Программирование как процесс процедурного решения задач. Понятие «алгоритм». Процесс проектирования и анализа алгоритмов

Программирование — это процесс разработки программ, представляющих собой формальное описание решения задачи в виде последовательности элементарных действий, выполняемых вычислительной системой. В рамках **процедурного подхода** решение задачи осуществляется путём пошагового выполнения команд, изменяющих состояние программы и данных.

Алгоритм — это точное, конечное и формализованное предписание, задающее порядок выполнения элементарных операций, обеспечивающее получение результата решения задачи за конечное число шагов.

Алгоритм обладает следующими свойствами:

1. **Дискретность** — алгоритм состоит из отдельных шагов.
2. **Определённость** — каждый шаг однозначно задан.
3. **Конечность** — выполнение алгоритма завершается за конечное число шагов.
4. **Результативность** — алгоритм приводит к получению результата.
5. **Массовость** — алгоритм применим к целому классу входных данных.

Алгоритм является абстрактной моделью решения задачи и не зависит от языка программирования и аппаратной реализации.

Программирование как процедурный процесс

В процедурном программировании программа рассматривается как последовательность команд, выполняемых в строго определённом порядке. Управление осуществляется с помощью базовых управляющих конструкций:

- последовательности,
- ветвления,
- циклов.

Программа представляет собой реализацию алгоритма на конкретном языке программирования.

Проектирование алгоритмов

Проектирование алгоритмов — это процесс разработки формального решения задачи до этапа программной реализации.

Основные этапы проектирования:

1. **Формальная постановка задачи** — определение входных и выходных данных.
2. **Выбор метода разработки алгоритма** (грубая сила, декомпозиция, уменьшение размера задачи, динамическое программирование, жадные методы, метод преобразования).
3. **Построение алгоритма** в виде словесного описания, псевдокода или блок-схемы.

4. **Проверка корректности алгоритма**.
5. **Выбор структур данных**, обеспечивающих эффективную реализацию.

Анализ алгоритмов

Анализ алгоритмов направлен на оценку эффективности решения задачи и проводится независимо от конкретной реализации.

Различают:

- **временную сложность** — количество элементарных операций,
- **пространственную сложность** — объём используемой памяти.

Для оценки эффективности применяется **асимптотический анализ**, основанный на обозначениях O, Ω и Θ. Анализ проводится для лучшего, среднего и худшего случаев выполнения алгоритма.

Заключение: таким образом, программирование как процесс процедурного решения задач основывается на понятии алгоритма. Проектирование и анализ алгоритмов позволяют получить корректное и эффективное решение задачи ещё до этапа программной реализации, а программа является конкретной реализацией алгоритма на языке программирования.

Понятие «язык программирования». Понятие «среда разработки». Структура программы на языке C++

Язык программирования — это формальная знаковая система, предназначенная для записи алгоритмов и структур данных в виде программ, выполняемых вычислительной системой. Язык программирования задаёт набор **лексических, синтаксических и семантических правил**, по которым формируется корректная программа.

Основные характеристики языка программирования:

- **алфавит языка** (ключевые слова, идентификаторы, символы);
- **синтаксис** — правила построения корректных конструкций;
- **семантика** — смысл выполняемых операций;
- **средства управления потоком выполнения**;
- **система типов данных**.

Язык программирования обеспечивает связь между абстрактным алгоритмом и его машинным исполнением.

Среда разработки — это программный комплекс, предназначенный для создания, редактирования, компиляции, отладки и тестирования программ.

Среда разработки, как правило, включает:

- редактор исходного кода;
- компилятор или интерпретатор;
- средства сборки программы;
- отладчик;
- средства запуска и анализа выполнения программы.

Среда разработки облегчает процесс программирования, но не влияет на сам алгоритм, который реализуется в программе.

Программа на языке C++ имеет строго определённую структуру.

Основные элементы структуры программы:

1. **Директивы препроцессора**

   Используются для подключения библиотек и определения макросов:

```
#include <iostream>
```

2. **Объявление функций и глобальных объектов**

   Функции могут быть объявлены до их использования.

3. **Главная функция main**

   Точка входа в программу:

```
int main() {
   return 0;
}
```

4. **Тело программы**

   Содержит последовательность операторов, реализующих алгоритм.

