: алгоритм работает правильно с заданной вероятностью.

Пример: проверка числа на простоту.

Классический метод требует деления на все числа до $\sqrt{n}$.

Вероятностный тест Миллера–Рабина быстро проверяет простоту, но может дать ошибку с очень малой вероятностью.

Классификация вероятностных алгоритмов

1. *Las Vegas – алгоритмы.*

Всегда выдают правильный результат.

Случайность влияет только на время работы.

Пример: случайные варианты алгоритма быстрой сортировки (quicksort).

2. *Monte Carlo – алгоритмы.*

Выдают результат быстро, но с небольшой вероятностью ошибки.

Пример: алгоритмы проверки простоты чисел.

3. *Алгоритмы для случайных процессов.*

Используются для моделирования реальных явлений (моделирование очередей, погоды, трафика в сети) (табл. 3.11).

Таблица 3.11

Сравнение видов алгоритмов

Тип алгоритма	*Результат*	*Роль случайности*	*Пример*
Детерминированный	Всегда один и тот же	Нет	Сортировка вставками
Las Vegas	Всегда правильный	Время выполнения	Quicksort с рандомизацией
Monte Carlo	Вероятностный, возможна ошибка	На корректность ответа	Тест Миллера – Рабина

Применение случайных алгоритмов

1. *Криптография.*

Генерация ключей требует случайных чисел.

Алгоритмы RSA используют вероятностные тесты простоты.

2. *Оптимизация.*

Генетические алгоритмы и муравьиные алгоритмы используют случайность для поиска решения.

Метод имитации отжига применяется в задачах планирования.

3. *Компьютерное моделирование.*

Случайные процессы применяются для имитации трафика, биологических систем, движения частиц.

4. *Обработка данных.*

Случайная выборка позволяет эффективно анализировать большие объемы информации (big data).

Связь со сложностью

Рандомизированные алгоритмы часто позволяют решать задачи быстрее, чем детерминированные. Иногда они являются единственными практическими методами.

Пример

Проверка числа на простоту: детерминированные алгоритмы слишком медленны, но вероятностные позволяют работать с числами в сотни и тысячи знаков.

Случайные процессы и алгоритмы представляют собой важное направление современной информатики. Они позволяют:

- эффективно решать задачи, которые слишком сложны для классических методов;
- моделировать реальные случайные явления;
- обеспечивать безопасность в криптографии.

Сочетание строгости алгоритмов и гибкости случайности делает такие методы незаменимыми в науке, инженерии и программировании.

Контрольные вопросы

1. Дайте определение понятия «алгоритм». Какие виды алгоритмов и формы их представления существуют?

2. Перечислите основные характеристики и свойства алгоритмов. Почему свойства детерминированности и конечности имеют фундаментальное значение?

3. Как связаны между собой понятия информации и алгоритмов ее обработки? Приведите примеры из практики.

4. Как формулировали определение алгоритма А. Н. Колмогоров и А. А. Марков? В чем заключалась новизна их подхода?

5. Опишите алгоритм Евклида. Почему он считается примером «вечного» алгоритма? Как великая теорема Ферма связана с современными алгоритмами?

6. Что такое алгоритмически неразрешимые задачи? Приведите примеры таких задач и поясните их значение для теории информатики.

7. Сформулируйте первую и вторую теоремы Геделя о неполноте. В чем их сходство с проблемой останова машины Тьюринга?

8. В чем заключается суть проблемы останова? Почему доказано, что универсального алгоритма ее решения не существует?

9. Какие еще примеры нерешаемых задач известны (совершенные числа, задача Поста, проблема слов в группах и др.)? Каково их значение?

10. Что изучает теория вычислимости и теория алгоритмической сложности? Какова роль случайных процессов и рандомизированных алгоритмов в современных вычислениях?

4. МАТЕМАТИЧЕСКИЕ ФОРМАЛИЗМЫ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ

Современная теория алгоритмов и программирования опирается на разнообразные математические модели вычислений. Эти модели позволяют не только формализовать сам процесс вычислений, но и сравнивать различные подходы к построению программ и алгоритмов, а также оценивать эффективность вычислительных систем.

4.1. Основные парадигмы программирования

Под парадигмой программирования понимается совокупность принципов и идей, определяющих стиль построения программ, методы описания алгоритмов и подходы к организации вычислительного процесса. Выбор парадигмы существенно влияет на структуру программ, способы их анализа и область применения.

В истории развития информатики сложились несколько базовых парадигм программирования.

1. Императивная (процедурная) парадигма.

Данный подход рассматривает программу как последовательность команд, изменяющих состояние вычислительной машины. В центре внимания находится понятие переменной и инструкции, выполняющей преобразование ее значения. Императивные языки (например, Pascal, C) близки по своей логике к архитектуре фон Неймана, что делает их естественными для описания шагов вычислений.

2. Функциональная парадигма.

Здесь вычисления представляются как вычисление значений функций, а сама программа трактуется как композиция функций. При этом отсутствует явное использование состояний и переменных в традиционном смысле. Основой служит λ-исчисление, предложенное А. Черчем. Примеры языков: Lisp, Haskell. Данный подход упрощает рассуждения о корректности программ и позволяет эффективно использовать параллелизм.

3. Логическая парадигма.

В этой парадигме вычисления рассматриваются как процесс логического вывода, а программа определяется в терминах фактов и правил. Исполнение программы сводится к поиску доказательства утверждения. В качестве основы используется предикатная логика первого порядка. Наиболее известный язык – Prolog. Логическое программирование особенно эффективно для задач искусственного интеллекта, систем автоматического доказательства теорем и экспертных систем.

4. Объектно-ориентированная парадигма.

Программа представляется как совокупность объектов, которые объединяют данные (состояния) и методы их обработки. Основные принципы – инкапсуляция, наследование и полиморфизм. Данный подход позволяет моделировать сложные системы в терминах взаимодействующих сущностей. Примеры языков: Java, C++, Python. Объектно-ориентированное программирование удобно для создания больших и сложных программных комплексов, ориентированных на долгосрочную эксплуатацию и расширяемость.

Таким образом, каждая из перечисленных парадигм отражает определенный взгляд на процесс вычислений и соотносится с конкретными математическими формализмами. Императивный подход опирается на модель машины Тьюринга, функциональный – на λ-исчисление, логический – на предикатное исчисление, а объектно-ориентированный – на более прикладные абстракции, связывающие вычисления с моделированием реальных систем. Каждая парадигма обладает своими преимуществами и ограничениями, и в современном программировании часто используются их комбинации.

4.2. Сравнительный анализ моделей

В рамках математической теории алгоритмов для формализации понятия вычислений были предложены различные модели, получившие в дальнейшем фундаментальное значение для информатики. Каждая из них строилась на основе различных интуитивных представлений о вычислительном процессе, однако все они привели к одному и тому же понятию вычислимой функции.

Классическим и наиболее известным формализмом является машина Тьюринга, предложенная А. Тьюрингом в 1936 г. Эта абстрактная конструкция описывает вычисления как последовательность операций над символами, записанными на бесконечной ленте, с использованием конечного набора состояний. Машина Тьюринга позволила не только строго определить алгоритм, но и сформулировать знаменитую задачу останова, сыгравшую ключевую роль в осознании пределов алгоритмических методов.

Параллельно с этим развивались и другие подходы. А. Черч разработал λ-исчисление, представляющее вычисления как применение функций к аргументам. В этой системе отсутствуют переменные в привычном понимании, а вычислительный процесс сводится к пошаговому упрощению выражений с использованием правил редукции. λ-исчисление оказало значительное влияние на становление функционального программирования и остается важным инструментом в области формальных методов.

Еще одним направлением стала теория рекурсивных функций, основанная на идее построения сложных функций из элементарных с помощью операций суперпозиции, примитивной рекурсии и минимизации. Эта модель развивалась в рамках математической логики и дала возможность связать понятие алгоритма с традиционными математическими объектами – функциями на множестве натуральных чисел.

Существенный вклад внес и А. А. Марков, предложивший систему нормальных алгоритмов, которые формализуют вычисления как последовательные подстановки строк по заданным правилам. В отличие от машины Тьюринга или λ-исчисления, нормальные алгоритмы особенно наглядны при описании преобразований символьных последовательностей, что сближает их с задачами лингвистики и теории формальных языков.

Наконец, широкое распространение получила теория автоматов, в основе которой лежит идея о вычислительном процессе как преобразовании входных последовательностей в выходные при помощи конечного числа состояний. Хотя автоматы менее мощны, чем машина Тьюринга (они не могут реализовать все вычислимые функции), их простота делает их незаменимыми при анализе конечных процессов, например в проектировании цифровых схем или распознавании формальных языков.

Сравнительный анализ перечисленных формализмов показывает, что такие модели, как машины Тьюринга, λ-исчисление, рекурсивные функции и нормальные алгоритмы Маркова, эквивалентны с точки зрения вычислимости: любая функция, вычислимая в одной из этих систем, вычислима и в других. Эта эквивалентность привела к формулировке тезиса Черча–Тьюринга, утверждающего, что именно данный класс функций и является совокупностью всех алгоритмически вычислимых функций.

Тем не менее эквивалентность моделей не устраняет различий между ними. Они проявляются прежде всего в удобстве описания алгоритмов и в области применения. Машина Тьюринга обеспечивает простоту и универсальность, но громоздка при описании реальных процедур. λ-исчисление естественным образом выражает идеи функционального программирования и обладает высокой степенью абстракции. Рекурсивные функции удобны для формальной связи вычислимости с классической математикой. Нормальные алгоритмы Маркова особенно наглядны при работе с текстовыми данными. Автоматы же, хотя и менее мощны, занимают свое место как эффективный инструмент анализа конечных процессов.

Таким образом, сравнительный анализ различных моделей вычислений выявляет как их фундаментальное единство, выражающееся в эквивалентности по вычислимой мощности, так и различия в удобстве применения, наглядности и степени абстракции (табл. 4.1). Это делает каждую модель

важной в своей области и оправдывает существование целого спектра подходов к формализации вычислительных процессов.

Таблица 4.1

Сравнительный анализ моделей вычислений

Модель	*Основа*	*Сильные стороны*	*Ограничения*
Машина Тьюринга	Абстрактная лента и конечное число состояний	Универсальность, простота формализации, фундамент для теории алгоритмов	Трудна для практического описания реальных процедур
λ-исчисление	Функции и редукция выражений	Высокая абстрактность, естественная основа функционального программирования	Сложность понимания, непривычность без переменных
Рекурсивные функции	Построение функций с помощью рекурсии и суперпозиции	Связь с классической математикой, строгая формализация вычислимости	Ограниченная наглядность при описании процедур
Нормальные алгоритмы Маркова	Последовательные подстановки строк	Простота и наглядность при работе с символьными данными и языками	Менее удобны для анализа общих алгоритмов
Автоматы	Состояния и переходы по входным символам	Эффективность для анализа конечных процессов, применение в схемотехнике и лингвистике	Ограниченная вычислительная мощность (не универсальны)

4.3. Методы оценки вычислительных моделей

При изучении вычислительных моделей важно учитывать не только их *вычислительную мощность*, то есть множество функций, которые могут быть реализованы в рамках той или иной модели, но и *эффективность* использования этих моделей в практике. Одной лишь теоретической эквивалентности с точки зрения вычислимости недостаточно для ответа на вопрос, насколько удобно и рационально применять конкретный формализм. Поэтому в теории алгоритмов и программирования выделяют несколько основных методов оценки вычислительных моделей.

Первым и наиболее распространенным является *асимптотический анализ*. Его целью является исследование порядка роста времени работы алгоритма и объема используемой памяти в зависимости от размера входных данных. Если две модели эквивалентны по вычислимой мощности, но в одной из них алгоритмы для решения определенного класса задач оказываются существенно более эффективными, это становится важным фактором при выборе модели. Асимптотический анализ позволяет абстрагироваться

от конкретной реализации и рассматривать эффективность вычислений в предельных, асимптотических масштабах.

Другим критерием является *сравнение по выразительности*. Под выразительностью понимается удобство и естественность описания тех или иных классов задач. Например, автоматы чрезвычайно удобны при работе с регулярными языками и конечными процессами, тогда как λ-исчисление более адекватно выражает функциональные преобразования. Таким образом, даже при равной вычислительной мощности модели могут существенно различаться в своей выразительности и наглядности.

Наконец, важным показателем является *практическая применимость* модели. Она определяется возможностью использования формализма при проектировании языков программирования, построении компиляторов и создании вычислительных систем. Так, машина Тьюринга играет фундаментальную роль в теории, но неудобна как инструмент описания прикладных алгоритмов. В то же время объектно-ориентированные модели, хотя и менее абстрактны, демонстрируют большую практическую значимость.

Таким образом, выбор вычислительной модели обусловлен не только теоретическими, но и практическими соображениями. С одной стороны, необходимо учитывать ее вычислительную мощность и общую эквивалентность с иными формализмами, с другой – удобство применения, эффективность и адекватность выражения конкретных задач (табл. 4.2). Именно совокупность этих факторов определяет ценность модели как в теоретическом, так и в прикладном плане.

Таблица 4.2

Методы оценки вычислительных моделей

Критерий	*Суть*	*Значение*
Асимптотический анализ	Изучение порядка роста времени и памяти в зависимости от размера входа	Позволяет сравнивать эффективность алгоритмов в предельных масштабах
Сравнение по выразительности	Удобство и естественность описания различных классов задач	Определяет, для каких типов задач модель наиболее адекватна и наглядна
Практическая применимость	Возможность применения модели при создании языков и вычислительных систем	Показывает ценность модели в прикладном программировании и инженерных решениях

Контрольные вопросы

1. Что такое парадигма программирования и какие парадигмы являются основными?

2. Чем различаются императивная и функциональная парадигмы программирования?

3. Каковы особенности логического программирования и где оно применяется?

4. В чем заключается объектно-ориентированный подход к построению программ?

5. Какие математические модели вычислений считаются базовыми?

6. Что общего у машины Тьюринга, λ-исчисления и рекурсивных функций?

7. Почему говорят, что разные вычислительные модели эквивалентны по вычислимости?

8. Какие различия существуют между моделями с точки зрения удобства описания алгоритмов?

9. В чем заключается асимптотический анализ алгоритмов?

10. Какие критерии используются для оценки вычислительных моделей?

11. Чем отличается понятие вычислимости от понятия эффективности вычислений?

12. Почему практическая применимость модели важна при выборе парадигмы программирования?

13. Как теория вычислительных моделей связана с разработкой языков программирования?

14. Какие преимущества и ограничения имеет функциональная парадигма?

15. Какая из моделей наиболее близка к архитектуре современных компьютеров и почему?

5. СОВРЕМЕННЫЕ АЛГОРИТМЫ И ТЕОРИЯ СВЯЗИ

Современная информатика и теория алгоритмов переживают этап бурного развития, связанный с глобальным распространением компьютерных сетей, усложнением задач обработки информации и повышением требований к безопасности данных. Если в классических разделах теории алгоритмов внимание уделялось главным образом вопросам вычислимости и общим принципам построения алгоритмов, то сегодня акцент все больше смещается к разработке эффективных и практически применимых методов.

Ключевыми направлениями становятся алгоритмы для распределенных вычислений и сетевого взаимодействия, средства оптимизации программного обеспечения, а также методы криптографии, основанные на свойствах простых чисел. Все большее значение приобретают вероятностные и эвристические подходы, позволяющие решать задачи, для которых строгие методы оказываются слишком трудоемкими. Важным стимулом для развития алгоритмов служат новые технологические вызовы: квантовые вычисления, блокчейн, искусственный интеллект, большие данные.

Таким образом, современная теория алгоритмов тесно связана с прикладными задачами науки и техники. Она не только исследует фундаментальные свойства вычислений, но и формирует основу для практических решений, определяющих развитие информационного общества. В настоящем разделе будут рассмотрены некоторые важнейшие современные алгоритмы и направления их применения.

5.1. Компьютерные сети и распределенные вычисления

Бурное развитие вычислительной техники во второй половине XX в. привело к необходимости объединения компьютеров в сети, которые обеспечивали бы совместный доступ к данным и вычислительным ресурсам. Появление компьютерных сетей не только изменило практику использования вычислительных систем, но и открыло новую область исследований, связанную с организацией передачи информации, распределением вычислений и координацией работы множества взаимодействующих узлов. Именно в этот период начали формироваться основы теории компьютерных сетей и распределенных систем, которая сегодня является неотъемлемой частью информатики.

Одной из центральных задач сетевых вычислений стала *эффективная передача данных*. Необходимо было разрабатывать методы построения маршрутов от отправителя к получателю с минимальными затратами времени и ресурсов. Для этого были предложены алгоритмы маршрутизации, обеспечивающие нахождение кратчайших путей, балансировку нагрузки и устойчивость к сбоям отдельных узлов или каналов связи. Эти алгоритмы легли в основу современных протоколов передачи данных.

Другой фундаментальной проблемой стала *синхронизация вычислений* между удаленными узлами. В распределенных системах множество процессов выполняется параллельно, и для корректной работы необходимо согласовывать порядок их взаимодействия. В условиях отсутствия глобальных часов разработаны специальные алгоритмы синхронизации и механизмы логического времени (например, метки Лэмпорта), позволяющие упорядочивать события и обеспечивать согласованность действий.

Особое место занимают *алгоритмы консенсуса*, позволяющие всем участникам распределенной системы достичь единого согласованного состояния, даже если часть узлов отказывает или ведет себя некорректно. Алгоритмы Paxos и Raft стали классическими решениями этой задачи и получили широкое применение в современных облачных вычислениях, распределенных базах данных и системах хранения.

Функционирование глобальных сетей невозможно без разработки стандартных *протоколов передачи данных*. Ключевое место занимает стек протоколов TCP/IP, обеспечивающий надежную доставку сообщений в сети Интернет. На прикладном уровне функционирование Всемирной паутины обеспечивается протоколами семейства HTTP, которые продолжают развиваться для поддержки новых требований к безопасности и скорости передачи данных. В некоторых областях активно применяются децентрализованные P2P-сети, где взаимодействие осуществляется без выделенного центра управления, что повышает устойчивость и масштабируемость систем.

Таким образом, компьютерные сети и распределенные вычисления опираются на целый комплекс алгоритмов, методов и протоколов. Их совместное использование обеспечивает надежность, отказоустойчивость и эффективность взаимодействия между узлами, что делает возможной работу глобальных коммуникационных систем, облачных сервисов и современных приложений, связанных с обработкой больших данных в реальном времени (табл. 5.1). Теоретические разработки в этой области напрямую находят применение в практике и продолжают определять развитие информационного общества.

Таблица 5.1

Ключевые задачи и алгоритмы распределенных систем

Задача	*Содержание*	*Примеры алгоритмов/протоколов*	*Назначение*
Маршрутизация	Поиск оптимальных путей передачи данных между узлами сети	Dijkstra, Bellman-Ford, OSPF, BGP	Обеспечение минимальной задержки, балансировки нагрузки и отказоустойчивости
Синхронизация	Согласование времени и порядка выполнения процессов в распределенной системе	Логические часы Лэмпорта, алгоритм Булли, алгоритм кольцевого выбора	Корректное взаимодействие процессов без глобальных часов
Консенсус	Достижение общего согласованного состояния между всеми узлами сети	Paxos, Raft, Byzantine Fault Tolerance (PBFT)	Обеспечение целостности данных и устойчивости к сбоям или злонамеренным узлам
Протоколы передачи	Стандартизированные правила обмена данными между узлами	TCP/IP, UDP, HTTP/HTTPS, FTP, P2P	Надежная доставка сообщений, взаимодействие приложений и пользователей

5.2. Оптимизация алгоритмов и программного обеспечения

Корректность алгоритма означает, что он в точности решает поставленную задачу и гарантированно выдает правильный результат для любого допустимого входа. Однако в реальной практике этого недостаточно: алгоритм, обладающий слишком высокой временной или пространственной сложностью, может быть неприменим даже для относительно небольших объемов данных. Именно поэтому проблема оптимизации алгоритмов и программного обеспечения занимает особое место в теории и практике вычислений.

Оптимизация алгоритмов осуществляется на нескольких уровнях.

Теоретический уровень

На теоретическом уровне оптимизация связана с выбором алгоритмов, обладающих *асимптотически более высокой эффективностью*. Сравнение проводится с использованием понятий теории сложности, таких как классы $O(n)$, $O(n \log n)$, $O(n^2)$ и т. д. Например, сортировка пузырьком имеет временную сложность порядка $O(n^2)$, тогда как сортировка слиянием или

быстрая сортировка работают за $O(n*\log n)$. При больших объемах данных именно асимптотические различия становятся определяющими: если один алгоритм способен обработать миллион элементов за секунды, а другой требует часов, то выбор очевиден.

В теории алгоритмов активно изучаются и другие аспекты оптимизации: нижние границы сложности задач, доказательства невозможности существования более быстрых алгоритмов для определенных классов, а также построение приближенных алгоритмов для NP-трудных задач. В этом контексте оптимизация включает не только улучшение отдельных решений, но и поиск фундаментальных пределов вычислительных возможностей.

Инженерный уровень

Даже алгоритмы, обладающие одинаковой асимптотической сложностью, могут существенно различаться по реальной скорости работы в зависимости от конкретной реализации. Здесь на первый план выходят *инженерные методы оптимизации*, учитывающие особенности архитектуры вычислительных систем.