Особенности структуры программы на C++

- выполнение программы начинается с функции main;
- программа состоит из операторов, выполняемых последовательно, с возможными ветвлениями и циклами;
- язык C++ является компилируемым языком;
- поддерживается модульность за счёт функций и файлов заголовков.

Заключение: язык программирования является формальным средством записи алгоритмов, среда разработки обеспечивает технические средства для создания программ, а структура программы на языке C++ определяет организацию кода и порядок выполнения программы.

Полный цикл разработки C++-приложения

Полный цикл разработки C++-приложения — это совокупность взаимосвязанных этапов создания программного продукта от формулировки задачи до сопровождения готового решения. Цель полного цикла — получение корректного, эффективного и сопровождаемого приложения.

Основные этапы полного цикла разработки

1. Постановка задачи и формирование требований

На данном этапе:

- формулируется цель разработки;
- определяются входные и выходные данные;
- задаются ограничения по времени, памяти и точности;
- уточняется область применения программы.

Результатом является чёткое понимание того, **что должна делать программа**.

2. Проектирование решения

Этап логического и архитектурного проектирования:

- выбор алгоритмов и методов решения;
- выбор структур данных;
- определение архитектуры приложения;
- декомпозиция задачи на функции и модули.

На этом этапе формируется **алгоритмическая и структурная модель программы**.

3. Реализация (кодирование)

Происходит перевод проектного решения в программный код на языке C++:

- реализация алгоритмов;
- описание функций и классов;
- работа с памятью и ресурсами;
- соблюдение синтаксических и семантических правил языка.

Результатом является исходный код программы.

4. Сборка и компиляция

На данном этапе:

- исходный код компилируется компилятором C++;
- выявляются и устраняются синтаксические и часть семантических ошибок;
- формируется исполняемый файл.

Компиляция обеспечивает преобразование программы в машинный код.

5. Тестирование

Проверяется корректность работы программы:

- тестирование на корректных и граничных данных;
- проверка соответствия требованиям;
- выявление логических ошибок;
- проверка устойчивости и корректности обработки ошибок.

Цель этапа — подтверждение правильности реализации алгоритмов.

6. Отладка

Этап исправления обнаруженных ошибок:

- пошаговое выполнение программы;
- анализ значений переменных;
- устранение логических и временных ошибок;
- оптимизация некорректных участков кода.

Отладка тесно связана с тестированием.

7. Оптимизация

При необходимости выполняется:

- оптимизация алгоритмов;
- улучшение использования памяти;
- снижение времени выполнения программы;
- устранение избыточных вычислений.

Оптимизация проводится **после получения корректного решения**.

8. Документирование

Создаётся документация:

- описание алгоритмов и структуры программы;
- комментарии в коде;
- инструкции по сборке и использованию.

Документация обеспечивает сопровождение и развитие приложения.

9. Сопровождение и развитие

После ввода программы в эксплуатацию:

- исправляются найденные ошибки;

- вносятся изменения и расширения;
- адаптация программы под новые требования.

Заключение: Полный цикл разработки C++-приложения представляет собой последовательный процесс от постановки задачи до сопровождения готового программного продукта. Соблюдение всех этапов цикла позволяет получить корректное, эффективное и устойчивое приложение, соответствующее заданным требованиям.

Основные парадигмы программирования: Императивное программирование. Декларативное программирование. Объектно-ориентированное программирование. Функциональное программирование. Логическое программирование

Парадигма программирования — это совокупность концепций и принципов, определяющих способ постановки и решения задач, а также стиль построения программ. Парадигма задаёт модель мышления программиста и подход к организации вычислений.

Императивное программирование основано на описании алгоритма в виде последовательности команд, явно изменяющих состояние программы.

Характерные особенности:

- программа — последовательность инструкций;
- явное управление потоком выполнения;
- изменение значений переменных;
- использование операторов присваивания, ветвлений и циклов.

Декларативное программирование ориентировано на описание того, *что* требуется получить, а не *как* это сделать.

Основные черты:

- отсутствие явного управления потоком выполнения;
- минимизация изменения состояния;
- акцент на описании свойств и отношений.

Примерами декларативного подхода являются языки запросов и логические языки программирования.

Объектно-ориентированное программирование (ООП) рассматривает программу как совокупность взаимодействующих объектов, объединяющих данные и методы их обработки.

Основные принципы ООП:

- **инкапсуляция** — объединение данных и методов;
- **абстракция** — выделение существенных характеристик;
- **наследование** — повторное использование кода;
- **полиморфизм** — единый интерфейс для различных реализаций.

ООП используется для разработки сложных и масштабируемых программных систем.

Функциональное программирование основано на вычислении значений функций без изменения состояния программы.

Характерные особенности:

- использование чистых функций;
- отсутствие побочных эффектов;
- неизменяемость данных;
- рекурсия вместо циклов.

Функциональный подход упрощает анализ и повышает надёжность программ.

Логическое программирование основано на формализации знаний в виде логических фактов и правил, а решение задачи сводится к логическому выводу.

Основные характеристики:

- описание отношений между объектами;
- автоматический вывод решений;
- отсутствие явного алгоритма решения.

Программирование заключается в формулировке условий, при которых утверждение считается истинным.

Заключение: Различные парадигмы программирования предлагают разные подходы к решению задач. Современные языки программирования, включая C++, поддерживают несколько парадигм, что позволяет выбирать наиболее эффективный стиль разработки в зависимости от характера задачи.

Язык C++. Синтаксис. Элементы синтаксиса. Правила синтаксиса. Синтаксические ошибки. Характеристики эффективности синтаксиса

Синтаксис языка программирования — это совокупность формальных правил, определяющих, каким образом из символов языка образуются корректные программные конструкции. Синтаксис языка C++ задаёт допустимую структуру программ и их элементов и является обязательным для корректной компиляции.

Элементы синтаксиса языка C++

К основным элементам синтаксиса C++ относятся:

1 Алфавит языка Включает: • латинские буквы, • цифры, • специальные символы (+, -, {}, ; и др.).	**2 Ключевые слова** Зарезервированные слова языка (int, if, while, return, const и др.), имеющие фиксированное значение и не допускаемые к использованию в качестве идентификаторов.	**3 Идентификаторы** Имена переменных, функций, типов и объектов. Правила: a. начинаются с буквы или _, b. могут содержать буквы, цифры и _, c. регистр символов имеет значение.
4 Литералы Числовые, символьные, строковые и логические константы.	**5 Операторы и выражения** Средства выполнения вычислений и управления программой.	**6 Разделители** Скобки, запятые, точка с запятой, определяющие структуру программы.

Правила синтаксиса C++

Основные синтаксические правила:

- каждый оператор завершается точкой с запятой;
- блоки кода ограничиваются фигурными скобками {} ;
- операторы должны располагаться в допустимой последовательности;
- соблюдается приоритет и ассоциативность операторов;
- идентификаторы должны быть объявлены до использования;
- структура программы должна содержать функцию main как точку входа.

Синтаксис C++ является **строгим и формализованным**, что позволяет однозначно интерпретировать программы компилятором.

Синтаксические ошибки — это ошибки, связанные с нарушением правил построения программных конструкций.

Примеры синтаксических ошибок:

- пропущенная точка с запятой;
- несоответствие открывающих и закрывающих скобок;
- неверное использование ключевых слов;
- неправильная структура выражений и операторов.

Синтаксические ошибки выявляются **на этапе компиляции** и препятствуют созданию исполняемой программы.

Характеристики эффективности синтаксиса

Эффективность синтаксиса языка C++ характеризуется следующими свойствами:

1. **Выразительность**

   Возможность компактно и точно описывать алгоритмы и структуры данных.

2. **Однозначность**

   Каждая корректная конструкция интерпретируется компилятором единственным образом.

3. **Строгость**

   Чёткие правила снижают вероятность неоднозначных интерпретаций.

4. **Близость к машинной модели**

   Синтаксис C++ позволяет эффективно управлять памятью и ресурсами.

5. **Поддержка различных парадигм**

   Синтаксис языка допускает процедурный, объектно-ориентированный и функциональный стили программирования.

Заключение: синтаксис языка C++ представляет собой формальную систему правил, определяющих структуру программы. Элементы синтаксиса и строгие правила языка обеспечивают однозначность и корректность программ, а высокая выразительность и близость к аппаратной реализации делают C++ эффективным инструментом для разработки производительных приложений.