Ключевыми направлениями являются:

Параллельные вычисления. Использование многопроцессорных систем, многоядерных процессоров и графических ускорителей позволяет распределять выполнение задач между несколькими потоками. Многие алгоритмы – в частности алгоритмы линейной алгебры, обработки изображений и анализа больших данных – получают значительный выигрыш при правильной параллельной реализации.

Оптимизация работы с памятью. В современных системах доступ к оперативной памяти часто оказывается узким местом. Алгоритмы, эффективно использующие кэш-память и локальность данных, демонстрируют существенно более высокую производительность. В этом направлении особое значение имеют алгоритмы блочного типа, которые перерабатывают данные не поэлементно, а блоками, соответствующими размеру кэша.

Кэширование и мемоизация. В ряде задач одни и те же вычисления повторяются многократно. Сохранение промежуточных результатов позволяет избежать избыточных затрат времени. Этот прием широко используется в динамическом программировании и при оптимизации рекурсивных процедур.

Выбор оптимальных структур данных. Одной и той же задачи можно достичь разными способами, однако эффективность будет зависеть от выбранных структур данных. Так, хеш-таблицы обеспечивают быстрый доступ за время $O(1)$ в среднем случае, деревья поиска позволяют упорядочивать данные и поддерживать эффективный поиск, а графовые структуры обеспечивают удобное представление сложных сетевых объектов.

Практический уровень

Оптимизация программного обеспечения тесно связана с инженерной практикой. Здесь учитываются такие аспекты, как энергопотребление, надежность и масштабируемость систем. В условиях роста значимости облачных вычислений и обработки больших данных встала задача создания алгоритмов, устойчивых к сбоям и эффективно работающих в распределенной среде. Для этого используются методы балансировки нагрузки, дублирования вычислений и распределенного кэширования.

Необходимо также подчеркнуть, что оптимизация в программной инженерии часто является компромиссом между скоростью, надежностью и читаемостью кода. Слишком агрессивная оптимизация может сделать программу трудно поддерживаемой, что в долгосрочной перспективе снижает ее ценность. Поэтому принципиальное значение имеет поиск баланса между теоретической эффективностью и практической реализуемостью (табл. 5.2).

Таблица 5.2

Методы оптимизации алгоритмов
и программного обеспечения

Метод оптимизации	*Суть*	*Пример использования*
Выбор асимптотически более эффективного алгоритма	Замена алгоритма с высокой сложностью на более быстрый по порядку роста	Сортировка: пузырьковая $O(n^2) \to$ быстрая/слиянием $O(n \log n)$)
Параллелизм	Распределение вычислений между несколькими процессорами или потоками	Параллельная обработка данных в многопроцессорных системах
Кэширование	Хранение промежуточных результатов для ускорения повторных обращений	Запоминание вычисленных значений в динамическом программировании
Оптимальные структуры данных	Использование структур, лучше всего соответствующих задаче	Хеш-таблицы для быстрого поиска, деревья поиска для сортировки

Таким образом, оптимизация алгоритмов и программного обеспечения представляет собой многогранный процесс, включающий как фундаментальные аспекты теории сложности, так и инженерные методы, учитывающие архитектуру современных вычислительных систем. Она выступает связующим звеном между абстрактной математической моделью алгоритма и его практической реализацией в программном обеспечении. Без оптимизации математически корректные процедуры остаются лишь теоретическими схемами, тогда как грамотное использование методов оптимизации

позволяет превратить их в эффективные инструменты, обеспечивающие высокую производительность и масштабируемость современных информационных технологий.

5.3. Алгоритмы и теория простых чисел (RSA, Ферма, Евклид)

Теория чисел, являющаяся одной из древнейших областей математики, в XXI в. приобрела особую актуальность благодаря ее применению в информатике и криптографии. Свойства простых чисел, долгое время изучавшиеся как чисто абстрактные объекты, сегодня лежат в основе защиты данных, электронных транзакций и коммуникационных протоколов. Таким образом, математические идеи, сформулированные еще в Античности, оказываются краеугольными камнями цифровой безопасности.

Алгоритм Евклида

Одним из старейших и фундаментальных алгоритмов является *алгоритм Евклида*, известный со времен Древней Греции. Его идея заключается в последовательном применении деления с остатком: если $a = bq + r$, то наибольший общий делитель чисел a и b совпадает с наибольшим общим делителем чисел b и r. Повторяя данный процесс, мы неизбежно приходим к случаю, когда остаток становится равным нулю, а последнее ненулевое значение и есть НОД.

Простота и эффективность алгоритма Евклида сделали его классическим примером вечного алгоритма: он используется на протяжении более двух тысяч лет и продолжает играть ключевую роль в современной криптографии. В частности, при построении открытых и закрытых ключей необходимо выполнять вычисления в арифметике по модулю, и нахождение обратных элементов напрямую связано с алгоритмом Евклида и его расширенной формой.

Малая теорема Ферма

Другим краеугольным результатом теории чисел является *малая теорема Ферма*. Она утверждает, что если p – простое число, а a – целое число, не делящееся на p, то выполняется сравнение:

$$a^{p-1} \equiv 1 \ (mod\ n).$$

Это утверждение, сформулированное еще в XVII в., имеет фундаментальное значение для построения тестов на простоту. Хотя малая теорема Ферма не дает универсального критерия (существуют составные числа, удовлетворяющие ее условию, – так называемые числа Кармайкла), она стала

основой для целого ряда вероятностных тестов, включая тест Миллера – Рабина. Эти тесты позволяют эффективно проверять простоту очень больших чисел, что особенно важно при генерации ключей в криптосистемах.

Криптосистема RSA

Наиболее ярким примером применения простых чисел в информатике является криптосистема RSA, предложенная в 1977 г. Р. Ривестом, А. Шамиром и Л. Адлеманом. Ее безопасность основана на односторонности задачи факторизации: вычислить произведение двух больших простых чисел легко, а разложить это произведение обратно на множители крайне трудно.

Суть метода заключается в следующем. Выбираются два больших простых числа p и q, после чего вычисляется их произведение $n = pq$. На основе функции Эйлера $\varphi(n) = (p - 1)(q - 1)$ подбираются параметры открытого и закрытого ключей. Шифрование сообщения выполняется возведением его в степень e по модулю n, а расшифрование – возведением в степень d, где d является мультипликативным обратным к е по модулю $\varphi(n)$. Таким образом, знание d позволяет расшифровать сообщение, но вычислить d без знания разложения числа n практически невозможно.

RSA стал фундаментом современной криптографии с открытым ключом и широко применяется в протоколах безопасной связи (SSL/TLS), цифровых подписях и системах аутентификации (табл. 5.3).

Таблица 5.3

Алгоритмы и теория простых чисел

Алгоритм / Теорема	*Суть*	*Значение*	*Современное применение*
Алгоритм Евклида	Вычисление НОД через последовательное деление с остатком	Один из старейших и эффективнейших алгоритмов; работает за логарифмическое время	Вычисление обратных элементов по модулю; криптография
Малая теорема Ферма	Если p – простое, то $a^{p-1} \equiv 1 \ (mod\ n)$ для любого a, не делящегося на p	Базовый результат теории чисел; основа тестов на простоту	Генерация простых чисел; тесты на простоту (Миллер – Рабин)
RSA	Шифрование на основе произведения двух больших простых чисел	Первая практическая криптосистема с открытым ключом; стойкость опирается на трудность факторизации	Защита данных, цифровые подписи, протоколы SSL/TLS, аутентификация

Алгоритм Евклида, малая теорема Ферма и криптосистема RSA представляют три уровня применения теории чисел: от античных алгоритмов до современной криптографической инфраструктуры. Они наглядно демонстрируют преемственность математических идей и их практическую значимость в условиях цифровой эпохи. Таким образом, теория простых чисел,

долгое время считавшаяся чисто теоретической областью, стала одним из ключевых факторов, обеспечивающих безопасность глобальных коммуникаций и информационных технологий.

5.4. Вероятностный тест Миллера – Рабина

Проблема проверки чисел на простоту является одной из центральных в теории чисел и имеет исключительное значение для криптографии. Многие криптосистемы, такие как RSA, требуют генерации больших простых чисел, обычно длиной в сотни или тысячи бит. Полный перебор или детерминированные методы проверки простоты оказываются слишком затратными для практического применения, поэтому особую роль играют вероятностные алгоритмы, обеспечивающие высокую эффективность при минимальной вероятности ошибки.

Если для какого-то целого (не кратного) условие $a^{p-1} \equiv 1\ (mod\ n)$ не выполняется, то число n с необходимостью является составным. В противном случае оно объявляется «вероятно простым».

Главная особенность теста Миллера – Рабина заключается в его вероятностном характере. Для каждого случайного выбора основания a вероятность того, что составное число будет ошибочно признано простым, не превышает 1/4. При многократном повторении теста вероятность ошибки экспоненциально уменьшается и быстро становится пренебрежимо малой. Так, при 20 независимых проверках ошибка становится практически невозможной для любых практических применений.

Эта комбинация высокой эффективности и надежности сделала тест Миллера – Рабина (табл. 5.4) одним из стандартных инструментов криптографии. Он используется при генерации ключей в системах с открытым ключом, при построении протоколов обмена и аутентификации, а также в программных библиотеках для работы с большими числами. В отличие от детерминированных тестов простоты, применимых лишь для относительно малых чисел, тест Миллера – Рабина сохраняет работоспособность и при длине входных данных в тысячи бит, что делает его незаменимым в современной вычислительной практике.

Таблица 5.4

Вероятностный тест Миллера – Рабина

Этап алгоритма	*Математическая идея*	*Значение в криптографии*
Разложение $n - 1 = 2^s \cdot d$	Представление числа $n - 1$ в виде степени двойки, умноженной на нечетное d	Подготовка числа к проверке свойств степеней по модулю
Выбор случайного свидетеля a	Случайный выбор основания a из диапазона $2 \le a \le n - 2$ для проверки сравнений по модулю n	Исключение «специальных» чисел, повышает надежность теста

Окончание табл. 5.4

Этап алгоритма	*Математическая идея*	*Значение в криптографии*
Вычисление $a^d \pmod n$	Использование критерия сильной псевдопростоты: для простого n либо $a^d \equiv 1 \pmod n$, либо найдется r, при котором $a^{2^r d} \equiv -1 \ (mod\ n)$	Начальная проверка простоты
Последовательное возведение в квадрат	Проверка равенства $a^{2^r d} \equiv -1 \ (mod\ n)$ для некоторого r, $0 \le r < s$	Дополнительная фильтрация составных чисел
Вероятностный вывод	Если ни одно из условий не выполнено – число составное; если выполнено – число «вероятно простое»	Дает надежный результат при многократных повторениях теста с разными свидетелями

Таким образом, тест Миллера–Рабина представляет собой пример того, как вероятностные методы позволяют находить баланс между теоретической строгостью и практической применимостью. Он наглядно демонстрирует, что в вычислительной математике абсолютная гарантия может быть заменена чрезвычайно высокой степенью вероятности, если это обеспечивает значительный выигрыш в производительности.

5.5. Алгоритм Рабина – Карпа

Одной из центральных задач обработки текстовой информации является поиск подстроки в строке. Эта задача возникает в самых различных областях – от традиционной работы с текстовыми массивами до анализа генетических последовательностей и построения поисковых систем. Наивный метод решения основан на последовательном сравнении шаблона с каждым возможным фрагментом текста. Однако при больших объемах данных он становится крайне неэффективным: в худшем случае его временная сложность достигает порядка $O(nm)$, где n – длина текста, а m – длина подстроки.

Существенно более эффективный подход был предложен М. Рабином и Р. Карпом в 1980 г. Их алгоритм опирается на идею хеширования, которая позволяет заменить прямое посимвольное сравнение более компактными числовыми операциями. Основное достоинство метода заключается в том, что сравнение строк сводится к сравнению их хеш-значений, вычисляемых по определенной функции.

Пусть требуется определить, содержится ли строка P длиной m в тексте T длиной n. Алгоритм Рабина – Карпа начинается с вычисления хеш-функции для шаблона P, а затем последовательно вычисляет хеши для всех

подстрок текста длины m. Если хеши различны, подстроки гарантированно не совпадают и дополнительная проверка не требуется. Если же хеши совпадают, выполняется прямое сравнение символов для устранения возможных коллизий, то есть случаев, когда разные строки имеют одинаковый хеш.

Для повышения эффективности используется метод скользящего хеша. Его идея заключается в том, что при переходе от подстроки T[i.. i+m−1] к следующей T[i+1.. i+m] новое значение хеша вычисляется из предыдущего за постоянное время, без полного пересчета. Это достигается за счет того, что из старого хеша вычитается вклад удаленного символа и добавляется вклад нового.

С точки зрения теории сложности алгоритм Рабина – Карпа в среднем работает за время порядка $O(n + m)$, что близко к линейному. Однако в худшем случае, при неблагоприятном распределении коллизий, его эффективность может снижаться до $O(nm)$. На практике такая ситуация встречается крайне редко при использовании корректно подобранных хеш-функций, и алгоритм демонстрирует высокую производительность.

Алгоритм Рабина – Карпа (табл. 5.5) нашел широкое применение в разнообразных областях. Он используется при построении систем текстового поиска и проверки плагиата, в биоинформатике для анализа длинных последовательностей, а также как часть составных методов индексирования и сопоставления данных. Особенно важным оказывается его свойство эффективно обрабатывать сразу несколько шаблонов, что значительно расширяет область его применения.

Таблица 5.5

Алгоритм Рабина – Карпа

Этап алгоритма	*Математическая идея*	*Значение*
Вычисление хеша шаблона и первой подстроки	Преобразование строки в числовое значение по модулю	Ускоряет сравнение: замена посимвольных операций на операции с числами
Сравнение хешей	Если хеши различны, строки точно различны	Позволяет быстро отсеивать неподходящие фрагменты
Проверка при совпадении хешей	Дополнительное посимвольное сравнение для исключения ошибок из-за коллизий	Гарантия корректности результата
Скользящий хеш	Пересчет нового хеша из предыдущего за константное время	Обеспечивает линейную среднюю сложность алгоритма
Обработка нескольких шаблонов	Сравнение сразу нескольких хешей	Эффективность при множественном поиске (например, в анализе текстовых данных)

Таким образом, алгоритм Рабина – Карпа является классическим примером удачного применения методов хеширования в теории алгоритмов. Он демонстрирует, каким образом использование компактных математических преобразований позволяет существенно ускорить решение традиционных задач. Сочетая простоту идеи с высокой практической эффективностью, данный алгоритм прочно вошел в арсенал методов обработки строковых данных и продолжает оставаться актуальным в современных вычислительных системах.

5.6. Генетические и муравьиные алгоритмы

В последние десятилетия особое внимание в теории алгоритмов и вычислительной практике привлекли методы, вдохновленные процессами, протекающими в природе. Эти подходы основаны не на строгих математических формулах, а на моделировании поведения биологических систем, что позволяет находить приближенные решения задач, трудноразрешимых традиционными методами. Ключевыми представителями данного направления являются *генетические* и *муравьиные алгоритмы*, относящиеся к классу эволюционных и коллективных эвристик.

Генетические алгоритмы

Генетические алгоритмы основаны на принципах естественного отбора и дарвиновской эволюции. Основная идея заключается в том, что множество возможных решений задачи рассматривается как популяция, которая эволюционирует во времени. Каждое решение кодируется в виде «хромосомы» – строки символов или чисел. Качество решения оценивается с помощью функции приспособленности (fitness function), аналогичной биологической способности к выживанию.

Процесс оптимизации имитирует биологические механизмы:

- *отбор* – более приспособленные решения имеют больше шансов «передать» свои свойства;
- *кроссовер* – комбинация частей различных решений для получения новых вариантов;
- *мутация* – случайное изменение отдельных элементов хромосомы для поддержания разнообразия.

Итеративное повторение этих операций позволяет популяции постепенно приближаться к оптимальному или субоптимальному решению. Генетические алгоритмы хорошо зарекомендовали себя в задачах глобальной оптимизации, где классические методы либо слишком затратны, либо застревают в локальных минимумах.

Муравьиные алгоритмы

Муравьиные алгоритмы восходят к наблюдениям за поведением реальных колоний муравьев, которые при поиске пищи находят кратчайшие маршруты. В основе этого процесса лежит механизм *феромонной коммуникации*: каждый муравей оставляет на пути химический след, интенсивность которого указывает другим особям вероятность выбора данного маршрута. Со временем наиболее короткие пути усиливаются, поскольку по ним проходит больше муравьев, а следы на неэффективных маршрутах постепенно исчезают.

В вычислительной форме данный принцип реализуется следующим образом: множество искусственных «агентов-муравьев» исследует граф задачи (например, сеть дорог или маршруты в логистике). При каждом перемещении они обновляют феромонные метки, которые отражают качество найденного пути. Таким образом, система самоорганизуется и постепенно выявляет эффективные решения. Муравьиные алгоритмы особенно эффективны для задач комбинаторной оптимизации, таких как задача коммивояжера, маршрутизация в сетях и распределение ресурсов.

Эвристический характер и практическая значимость

И генетические, и муравьиные алгоритмы относятся к *эвристическим методам*. Это означает, что они не гарантируют нахождения точного оптимального решения, однако способны эффективно находить хорошие приближенные решения в разумные сроки. Их применение особенно оправдано в тех случаях, когда строгие математические методы либо слишком сложны, либо требуют неприемлемых вычислительных затрат.

Благодаря своей универсальности и относительной простоте реализации эти алгоритмы нашли применение в инженерии, биоинформатике, экономике, логистике и анализе данных. Они иллюстрируют принципиально новый подход к построению вычислительных методов – опору на имитацию природных механизмов и коллективного поведения, что делает возможным решение задач, ранее считавшихся слишком сложными для классических алгоритмов.

Таким образом, генетические и муравьиные алгоритмы демонстрируют, каким образом идеи, почерпнутые из биологии, находят успешное воплощение в информатике. Эти методы расширяют границы возможностей алгоритмических систем, позволяя решать задачи глобальной оптимизации и комбинаторного поиска, где традиционные подходы оказываются недостаточными. Их развитие представляет собой важный шаг в сторону интеграции естественно-научных принципов и вычислительных технологий.

5.7. Новые технологии и новые проблемы

Современное развитие вычислительной техники и информационных технологий обусловило появление принципиально новых типов алгоритмов, а вместе с ними и новых вызовов для теории и практики. Каждое технологическое достижение приводит к пересмотру границ вычислимого и эффективного, формируя новые исследовательские задачи и стимулируя развитие алгоритмических методов.

Одним из наиболее значимых направлений являются *квантовые вычисления*, основанные на принципах квантовой механики. Квантовые алгоритмы, такие как алгоритм Шора для факторизации больших чисел и алгоритм Гровера для ускоренного поиска в неструктурированных массивах данных, демонстрируют возможности, принципиально недостижимые для классических вычислительных моделей. Вместе с тем именно эти алгоритмы создают серьезную угрозу для традиционных криптографических систем. Так, применение алгоритма Шора делает уязвимыми все методы шифрования, основанные на сложности факторизации, включая RSA. Этот вызов стал мощным стимулом для формирования направления *постквантовой криптографии*, целью которого является разработка алгоритмов защиты, устойчивых к атакам с использованием квантовых компьютеров.

Не менее важным направлением стало развитие *искусственного интеллекта* и методов машинного обучения. Нейронные сети, вероятностные модели и эволюционные алгоритмы позволяют решать широкий круг задач – от анализа изображений до прогнозирования сложных систем. Однако вместе с новыми возможностями возникают и новые проблемы: интерпретируемость и прозрачность работы таких алгоритмов, их устойчивость к искажениям данных, а также этические аспекты использования искусственного интеллекта в обществе.

Третьим фактором современного этапа является распространение *распределенных реестров и технологии блокчейн*. Эти системы опираются на алгоритмы консенсуса, криптографические методы и принципы децентрализации. Они открывают новые возможности для построения прозрачных и устойчивых систем учета, финансовых транзакций и взаимодействия без посредников. В то же время они ставят перед исследователями задачи повышения масштабируемости, снижения энергопотребления и обеспечения безопасности в условиях открытых децентрализованных сред.

Развитие квантовых вычислений, искусственного интеллекта и распределенных систем показывает, что теория алгоритмов не является статичной дисциплиной. Она развивается в тесной связи с технологическим прогрессом, постоянно адаптируясь к новым условиям. Каждое новое направление

не только расширяет спектр известных алгоритмических методов, но и формирует новые фундаментальные вопросы, решение которых определяет будущее информатики и вычислительных технологий.

Контрольные вопросы

1. Какие задачи решаются с помощью алгоритмов компьютерных сетей и распределенных вычислений?
2. В чем заключается цель оптимизации алгоритмов и программного обеспечения?
3. Как алгоритм Евклида используется в криптографических системах?
4. Как малая теорема Ферма применяется для проверки чисел на простоту?
5. На каком принципе основана криптосистема RSA и чем объясняется ее устойчивость?
6. В чем состоит суть вероятностного теста Миллера – Рабина и как достигается малая вероятность ошибки?
7. Для чего применяется алгоритм Рабина – Карпа и на чем основана его эффективность?
8. Каковы основные идеи генетических алгоритмов и какие задачи они позволяют решать?
9. В чем заключается принцип работы муравьиных алгоритмов?
10. Какие новые угрозы для алгоритмов возникают в связи с развитием квантовых вычислений?
11. Что общего у эвристических методов (генетических и муравьиных алгоритмов) и чем они отличаются от классических методов оптимизации?
12. Почему алгоритмы, основанные на простых числах, занимают центральное место в современной криптографии?
13. Как хеширование используется в алгоритме Рабина – Карпа?
14. В чем состоит различие между детерминированными и вероятностными алгоритмами проверки простоты?
15. Почему развитие технологий напрямую влияет на развитие теории алгоритмов?