Язык C++. Типы данных и константы

Типы данных в языке C+

Тип данных определяет множество допустимых значений переменной, объём занимаемой памяти и допустимые операции над данными. Типизация в C++ является **статической**, то есть типы всех объектов определяются на этапе компиляции.

Классификация типов данных

Фундаментальные (базовые) типы

К фундаментальным типам относятся:

- целочисленные: int, short, long, long long, char, bool;
- вещественные: float, double, long double;
- модификаторы знака и размера: signed, unsigned.

Размеры типов зависят от реализации, но подчиняются стандартным ограничениям языка.

Производные и составные типы

К производным типам относятся:

- массивы;
- указатели;
- ссылки;
- функции.

Эти типы формируются на основе фундаментальных типов и используются для построения более сложных структур данных.

Пользовательские типы

C++ позволяет создавать пользовательские типы:

- struct, class;
- enum, enum class;
- typedef и using.

Пользовательские типы обеспечивают абстракцию и структурирование данных.

Константы в языке C+

Константа — это объект, значение которого не может быть изменено после инициализации.

Ключевое слово const

Используется для объявления неизменяемых объектов:

const int n = 10;

Гарантирует защиту значения от изменения в процессе выполнения программы.

constexpr

Используется для задания констант, вычисляемых **на этапе компиляции**:

constexpr int size = 100;

Позволяет оптимизировать вычисления и использовать значения в контексте, требующем константного выражения.

Литералы как константы

К числовым и символьным константам относятся:

- целочисленные и вещественные литералы;
- символьные и строковые литералы;
- логические литералы true и false.

Преобразование типов

C++ поддерживает:

- **неявные преобразования типов**;
- **явные преобразования** (приведения типов).

Использование инициализации в фигурных скобках {} позволяет запретить сужающие преобразования.

Роль типов и констант

Типы данных и константы:

- обеспечивают корректность вычислений;
- повышают надёжность программ;
- позволяют компилятору выполнять оптимизацию;
- являются основой типобезопасности языка.

Заключение: Типы данных в языке C++ определяют представление и обработку информации в программе, а использование констант позволяет зафиксировать неизменяемые значения и повысить надёжность и эффективность программ.

Язык C++. Переменные и операторы

Переменная — это именованная область памяти, предназначенная для хранения данных определённого типа, значение которых может изменяться в процессе выполнения программы.

Основные характеристики переменной:

- **тип данных** — определяет допустимые значения и операции;
- **имя (идентификатор)** — используется для обращения к переменной;
- **значение** — текущее содержимое памяти;
- **область видимости** — часть программы, в которой переменная доступна;
- **время жизни** — период существования переменной в памяти.

Переменные должны быть объявлены до использования и могут быть инициализированы при объявлении.

Инициализация переменных

В C++ используются различные способы инициализации:

- копирующая инициализация (=),
- прямая инициализация (()),
- инициализация списком ({}).

Инициализация списком предотвращает неявные сужающие преобразования типов.

Оператор — это конструкция языка, задающая действие, выполняемое над операндами.

Основные группы операторов

1. **Арифметические операторы** +, -, *, /, %	2. **Операторы присваивания** =, +=, -=, *=, /=	3. **Операторы сравнения** ==, !=, <, >, <=, >=
4. **Логические операторы** &&, ||, !	5. **Побитовые операторы** &, |, ^, ~, <<, >>	**6 Операторы инкремента и декремента** ++, --
7 Тернарный оператор ?:		

Приоритет и ассоциативность операторов

Операторы обладают:

- **приоритетом** — определяет порядок выполнения операций;
- **ассоциативностью** — определяет порядок вычислений при равных приоритетах.

Корректное использование операторов требует учёта этих свойств либо применения скобок.

Выражение — это комбинация операторов и операндов, которая в результате вычисления даёт значение.

Выражения являются основой вычислительной части программы.

Роль переменных и операторов

Переменные и операторы обеспечивают:

- хранение и изменение данных;
- выполнение вычислений;
- управление логикой программы;
- реализацию алгоритмов.

Заключение: Переменные и операторы являются базовыми элементами языка C++, обеспечивающими представление данных и выполнение вычислений. Их корректное использование определяет правильность и эффективность программ.