6. МНОЖЕСТВА

Теория множеств занимает фундаментальное место в современной математике и информатике. Она предоставляет строгий язык для описания объектов и их взаимосвязей, формирует основу логики и алгебраических структур, служит базой для построения теории алгоритмов, вероятностей и комбинаторики. Именно через понятие множества удалось унифицировать многие разделы математики, сведя их к исследованиям отношений между элементами и совокупностями объектов. Понятие множества позволяет формализовать интуитивные представления об объединениях элементов, их принадлежности или непринадлежности, а также о взаимодействии различных совокупностей. Операции над множествами образуют основу для дальнейшего изучения логических систем, анализа вычислительных процессов и построения структур данных. В информатике теория множеств играет двоякую роль. С одной стороны, она используется как формальный аппарат при доказательствах корректности алгоритмов и формулировке задач. С другой стороны, множества находят непосредственное применение в программировании, например, при описании коллекций данных, работе с базами данных, анализе отношений между элементами или построении моделей реальных процессов. Таким образом, теория множеств выступает универсальным инструментом, обеспечивающим связующее звено между абстрактной математикой и прикладной информатикой. Ее изучение является необходимым этапом для понимания более сложных структур и методов, которые будут рассмотрены в последующих разделах.

6.1. Определение множеств

Основные понятия множеств

Множеством M называется объединение в единое целое определенных различимых однотипных объектов *a*, которые называются *элементами* множества:

$$a \in M.$$

Множество можно описать, указав какое-то свойство, присущее всем элементам этого множества.

Замечание. Вообще говоря, понятие множества считается первичным (исходным) понятием и как таковое не определяется. Приведенное выше определение следует, скорее, считать уточнением понятия множества.

Множество, все элементы которого являются числами, называется *числовым*. В дальнейшем мы будем, прежде всего, рассматривать именно такие

множества. Множество, элементами которого являются другие множества, называется *классом* или *семейством*.

Множество, содержащее конечное число элементов, называется *конечным*. При подсчете количества элементов учитываются только различные (неповторяющиеся) элементы.

Множество, не содержащее элементов, называется *пустым* и обозначается символом $\varnothing$.

Множество может быть задано перечислением (списком) своих элементов, порождающей процедурой или описанием характеристических свойств (предикатом), которым должны обладать его элементы. Причем в последнем случае необходимо формулировать описание характеристических свойств элементов множества достаточно корректно, для того, чтобы множество было определено вполне однозначно. Добавим, что многие числовые множества могут быть заданы всеми тремя указанными способами (например, множество четных однозначных чисел).

Пример 1.

Некоторые примеры множеств, заданных различными способами:

а) $M_1 = \{1; 2; 3; 4\}$;

б) $M_2 = \{x \; x \in Z, -4 < x < 9\}$;

в) $M_3 = \{x \; x = 2n + 1, n \in N\}$.

Мощностью конечного множества M называется количество его элементов.

Обозначается M. Если $A = B$, то множества A и B называются равномощными.

Если все элементы множества A являются также элементами множества B, то говорят, что множество A *включается (содержится)* в множестве B:

$$A \subset B.$$

Если $A \subseteq B$, то множество A называется *подмножеством* множества B (также говорят, что B покрывает A). Если при этом $A \neq B$, то множество A называется *собственным подмножеством* множества B и обозначается $A \subset B$.

Замечание. Не следует считать равносильными отношения принадлежности ($\in$) и вхождения одного множества в другое ($\subset$).

Можно привести следующий пример. Пусть A – множество всех студентов данной группы, а B – множество всех учебных групп данного института. Здесь $A \in B$, но $A \not\subset B$, поскольку элементы этих множеств разнородны. Этот пример показывает также, что элементами множеств могут являться другие множества.

Парадокс Рассела. Задание множеств характеристическим предикатом может привести к противоречиям. Рассмотрим множество всех множеств, не содержащих себя в качестве элемента: $Y = \{X: X \notin X\}$. Если такое множество существует, то можно ответить на следующий вопрос: принадлежит ли оно само себе. С одной стороны, если $Y \in Y$, то $Y \notin Y$. С другой стороны, если $Y \notin Y$, то $Y \in Y$! Это противоречие можно разрешить различными способами, в целом сводящимся к тому, что Y не является множеством.

Для трех множеств A, B, C справедливы следующие соотношения:

$$A \subseteq A;\ A \not\subset A;$$
$$A \subseteq B \wedge B \subseteq C \to A \subseteq C;$$
$$A \subseteq B \wedge B \subset C \to A \subset C.$$

Связь между включением и равенством множеств устанавливается следующим соотношением:

$$A = B \leftrightarrow A \subseteq B \wedge B \subseteq A.$$

Здесь знак $\wedge$ обозначает *конъюнкцию* (логическое «и»).

В заключение добавим, что Г. Кантор предложил использовать понятие «универсального множества» (*универсум*), как бы противоположного понятию пустого множества $\varnothing$. По мысли Кантора, универсальное множество U содержит все мыслимые множества, и при этом оно само содержится во множестве своих подмножеств в качестве элемента. В дальнейшем смысл и содержание понятия универсального множества будут раскрыты более подробно.

Операции над множествами и их свойства

Объединением множеств A и B называется множество C, элементы которого принадлежат хотя бы одному из множеств A и B.

Обозначается $C = A \cup B$.

Геометрическое изображение множеств в виде области на плоскости называется диаграммой Эйлера – Вэйна.

Пересечением множеств A и B называется множество C, элементы которого принадлежат каждому из множеств A и B.

Обозначение $C = A \cap B$ Для множеств A, B и C справедливы следующие свойства:

$$A \cap A = A \cup A = A;$$
$$A \cup B = B \cup A;$$
$$A \cap B = B \cap A;$$
$$(A \cap B) \cap C = A \cap (B \cap C);$$
$$(A \cup B) \cup C = A \cup (B \cup C);$$

$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$;
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$;
$A \cup (A \cap B) = A$;
$A \cap (A \cup B) = A$;
$A \cup \varnothing = A$;
$A \cap \varnothing = \varnothing$.

Разностью множеств A и B называется множество, состоящее из элементов множества A, не принадлежащих множеству B.

Обозначается: $C = A \setminus B$.

Симметрической разностью множеств A и B называется множество C, элементы которого принадлежат в точности одному из множеств A или B.

Обозначается: $A \Delta B$.

$$A \Delta B = (A \setminus B) \cup (B \setminus A).$$

CE называется *дополнением* множества A относительно множества E, если $A \subseteq E$ и $CE = E \setminus A$.

Для множеств A, B и C справедливы следующие соотношения:

$A \setminus B \subseteq A$;
$A \setminus A = \varnothing$;
$A \setminus (A \setminus B) = A \cap B$;
$A \Delta B = B \Delta A$;
$A \Delta B = (A \cup B) \setminus (A \cap B)$;
$A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$;
$A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$;
$(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$;
$(A \cap B) \setminus C = (A \setminus C) \cap (B \setminus C)$;
$A \setminus (B \setminus C) = (A \setminus B) \cup (A \cap C)$;
$(A \setminus B) \setminus C = A \setminus (B \cup C)$;
$(A \Delta B) \Delta C = A \Delta (B \Delta C)$;
$A \cap (B \Delta C) = (A \cap B) \Delta (A \cap C)$;
$A \cup CEA = E$;
$A \cap CEA = \varnothing$;
$CEE = \varnothing$;
$CE\varnothing = E$;
$CECEA = A$;
$CE(A \cup B) = CEA \cap CEB$;
$CE(A \cap B) = CEA \cup CEB$.

Пример 2.

Исходя из определения равенства множеств и операций над множествами, доказать тождество и проверить его с помощью диаграммы Эйлера – Вэйна.

$A \setminus B = A \setminus (A \cap B)$.

Из записанных выше соотношений видно, что

$A \setminus (A \cap B) = (A \setminus A) \cup (A \setminus B) = \varnothing \cup (A \setminus B) = \mathrm{A} \setminus \mathrm{B}$

Что и требовалось доказать.

Для иллюстрации полученного результата построим диаграммы Эйлера – Вэйна (рис. 6.1).

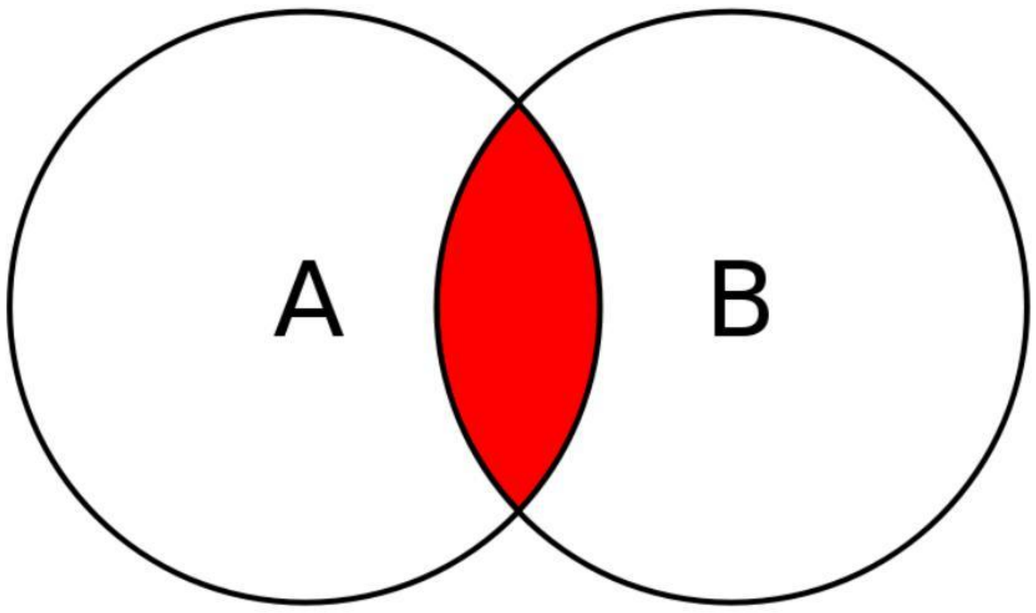

Рис. 6.1. Диаграмма Эйлера – Вэйна

Пример 3.

Исходя из определения равенства множеств и операций над множествами, доказать тождество.

$A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$.

Если некоторый элемент $x \in A \setminus (B \cup C)$, то это означает, что этот элемент принадлежит множеству А, но не принадлежит множествам B и C.

Множество $A \setminus B$ представляет собой множество элементов множества A, не принадлежащих множеству B.

Множество $A \setminus C$ представляет собой множество элементов множества A, не принадлежащих множеству C.

Множество $(A \setminus B) \cap (A \setminus C)$ представляет собой множество элементов, которые принадлежат множеству A, но не принадлежат ни множеству B, ни множеству C.

Таким образом, тождество можно считать доказанным.

Векторы и прямые произведения

Вектором (кортежем) в линейной алгебре и дискретной математике называют упорядоченный набор элементов. Это не есть определение вектора, поскольку целесообразнее это понятие считать основным.

Элементы, определяющие вектор, называются координатами или компонентами. Координаты нумеруются слева направо, а их число называется длиной или размерностью вектора. В отличие от элементов множества, координаты вектора могут совпадать.

Координаты вектора заключаются в круглые скобки, например $(a_1, a_2, ..., a_n)$. Иногда скобки или запятые опускаются. Часто векторы длины 2 называются упорядоченными парами, длины 3 – тройками и т. д.

Два вектора равны, если они имеют равную длину и их соответствующие координаты равны. Иначе говоря, векторы $(a_1, ..., a_n)$ и $(b_1, ..., b_m)$ равны, если $n = m$ и $a_1 = b_1, ..., a_n = b_m$.

Прямым произведением множеств A и B (обозначение $A \times B$) называется множество всех упорядоченных пар $(a; b)$, таких, что $a \in A$, $b \in B$. В частности, если $A = B$, то обе координаты принадлежат множеству A, такое произведение обозначается A^2. Аналогично, прямым произведением множеств $A_1, A_2, ..., A_n$ называется множество всех векторов $(a_1, a_2, ..., a_n)$ длины n, таких, что $a_1 \in A_1, a_2 \in A_2, ..., a_n \in A_n$.

Пример 4. Множество $R \times R = R^2$ – это множество всех упорядоченных пар действительных чисел, геометрической интерпретацией которого служит декартова координатная плоскость.

Координатное представление точек плоскости было впервые предложено Р. Декартом и исторически является первым примером прямого произведения. Поэтому часто прямое произведение множеств называют декартовым произведением.

Пример 5. Даны множества $A = \{a; b; c; d; e; f; g; h\}$ и $B = \{1; 2; 3; 4; 5; 6; 7; 8\}$. Тогда $A \times B$ есть множество обозначений клеток шахматной доски.

Вообще конечное множество, элементами которого являются какие-либо символы (буквы, цифры, знаки препинания, знаки операций и т. д.) называется алфавитом. Любые элементы множества A^n в этом случае являются *словами* длины n в алфавите A. Например, десятичное целое число – это слово в алфавите цифр.

Проекцией вектора $a = (a_1, a_2, ..., a_n)$ на некоторую ось i называется его компонента (координата) с соответствующим порядковым номером (обозначается $\text{пр}_i\ a$). Например, проекция точки плоскости на 1-ю ось есть ее абсцисса (первая координата).

Теорема 1.1. Мощность произведения конечных множеств $A_1, A_2, ..., A_n$ равна произведению мощностей этих множеств: $|A_1 \times A_2 \times ... \times A_n| = |A_1| \cdot |A_2| \cdot ... \cdot |A_n|$.

Следствие. $|A^n| = |A|^n$.

Эта простая теорема и ее следствие впоследствии широко используются в комбинаторике.

6.2. Теоретико-множественные операции

Одним из центральных направлений теории множеств является изучение операций, позволяющих из одних множеств строить другие. Эти операции являются аналогом арифметических действий над числами, но применяются к совокупностям объектов. Они образуют замкнутую систему, в рамках которой можно описывать самые разнообразные математические и прикладные задачи.

Объединение множеств

Операция объединения соответствует интуитивному «сложению» множеств. Она позволяет построить новое множество, включающее все элементы исходных множеств. При этом если один и тот же элемент встречается в обоих множествах, он включается в объединение только один раз. Таким образом, объединение отражает принцип расширения охвата: оно показывает, какие объекты принадлежат хотя бы одному из рассматриваемых множеств.

С логической точки зрения объединение соответствует операции дизъюнкции («ИЛИ»). Если рассматривать принадлежность элемента как логическое высказывание, то факт вхождения в объединение выражает истинность хотя бы одного из условий: принадлежность первому множеству или второму.

Пересечение множеств

Пересечение реализует противоположный принцип: оно выделяет те элементы, которые одновременно принадлежат обоим множествам. Эта операция выражает идею «общего» для совокупностей и используется для анализа совпадений, пересечений интересов или общих свойств объектов.

С точки зрения логики пересечение соответствует конъюнкции («И»): элемент входит в пересечение тогда и только тогда, когда он принадлежит и первому, и второму множеству.

Разность множеств

Разность множеств описывает ситуацию исключения: она формирует множество тех элементов, которые принадлежат одному множеству, но отсутствуют в другом. Эта операция особенно важна в задачах фильтрации

и уточнения информации. В логических терминах она соответствует связке «И НЕ».

Симметрическая разность

Симметрическая разность является комбинацией предыдущих операций и описывает множество элементов, которые принадлежат только одному из множеств, но не обоим сразу. Она выражает идею различий между множествами и часто используется при сравнении данных.

Дополнение множества

Если в задаче фиксировано универсальное множество, включающее все допустимые объекты, то для каждого подмножества можно определить его дополнение. Дополнение состоит из всех элементов универсального множества, которые не принадлежат данному подмножеству.

Эта операция особенно важна в логике, так как соответствует отрицанию. Если принадлежность элемента множеству трактовать как истинное высказывание, то принадлежность дополнению означает ложность этого высказывания.

Свойства операций

Теоретико-множественные операции обладают рядом фундаментальных свойств, которые делают их схожими с арифметическими действиями. Среди них: коммутативность, ассоциативность, дистрибутивность. Кроме того, для объединения и пересечения определены нейтральные элементы: пустое множество и универсальное множество. Эти свойства образуют алгебру множеств, также известную как булева алгебра.

Практическое значение

В информатике операции над множествами имеют непосредственные приложения. Они используются в построении поисковых запросов (например, объединение и пересечение условий), в реляционных базах данных, при анализе графов, в системах управления доступом и при обработке больших массивов информации.

Таким образом, операции над множествами формируют не только фундамент математической логики, но и практический инструмент анализа данных и построения алгоритмов. Их изучение является необходимым шагом для понимания последующих разделов, связанных с комбинаторикой, вероятностями и структурой алгоритмов.

6.3. Комбинаторика и множества: примеры, упражнения

Комбинаторика – это раздел математики, тесно связанный с теорией множеств, предметом которого является изучение способов выбора, упорядочивания и комбинирования элементов конечных множеств в соответствии с заданными условиями. Основная задача комбинаторики заключается в подсчете числа возможных комбинаций или структур, которые могут быть построены на основе заданного множества.

Значимость комбинаторных методов чрезвычайно велика. Они применяются в информатике при анализе сложности алгоритмов, в криптографии при оценке стойкости шифров и генерации ключей, в теории вероятностей при вычислении вероятностей сложных событий, а также в многочисленных практических областях – от статистики и биоинформатики до организации вычислительных процессов.

В основе комбинаторики лежат фундаментальные правила, позволяющие переходить от простых задач к более сложным конструкциям.

Правило суммы утверждает, что если некоторое действие может быть выполнено либо одним способом, либо другим, то общее число способов равно сумме числа способов каждого варианта.

Правило произведения применяется, когда действие состоит из последовательности этапов: если первый этап можно выполнить n-способами, а второй – m-способами, то общее число вариантов равно произведению $n \cdot \times m$.

На базе этих правил формируются более развитые понятия: *перестановки, размещения и сочетания*.

Перестановкой называют любое упорядоченное расположение всех элементов множества.

Размещением – упорядоченный выбор части элементов из множества, причем порядок выбора играет роль.

Сочетанием – выбор подмножества без учета порядка элементов, когда важен состав, но не последовательность.

Для наглядности рассмотрим пример. Пусть дано множество $\{a, b, c\}$. Тогда все перестановки его элементов составят шесть различных упорядоченных строк: abc, acb, bac, bca, cab, cba. Если требуется выбрать два элемента без учета порядка, то возможны три сочетания: $\{a, b\}$, $\{a, c\}$, $\{b, c\}$. Этот простой пример демонстрирует различие между упорядоченным и неупорядоченным выбором.

Примеры задач:

1. *Составление пароля.* Сколько существует способов составить пароль из трех символов, если каждый символ может быть выбран из десяти цифр (0–9)?

По правилу произведения число способов равно $10 \times 10 \times 10 = 1000$.

2. *Рассадка студентов.* Сколькими способами можно рассадить четырех студентов за четыре различных места в аудитории?

Эта задача сводится к числу перестановок из четырех элементов: $4! = 24$.

3. *Выбор участников.* В группе из пяти человек необходимо выбрать двух для участия в олимпиаде. Сколькими способами это можно сделать?

Порядок выбора не имеет значения, поэтому используется число сочетаний.

Контрольные вопросы

1. Дайте определение множества и объясните, чем оно отличается от обычного списка или последовательности.

2. Как обозначаются множества и их элементы в математике?

3. Что означает запись $a \in Aa\backslash in\ Aa \in A$ и $a \notin Aa\backslash \text{notin}\ Aa \in /A$?

4. Что называют пустым множеством и как оно обозначается?

5. Перечислите основные операции над множествами и объясните их смысл.

6. В чем заключается различие между объединением и пересечением множеств?

7. Что понимается под разностью множеств и симметрической разностью?

8. Дайте определение дополнения множества относительно универсального множества.

9. Сформулируйте и поясните правило суммы в комбинаторике.

10. Сформулируйте и поясните правило произведения в комбинаторике.

11. Дайте определение перестановки. Приведите пример для множества из трех элементов.

12. Чем размещения отличаются от перестановок?

13. Что такое сочетания и в каких задачах они используются?

14. Какова практическая значимость комбинаторики для информатики и криптографии?

15. Почему теория множеств считается фундаментом для дальнейшего изучения логики, теории алгоритмов и теории вероятностей?

7. ФОРМАЛЬНЫЕ ТЕОРИИ И ЯЗЫКИ

Формальные системы и языки занимают фундаментальное место в современной математике, логике и информатике. Их назначение состоит в создании строгого и недвусмысленного аппарата для описания вычислительных процессов, алгоритмов и структур данных. В отличие от естественного языка, который допускает многозначность и интерпретационные трудности, формальные языки основаны на четко определенных правилах построения и преобразования символов.

В основе любой формальной системы лежит конечный набор символов (алфавит), правила образования правильных выражений (синтаксис) и механизмы их преобразования (правила вывода). Именно эта строгость позволяет использовать формальные системы для описания логических рассуждений, построения алгоритмов и анализа вычислительных процессов.

Значение формальных языков трудно переоценить. В теории они служат инструментом для исследования границ алгоритмически вычислимого, для формулирования и доказательства теорем, а также для построения моделей абстрактных вычислительных машин. В прикладной области они лежат в основе языков программирования, методов автоматической проверки корректности программ, систем компьютерной верификации, а также технологий искусственного интеллекта.

Изучение формальных систем и языков позволяет объединить абстрактно-математические конструкции с реальными задачами информатики, формируя основу для понимания как фундаментальных теорий, так и практических технологий.

7.1. Построение формальной системы

Формальная система предназначена для строгого описания рассуждений и вычислений на уровне символов. При ее построении прежде всего фиксируют объектный язык – совокупность выражений, с которыми система будет работать, – и отделяют его от метаязыка, в котором даются определения и ведется обсуждение свойств самой системы. Это разграничение исключает двусмысленность и позволяет формулировать утверждения о языке без смешения уровней.

Отправной точкой служит алфавит – конечный набор элементарных символов (переменные, функциональные и предикатные знаки, логические связки, кванторы, разделители). На основе алфавита через индуктивные правила построения вводится множество корректных выражений, или правильных формул. Эти правила определяют, какие строки символов признаются термами, а какие – формулами, и обеспечивают единственность

синтаксического разбора (каждая формула имеет однозначную структуру). Таким образом задается чисто синтаксическая сторона языка, независимая от интерпретации.

Далее фиксируется вычислительное ядро языка – аксиомы и правила вывода. Аксиомы чаще всего задаются не одной формулой, а схемами аксиом, позволяющими подставлять в них произвольные формулы подходящего вида. Правила вывода регламентируют переход от уже признанных формул к новым; центральным примером является *modus ponens*, но в зависимости от выбранного исчисления применяются и другие правила (обобщение по квантору, правила введения и удаления связок, структурные правила). Последовательность формул, каждая из которых либо аксиома, либо получается из предыдущих по правилам вывода, называется выводом; конечная формула вывода – теоремой системы. На этом уровне формальная система остается сугубо синтаксической: она оперирует символами и их допустимыми преобразованиями, не обращаясь к значениям вне языка.

Чтобы связать синтаксис с содержанием, вводят семантику языка. Семантика задается через интерпретации (или модели), которые приписывают термам элементы некоторой области, а формулам – логические значения в зависимости от выбранных истинностных условий. Говорят, что интерпретация удовлетворяет формулу, если последняя истинна при данной интерпретации, и что модель удовлетворяет множество формул, если истинны все формулы из этого множества. Семантическая связь между множеством предпосылок и заключением выражается отношением логического следования: заключение истинно во всех интерпретациях, в которых истинны предпосылки.

Ключевое место занимают два моста между синтаксисом и семантикой – корректность (зву́чность) и полнота. Корректность утверждает, что все выводимое в системе семантически истинно: если формула доказуема, то она является логическим следствием аксиом. Полнота, напротив, гарантирует, что всякая семантически истинная формула выводима: если формула истинна во всех моделях аксиом, то существует ее синтаксический вывод. Совместно эти свойства обеспечивают адекватность выбранного исчисления своей семантике и позволяют свободно переходить от доказательств к моделям и обратно.

Наряду с корректностью и полнотой изучаются метатеоретические характеристики формальной системы. Непротиворечивость означает невозможность вывести формулу вместе с ее отрицанием; независимость аксиом фиксирует, что ни одна из них не выводится из остальных; разрешимость (или вычислимая выводимость) и получислимость описывают, существует ли алгоритм, распознающий все теоремы языка. Эти свойства уже не формулируются внутри системы, а доказываются во внешней,

метаматематической перспективе, опираясь на понятия вычислимости и алгоритмической сложности.

Общий шаблон построения удобно иллюстрируется на примере исчислений логики. В пропозициональной логике алфавит включает пропозициональные переменные и логические связки; правила построения порождают класс формул; аксиоматика и правило *modus ponens* образуют доказательственную систему (в стиле Гильберта или натурального вывода), тогда как семантика задается булевыми оценками и таблицами истинности. Для этой логики известны и корректность, и полнота, а также алгоритмическая разрешимость: существует конечная процедура, решающая задачу выводимости. В логике первого порядка к алфавиту добавляются функциональные и предикатные символы, кванторы и термы; семантика опирается на интерпретации в структурах с заданными операциями и отношениями. Здесь также выполняются теоремы о корректности и полноте, однако общая задача выводимости уже неразрешима в алгоритмическом смысле, что демонстрирует ограниченность механических процедур в более выразительных языках.

Таким образом, построение формальной системы проходит последовательные стадии: задание алфавита и правил образования выражений, фиксация аксиом и правил вывода, введение семантики и установление отношений между синтаксисом и моделями. Эта конструкция задает универсальный каркас для дальнейшего исследования – от формальных языков и автоматов до программных спецификаций и верификации, – позволяя единым образом формулировать и решать как теоретические, так и прикладные задачи.

7.2. Формальные языки и объекты информатики

Формальные языки представляют собой одно из ключевых понятий современной информатики, поскольку именно они обеспечивают строгий и однозначный аппарат описания алгоритмов, программ и данных. Под формальным языком понимается множество последовательностей символов, называемых словами, которые строятся из фиксированного конечного алфавита в соответствии с определенными правилами. В отличие от естественных языков, в которых допускается многозначность и неоднозначность интерпретации, формальные языки исключают подобные неопределенности, что делает их особенно важными для автоматической обработки информации.

Формальные языки тесно связаны с объектами информатики на различных уровнях. Во-первых, они применяются для описания алгоритмов

и программ. Каждый язык программирования, от машинных команд до языков высокого уровня, представляет собой формальный язык с четко заданным синтаксисом и семантикой. Именно благодаря этому удается создавать программы, которые могут быть однозначно интерпретированы вычислительными машинами. Во-вторых, формальные языки используются при описании структур данных и протоколов взаимодействия. Так, грамматики позволяют задавать правила построения корректных программ, спецификации определяют условия корректности их выполнения, а формальные модели коммуникации фиксируют требования к обмену сообщениями в распределенных системах.

Особое место в теории формальных языков занимает вопрос о различиях в их выразительной мощности. Наиболее известным способом классификации является иерархия Хомского, выделяющая несколько уровней в зависимости от строгости и сложности правил построения слов. На самом низком уровне находятся регулярные языки, которые могут быть описаны с помощью регулярных выражений и распознаются конечными автоматами. Эти языки находят широкое применение в практике, например, при поиске текстовых шаблонов, построении лексических анализаторов и проектировании сетевых протоколов. Более сложным классом являются контекстно-свободные языки, задаваемые контекстно-свободными грамматиками и распознаваемые автоматами с магазинной памятью. Они позволяют описывать вложенные структуры и иерархические зависимости, что делает их особенно удобными для синтаксического анализа языков программирования.

Следующим уровнем являются контекстно-зависимые языки, которые требуют более мощных правил преобразования и, соответственно, более сложных автоматов для распознавания. Их использование связано с задачами, где необходимо учитывать дополнительные ограничения, например, проверку корректности типов в программировании или анализ некоторых специализированных форматов данных. Наконец, наивысший уровень иерархии образуют рекурсивно перечислимые языки, которые эквивалентны множествам слов, распознаваемым машинами Тьюринга. Этот класс охватывает все алгоритмически вычислимые языки и тем самым определяет границы возможного в автоматической обработке информации.

Практическое значение формальных языков выходит далеко за рамки их математической классификации. Они составляют основу процессов компиляции и интерпретации программ, где осуществляется последовательный анализ исходного текста: от лексического и синтаксического анализа до генерации машинного кода. Формальные языки применяются при описании запросов к базам данных, в которых четкие правила построения предложений позволяют системам управления эффективно обрабатывать большие

объемы информации. Кроме того, они лежат в основе спецификаций протоколов передачи данных, что обеспечивает корректность взаимодействия между элементами компьютерных сетей.

Теория формальных языков тесно связана с теорией автоматов, которая описывает устройства, способные распознавать или порождать языки. Автоматы предоставляют формальный механизм проверки принадлежности слова языку, что имеет важное значение в задачах верификации и анализа программ. С помощью автоматов и грамматик становится возможным строго описывать свойства вычислительных систем и проверять их корректность.

Формальные языки представляют собой не только абстрактную математическую конструкцию, но и универсальный инструмент, соединяющий фундаментальную теорию вычислений с прикладными задачами информатики. Их использование позволяет формализовать описание программ и данных, исключить неоднозначность интерпретаций и обеспечить возможность автоматической проверки корректности вычислительных процессов. Именно поэтому они занимают центральное место в теории и практике построения современных информационных систем

7.3. Алфавиты, слова, языки

Формальные теории опираются на строгое определение элементарных понятий, без которых невозможно дальнейшее развитие системной логики и построение вычислительных моделей. К числу таких понятий относятся алфавит, слово и язык.

Под *алфавитом* понимается конечное множество символов, из которых формируются все последующие конструкции. В зависимости от целей исследования алфавит может содержать буквы, цифры, специальные знаки, операторы, логические константы или команды программирования. Фундаментальным требованием является конечность алфавита, что обеспечивает дискретность и обозримость всех операций.

Из символов алфавита строятся конечные последовательности, которые называются *словами*. Длина слова определяется числом символов в нем, и хотя она может быть произвольной, каждое конкретное слово всегда конечно. Важным элементом является пустое слово, обозначаемое символом ε, которое не содержит символов и имеет длину ноль. Пустое слово играет особую роль в теории формальных языков, поскольку позволяет корректно описывать процессы построения и преобразования строк, а также формулировать аксиоматические правила в грамматиках.

Совокупность всех слов, которые могут быть образованы из данного алфавита, называется замыканием алфавита и обозначается Σ. Это множество бесконечно, так как оно включает слова любой длины, начиная с пустого. Внутри этого пространства выделяются подмножества, обладающие определенными структурными свойствами. Такие подмножества и называются формальными языками.

Таким образом, *формальный язык* определяется как множество слов над фиксированным алфавитом. Языки могут быть конечными, если количество слов в них ограничено (например, язык, состоящий только из всех слов длины три), и бесконечными, если их число неограниченно (например, язык всех корректных арифметических выражений). Структура языка задается специальными правилами построения, которые фиксируют, какие слова принадлежат языку, а какие нет.

Среди таких правил важнейшую роль играют грамматики, формализующие процесс построения слов. Наиболее известной системой классификации является иерархия Хомского, которая выделяет регулярные, контекстно-свободные, контекстно-зависимые и рекурсивно перечислимые языки. Каждому из этих уровней соответствует определенный тип грамматик и автоматов, что позволяет связать формальные языки с вычислительными моделями.

Необходимо различать два уровня описания языка – синтаксис и семантику. Синтаксис определяет допустимые комбинации символов и правила построения слов, задавая тем самым форму корректных выражений. Семантика же придает этим выражениям смысл, связывая их с операциями, вычислительными процессами или объектами предметной области. В информатике это различие имеет принципиальное значение: язык программирования задается синтаксическими правилами, однако именно семантика определяет, какие вычисления выполняет программа при запуске.

Практическая значимость понятий алфавита, слова и языка проявляется во многих областях. Алфавит двоичных символов лежит в основе систем кодирования информации и архитектуры вычислительных машин. Слова формируют машинные инструкции, тексты программ и данные, обрабатываемые компьютером. Формальные языки определяют синтаксис языков программирования, спецификации протоколов передачи данных и структуры запросов в базах данных.

Таким образом, триада «алфавит – слово – язык» представляет собой фундаментальную основу формальных систем. Она обеспечивает переход от элементарных символов к сложным структурам, позволяя формализовать построение программ, описывать вычислительные процессы и задавать строгие модели информационных объектов. Именно на этих понятиях

базируется все дальнейшее развитие теории формальных языков и автоматов, что делает их ключевым элементом как в математике, так и в информатике.

7.4. Алгоритмические проблемы: принадлежность, выполнимость, оптимизация

Формальные языки, как математические объекты, позволяют не только описывать структуры и процессы, но и ставить задачи, связанные с их исследованием. Важнейшее направление здесь образуют так называемые алгоритмические проблемы, которые формулируются в терминах операций над словами и языками и имеют ключевое значение в теории вычислений и информатики. Среди них особое место занимают три группы задач: проблема принадлежности, проблема выполнимости и задачи оптимизации.

Проблема принадлежности заключается в следующем: дано слово и формальный язык, требуется определить, принадлежит ли это слово данному языку. Формально, если язык L задан грамматикой или автоматом, необходимо установить, верно ли, что слово $w \in L$. Эта задача является фундаментальной для синтаксического анализа. В частности, в компиляторах языков программирования именно она решается при проверке корректности исходного кода. Для регулярных и контекстно-свободных языков проблема принадлежности разрешима, и для нее разработаны эффективные алгоритмы. Например, конечные автоматы за линейное время проверяют принадлежность слова регулярному языку, а алгоритм Кока – Янгера – Касами и метод рекурсивного спуска позволяют решать ту же задачу для контекстно-свободных грамматик.

Проблема выполнимости в общем виде формулируется как вопрос о том, существует ли хотя бы одно слово, принадлежащее заданному языку. Иными словами, пуст ли язык, определяемый данной грамматикой или автоматом. Эта задача имеет теоретическое значение при исследовании формальных систем, так как позволяет определить, не сводится ли язык к пустому множеству вследствие слишком жестких ограничений. Для регулярных языков выполнимость проверяется алгоритмически, тогда как для более сложных классов языков задача может оказаться трудноразрешимой или вовсе неразрешимой. В практическом плане проблема выполнимости связана с верификацией программных спецификаций: если спецификация задает язык, необходимо проверить, существуют ли реализации, удовлетворяющие ее требованиям.

Третью важную категорию составляют *задачи оптимизации*. Они возникают тогда, когда принадлежность или выполнимость установлены,

но требуется найти решение, обладающее некоторыми минимальными или максимальными свойствами. Примером может служить поиск кратчайшего слова, принадлежащего языку, или построение минимального автомата, распознающего данный язык. Задачи оптимизации особенно важны в прикладных областях информатики: в компиляции они связаны с минимизацией синтаксических деревьев, в сетевых протоколах – с оптимизацией последовательностей сообщений, в алгоритмах сжатия данных – с построением наиболее компактных кодов.

Следует отметить, что алгоритмические проблемы над формальными языками различаются по своей вычислительной сложности. Некоторые задачи решаются эффективно, в полиномиальное время, другие же оказываются NP-трудными или вовсе неразрешимыми. Именно поэтому теория формальных языков и автоматов тесно связана с теорией вычислимости и теорией сложности, которые определяют границы возможного и практическую применимость тех или иных методов.

Алгоритмические проблемы принадлежности, выполнимости и оптимизации образуют основу прикладных и теоретических исследований формальных языков. Они обеспечивают строгий аппарат для проверки корректности программ, анализа грамматик, проектирования алгоритмов и оптимизации вычислительных процессов. Изучение этих проблем позволяет глубже понять как фундаментальные свойства формальных систем, так и практические аспекты их использования в информатике.

7.5. Сложность по Колмогорову и ее применение

Одним из важнейших направлений теории алгоритмов является разработка методов количественной характеристики сложности объектов. В отличие от классического подхода, где измеряются ресурсы, необходимые алгоритму для решения задачи (время, память и т. д.), в 1960-е гг. была предложена новая концепция, сосредоточенная на самих объектах, а не на процессах их обработки. Эта концепция получила название *алгоритмической сложности* или *сложности по Колмогорову*.

Идея была независимо сформулирована А. Н. Колмогоровым, Р. Соломоновым и Г. Чейтином. Несмотря на различные исходные мотивации, все они исходили из одной фундаментальной мысли: степень упорядоченности объекта можно определить через длину его кратчайшего описания.

Формально под Колмогоровской сложностью слова понимается длина наименьшей программы, которая при выполнении на универсальной машине Тьюринга порождает это слово и завершает работу. Если объект может быть описан короткой программой, его сложность невелика; если

же для воспроизведения требуется программа, практически равная по длине самому объекту, сложность считается высокой. Таким образом, сложность в колмогоровском смысле измеряет информационное содержание объекта.

Пример иллюстрирует различие между простыми и случайными последовательностями. Строка вида:

01010101…

(100 повторений блока «01») имеет низкую сложность, так как может быть описана короткой программой «вывести '01' 100 раз». Напротив, случайная строка длиной 200 символов, лишенная внутренней регулярности, практически не поддается сжатию, и ее Колмогоровская сложность будет близка к 200.

Важнейшей особенностью данной теории является ее *универсальность*. Колмогоровская сложность не зависит от конкретного языка программирования или вычислительной модели. Различия в выборе машины выражаются лишь в постоянной добавке, не изменяющей асимптотические оценки. Это свойство делает концепцию объективной и фундаментальной для всей теории информации.

Следует отметить, что величина Колмогоровской сложности невычислима. Доказано, что не существует алгоритма, который для произвольной строки позволял бы определить ее точную сложность. Этот факт является прямым следствием общих результатов теории вычислимости, в частности неразрешимости *проблемы останова*. Тем не менее существуют методы оценки и приближения, которые широко применяются на практике.

Практическое значение теории сложности по Колмогорову многообразно. В теории информации она дает строгий критерий случайности: строка считается случайной, если ее Колмогоровская сложность близка к ее длине. В криптографии этот подход используется для анализа стойкости шифров, так как криптотексты должны обладать высокой алгоритмической сложностью, чтобы исключить возможность упрощенного восстановления исходных данных. В области сжатия данных Колмогоровская сложность объясняет фундаментальные пределы: сжать можно только те данные, которые содержат избыточность, то есть имеют относительно малую сложность.

Кроме того, концепция алгоритмической сложности оказала влияние на современные направления исследований. В машинном обучении и искусственном интеллекте идея кратчайшего описания данных лежит в основе принципа минимальной достаточности моделей: предпочтение отдается тем описаниям, которые одновременно компактны и адекватно объясняют наблюдения. Аналогичные принципы используются в статистическом моделировании и при построении вероятностных алгоритмов.

Таким образом, теория сложности по Колмогорову занимает особое место в теории алгоритмов и информатики. Она позволяет строго оценивать степень упорядоченности и случайности объектов, устанавливает предельные возможности сжатия и кодирования, формирует основу криптографических методов и современных подходов к анализу данных. Несмотря на невычислимость в строгом смысле, данная концепция служит универсальным инструментом для понимания фундаментальных свойств информации и вычислительных процессов.

Контрольные вопросы

1. Что понимается под формальной системой и какие компоненты она включает?
2. Какова роль аксиом и правил вывода в построении формальных систем?
3. Чем формальные языки отличаются от естественных и почему они применяются в информатике?
4. Какие объекты информатики описываются средствами формальных языков?
5. Дайте определение алфавита. Какие примеры алфавитов встречаются в информатике?
6. Что называется словом в теории формальных языков? Какое значение имеет пустое слово?
7. Что такое замыкание алфавита Σ и чем оно отличается от формального языка?
8. В чем состоит различие между синтаксисом и семантикой формальных языков?
9. Как формальные грамматики используются для описания языков программирования?
10. Какие алгоритмические задачи возникают при работе с формальными языками (принадлежность, выполнимость, оптимизация)?
11. В чем заключается идея алгоритмической сложности по Колмогорову?
12. Как определяется Колмогоровская сложность слова и чем она отличается от длины слова?
13. Почему Колмогоровская сложность является невычислимой величиной?
14. Как связаны Колмогоровская сложность и понятие случайности последовательности?
15. Какие практические применения имеет теория сложности по Колмогорову (например, в криптографии, теории информации или машинном обучении)?

8. КОНЕЧНЫЕ АВТОМАТЫ

Теория конечных автоматов является одной из фундаментальных областей теории вычислительных процессов и формальных языков. Конечные автоматы представляют собой абстрактные математические модели, описывающие системы с конечным числом состояний и правилами перехода между ними. Несмотря на простоту своей структуры, они обладают значительной выразительной силой и позволяют моделировать широкий спектр вычислительных и информационных процессов.

Ключевая идея заключается в том, что поведение сложной системы можно свести к последовательности дискретных состояний, каждое из которых однозначно определяется входными данными и текущим положением. Конечный автомат, таким образом, выполняет роль преобразователя входных последовательностей символов (слов) в выходные действия или решения задач о принадлежности слова заданному языку.

Особое значение конечные автоматы имеют в области формальных языков: именно с их помощью задаются и распознаются регулярные языки, которые образуют один из важнейших классов в иерархии Чомского. Это делает автоматы незаменимыми в таких задачах, как лексический анализ программ, построение протоколов взаимодействия в компьютерных сетях, моделирование цифровых устройств и проектирование управляющих систем.

Исторически теория конечных автоматов сыграла важную роль в развитии общей теории алгоритмов и вычислимости, а также в становлении программной инженерии. Современные приложения охватывают самые разные области: от компиляторов и поисковых систем до биоинформатики и систем искусственного интеллекта.

Простота и наглядность конечных автоматов делают их удобным инструментом для изучения фундаментальных вопросов вычислительных моделей, анализа сложности и построения алгоритмов. В то же время их универсальность позволяет использовать автоматы как связующее звено между абстрактной теорией и прикладными задачами обработки информации.