Язык C++. Массивы

Массив — это упорядоченная совокупность элементов одного типа данных, размещённых в памяти **непрерывно** и доступных по общему имени с использованием индекса. Массивы применяются для хранения и обработки наборов однотипных данных.

Объявление и инициализация массивов

В языке C++ массив объявляется с указанием типа элементов и количества элементов:

```
int a[10];
```

Инициализация массива может выполняться:

```
int b[5] = {1, 2, 3, 4, 5};
```

Если количество инициализаторов меньше размера массива, оставшиеся элементы инициализируются нулевыми значениями.

Индексация массивов

- Индексация элементов массива начинается с **нуля**.
- Доступ к элементу осуществляется по индексу:

```
a[i]
```

- Допустимые индексы находятся в диапазоне от 0 до n − 1.

Язык C++ **не выполняет проверку выхода за границы массива**, что может привести к неопределённому поведению программы.

Многомерные массивы

C++ поддерживает многомерные массивы, наиболее распространёнными являются двумерные массивы:

```
int m[3][4];
```

Элементы двумерного массива размещаются в памяти построчно.

Связь массивов и памяти

Массив хранится в непрерывной области памяти.

Имя массива в большинстве выражений неявно преобразуется в указатель на первый элемент массива.

Ограничения массивов

- размер массива должен быть известен на этапе компиляции;
- невозможность изменения размера массива;
- отсутствие встроенной проверки границ.

Для динамических массивов используются средства динамической памяти.

Роль массивов

Массивы являются:

- базовой структурой данных;
- основой для реализации алгоритмов сортировки и поиска;
- фундаментом для более сложных структур данных.

Заключение: массивы в языке C++ представляют собой простую и эффективную структуру данных для хранения однотипных элементов. Их использование требует аккуратного обращения с индексами и памятью, так как контроль границ массива возлагается на программиста.

Язык C++. Операторы принятия решений. Операторы циклов

Операторы принятия решений предназначены для выбора одного из возможных вариантов выполнения программы в зависимости от заданного условия.

Условный оператор if

Оператор if выполняет блок кода при истинности логического выражения:

```
if (condition) {
    // действия
}
else {
    // альтернативные действия
}
```

Допускается каскадное использование else if для проверки нескольких условий.

Оператор выбора switch

Оператор switch используется для выбора варианта выполнения программы по значению выражения целочисленного типа:

```
switch (expr) {
    case value1:
        break;
    case value2:
        break;
    default:
        break;
}
```

Особенностью оператора является необходимость использования break для предотвращения перехода к следующему варианту.

Операторы циклов

Циклы предназначены для многократного выполнения фрагмента программы.

Цикл for

Используется, когда число повторений известно заранее:

```
for (init; condition; iteration) {
    // тело цикла
}
```

Цикл while

Используется, когда число повторений заранее неизвестно:

```
while (condition) {
   // тело цикла
}
```

Цикл do-while

Гарантирует выполнение тела цикла хотя бы один раз:

```
do {
   // тело цикла
} while (condition);
```

Управление выполнением циклов

Для управления выполнением циклов используются:

- break — немедленный выход из цикла;
- continue — переход к следующей итерации.

Связь с алгоритмами

Операторы принятия решений и циклы реализуют базовые управляющие конструкции алгоритмов:

- **ветвление**;
- **повторение**.

Они являются основой реализации процедурных алгоритмов в языке C++.

Заключение: Операторы принятия решений и циклов в языке C++ обеспечивают управление потоком выполнения программы и позволяют реализовывать алгоритмы любой сложности на основе условий и повторяющихся действий.

Язык C++. Указатели

Указатель — это переменная, значением которой является **адрес области памяти**, в которой хранится объект определённого типа. Указатель не хранит само значение объекта, а хранит адрес этого объекта в памяти.

Объявление указателя:

```
int* p;
```

Основные операции с указателями

1. **Оператор взятия адреса &**

   Используется для получения адреса переменной:

```
int x;
int* p = &x;
```

1.
2. **Оператор разыменования ***

   Используется для доступа к значению по адресу:

```
*p = 10;
```

2.
3. **Присваивание указателей**

   Указатель может указывать на другой объект того же типа.