8.1. Конечные автоматы как модель вычислений

Конечный автомат является одной из наиболее простых и в то же время фундаментальных моделей вычислений. Его работа основана на идее, что система может находиться только в одном из конечного множества состояний, переходя из одного состояния в другое под воздействием входных символов. Такой подход позволяет свести описание поведения системы к строго

формализованным правилам, исключающим двусмысленность и неопределенность.

Исторически концепция конечных автоматов оформилась в середине XX в. в работах У. Маккалока и У. Питтса, которые исследовали формализацию работы нейронных сетей, а затем получила развитие в трудах С. Клини, М. Рабина, Д. Скотта и других исследователей. Конечные автоматы оказались удобной моделью не только для изучения вычислительных процессов, но и для построения формальных языков, что предопределило их центральное место в информатике.

Формально конечный автомат задается пятеркой.

В зависимости от способа задания функции переходов различают детерминированные конечные автоматы (ДКА) и недетерминированные конечные автоматы (НКА). В детерминированном автомате каждому состоянию и входному символу соответствует ровно одно следующее состояние. В недетерминированном автомате допускается несколько возможных переходов или даже переход без чтения символа (ε-переход). Несмотря на различие в определении, оба типа автоматов эквивалентны по вычислительной мощности: любой язык, распознаваемый НКА, может быть распознан некоторым ДКА.

Языки, допускаемые конечными автоматами, называются *регулярными языками*. Они образуют важный класс в иерархии Хомского и обладают рядом полезных свойств: замкнуты относительно объединения, пересечения, конкатенации и звезды Клини. Регулярные языки широко применяются в описании синтаксиса простых конструкций, определении лексических структур программных языков и спецификации протоколов связи.

Практическое значение конечных автоматов чрезвычайно велико. В программировании они используются при построении компиляторов и интерпретаторов, где лексический анализатор фактически реализует автомат, распознающий ключевые слова и структуры кода. В инженерии автоматы применяются для проектирования цифровых схем, контроллеров и систем управления. В теории вычислений они служат базовой моделью для анализа алгоритмов, а в прикладных задачах – инструментом для обработки потоков информации, распознавания шаблонов и моделирования процессов с конечным числом состояний.

Таким образом, конечные автоматы являются не только важнейшей моделью вычислений, но и универсальным инструментом для формализации и анализа дискретных процессов. Их простота и строгость делают их идеальной отправной точкой для изучения формальных языков и алгоритмических систем, а их применимость обеспечивает прочную связь между теоретической информатикой и практическими задачами современной вычислительной техники.

Автоматы Мили и Мура

В теории конечных автоматов различают две близкие по выразительной мощности модели – автомат Мили и автомат Мура. Обе модели описывают дискретные системы с конечным числом состояний и переходами между ними, однако по-разному трактуют зависимость выходных сигналов от входов и состояний. Автоматы Мили и Мура представляют собой разновидности детерминированных конечных автоматов, в которых выходные значения формируются по-разному. В автомате Мили выход зависит от текущего состояния и входного символа, поэтому подписи делаются на переходах. В автомате Мура выход зависит только от состояния, поэтому каждому состоянию приписывается фиксированное выходное значение.

Основные различия:

1. Автомат Мили – выход связан с ребрами переходов (состояние + вход).

2. Автомат Мура – выход связан с вершинами (состояния).

3. Автомат Мили, как правило, имеет меньше состояний, чем эквивалентный автомат Мура.

4. Автомат Мили определяется кортежем $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$, где Q – конечное множество состояний, Σ – входной алфавит, Γ – выходной алфавит, δ: $Q \times \Sigma \to Q$ – функция переходов, λ: $Q \times \Sigma \to \Gamma$ – выходная функция, а $q_0 \in Q$ – начальное состояние. Особенность модели состоит в том, что выход в каждый момент времени зависит одновременно от текущего состояния и текущего входного символа.

5. Автомат Мура задается кортежем $M' = (Q', \Sigma, \Gamma, \delta', \lambda', q'_0)$, с теми же алфавитами, но с выходной функцией λ': $Q' \to \Gamma$, зависящей только от состояния. В этой модели каждое состояние несет метку выходного символа, и выход изменяется только при входе в новое состояние.

Обе модели эквивалентны по распознаваемым регулярным языкам. Преобразование из Мили в Мура достигается «разворачиванием» выходов на переходах в выходы состояний. Обратное преобразование из Мура в Мили позволяет уменьшить количество состояний.

С точки зрения цифровой реализации автомат Мили обладает меньшей латентностью реакции, так как выход зависит от пары (q, a). Автомат Мура обеспечивает более стабильный выход, так как он зависит только от состояния.

Практический выбор модели определяется требованиями к задержке, устойчивости выходов и удобству спецификации. В таблице ниже представлены основные отличия автоматов Мили и Мура (табл. 8.1).

Таблица 8.1

Основные отличия

Автомат Мили	*Автомат Мура*
Выход зависит от состояния и входа	Выход зависит только от состояния
Мгновенная реакция на вход	Задержка выхода на один такт
Компактное описание, меньше состояний	Более стабильные выходы, удобен в аппаратуре

Конечные автоматы, несмотря на кажущуюся простоту своей структуры, занимают фундаментальное место в теории вычислений и формальных языков. Они позволяют описывать широкий спектр дискретных процессов и обеспечивают строгий математический аппарат для анализа систем с конечным числом состояний.

Различие между детерминированными и недетерминированными автоматами показывает, что при разных подходах к построению переходов сохраняется эквивалентная вычислительная мощность, что подчеркивает устойчивость самой модели. В свою очередь, включение выходных сигналов в автоматы Мили и Мура расширяет область применения теории автоматов, позволяя моделировать реальные устройства, протоколы и управляющие системы.

Таким образом, конечные автоматы представляют собой не только абстрактную модель для изучения вычислительных процессов, но и универсальный инструмент для инженерной практики. Их применение охватывает компиляторы и протоколы связи, цифровые схемы и системы управления, средства распознавания и обработки информации. Именно поэтому они остаются отправной точкой для дальнейшего изучения более сложных вычислительных моделей и методов формального анализа в информатике.

8.2. Распознавание языков

Одним из важнейших применений конечных автоматов является распознавание формальных языков. Под распознаванием понимается процесс проверки того, принадлежит ли данное слово множеству слов, образующих язык. Формально язык L называется распознаваемым автоматом $A = (Q, \Sigma, \delta, q_0, F)$, если для любого слова $w \in \Sigma^*$ автомат, начав работу в состоянии q_0 и последовательно обработав символы слова согласно функции переходов δ, завершает работу в состоянии из множества F тогда и только тогда, когда w принадлежит L.

Множество языков, распознаваемых конечными автоматами, образует класс регулярных языков. Этот класс занимает важное место в иерархии Хомского и характеризуется замкнутостью относительно операций объединения, пересечения, дополнения, конкатенации и звезды Клини. Регулярные языки эквивалентны регулярным выражениям и могут быть описаны как грамматиками 3-го типа, так и автоматами (табл. 8.2).

Таблица 8.2

Пример таблицы переходов

Состояние	Вход: 0	Вход: 1
q_0	q_0	q_1
q_1	q_0	q_1

Каждому конечному автомату соответствует регулярное выражение, описывающее тот же язык. Например, язык двоичных слов, оканчивающихся на 1, задается выражением (0|1)*1, а язык слов с четным числом нулей может быть описан более сложным регулярным выражением, использующим конкатенацию и итерацию.

Формулы распознавания языков

В разделе «Распознавание языков» важное место занимает формальное определение конечных автоматов и математические формулы, которые задают правила их работы. Ниже приведены основные определения и соотношения.

1. Формальное определение конечного автомата

Конечный автомат задается пятеркой:

$A = (Q, \Sigma, \delta, q_0, F)$, где Q – конечное множество состояний; Σ – входной алфавит; δ: $Q \times \Sigma \to Q$ – функция переходов; $q_0 \in Q$ – начальное состояние; $F \subseteq Q$ – множество допускающих состояний.

2. Расширенная функция переходов

Для обработки слова $w \in \Sigma^*$ вводится расширенная функция переходов:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$$

Здесь $x \in \Sigma^*$, $a \in \Sigma$.

3. Язык, распознаваемый автоматом

Множество всех слов, которые автомат допускает, определяется как:

$$L(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

4. Эквивалентность ДКА и НКА

Для недетерминированного конечного автомата (НКА) функция переходов имеет вид:

δ: $Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$.

Несмотря на возможность ε-переходов и недетерминизма, для любого НКА существует эквивалентный ДКА, распознающий тот же язык: $L(NFA) = L(DFA)$.

5. Свойства регулярных языков

Регулярные языки замкнуты относительно основных операций. Если $L_1, L_2 \subseteq \Sigma^*$, то:

$L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 * L_2$, L_1^*, $\overline{L}_1$ – тоже регулярные языки.

Пример конечного автомата

Рассмотрим конечный автомат, распознающий язык $L = \{w \in \{0,1\}^* \mid w$ оканчивается на 01$\}$. Его работа описывается с помощью таблицы переходов (табл. 8.3).

Таблица 8.3

Пример таблицы переходов

Состояние	Символ входа	Следующее состояние
q_0	0	q_1
q_0	1	q_0
q_1	0	q_1
q_1	1	q_2
q_2	0	q_1
q_2	1	q_0

Допускающее состояние: q_2.

Распознавание языков с помощью конечных автоматов представляет собой фундаментальный метод формальной информатики. Регулярные языки, несмотря на ограничения, охватывают широкий спектр практических задач: лексический анализ в компиляторах, обработка протоколов связи, поиск и фильтрация текстов. Простота и строгость конечных автоматов делает их ключевым инструментом в теории и практике вычислений.

Контрольные вопросы

1. Что означает понятие «распознавание языка» в теории формальных языков?

2. Как формально определяется конечный автомат, используемый для распознавания языков?

3. Что представляет собой расширенная функция переходов δ и для чего она вводится?

4. Как определяется множество слов, допускаемых автоматом?

5. В чем заключается принципиальное различие между детерминированными (ДКА) и недетерминированными (НКА) конечными автоматами?

6. Какую роль играют ε-переходы в недетерминированных конечных автоматах?

7. Докажите или объясните, почему НКА и ДКА эквивалентны по вычислительной мощности.

8. Какие языки называются регулярными и как они связаны с конечными автоматами?

9. Какие операции сохраняют замкнутость класса регулярных языков?

10. Как с помощью конечного автомата можно описать язык всех слов, оканчивающихся на подстроку «01»?

11. Как регулярные выражения связаны с конечными автоматами?

12. Почему конечные автоматы не могут распознавать контекстно-свободные языки в общем случае?

13. В чем заключается практическая значимость распознавания языков для программирования?

14. Как применяется распознавание регулярных языков в лексическом анализе компиляторов?

15. Приведите пример языка, который не может быть распознан конечным автоматом, и объясните причину.

9. МАШИНЫ ТЬЮРИНГА

Машина Тьюринга является одной из центральных моделей вычислений, предложенной Аланом Тьюрингом в 1936 г. Она представляет собой абстрактное устройство, способное выполнять алгоритмы любого уровня сложности, которые могут быть формализованы. Именно эта модель послужила основой для формулировки тезы Черча – Тьюринга: всякая вычислимая функция может быть реализована на машине Тьюринга.

Главное значение машины Тьюринга состоит в том, что она устанавливает границы алгоритмически разрешимого. Несмотря на ее простоту, модель способна имитировать работу любых современных вычислительных систем, что делает ее универсальным инструментом для теоретической информатики.

9.1. Структура машины Тьюринга

Машина Тьюринга представляет собой формальную вычислительную модель, которая была введена Аланом Тьюрингом в 1936 г. с целью строгого анализа понятия алгоритма и вычислимости. Данная модель приобрела фундаментальное значение в теории алгоритмов и математической логике, став универсальным эталоном для сравнения вычислительной мощности различных систем. В отличие от более простых устройств, таких как конечные автоматы, машина Тьюринга обладает неограниченной (хотя и абстрактной) памятью и способна выполнять вычисления любого уровня сложности, которые можно формализовать в алгоритмической форме.

Машина Тьюринга состоит из трех основных компонентов: бесконечной ленты, устройства управления и головки чтения/записи.

Структура машины Тьюринга включает несколько основных компонентов, каждый из которых выполняет определенную функцию:

1. Лента

Лента представляет собой бесконечную последовательность ячеек, каждая из которых может содержать символ из заданного алфавита. В простейшем случае алфавит состоит из двух символов: «пустой» и «1». Лента одновременно является носителем входных данных, рабочей памятью и областью вывода результатов.

2. Головка чтения/записи

Головка может перемещаться по ленте влево или вправо, считывать символ в текущей ячейке и изменять его. Все вычисления машины Тьюринга реализуются через последовательность таких операций.

3. Устройство управления

Состояние машины определяется ее внутренним регистром и положением головки. Устройство управления действует по заранее заданной программе, которая задается в виде таблицы переходов. Каждое правило имеет вид:

$$(q_i, a) \rightarrow (q_j, b, D),$$

где q_i – текущее состояние, a – символ, считанный с ленты, q_j – новое состояние, b – символ, который записывается в текущую ячейку, $D \in \{L, R\}$ – направление перемещения головки (влево или вправо).

4. *Свойства машины Тьюринга:*

4.1. *Универсальность.* Существуют так называемые универсальные машины Тьюринга, способные имитировать работу любой другой машины Тьюринга при условии, что ее программа и данные заданы на ленте. Это свойство лежит в основе концепции универсального компьютера.

4.2. *Дискретность.* Вычисления выполняются пошагово, каждый шаг представляет собой переход из одного состояния в другое при обработке одного символа.

4.3. *Детерминированность.* В классической модели каждому сочетанию «состояние + символ» соответствует ровно одно правило перехода. Тем не менее существуют и недетерминированные модификации, в которых допускается несколько возможных переходов.

4.4. *Неограниченность памяти.* Бесконечность ленты обеспечивает принципиальную возможность хранения неограниченного объема данных, что делает модель Тьюринга максимально приближенной к абстрактному понятию алгоритма.

9.2. Такт работы и пример программы

Работа машины Тьюринга организуется в виде последовательности дискретных шагов, которые называются тактами. Каждый такт представляет собой строго формализованную операцию преобразования текущего состояния машины и содержимого ячейки ленты. Несмотря на простоту такого шага, именно последовательность тактов обеспечивает реализацию алгоритмов любой сложности. Каждый такт включает в себя три основных действия.

1. Считывание символа. Головка машины фиксирует символ $a \in \Sigma$, находящийся в текущей ячейке ленты.

2. Выполнение правила перехода. Устройство управления обращается к таблице переходов и выбирает правило, соответствующее паре (q_i, a), где q_i – текущее состояние. В результате выбирается тройка (q_j, b, D).

3. Запись и перемещение. В текущую ячейку записывается новый символ b, после чего головка смещается на одну ячейку влево ($D = L$) или вправо ($D = R$), а текущее состояние изменяется на q_j. Таким образом, такт можно описать как функцию: δ: $Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R\}$.

Работа машины продолжается до тех пор, пока не будет достигнуто одно из заключительных состояний или пока не исчерпается программа.

Пример программы для машины Тьюринга

Рассмотрим простейшую задачу: прибавление единицы к двоичному числу, записанному на ленте. Пусть лента содержит последовательность из нулей и единиц, завершающуюся пустым символом (например, 1011□). Алгоритм работы следующий (табл. 9.1).

1. Переместиться вправо до первого пустого символа.

2. Начать операцию прибавления: если последний символ равен 0, заменить его на 1 и завершить работу.

3. Если последний символ равен 1, заменить его на 0 и сместиться влево. Продолжать перенос разряда до тех пор, пока не встретится 0 или не будет достигнут край числа.

4. Если все символы оказались равны 1, в начале числа записывается дополнительная 1.

*Таблица 9.*1

Пример таблицы переходов

Состояние	*Считан символ*	*Новый символ*	*Движение*	*Новое состояние*
q_0	0	1	N	q_f
q_0	1	0	L	q_0
q_0	□	1	N	q_f

Здесь: q_0 – рабочее состояние, q_f – финальное состояние, символ □ обозначает пустую ячейку, N – отсутствие движения (головка остается на месте).

Если на ленте записано число 1011 (что соответствует десятичному 11), то после выполнения программы результат будет 1100 (число 12). Таким образом, данная машина Тьюринга реализует алгоритм инкремента двоичного числа.

Контрольные вопросы

1. Кто предложил модель машины Тьюринга и в каком году?
2. Какова основная цель введения машины Тьюринга в теорию вычислений?
3. Из каких основных элементов состоит машина Тьюринга?
4. Как определяется алфавит машины Тьюринга?
5. Что такое лента машины Тьюринга и каковы ее свойства?
6. Какую роль играет головка в машине Тьюринга?
7. Что представляет собой функция переходов машины Тьюринга?
8. Что понимается под состоянием машины Тьюринга?
9. Объясните, что такое допускающее и завершающее состояние машины Тьюринга.
10. Опишите один такт работы машины Тьюринга. Какие действия выполняются за этот такт?
11. Как можно формализовать алгоритм при помощи таблицы переходов машины Тьюринга?
12. В чем заключается отличие машины Тьюринга от конечного автомата?
13. Почему машина Тьюринга считается универсальной моделью вычислений?
14. Сформулируйте тезу Черча – Тьюринга. Какое значение она имеет для информатики?
15. Приведите пример простой задачи, которая может быть решена на машине Тьюринга, и объясните ее выполнение.

10. ТЕОРИЯ ВЫЧИСЛИМОСТИ

Теория вычислимости является одним из центральных направлений теоретической информатики и математической логики. Ее предметом является исследование границ и возможностей алгоритмических методов, т. е. определение того, какие задачи могут быть решены с помощью алгоритмов, а какие принципиально не допускают алгоритмического решения.

Возникновение этой теории связано с работами А. Тьюринга, А. Черча, Э. Поста и А. Маркова в 1930–1940-е гг., когда были предложены различные формальные модели вычислений: машина Тьюринга, λ-исчисление, нормальные алгоритмы Маркова, рекурсивные функции. Несмотря на различие в подходах, все эти модели оказались эквивалентными с точки зрения вычислимой мощности, что позволило сформулировать единое и строгое понятие вычислимости.

Теория вычислимости изучает не только существование алгоритмов для решения тех или иных задач, но и доказательства их отсутствия. Именно здесь важнейшую роль играют специальные методы, такие как *диагонализация* и *сводимость*. Первый метод позволяет строить объекты, которые не могут быть перечислены никаким алгоритмом, тем самым демонстрируя существование принципиально неразрешимых задач. Второй метод дает возможность устанавливать неразрешимость новых задач путем сведения их к уже известным.

Значение теории вычислимости выходит далеко за рамки чисто математических интересов. Эта теория формирует основу для анализа алгоритмической сложности, разработки криптографических протоколов, построения формальных языков и доказательных систем. Более того, именно в рамках этой теории были впервые определены такие фундаментальные понятия, как «разрешимая» и «неразрешимая» задача, «вычислимая функция» и «полнота по сводимости».

Таким образом, теория вычислимости не только очерчивает границы возможного для алгоритмов, но и формирует основу для понимания самой природы вычислений, а также фундаментальных ограничений, накладываемых на процессы обработки информации.

10.1. Метод диагонализации

Метод диагонализации является одним из важнейших приемов в математической логике и теории вычислимости. Его впервые применил Георг Кантор в конце XIX в. для доказательства несчетности множества действительных чисел, а позднее этот метод был развит и адаптирован к проблемам

алгоритмов Аланом Тьюрингом, Куртом Геделем и другими исследователями. В современной информатике диагонализация используется для строгого обоснования существования неразрешимых задач и для установления пределов алгоритмических методов. Суть диагонализации заключается в следующем: если предполагается, что некоторое множество объектов можно перечислить в последовательности, то, рассмотрев их в виде матрицы (таблицы), можно построить новый объект, который гарантированно не принадлежит этому перечислению. Такой объект строится изменением элементов на диагонали матрицы, что обеспечивает его отличие от каждого элемента исходного списка хотя бы в одной позиции. Именно эта простая идея позволяет доказать, что множество всех действительных чисел несчетно, множество всех подмножеств натуральных чисел неизмеримо, а также что существуют функции, которые не могут быть вычислены никаким алгоритмом. Рассмотрим пример с последовательностями, состоящими из нулей и единиц. Пусть все бесконечные бинарные последовательности пронумерованы:

$s_1 = 0\ 1\ 0\ 0\ 1$

$s_2 = 1\ 1\ 1\ 0\ 0$

$s_3 = 0\ 0\ 1\ 1\ 0$

$s_4 = 1\ 0\ 1\ 1\ 1$

Построим новую последовательность $t = t_1, t_2, t_3...$ по правилу:

$t_i = 1$, если на диагонали стоит 0;

$t_i = 0$, если на диагонали стоит 1.

То есть элемент t_i выбирается противоположным диагональному элементу строки s_i. Полученная последовательность t отличается от первой строки в первом элементе, от второй – во втором, от третьей – в третьем и так далее. Следовательно, t не совпадает ни с одной последовательностью из перечисленного списка, что доказывает невозможность полного перечисления.

Применение к теории вычислимости

Предположим, что все вычислимые функции можно пронумеровать: $f_1(x)$, $f_2(x)$, $f_3(x)$... и каждая из них вычисляется некоторой машиной Тьюринга. Используя метод диагонализации, можно построить функцию $g(x) = f_x(x) + 1$, которая для каждого аргумента x отличается от функции $f_x(x)$. Это значит, что функция g не совпадает ни с одной функцией списка, т. е. она невычислима. Таким образом, метод диагонализации позволяет строго показать существование невычислимых функций.

Пример. Проблема останова

Метод диагонализации ярко проявляется в доказательстве неразрешимости проблемы останова. Пусть существует универсальная программа $H(P, x)$, которая определяет, завершится ли программа P на входе x. Тогда можно построить программу Q, работающую следующим образом:

– вызывает $H(Q, Q)$ для определения, остановится ли она сама;

– если H предсказывает остановку, программа уходит в бесконечный цикл;

– если H предсказывает бесконечный цикл, программа завершает работу.

Такое построение приводит к противоречию: программа ведет себя противоположно прогнозу алгоритма H. Следовательно, универсальный алгоритм останова не существует.

Значение метода

Метод диагонализации играет фундаментальную роль в теории вычислимости и логике:

– он показывает существование невычислимых функций;

– доказывает неразрешимость ряда задач (проблема останова, проблема Поста, десятая проблема Гильберта);

– лежит в основе доказательств теоремы Геделя о неполноте;

– используется в теории сложности для разграничения классов задач.

Таким образом, диагонализация является универсальным инструментом, позволяющим показать границы алгоритмически возможного и построить строгие примеры задач, выходящих за пределы разрешимого.

10.2. Метод сводимости

Метод сводимости является одним из ключевых инструментов теории вычислимости и теории сложности, позволяющим устанавливать взаимосвязь между различными задачами. Основная идея метода заключается в том, что трудность одной задачи можно передать другой, сведя решение первой к решению второй. Если удается построить алгоритм, который использует решение некоторой задачи B для нахождения ответа на задачу A, то говорят, что задача A сводится к задаче B.

Формально сводимость определяется следующим образом:

Пусть даны две задачи A и B. Задача A сводится к задаче B ($A \leq B$), если существует вычислимая функция f, которая преобразует любой экземпляр задачи A в такой экземпляр задачи B, что решение задачи B для этого экземпляра позволяет однозначно восстановить решение задачи A. При этом функция f должна вычисляться алгоритмически за конечное число шагов.

Сводимость является мощным инструментом для доказательства неразрешимости задач. Если известно, что задача *B* неразрешима, и задача *A* сводится к задаче *B*, то задача *A* также является неразрешимой. Таким образом, достаточно установить неразрешимость некоторой базовой задачи, после чего множество других задач можно объявить неразрешимыми с помощью сводимости.

Классическим примером является сведение различных задач к *проблеме останова*. Так, доказав, что задача об определении завершения работы произвольной программы является неразрешимой, можно показать неразрешимость множества других задач, связанных с анализом программ. Например, задача проверки эквивалентности двух программ или задача установления, напечатает ли программа хотя бы один символ при работе, также оказываются неразрешимыми, поскольку они сводятся к проблеме останова.

В теории алгоритмической сложности метод сводимости играет не менее важную роль. Он используется для выделения классов задач и установления их относительной трудности. Наиболее известный пример связан с классами P и NP. Многие задачи сводятся друг к другу за полиномиальное время. В частности, NP-полные задачи определяются именно через сводимость: задача считается NP-полной, если она принадлежит классу NP и любая задача из NP сводится к ней за полиномиальное время.

Таким образом, метод сводимости выполняет двойную функцию. С одной стороны, он служит строгим средством для доказательства неразрешимости, позволяя переносить это свойство с одних задач на другие. С другой стороны, он является основным инструментом классификации задач по сложности, обеспечивая фундамент для построения теории алгоритмической трудности (табл. 10.1).

Таблица 10.1

Примеры сводимости задач

Исходная задача (A)	*Задача, к которой выполняется сведение (B)*	*Результат*
Определить, завершится ли программа на входе	Проблема останова	Проблема останова неразрешима → исходная задача также неразрешима
Проверка: напечатает ли программа хотя бы один символ	Проблема останова	Неразрешима, так как сводится к останову
Проверка эквивалентности двух программ	Проблема останова	Неразрешима через сводимость
Проверка: имеет ли данное диофантово уравнение решение	Десятая проблема Гильберта	Доказано неразрешимым (Матиясевич, 1970)

Окончание табл. 10.1

Исходная задача (A)	*Задача, к которой выполняется сведение (B)*	*Результат*
Любая задача из класса NP	SAT (задача выполнимости булевых формул)	SAT NP-полна (теорема Кука – Левина)
NP-задача оптимизации (например, задача коммивояжера)	SAT или 3-SAT	Сведение за полиномиальное время → NP-полнота

Контрольные вопросы

1. Что изучает теория вычислимости и какое место она занимает в теоретической информатике?
2. В чем заключается основная идея метода диагонализации?
3. Как Кантор использовал метод диагонализации в доказательстве несчетности действительных чисел?
4. Как метод диагонализации применяется для построения невычислимых функций?
5. Почему диагонализация используется при доказательстве неразрешимости проблемы останова?
6. Как связаны метод диагонализации и теорема Геделя о неполноте?
7. Дайте формальное определение сводимости задач.
8. Как метод сводимости используется для доказательства неразрешимости задач?
9. Приведите пример задачи, сводимой к проблеме останова.
10. Почему проверка эквивалентности программ является неразрешимой задачей?
11. Как метод сводимости применяется в теории сложности для анализа классов задач?
12. Что такое NP-полные задачи и как они определяются через сводимость?
13. В чем состоит различие между доказательством неразрешимости методом диагонализации и методом сводимости?
14. Какова роль метода сводимости в установлении границ между «трудными» и «легкими» задачами?
15. Объясните, почему метод сводимости является универсальным инструментом как в теории вычислимости, так и в теории сложности.

11. ТЕОРИЯ СЛОЖНОСТИ

11.1. Классы сложности P, NP, NPC

Алгоритмическая сложность является центральным понятием теоретической информатики. Даже если задача алгоритмически разрешима, эффективность ее решения может различаться радикально: одни задачи решаются за секунды, другие – за миллионы лет даже на самых мощных компьютерах.

Для классификации задач по их «трудности» введены специальные классы сложности. Они объединяют задачи в группы в зависимости от того, сколько ресурсов требуется для их решения. Наиболее изученными и фундаментальными являются классы P, NP и NP-полные задачи (NPC).

Исторический контекст

1. 1930-е гг. А. Тьюринг и А. Черч заложили основы теории алгоритмов. Вопросы сложности еще не выделялись.

2. 1950–1960-е гг. Появляются первые оценки трудоемкости алгоритмов, развивается асимптотический анализ.

3. 1971 г. Стивен Кук формулирует теорему о NP-полноте (Cook's theorem): задача о выполнимости булевых формул (SAT) является NP-полной.

4. 1972 г. Ричард Карп публикует список из 21 NP-полной задачи и формулирует основу всей современной теории NP-полноты.

Эти работы показали, что многие практические задачи обладают фундаментальной вычислительной сложностью, и это ограничение не связано с конкретными компьютерами, а носит принципиальный характер.

Важнейшей частью теории алгоритмов и вычислительной сложности является классификация задач в зависимости от их вычислительной трудоемкости. Для формализации этого подхода вводятся специальные классы задач, отражающие различие между эффективно решаемыми задачами, задачами, допускающими быструю проверку решения, и задачами, для которых известные алгоритмические методы оказываются крайне неэффективными. Основные среди этих классов – P, NP, NP-полные и NP-трудные задачи.

Класс P

Класс P (Polynomial time) определяется как множество задач, для которых существует детерминированный алгоритм, работающий за время, ограниченное многочленом от размера входа. Другими словами, задача относится к классу P, если существует алгоритм A и натуральное число k, такие что для любого входа размера n работа алгоритма завершается за время $O(n^k)$.

Считается, что алгоритмы, работающие за полиномиальное время, обладают приемлемой сложностью. Это объясняется тем, что рост числа шагов с увеличением размера входных данных остается контролируемым

и не приводит к экспоненциальному взрыву трудоемкости. Именно поэтому класс P рассматривается как модель множества «эффективно решаемых» задач.

Примеры задач, принадлежащих к этому классу, включают сортировку массива (например, алгоритм быстрой сортировки со сложностью $O(n \log n)$), поиск кратчайшего пути в графе (алгоритм Дейкстры с трудоемкостью $O(n^2)$ при матричном представлении), проверку синтаксической корректности арифметических выражений, а также операции над матрицами, такие как умножение ($O(n^3)$ в базовом варианте).

Класс NP

Класс NP (Nondeterministic Polynomial time) включает задачи, обладающие более слабым условием: для них может не существовать известного полиномиального алгоритма поиска решения, однако любое предложенное решение можно проверить за полиномиальное время. Формально это означает, что задача принадлежит NP, если существует алгоритм-верификатор, который за полиномиальное время проверяет корректность входного кандидата-решения.

Таким образом, принадлежность к NP не гарантирует существования эффективного алгоритма решения, но утверждает, что процесс проверки результата выполняется за разумное время. Примером является задача коммивояжера: хотя неизвестен алгоритм, позволяющий быстро найти оптимальный маршрут, для предложенного маршрута можно за полиномиальное время вычислить его длину и убедиться, удовлетворяет ли он заданному условию. Сюда же относятся задачи о рюкзаке, раскраске графа в k цветов и многие другие.

Важно отметить, что все задачи из P также входят в NP. Если задачу можно эффективно решить, то ее решение можно столь же эффективно и проверить. Это означает, что $P \subseteq NP$.

NP-полные задачи

Особое место в теории вычислительной сложности занимают NP-полные задачи. Они представляют собой наиболее трудные задачи внутри класса NP.

Задача Z называется NP-полной, если выполняются два условия: во-первых, Z принадлежит NP; во-вторых, любая другая задача из NP сводится к задаче Z за полиномиальное время. Таким образом, NP-полные задачи образуют ядро класса NP, концентрируя в себе всю его сложность.

Если бы удалось построить полиномиальный алгоритм хотя бы для одной NP-полной задачи, это означало бы существование полиномиальных алгоритмов для всех задач NP. В таком случае выполнялось бы равенство $P = NP$, которое является одной из важнейших открытых проблем современной математики и теоретической информатики.

Примеры NP-полных задач хорошо известны. Наиболее фундаментальной из них является задача выполнимости булевых формул (SAT), сформулированная в 1971 г. Стивеном Куком. Классическими NP-полными задачами также являются задача о клике, задача коммивояжера в оптимизационной постановке и задача о рюкзаке.

NP-трудные задачи

Помимо задач класса NP выделяют еще более общий класс NP-трудных (NP-hard) задач.

К этому классу относятся задачи, к которым сводятся все NP-задачи, однако сами они не обязаны принадлежать NP. Это означает, что проверка предложенного решения NP-трудной задачи может оказаться не полиномиальной или даже не алгоритмически выполнимой.

Классическим примером NP-трудной задачи является проблема останова для машины Тьюринга. Она формулируется как вопрос: завершит ли данная программа свою работу на заданном входе или будет работать бесконечно. Доказано, что эта задача неразрешима, и поэтому она выходит за пределы NP, хотя при этом сводимость от NP-задач к ней сохраняется.

Значение классификации

Классы P, NP, NP-полные и NP-трудные образуют основу современной теории сложности вычислений. Они позволяют строго разграничить задачи, которые решаются эффективно, задачи, для которых пока не известны быстрые методы решения, и задачи, принципиально не поддающиеся алгоритмической обработке.

Эта классификация играет ключевую роль не только в теоретической информатике, но и в практических приложениях. В частности, многие задачи оптимизации в экономике, логистике и биоинформатике оказываются NP-полными. Это объясняет необходимость разработки приближенных и эвристических методов, обеспечивающих приемлемые решения там, где точное вычисление слишком затратно или невозможно (табл. 11.1).

Таблица 11.1

Сравнение классов сложности

Класс	*Определение*	*Примеры*	*Особенности*
P	Решаемы за полиномиальное время	Сортировка, поиск пути	Эффективные алгоритмы известны
NP	Решения проверяются за полиномиальное время	Рюкзак, раскраска графа	Эффективного алгоритма поиска неизвестно
NP-полные	«Самые трудные» в NP	SAT, коммивояжер	Решение хотя бы одной → P = NP
NP-трудные	Не обязательно в NP, но не проще NP-задач	Задача останова	Проверка решения может быть сложнее P

Визуализация

P ⊆ NP
└── NP-полные
└── NP-трудные (шире, чем NP)

Проблема P vs NP

Главный открытый вопрос: равны ли классы P и NP?

Если P = NP → все NP-задачи можно решать быстро, что приведет к революции в науке и технологиях (включая криптографию).

Если P ≠ NP → существует целый класс задач, для которых решения можно проверять быстро, но находить их принципиально трудно.

Этот вопрос включен в список семи «Задач тысячелетия» Клэя. За его решение обещан миллион долларов.

Практическое значение:

1. Криптография.

Алгоритмы RSA и ECC основаны на предположении, что некоторые задачи из NP (факторизация, дискретный логарифм) не имеют полиномиальных решений.

Если P = NP, криптография в ее современном виде станет небезопасной.

2. Оптимизация.

Задачи логистики, планирования производства, маршрутизации – часто NP-трудные.

Используются эвристики и приближенные алгоритмы (генетические алгоритмы, муравьиные колонии).

3. Искусственный интеллект.

Множество задач машинного обучения и распознавания образов связаны с NP-трудными задачами.

Итог

Класс P – задачи, решаемые эффективно.

Класс NP – задачи, для которых решение можно быстро проверить, но не всегда найти.

NP-полные задачи – наиболее трудные в NP; их исследование связано с вопросом P vs NP.

NP-трудные задачи шире, чем NP, и включают даже такие, для которых проверка решения не является быстрой.

Знание классов сложности необходимо не только для теоретической информатики, но и для практических областей: от криптографии до оптимизационных задач в логистике и промышленности.

Историческое происхождение NP-трудных задач

1. Начало: класс NP.

В 1960–1970-е гг. ученые (Кук, Левин, Карп) сформулировали понятие класса NP-задач, решения которых можно проверить за полиномиальное время. Вскоре появилась идея NP-полных задач: это те задачи в NP, к которым можно свести все остальные задачи из NP.

Пример:

1. SAT (задача выполнимости булевой формулы) – первая NP-полная задача, доказанная Стивеном Куком (1971).

2. Проблема расширения.

После введения NP-полных задач стало понятно, что есть и более «трудные» задачи: они не принадлежат NP, потому что для них решение нельзя проверить за полиномиальное время, но к ним можно свести задачи из NP.

То есть такие задачи не проще NP-задач, а значит, они находятся «за пределами NP».

3. Введение понятия NP-hard (NP-трудные).

Чтобы описать эти задачи, ученые ввели отдельный класс – NP-hard.

Определение: задача называется NP-трудной, если любая задача из NP сводится к ней за полиномиальное время.

Но при этом сама NP-трудная задача не обязана находиться в NP.

Она может быть даже неразрешимой.

Примеры происхождения

1. Задача коммивояжера (оптимизационная версия):

- формулировка: найти кратчайший маршрут, проходящий через все города;
- проверить оптимальность решения сложно (нужно убедиться, что нет лучшего маршрута);
- поэтому эта версия относится к NP-hard, а не к NP.

2. Задача останова машины Тьюринга.

- определить, завершится ли программа на заданном входе;
- это задача вообще неразрешимая, поэтому она даже не в NP, но к ней можно свести NP-задачи. Следовательно, она NP-трудная.

Визуализация

P ⊆ NP
└── NP-полные (NPC)
└── NP-трудные (шире, чем NP, включая неразрешимые задачи)

Итог

NP-трудные задачи возникли как обобщение NP-полных задач.

Ученые поняли, что существуют задачи не проще NP, но которые нельзя поместить в NP (например, их решения нельзя проверить за полиномиальное время).

Поэтому выделили новый класс – NP-hard.

11.2. Сложность по времени и памяти

Анализ алгоритмов в теоретической информатике не ограничивается фактом существования решения задачи. Для практического применения критически важно оценить ресурсы, которые требуются для выполнения алгоритма. Наиболее значимыми ресурсами считаются:

1. Время выполнения алгоритма – число шагов (операций), необходимых для решения задачи в зависимости от размера входа n.

2. Память – объем используемых ячеек памяти для входных данных, промежуточных вычислений и результата.

Обе характеристики тесно взаимосвязаны и составляют основу анализа сложности.

Модели вычислений

Для формализации понятия сложности используются абстрактные модели:

Машина Тьюринга. Сложность измеряется числом тактов работы машины.

RAM-модель (Random Access Machine). Современная модель: предполагается, что доступ к памяти занимает $O(1)$, а каждая простая арифметическая операция выполняется за константное время.

На практике именно RAM-модель ближе к реальным компьютерам, поэтому временную и пространственную сложность обычно анализируют в ее рамках.

Временная сложность

Определение: временная сложность алгоритма – это функция $T(n)$, показывающая, сколько элементарных операций требуется для обработки входа длиной n (табл. 11.2).

Асимптотические обозначения:

- $O(f(n))$: верхняя граница роста функции (худший случай);
- $\Omega(f(n))$: нижняя граница (лучший случай);
- $\Theta(f(n))$: точная асимптотическая оценка.

Пример:

- сортировка вставками: $O(n^2)$, $\Omega(n)$, $\Theta(n^2)$;
- бинарный поиск: $O(\log n)$, $\Omega(1)$, $\Theta(\log n)$.

Таблица 11.2

Временная сложность алгоритмов

Сложность	Рост функции	Пример алгоритма
$O(1)$	Константная	Доступ к элементу массива
$O(\log n)$	Логарифмическая	Бинарный поиск
$O(n)$	Линейная	Линейный поиск
$O(n \log n)$	Квазилинейная	Быстрая сортировка (в среднем)
$O(n^2)$	Квадратичная	Сортировка вставками
$O(2^n)$	Экспоненциальная	Перебор всех подмножеств
$O(n!)$	Факториальная	Задача коммивояжера (перебор)

Интерпретация: алгоритм с экспоненциальной сложностью становится непрактичным уже при $n \approx 50$, в то время как алгоритм с $O(n \log n)$ может работать и при миллионах элементов.

Пространственная сложность

Пространственная сложность алгоритма – функция $S(n)$, определяющая количество памяти, необходимое для работы алгоритма при входе длиной n.

Составляющие памяти:

1. Память для входных данных.

Объем напрямую зависит от размера входа.

2. Дополнительная память.

Используется для временных переменных, структур данных (стек, массивы).

3. Память стека вызовов.

Актуально для рекурсивных алгоритмов (например, быстрая сортировка требует $O(\log n)$ памяти для стека).

Примеры пространственной сложности:

Сортировка выбором: $O(1)$ дополнительной памяти.

Сортировка слиянием: $O(n)$ дополнительной памяти.

Рекурсивный поиск в глубину: $O(h)$, где h – глубина дерева.

Вывод: часто алгоритмы «in-place» (работающие на месте, без доп. памяти) выгодны, но могут проигрывать по времени.

Компромиссы «время-память»

Между временем и памятью существует фундаментальный компромисс.

Ускорение работы часто требует использования дополнительных структур данных.

Пример: поиск в хэш-таблице выполняется за $O(1)$, но требует дополнительной памяти.

Экономия памяти может привести к увеличению времени работы.

Пример. Вычисление чисел Фибоначчи:

- рекурсивный алгоритм: экспоненциальное время, малая память;
- динамическое программирование: линейное время, но $O(n)$ памяти.

Таблица 11.3

Примеры компромисса

Алгоритм	*Время*	*Память*	*Особенность*
Линейный поиск	$O(n)$	$O(1)$	Минимум памяти, медленно
Бинарный поиск	$O(\log n)$	$O(1)$	Быстро, но требует сортировки
Хэш-таблица	$O(1)$	$O(n)$	Максимальная скорость, больше памяти

Практическое значение:

1. Программирование и разработка ПО.

Алгоритм сортировки выбирается в зависимости от доступной памяти и ожидаемого объема данных.

2. Системное администрирование.

Администратор при планировании серверов учитывает, что индексация в БД ускоряет поиск (экономия времени), но требует дополнительного места (затраты памяти).

3. Криптография.

Стойкость алгоритмов измеряется не только временем атаки, но и требуемым объемом памяти.

Пример: атака по словарю требует O(n) памяти для хранения паролей; атака перебором требует экспоненциального времени, но минимальной памяти.

4. Большие данные (Big Data).

Для массивов в миллиарды записей эффективны только алгоритмы с временной сложностью $O(n \log n)$ или лучше, а также с пространственной сложностью не выше $O(n)$.

5. Визуализация роста функций (табл. 11.4).

Таблица 11.4

Рост сложности

n	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(2^n)$
10	~3	10	30	100	1024
100	~7	100	700	10 000	1,27e30
1000	~10	1000	10 000	1 000 000	$\approx 10^{301}$

Видно, что при $n = 1000$ экспоненциальные алгоритмы становятся абсолютно непрактичными, тогда как $O(n \log n)$ остается приемлемым.

Итог: временная сложность отражает скорость работы алгоритма, пространственная – объем требуемой памяти.

Для анализа используются асимптотические обозначения O, Ω и Θ.

Существует компромисс: ускорение часто требует дополнительных затрат памяти.