Арифметика указателей

Арифметика указателей основана на размере типа, на который указывает указатель:

- p + 1 указывает на следующий элемент массива;
- p - 1 — на предыдущий элемент;
- разность двух указателей даёт количество элементов между ними.

Арифметика указателей допустима только в пределах одного массива.

Связь массивов и указателей

Имя массива в большинстве выражений неявно преобразуется в указатель на его первый элемент:

```
int a[5];
int* p = a;
```

Доступ к элементам массива возможен как через индекс, так и через указатели.

Константы и указатели

Различают следующие варианты использования const с указателями:

1. **Указатель на константу**

const int* p;

Нельзя изменять значение объекта, на который указывает указатель.

2. **Константный указатель**

int* const p = &x;

Нельзя изменять адрес, хранимый в указателе.

3. **Константный указатель на константу**

const int* const p = &x;

Нельзя изменять ни адрес, ни значение.

Назначение указателей

Указатели используются для:

- работы с динамической памятью;
- передачи аргументов в функции;
- реализации сложных структур данных;
- эффективной работы с массивами.

Заключение: указатели в языке C++ являются мощным средством управления памятью. Их корректное использование требует строгого соблюдения правил работы с адресами, арифметикой указателей и константностью.

Массивы и указатели. Динамическая память и smart-указатели в языке C++

Связь массивов и указателей

В языке C++ массивы и указатели тесно связаны через модель памяти.

- Элементы массива располагаются **непрерывно в памяти**.
- Имя массива в большинстве выражений **неявно преобразуется в указатель** на его первый элемент.
- Доступ к элементам массива возможен:
  - по индексу: a[i],
  - через указатель: *(a + i).

Таким образом, индексация массива является формой арифметики указателей.

Арифметика указателей и массивы

При выполнении операций над указателями учитывается размер типа:

- p + i указывает на i-й элемент массива;
- разность указателей даёт количество элементов между ними.

Арифметика указателей допустима **только в пределах одного массива**.

Динамическая память — это область памяти, выделяемая и освобождаемая во время выполнения программы.

В C++ для работы с динамической памятью используются операторы:

- new — выделение памяти,
- delete — освобождение памяти.

Пример:

```
int* p = new int;
delete p;
```

Для динамических массивов используются:

```
int* a = new int[n];
delete[] a;
```

Проблемы ручного управления памятью

При использовании обычных указателей возможны:

- утечки памяти;
- обращение к освобождённой памяти;
- двойное освобождение памяти.

Эти проблемы требуют строгого контроля со стороны программиста.

Smart-указатели — это объекты стандартной библиотеки, которые автоматически управляют временем жизни динамически выделенной памяти.

Основные виды smart-указателей:

1. **std::unique_ptr**

   Обладает единственным владельцем ресурса.

2. **std::shared_ptr**

   Поддерживает совместное владение ресурсом.

3. **std::weak_ptr**

   Используется для наблюдения за объектом без владения.

Smart-указатели реализуют принцип **RAII** — освобождение ресурса происходит автоматически при выходе объекта из области видимости.

Использование smart-указателей:

- повышает надёжность программ;
- предотвращает утечки памяти;
- упрощает управление ресурсами;
- рекомендуется вместо обычных указателей при работе с динамической памятью.

Заключение: связь массивов и указателей лежит в основе модели памяти C++. Динамическая память расширяет возможности программ, а smart-указатели обеспечивают безопасное и автоматическое управление ресурсами, снижая вероятность ошибок.

Язык C++. Функции

Функция — это именованный фрагмент программы, предназначенный для выполнения определённой подзадачи. Функции позволяют структурировать программу, реализовать повторное использование кода и упростить разработку и сопровождение программ.

Объявление и определение функций

Функция в C++ может быть:

- **объявлена** (прототип функции),
- **определена** (реализация функции).

Пример:

```
int sum(int a, int b);   // объявление
int sum(int a, int b) { // определение
    return a + b;
}
```

Объявление функции сообщает компилятору её интерфейс.

Параметры и аргументы функций

Функции могут принимать параметры, передаваемые:

- **по значению** — создаётся копия аргумента;
- **по ссылке** — функция работает с оригинальным объектом;
- **через указатель** — передаётся адрес объекта.