В реальных системах выбор алгоритма определяется сочетанием времени, памяти и специфики задачи.

11.3. Многоленточная машина Тьюринга

Исторический контекст

Классическая одноленточная машина Тьюринга (МТ), предложенная А. Тьюрингом в 1936 г., стала первой строгой моделью алгоритма. Однако она оказалась слишком «узкой» для практического анализа времени работы: многие алгоритмы, легко реализуемые на компьютере, на одноленточной машине требуют чрезмерного количества шагов.

В середине XX в. математики и кибернетики начали искать модели, более адекватные реальным вычислительным устройствам. Одним из таких расширений стала многоленточная машина Тьюринга (ММТ), где у машины есть несколько рабочих лент, что приближает ее к современным архитектурам с оперативной памятью, регистрами и внешними носителями.

Устройство многоленточной машины

ММТ состоит из следующих компонентов:

1. Множество лент ($k \geq 2$). Каждая лента бесконечна в обе стороны и разделена на ячейки, содержащие символы из рабочего алфавита Г.

2. Набор головок. У каждой ленты есть отдельная головка чтения/записи. На каждом шаге машина одновременно считывает символы под всеми головками.

3. Управляющее устройство.

Оно хранит текущее состояние $q \in Q$ и определяет:

- какие символы записать на каждую ленту;
- куда сдвинуть каждую головку (влево, вправо или оставить на месте);
- в какое состояние перейти.

Формальное определение

ММТ определяется как кортеж:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r),$$

где Q – конечное множество состояний;

Σ – входной алфавит;

Γ – рабочий алфавит, включающий Σ и пустой символ ␣;

δ – функция переходов:

$$\delta\colon \mathrm{Q} \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, \mathrm{S}\}^k,$$

где L – движение влево;

R – вправо;

S – остаться на месте;

q_0 – начальное состояние;

q_a – принимающее состояние (accepting);

q_r – отвергающее состояние (rejecting).

В отличие от классической машины Тьюринга, функция переходов учитывает k символов сразу (по одному с каждой ленты).

Виды многоленточных машин

Двуленточная МТ. Минимальное расширение, уже дающее существенный прирост эффективности.

Трехленточная и более. Позволяют хранить промежуточные данные, реализовывать параллельные процессы (например, чтение с одной ленты, обработку на второй, запись на третьей).

Примеры алгоритмов:

1. Копирование строки.

Одноленточная МТ: нужно многократно возвращаться в начало строки, сложность $\sim O(n^2)$.

Двуленточная МТ: достаточно пройти по строке один раз, одновременно копируя на вторую ленту. Сложность $O(n)$.

2. Проверка палиндрома.

Одноленточная МТ: удаление первой и последней буквы, постоянные возвраты к началу, сложность $O(n^2)$.

Двуленточная МТ: переписывание строки в обратном порядке на вторую ленту и сравнение, сложность $O(n)$.

3. Умножение чисел.

На одной ленте: требуется многократное перемещение и переписывание промежуточных результатов.

На трех лентах:

- первая – первый множитель;
- вторая – второй множитель;
- третья – результат.

Это существенно сокращает количество шагов.

Эквивалентность моделей

Теорема. Любую многоленточную машину Тьюринга можно смоделировать на одноленточной.

Идея доказательства:

Одноленточная МТ может хранить все k лент многоленточной, чередуя их содержимое на одной ленте и отделяя специальными разделителями.

Положение головок кодируется специальными метками.

Один шаг многоленточной МТ симулируется последовательностью шагов одноленточной.

Замедление симуляции: если многоленточная машина работает за $T(n)$ шагов, то эквивалентная одноленточная выполняет $O(T(n)^2)$ шагов.

Вывод: многоленточные машины не увеличивают вычислительной мощности, но позволяют формулировать алгоритмы более эффективно.

Связь со сложностью времени и памяти

1. Временная сложность на многоленточных машинах ближе к реалиям современных вычислений, где параллельно можно работать с несколькими областями памяти.

2. Пространственная сложность отражает количество используемых лент и объем информации на каждой.

3. Определение классов P и NP принято формулировать именно на многоленточных машинах, так как они дают корректное и устойчивое определение «полиномиального времени».

Аналогия с современными компьютерами

Ленты можно сопоставить с устройствами памяти (оперативная память, кэш, внешние носители).

Многоленточность отражает возможность параллельного доступа к разным сегментам данных.

Многопроцессорность и многопоточность в современных системах могут рассматриваться как дальнейшее развитие идей многоленточной модели.

Итог:

Многоленточная машина Тьюринга – это расширение классической модели, где имеется несколько рабочих лент и головок чтения/записи.

Она не расширяет множество вычислимых функций, но сокращает время работы многих алгоритмов с квадратичного до линейного.

ММТ является эталонной моделью в теории сложности, так как именно на ее основе определяются классы P и NP.

Ее значение состоит в том, что она позволяет приближенно моделировать реальные компьютеры и одновременно сохраняет строгую математическую основу.

Контрольные вопросы

1. Дайте определение классов P и NP. Чем они отличаются?
2. Что означает понятие NP-полной задачи? Приведите примеры.
3. В чем заключается проблема «P vs NP» и почему она является одной из ключевых в теоретической информатике?
4. Объясните, что такое NP-трудные задачи. Чем они отличаются от NP-полных?
5. Что понимается под временной сложностью алгоритма?
6. Что означает пространственная сложность алгоритма? Какие факторы на нее влияют?
7. Приведите примеры компромисса «время-память» в алгоритмах.
8. Опишите устройство многоленточной машины Тьюринга. Чем она отличается от классической одноленточной?
9. Почему многоленточная машина Тьюринга не расширяет множество разрешимых задач, но используется для определения классов сложности?
10. Приведите примеры задач, которые на одноленточной машине решаются за квадратичное время, а на многоленточной – за линейное.

12. КОМПЬЮТЕРНЫЕ СЕТИ И БЕЗОПАСНОСТЬ

12.1. Основные понятия безопасности

Информационная безопасность (ИБ) является одной из фундаментальных дисциплин современной информатики и программной инженерии. В условиях глобальной цифровизации, интеграции информационных технологий в экономику, государственное управление, оборону и повседневную жизнь вопросы защиты информации приобрели первостепенное значение. Нарушение безопасности компьютерных систем приводит не только к финансовым потерям, но и к рискам для национальной безопасности и жизни людей.

Цель дисциплины «Информационная безопасность» состоит в том, чтобы сформировать системное понимание угроз, принципов защиты и механизмов обеспечения сохранности информации в компьютерных сетях. Для этого вводится понятийный аппарат, который является отправной точкой всей последующей теории и практики.

Основные категории информационной безопасности

1. *Информация* – это сведения, зафиксированные на носителе или передаваемые по каналу связи, которые имеют смысл для пользователя. В контексте ИБ информация рассматривается как главный объект защиты.

2. *Информационные ресурсы* – совокупность данных, программного обеспечения, технических и организационных средств, без которых невозможно функционирование информационной системы.

3. *Угроза безопасности информации* – потенциальное событие или действие, которое может привести к нарушению целостности, конфиденциальности или доступности информации.

4. *Уязвимость* – свойство системы, создающее возможность для реализации угрозы. Это «слабое место», которым может воспользоваться злоумышленник.

5. *Атака* – конкретное реализованное действие по использованию уязвимости с целью нарушения безопасности.

6. *Риск* – мера, характеризующая вероятность реализации угрозы и масштаб возможных последствий. Риск формализует связь между угрозами и уязвимостями.

Базовые свойства информации: триада CIA

В международной практике информационной безопасности основой защиты является так называемая *триада CIA (Confidentiality, Integrity, Availability)*.

1. *Конфиденциальность (Confidentiality).*

Информация должна быть доступна только тем субъектам, которые обладают соответствующими правами.

Механизмы обеспечения: криптографическое шифрование, разграничение прав доступа, аутентификация пользователей.

Пример: медицинские карты пациентов могут просматривать только врачи, участвующие в лечении.

2. *Целостность (Integrity).*

Информация должна *сохранять* свою корректность и полноту, не подвергаться несанкционированным изменениям.

Механизмы: контрольные суммы, электронная подпись, системы журналирования изменений.

Пример: бухгалтерский отчет не должен быть изменен посторонним лицом без ведома ответственного сотрудника.

3. *Доступность (Availability).*

Информация и сервисы должны быть доступны авторизованным пользователям в нужное время.

Механизмы: резервирование, отказоустойчивые кластеры, защита от атак типа «отказ в обслуживании» (DoS).

Пример: интернет-банкинг обязан работать круглосуточно независимо от нагрузки.

Дополнительные свойства

Помимо базовых свойств, в современных стандартах (например, ISO/IEC 27001) рассматриваются:

Аутентичность – подтверждение подлинности источника информации.

Неотказуемость – невозможность отрицания факта действия (например, отправки сообщения).

Подотчетность – фиксация действий пользователей и систем для последующего анализа.

Уровни обеспечения безопасности

1. *Физический уровень.*

Контроль доступа в помещения, защита серверов, видеонаблюдение.

Пример: ограничение входа в серверную.

2. *Технический уровень.*

Сетевые экраны, криптографические протоколы, средства защиты каналов связи.

Пример: использование TLS для передачи данных.

3. *Программный уровень.*

Антивирусы, системы обнаружения вторжений, регулярные обновления программного обеспечения.

4. *Организационный уровень.*

Политики безопасности, инструктаж сотрудников, контроль прав доступа.

Классификация угроз

1. *По источнику:*

- внешние (атаки из Интернета, вредоносное ПО);
- внутренние (злоупотребление правами, ошибки персонала).

2. *По характеру воздействия:*

- пассивные (перехват трафика, сбор информации);
- активные (модификация данных, уничтожение файлов).

3. *По последствиям:*

- нарушение конфиденциальности (утечка данных);
- нарушение целостности (искажение базы данных);
- нарушение доступности (отказ в обслуживании).

Модели безопасности (табл. 12.1)

Таблица 12.1

Сравнение моделей безопасности

Модель	*Основное свойство*	*Ключевое правило*
Белла – Лападулы	Конфиденциальность	No read up, no write down
Биба	Целостность	No write up, no read down
Кларка – Уилсона	Корректность	Доступ только через сертифицированные программы

Для формализации защиты информации разработаны специальные модели.

1. *Модель Белла – Лападулы (Bell – LaPadula, 1973).*

Ориентирована на обеспечение *конфиденциальности*.

Основные правила:

- субъект не может читать данные более высокого уровня секретности («no read up»);
- субъект не может записывать данные на более низкий уровень (no write down).

2. *Модель Биба (Biba, 1977).*

Ориентирована на обеспечение *целостности*.

Основные правила:

- субъект не может изменять объекты более высокого уровня целостности («no write up»);
- субъект не может читать объекты более низкого уровня («no read down»).

3. *Модель Кларка – Уилсона (Clark – Wilson, 1987).*

Делает акцент на *корректности бизнес-процессов*.

Основной принцип: доступ к данным осуществляется только через сертифицированные программы, которые поддерживают их целостность.

Примеры атак в контексте триады CIA (табл. 12.2).

Таблица 12.2

Примеры атак по свойствам CIA

Свойство	*Определение*	*Примеры атак*
Конфиденциальность	Доступ к данным посторонних лиц	Фишинг, MITM, утечки баз
Целостность	Несанкционированное изменение данных	SQL-инъекции, вирусы, подмена транзакций
Доступность	Недоступность информации или сервиса	DDoS, ransomware, отказ оборудования

1. Нарушение конфиденциальности.

Определение: доступ к информации получает субъект, который не имеет на это права.

Примеры:

Фишинг – злоумышленник выманивает логины и пароли через поддельные письма или сайты.

Перехват трафика (sniffing). Атака «человек посередине» (MITM), когда данные (например, пароли в Wi-Fi) считываются при передаче.

Несанкционированный доступ к базе данных. Например, утечки персональных данных пользователей из-за слабого пароля администратора.

2. Нарушение целостности.

Определение: данные подвергаются изменению или подмене без разрешения.

Примеры:

SQL-инъекция. Вредоносный код внедряется в запрос, что позволяет изменять данные в базе (например, менять баланс в интернет-банке).

Вирусы и трояны. Вредоносные программы модифицируют системные файлы или документы.

Подмена транзакций. В ходе кибератаки хакер изменяет номер счета получателя в платежном поручении.

3. Нарушение доступности.

Определение: пользователи не могут получить доступ к информации или сервисам в нужный момент.

Примеры:

DDoS-атака (Distributed Denial of Service). Сотни тысяч зараженных компьютеров начинают одновременно обращаться к серверу, перегружая его.

Шифровальщики (ransomware). Вредоносное ПО шифрует все файлы на диске, делая их недоступными до выкупа.

Отказ оборудования. Сбой жесткого диска без резервного копирования приводит к потере доступа к данным.

Основные понятия безопасности – угроза, уязвимость, атака, риск, а также базовые свойства информации (конфиденциальность, целостность, доступность) – составляют основу дисциплины информационной безопасности. Понимание этих принципов позволяет правильно строить архитектуру систем, разрабатывать защитные механизмы и оценивать уровень рисков.

Формальные модели (Белла – Лападулы, Биба, Кларка – Уилсона) обеспечивают строгую теоретическую базу, которая используется в реальных политиках безопасности операционных систем, сетевых средств защиты и корпоративных приложений.

12.2. Системный подход к защите

Развитие информационных технологий и глобализация сетевой инфраструктуры в XX–XXI вв. привели к тому, что безопасность компьютерных систем стала рассматриваться не как вспомогательная функция, а как неотъемлемая часть проектирования, эксплуатации и управления.

В первые годы компьютерной эры меры защиты имели локальный и изолированный характер: пароли для пользователей, физическая охрана оборудования, отдельные криптографические методы. Однако по мере усложнения систем и появления распределенных сетей стало очевидно, что разрозненные меры не способны обеспечить достаточный уровень защищенности.

Так возникла концепция *системного подхода к защите информации* – подхода, при котором безопасность интегрируется во все уровни информационной системы, обеспечивая ее целостное функционирование даже в условиях воздействия угроз.

Сущность системного подхода

Системный подход предполагает:

1. *Рассмотрение безопасности как свойства всей системы.*

Даже если отдельные компоненты защищены, отсутствие координации между ними приводит к уязвимостям.

2. *Интеграцию технических, программных, организационных и правовых мер.*

Только их совокупность позволяет снизить риск реализации угроз.

3. *Учет жизненного цикла информационной системы.*

Защита должна сопровождать систему на всех этапах: от анализа требований и проектирования до вывода из эксплуатации.

4. *Согласование с целями организации.*

Меры безопасности не должны мешать бизнес-процессам, а должны быть встроены в них.

Принципы системного подхода

1. *Комплексность.*

Защита должна охватывать все уровни системы: физический, сетевой, программный, организационный.

2. *Многоуровневость (defense in depth).*

Создаются несколько последовательных рубежей защиты, которые повышают устойчивость к атакам.

3. *Принцип минимальных привилегий.*

Пользователи и процессы получают только минимально необходимые права.

4. *Разделение полномочий.*

Функции администрирования и контроля должны распределяться между несколькими лицами, чтобы снизить риск злоупотреблений.

5. *Адаптивность.*

Система защиты должна развиваться вместе с изменением угроз и технологий.

6. *Непрерывность.*

Информационная безопасность – это постоянный процесс, включающий мониторинг, аудит, обновление средств защиты.

7. *Пропорциональность затрат.*

Уровень защиты должен соответствовать ценности защищаемых ресурсов и масштабу рисков.

Структура уровней защиты

Архитектура системного подхода к защите

Уровень 5: Управление и аудит.

Уровень 4: Организационные меры.

Уровень 3: Программная защита.

Уровень 2: Технические средства (сети, оборудование).

Уровень 1: Физическая безопасность.

Уровень 1. Физическая безопасность:

- контроль доступа в помещения (замки, электронные карты, биометрия);

- системы пожаротушения, климат-контроль, бесперебойное питание;
- защита от стихийных бедствий и аварий.

Уровень 2. Технические средства:

- межсетевые экраны, маршрутизаторы с фильтрацией пакетов;
- VPN, криптографическая защита каналов связи;
- IDS/IPS (системы обнаружения и предотвращения вторжений).

Уровень 3. Программная защита:

- антивирусные системы, средства защиты от спама;
- обновления и патчи для ОС и приложений;
- безопасное программирование (устойчивость к SQL-инъекциям, XSS).

Уровень 4. Организационные меры:

- политики безопасности (регламенты по паролям, доступу к ресурсам);
- обучение персонала и проведение тренировок по реагированию на инциденты;
- система распределения полномочий и ответственность сотрудников.

Уровень 5. Управление и аудит:

- внедрение стандартов ISO/IEC 27001, NIST CSF, отечественных ГОСТов;
- постоянный мониторинг (SIEM-системы);
- анализ рисков и регулярные проверки на соответствие требованиям.

Жизненный цикл информационной системы и безопасность

1. *Проектирование:* закладывание требований безопасности, архитектурные решения (например, сегментация сети).
2. *Разработка:* применение практик secure coding, статический анализ исходного кода.
3. *Внедрение:* тестирование защиты, настройка аутентификации и авторизации.
4. *Эксплуатация:* обновления, мониторинг, реагирование на инциденты.
5. *Вывод из эксплуатации:* безопасное удаление данных и утилизация носителей.

Модели и стандарты

- *Белла–Лападулы (Bell– LaPadula):* ориентирована на сохранение конфиденциальности;
- *Биба (Biba):* ориентирована на сохранение целостности;
- *Кларка–Уилсона (Clark–Wilson):* ориентирована на корректность бизнес-процессов;

- *ISO/IEC 27001:* международный стандарт управления ИБ;
- *NIST Cybersecurity Framework:* американская методология построения защиты;
- *ГОСТ Р ИСО/МЭК 27002–2021:* отечественный стандарт по мерам ИБ.

Типовые ошибки при отсутствии системного подхода

1. Полная ориентация на технические средства (например, установка только антивируса при отсутствии политики безопасности).
2. Игнорирование человеческого фактора (отсутствие обучения сотрудников → рост фишинговых атак).
3. Недооценка организационных мер (отсутствие разграничения прав доступа).
4. Отсутствие резервного копирования и планов восстановления (в случае атак ransomware приводит к полной потере данных).
5. Ориентация на прошлые угрозы при игнорировании новых (например, защита от вирусов без учета современных DDoS-атак).

Пример

Банковская система:

Физический уровень: охрана серверных, контроль доступа по биометрии.

Технический уровень: TLS-шифрование транзакций, межсетевые экраны.

Программный уровень: двухфакторная аутентификация, защита от SQL-инъекций.

Организационный уровень: обучение сотрудников распознаванию фишинга.

Управленческий уровень: аудит соответствия требованиям ЦБ РФ и ISO/IEC 27001.

Системный подход к защите информации – это комплексная, многоуровневая и непрерывная деятельность, интегрированная во все процессы организации. Он предполагает сочетание технических, программных, организационных и управленческих мер, основанных на международных и национальных стандартах.

Только такой подход позволяет достичь баланса между уровнем защиты, затратами и эффективностью, обеспечивая устойчивость информационной системы в условиях постоянно меняющихся угроз.

12.3. Базовые технологии безопасности

Базовые технологии информационной безопасности (ИБ) – это совокупность методов, средств и процессов, направленных на обеспечение конфиденциальности, целостности и доступности информации. Их можно рассматривать как «строительные блоки» любой системы защиты: независимо от масштаба организации (малый бизнес, государственная структура, международная корпорация), именно они формируют фундамент защиты данных и сервисов.

В отличие от частных решений (например, конкретного антивируса или межсетевого экрана), базовые технологии безопасности представляют собой *универсальные подходы*, которые сохраняют актуальность при изменении программных платформ, сетевой архитектуры и даже типов угроз.

Идентификация, аутентификация и авторизация

1. *Идентификация* – процесс, при котором субъект (пользователь, устройство или процесс) заявляет свою «личность» системе.

Пример: ввод логина.

2. *Аутентификация* – проверка подлинности заявленной личности. Система сравнивает предоставленные данные с эталоном.

Методы аутентификации:

- по знанию (пароль, PIN-код);
- по обладанию (карта, токен, смартфон);
- по биометрии (отпечатки пальцев, лицо, радужка глаза).

Современный стандарт – *многофакторная аутентификация (MFA)*, которая комбинирует два и более метода.

3. *Авторизация* – предоставление пользователю определенных прав после успешной аутентификации.

Пример: вход в корпоративную сеть: сотрудник вводит пароль, подтверждает вход через SMS-код и получает доступ только к тем ресурсам, которые соответствуют его роли.

Управление доступом

Это система правил и процедур, регулирующих, кто и что может делать в информационной системе.

1. Мандатное управление доступом (MAC).

Используется в системах с повышенными требованиями к безопасности (например, в военных структурах). Доступ определяется политикой безопасности, зависящей от уровня секретности данных.

2. Дискреционное управление доступом (DAC).

Контроль за доступом принадлежит владельцу ресурса.

Пример: владелец файла сам назначает права «чтение/запись».

3. Ролевое управление доступом (RBAC).

Доступ определяется ролью пользователя в организации (например, «администратор», «кассир», «аналитик»).

Пример: в больнице врач может читать и редактировать медицинские карты пациентов, но не имеет доступа к финансовым данным учреждения.

Криптографические технологии

Криптография является одной из ключевых технологий защиты информации. Она применяется для обеспечения конфиденциальности, целостности и аутентичности.

Основные направления:

1. Шифрование данных:

- симметричное (AES, DES, ГОСТ 28147–89): один ключ используется и для шифрования, и для дешифрования;
- асимметричное (RSA, ElGamal, ECC): используются два ключа – открытый и закрытый.

2. Хэширование.

Невозвратное преобразование данных фиксированной длины (MD5, SHA-1, SHA-256). Используется для проверки целостности и хранения паролей.

3. Электронная цифровая подпись (ЭЦП).

Обеспечивает аутентификацию отправителя и защиту от подделки документов.

4. Протоколы криптографической защиты.

TLS/SSL – для защиты веб-трафика, IPSec – для шифрования каналов VPN.

Пример: при оплате покупки в интернет-магазине данные карты шифруются протоколом TLS, а транзакция подтверждается электронной подписью банка.

Антивирусные и анти-мальварные технологии

Сигнатурный анализ. Определение вредоносного кода по базе известных «отпечатков».

Эвристический анализ. Определение подозрительного поведения программ.

Поведенческий анализ. Запуск приложений в «песочнице» (sandbox) для выявления вредоносной активности.

Пример: вредоносная программа пытается шифровать файлы на компьютере. Антивирус обнаруживает подозрительное массовое изменение файлов и блокирует процесс.

Сетевые технологии защиты

Межсетевые экраны (firewalls). Фильтрация пакетов на основе правил доступа.

VPN (Virtual Private Network). Создание защищенных туннелей поверх публичных сетей.

IDS/IPS (Intrusion Detection/Prevention Systems). Обнаружение и предотвращение сетевых атак.

Сегментация сети. Разделение сети на зоны (например, DMZ для публичных серверов).

Пример: корпоративная сеть банка разделена на сегменты: клиентский портал в DMZ, база данных – во внутренней сети, доступ только по VPN.

Резервное копирование и восстановление

Резервное копирование – технология защиты данных от потери из-за сбоев или атак.

Полное копирование: создается полный дубликат данных.

Инкрементальное копирование: копируются только изменения с последнего бэкапа.

Дифференциальное копирование: копируются изменения с момента последнего полного бэкапа.

Пример: после атаки программы-шифровальщика предприятие восстанавливает рабочую базу данных из резервной копии, хранящейся на внешнем сервере.

Мониторинг и аудит

Журналирование событий. Ведение логов о действиях пользователей и системных процессах.

SIEM-системы (Security Information and Event Management). Автоматический сбор, корреляция и анализ логов.

Реагирование на инциденты. Формирование планов IRP (Incident Response Plan).

Пример: SIEM-система обнаруживает аномальный рост запросов с одного IP-адреса и автоматически блокирует его как источник атаки.

Управление уязвимостями и обновлениями

Регулярные обновления операционных систем и приложений.

Использование систем сканирования уязвимостей (Nessus, OpenVAS).

Применение практик безопасной разработки (secure coding).

Пример: предотвращение атаки WannaCry стало возможным благодаря своевременному обновлению ОС Windows, где была закрыта уязвимость EternalBlue.

Взаимосвязь технологий и триады безопасности представлены в таблице ниже (табл. 12.3)

Таблица 12.3

Связь базовых технологий с триадой CIA

Технология	*Конфиденциальность*	*Целостность*	*Доступность*
Аутентификация и доступ	+	+	–
Криптография	+	+	–
Антивирусные средства	–	+	+
Сетевые технологии (VPN, IDS)	+	+	+
Резервное копирование	–	+	+
Мониторинг и аудит	+	+	–
Управление уязвимостями	+	+	+

Базовые технологии безопасности представляют собой ядро информационной защиты. Они реализуют функции контроля доступа, защиты данных при передаче и хранении, выявления и предотвращения атак, а также восстановления после инцидентов.

Эти технологии тесно интегрированы между собой: криптография неэффективна без управления ключами, антивирус не справится без обновлений, а межсетевой экран бесполезен без политики безопасности.

Современные стандарты ИБ (ISO/IEC 27001, ГОСТ Р ИСО/МЭК 27002–2021, NIST Cybersecurity Framework) прямо указывают, что именно базовые технологии являются минимально необходимым уровнем защиты, на котором строятся все более сложные архитектуры информационной безопасности.

Примеры атак и технологии защиты (табл. 12.4)

Таблица 12.4

Атаки и меры защиты

Атака	*Нарушаемое свойство*	*Технологии защиты*
MITM (перехват трафика)	Конфиденциальность	VPN, TLS, MFA
SQL-инъекция	Целостность	Валидация ввода, WAF, обновления
DDoS-атака	Доступность	Балансировка, фильтрация трафика, IDS/IPS
Ransomware	Доступность, целостность, конфиденциальность	Антивирусы, патчи, резервное копирование
Фишинг	Конфиденциальность, аутентичность	Антиспам, MFA, обучение пользователей

1. Нарушение конфиденциальности: атака «человек посередине» (MITM).

Сценарий: пользователь подключается к публичной Wi-Fi сети в кафе. Злоумышленник перехватывает весь трафик и получает логины, пароли, номера карт.

Нарушается свойство: конфиденциальность.

Технологии защиты:

- VPN (Virtual Private Network) – весь трафик шифруется внутри «туннеля»;
- TLS (HTTPS) – защита веб-сессий;
- двухфакторная аутентификация – даже при краже пароля доступ будет ограничен.

2. Нарушение целостности: SQL-инъекция.

Сценарий: веб-приложение интернет-магазина не фильтрует пользовательский ввод. Хакер вводит в поле «логин» фрагмент SQL-кода и получает возможность изменять данные в базе (например, менять цены или баланс пользователей).

Нарушается свойство: целостность данных.

Технологии защиты:

- контроль ввода данных (input validation);
- использование параметризованных запросов (prepared statements);
- регулярные обновления ПО и систем управления базами данных;
- WAF (Web Application Firewall), который фильтрует вредоносные запросы.

3. Нарушение доступности: DDoS-атака (Distributed Denial of Service).

Сценарий: на сайт банка обрушивается поток запросов от десятков тысяч зараженных компьютеров (ботнет). Сервер перегружается и перестает отвечать на легитимные запросы клиентов.

Нарушается свойство: доступность.

Технологии защиты:

- балансировка нагрузки (load balancing);
- фильтрация трафика на уровне провайдера;
- IDS/IPS для распознавания аномальных запросов;
- распределенные системы защиты от DDoS (Cloudflare, Arbor).

4. Атака с нарушением сразу нескольких свойств: вредоносное ПО (ransomware).

Сценарий: программа-шифровальщик (например, WannaCry) проникает в систему и шифрует все файлы. Пользователь теряет доступ к данным, злоумышленники требуют выкуп.

Нарушаются свойства:

- доступность (данные заблокированы);
- целостность (данные искажены шифрованием);
- конфиденциальность (при утечке ключей).

Технологии защиты:

- антивирусные и анти-мальварные средства;
- регулярные обновления ОС и приложений (патчи);
- резервное копирование (offline-backup);
- ограничение прав пользователей (минимальные привилегии).

5. Нарушение аутентичности: фишинг.

Сценарий: пользователь получает письмо «от банка» с просьбой ввести данные карты. Введенные сведения попадают злоумышленнику.

Нарушается свойство: конфиденциальность (утечка данных) и аутентичность.

Технологии защиты:

- антиспам-фильтры и системы анализа писем;
- двухфакторная аутентификация;
- обучение пользователей основам кибергигиены.

Рассмотренные примеры показывают, что *каждая категория атак напрямую связана с определенным свойством безопасности*. Соответственно, именно базовые технологии защиты (VPN, криптография, WAF, IDS, резервное копирование, антивирусы) формируют первый рубеж обороны.

12.4. Алгоритмы шифрования и защита информации

Шифрование является одной из древнейших и одновременно наиболее динамично развивающихся технологий защиты информации. Его главная задача – сделать данные недоступными для неавторизованных лиц, сохраняя при этом возможность их восстановления для законного получателя.

В эпоху цифровой трансформации шифрование применяется во всех сферах: от защиты военных коммуникаций и банковских транзакций до бытового обмена сообщениями в мессенджерах. На практике оно решает четыре ключевые задачи информационной безопасности:

1. *Конфиденциальность* – предотвращение раскрытия данных третьим лицам.

2. *Целостность* – обеспечение неизменности информации.

3. *Аутентичность* – подтверждение подлинности источника.

4. *Неотказуемость* – невозможность отрицания факта совершенного действия.

Исторический обзор криптографии

1. *Античность.* Шифр Цезаря – простой метод замены букв, использовавшийся для военных посланий.

2. *Средневековье.* Полиалфавитные шифры Виженера, первые криптоанализаторы в Арабском халифате.

3. *XIX в.* Механические шифраторы (диск Ж. Ж. Базери, машина Дж. Джефферсона).

4. *XX в.* Электромеханические машины (Энигма в Германии, «Сигаба» в США). Криптография становится ключевым фактором военных операций.

5. *1970-е гг.* Появление стандарта DES (Data Encryption Standard). Начало массового использования криптографии в гражданских целях.

6. *1976 г.* Открытие асимметричной криптографии (Уитфилд Диффи и Мартин Хеллман).

7. *1990-е гг.* Распространение Интернета и протоколов SSL/TLS.

8. *XXI в.* Переход на AES, внедрение асимметричных алгоритмов RSA и ECC. Появление угрозы квантовых компьютеров.

Симметричные алгоритмы

Принцип работы: один и тот же ключ используется как для шифрования, так и для расшифрования.

1. *Блочные шифры.*

Преобразуют данные блоками фиксированного размера.

1.1. DES (56-битный ключ, устарел из-за уязвимости к перебору).

1.2. AES (128/192/256 бит, современный международный стандарт).

1.3. ГОСТ 28147–89 (российский стандарт).

2. *Потоковые шифры.*

Генерируют псевдослучайный поток битов, который складывается с открытым текстом.

2.1. RC4 (ранее широко использовался, но признан небезопасным).

2.2. Salsa20/ChaCha20 (современные быстрые алгоритмы).

Проблема симметричных алгоритмов: безопасный обмен ключами.

Асимметричные алгоритмы

Принцип работы: используются два ключа – открытый и закрытый.

1. *RSA.*

Основан на сложности факторизации больших чисел. Используется для шифрования, ЭЦП и обмена ключами.

2. *Диффи–Хеллман.*

Позволяет двум сторонам выработать общий секретный ключ по открытому каналу.

3. *ECC (эллиптические кривые).*

Обеспечивает ту же безопасность, что RSA, но при меньших размерах ключей (эффективен для мобильных устройств).

Асимметричные алгоритмы решают проблему обмена ключами, но работают медленнее симметричных. Поэтому на практике используется комбинация: ключ передается асимметрично, а сами данные шифруются симметрично.

Режимы шифрования

Блочные шифры применяются в различных режимах:

1. *ECB (Electronic Codebook).* Простой, но небезопасный – одинаковые блоки дают одинаковый результат.

2. *CBC (Cipher Block Chaining).* Каждый блок зависит от предыдущего, что повышает стойкость.

3. *CFB и OFB.* Режимы потокового шифрования.

4. *CTR (Counter Mode).* Использует счетчик, обеспечивает высокую скорость.

Пример: AES в режиме CTR широко применяется в VPN и мессенджерах.

Хеш-функции

Определение: односторонние преобразования данных в «отпечаток» фиксированной длины.

1. *Свойства:* устойчивость к коллизиям, необратимость, равномерное распределение.

2. *Примеры:*

- устаревшие: MD5, SHA-1 (слабые, могут быть взломаны);
- современные: SHA-2 (SHA-256, SHA-512), SHA-3 (на основе алгоритма Keccak).

Применение: проверка целостности файлов, хранение паролей (с солью), цифровые подписи.

Электронная цифровая подпись (ЭЦП) и PKI

ЭЦП обеспечивает аутентификацию и неотказуемость.

1. *Процесс:*

1.1. Вычисляется хэш от сообщения.

1.2. Хэш шифруется закрытым ключом отправителя.

1.3. Получатель расшифровывает его открытым ключом и сравнивает с вычисленным.

2. *Инфраструктура открытых ключей (PKI).*

Использует сертификаты X.509, удостоверяющие личность владельца ключа. Сертификаты выпускают центры сертификации (CA).

Пример: HTTPS использует PKI для проверки подлинности сайтов.

Реальные атаки на криптосистемы

1. *Атаки перебором (brute-force).* Решение: увеличение длины ключей.

2. *Криптоанализ на основе известных текстов.* Решение: использование стойких режимов шифрования.

3. *Атаки на хеш-функции (коллизии). Пример*: взлом SHA-1. Решение: переход на SHA-256/3.

4. *Атаки на протоколы.* MITM-атаки при использовании SSL без проверки сертификата. Решение: строгая проверка цепочки доверия.

5. *Социальная инженерия.* Кража паролей или ключей у пользователей. Решение: MFA, аппаратные токены.

Современные вызовы криптографии

1. *Рост вычислительных мощностей.* Алгоритмы с короткими ключами становятся небезопасными.

2. *Квантовые компьютеры.* Алгоритм Шора может взломать RSA и ECC.

3. *Постквантовая криптография.* Разрабатываются новые алгоритмы (на основе решеток, хеш-функций, кодов).

4. *Баланс между безопасностью и производительностью.* Особенно актуален для мобильных устройств и IoT.

Примеры применения

1. *Банковская сфера:* защита транзакций с помощью TLS и цифровых подписей.

2. *Государственные системы:* электронные паспорта с криптографической защитой.

3. *Облачные сервисы:* хранение данных с применением AES-256.

4. *Мессенджеры:* протокол Signal (комбинация RSA, AES и Diffie – Hellman).

Алгоритмы шифрования являются краеугольным камнем информационной безопасности. Они реализуют базовые свойства защиты – конфиденциальность, целостность, аутентичность и неотказуемость.

Сочетание симметричных и асимметричных методов, применение хеш-функций и инфраструктуры открытых ключей позволяет строить современные защищенные системы. Однако развитие вычислительных технологий, особенно в области квантовых компьютеров, требует постоянного совершенствования криптографических алгоритмов и перехода к постквантовым решениям.

Контрольные вопросы

1. Что понимается под информационной безопасностью и какие ее основные свойства составляют так называемую триаду CIA?

2. Дайте определения понятий «угроза», «уязвимость», «риск» и «атака». Как эти понятия взаимосвязаны?

3. В чем заключается системный подход к защите информации? Перечислите его ключевые принципы.

4. Какие уровни защиты выделяются в системном подходе (физический, технический, программный, организационный, административный) и какие меры относятся к каждому из них?

5. Что такое аутентификация, идентификация и авторизация? Приведите примеры методов аутентификации.

6. Перечислите основные криптографические технологии и объясните, как они обеспечивают конфиденциальность, целостность и аутентичность информации.

7. Чем отличаются симметричные и асимметричные алгоритмы шифрования? Укажите их преимущества и недостатки.

8. Объясните роль хеш-функций и цифровой подписи в системах защиты информации. Приведите примеры их практического применения.

9. Какие реальные атаки угрожают криптосистемам (например, brute-force, атаки на хэши, MITM)? Какие меры защиты применяются против них?

10. Какие современные вызовы стоят перед криптографией (увеличение вычислительных мощностей, квантовые компьютеры, развитие постквантовых алгоритмов)?

СПИСОК РЕКОМЕНДУЕМЫХ ИСТОЧНИКОВ

1. *Кормен, Т. Х.* Алгоритмы. Построение и анализ / Т. Х. Кормен, Ч. Е. Лейзерсон, Р. Л. Ривест, К. Штайн ; пер. с англ. И. В. Красикова. – 3-е изд. – Москва : Вильямс, 2013. – 1328 с.

2. *Ахо, А. В.* Компиляторы: принципы, технология и инструменты / А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман ; пер. с англ. А. М. Шень. – 2-е изд. – Санкт-Петербург : Питер, 2008. – 784 с.

3. *Таненбаум, А. С.* Современные операционные системы : пер. с англ. / А. С. Таненбаум, Х. Бос. – 4-е изд. – Санкт-Петербург : Питер, 2015. – 1120 с.

4. *Семенов, Ю. А.* Компьютерные сети и телекоммуникации : учебное пособие / Ю. А. Семенов. – 2-е изд., испр. и доп. – Москва : Бином, 2014. – 448 с.

5. *Мейер, Б.* Объектно-ориентированное конструирование программного обеспечения / Б. Мейер ; пер. с англ. С. В. Назаренко. – 2-е изд. – Москва : Русская редакция, 2005. – 608 с.

6. *Шалыто, А. А.* Технология программирования и компьютерная грамотность / А. А. Шалыто. – Санкт-Петербург : Наука и Техника, 2010. – 560 с.

7. *Рассел, С.* Искусственный интеллект: современный подход : пер. с англ. / С. Рассел, П. Норвиг. – 3-е изд. – Москва : Вильямс, 2015. – 1408 с.

8. *Льюис, Г. Р.* Основания информатики : пер. с англ. / Г. Р. Льюис, Х. Р. Бернштейн. – Москва : Мир, 1988. – 504 с.

9. *Марков, А. А.* Теория алгорифмов / А. А. Марков. – Москва : Наука, 1954. – 376 с.

10. *Роджерс, Х.* Теория рекурсивных функций и эффективная вычислимость / Х. Роджерс ; пер. с англ. М. И. Канского. – Москва : Мир, 1972. – 616 с.

11. *Манин, Ю. И.* Введение в современную логику / Ю. И. Манин. – Москва : МЦНМО, 2010. – 144 с.

12. *Мальцев, А. И.* Алгоритмы и рекурсивные функции / А. И. Мальцев. – 2-е изд., испр. – Москва : Наука, 1986. – 368 с.

13. *Абрамов, С. А.* Элементы программирования / С. А. Абрамов. – 3-е изд. – Москва : МЦНМО, 2007. – 320 с.

14. *Себеста, Р. У.* Концепции языков программирования / Р. У. Себеста ; пер. с англ. С. В. Сокирко. – 9-е изд. – Москва : Вильямс, 2015. – 848 с.

15. *Гоулд, Г.* Дизайн языков программирования : пер. с англ. / Г. Гоулд. – Москва : Вильямс, 2014. – 640 с.

16. *Плотницкий, Ю. М.* Формальные языки и трансляция / Ю. М. Плотницкий. – Санкт-Петербург : СПбГУ, 2006. – 304 с.

17. *Шнайер, Б.* Прикладная криптография. Протоколы, алгоритмы и исходные тексты на языке Си : пер. с англ. / Б. Шнайер. – 2-е изд. – Москва : Триумф, 2002. – 816 с.

18. *Кобец, П. В.* Криптографические алгоритмы и протоколы передачи данных : учебное пособие / П. В. Кобец, В. В. Быков. – Санкт-Петербург : ИТМО, 2012. – 148 с.

19. *Панасенко, С. П.* Алгоритмы шифрования : специальный справочник / С. П. Панасенко. – Санкт-Петербург : БХВ-Петербург, 2009. – 576 с.

20. *Гордеев, В. А.* Организация безопасности информационных систем : учебное пособие / В. А. Гордеев, А. С. Саприкин ; СПбГУТ. – Санкт-Петербург, 2015. – 182 с.

21. *Бут, Г.* Визуальное моделирование с UML : пер. с англ. / Г. Бут, Дж. Рамбо, И. Якобсон. – 2-е изд. – Москва : БИНОМ, 2004. – 432 с.

22. *Гамма, Э.* Приемы объектно-ориентированного проектирования. Паттерны проектирования : пер. с англ. / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – Санкт-Петербург : Питер, 2007. – 366 с.

23. *Лидтке, Э.* Фундаментальные принципы эффективной разработки программного обеспечения : учебное пособие : пер. с англ. / Э. Лидтке. – Москва : Вильямс, 2004. – 464 с.

24. *Куратовский, К.* Теория множеств : пер. с англ./ К. Куратовский, А. Мостовский. – Москва : Мир, 1970. – 416 с.

25. *Колмогоров, А. Н.* Математическая логика / А. Н. Колмогоров, А. Г. Драгалин. – 2-е изд. – Москва : URSS, 2006. – 240 с.

26. *Успенский, В. А.* Что такое аксиоматический метод? / В. А. Успенский, В. В. Успенский. – Москва : МЦНМО, 2009. – 56 с.

27. ISO/IEC 12207:2017 Systems and software engineering – Software life cycle processes. – Женева : International Organization for Standardization, 2017.

28. ISO/IEC 25010:2011 Systems and software engineering – System and software quality models. – Женева : International Organization for Standardization, 2011.

29. *Бежанова, М. В.* Программная инженерия : курс лекций / М. В. Бежанова ; СПбГУТ. – Санкт-Петербург, 2016. – 196 с.

Бирюков Михаил Александрович
Ермакова Татьяна Вячеславовна
Попов Андрей Иванович

АЛГОРИТМИЧЕСКИЕ ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

Учебное пособие

Редакторы: *В. В. Повержук, Е. А. Матюшкина*
Компьютерная верстка *С. Н. Скляровой*

План издания 2026 г., п. 32

Подписано к печати 22.05.2026
Объем 11,75 уч.-изд. л. Тираж 25 экз. Заказ 1653
Редакционно-издательский отдел СПбГУТ
193232 СПб., пр. Большевиков, 22
Отпечатано в СПбГУТ