Использование const в параметрах предотвращает изменение аргументов.

Возврат значений

Функция может возвращать:

- значение фундаментального типа;
- объект пользовательского типа;
- ссылку или указатель.

Тип возвращаемого значения указывается в объявлении функции.

Использование оператора return завершает выполнение функции.

Область видимости и время жизни

- Параметры и локальные переменные функции имеют **локальную область видимости**.
- Локальные переменные уничтожаются при выходе из функции.

Рекурсия

Функция может вызывать саму себя.

Рекурсивная функция должна иметь:

- условие завершения;
- корректное уменьшение задачи.

Рекурсия используется для реализации алгоритмов декомпозиции.

Назначение функций

Функции обеспечивают:

- модульность программы;
- структурирование кода;
- повторное использование алгоритмов;
- повышение читаемости и надёжности.

Заключение: Функции в языке C++ являются основным средством декомпозиции программ. Они позволяют организовать код в виде логически завершённых модулей и являются фундаментом для построения сложных программных систем.

Указатели на функции. Перегрузка функций в языке C++. Шаблоны функций

Указатель на функцию — это переменная, значением которой является адрес функции с определённой сигнатурой (тип возвращаемого значения и список параметров).

Объявление указателя на функцию:

```
int (*pf)(int, int);
```

Основные свойства:

- указатель может хранить адрес любой функции с совпадающей сигнатурой;
- вызов функции осуществляется через указатель;
- широко используются для реализации обратных вызовов (callback), таблиц функций и стратегий поведения.

Указатели на функции позволяют отделить выбор выполняемой функции от места её вызова.

Перегрузка функций — это возможность определения нескольких функций с одним и тем же именем, но с различными списками параметров.

Условия перегрузки:

- функции должны различаться **типами и/или количеством параметров**;
- тип возвращаемого значения **не участвует** в разрешении перегрузки.

Пример:

```
int max(int a, int b);
double max(double a, double b);
```

Выбор конкретной функции осуществляется компилятором на этапе компиляции на основе типов аргументов.

Назначение перегрузки:

- повышение выразительности кода;
- единый интерфейс для однотипных операций;
- улучшение читаемости программ.

Шаблон функции — это параметризованное описание функции, позволяющее создавать функции для различных типов данных без дублирования кода.

Объявление шаблона функции:

```
template <typename T>
T max(T a, T b) {
   return (a > b) ? a : b;  }
```

Основные характеристики:

- параметр типа задаётся с помощью template;
- конкретная функция создаётся компилятором при использовании шаблона;
- обеспечивает обобщённое программирование.

Шаблоны позволяют:

- реализовать универсальные алгоритмы;
- обеспечить типобезопасность;
- избежать дублирования кода.

Связь перегрузки и шаблонов

- шаблонные функции могут перегружаться;
- при наличии обычной и шаблонной функции предпочтение отдаётся более точному совпадению;
- выбор осуществляется компилятором на этапе компиляции.

Заключение: указатели на функции обеспечивают гибкость управления выполнением программ, перегрузка функций позволяет использовать единое имя для логически одинаковых операций, а шаблоны функций реализуют обобщённое программирование и повышают универсальность и эффективность кода в языке C++.

14. Асимптотический анализ алгоритмов

Асимптотический анализ алгоритмов — это метод оценки эффективности алгоритмов, основанный на исследовании роста требуемых вычислительных ресурсов при увеличении размера входных данных. Анализ проводится **независимо от конкретной реализации и аппаратной платформы**.

Цели асимптотического анализа

- сравнение алгоритмов;
- оценка масштабируемости решений;
- обоснованный выбор алгоритма.

Показатели сложности

1. **Временная сложность** — количество элементарных операций.
2. **Пространственная сложность** — объём используемой памяти.

Асимптотические обозначения

- **O(f(n))** — верхняя оценка сложности;
- **Ω(f(n))** — нижняя оценка;
- **Θ(f(n))** — точная асимптотическая оценка.

При анализе:

- отбрасываются константные множители;
- учитывается доминирующий член функции сложности.

Случаи анализа

- лучший случай;
- средний случай;
- худший случай.

Методы анализа

- анализ циклов и вложенных циклов;
- анализ рекурсивных алгоритмов;
- использование рекуррентных соотношений.

Заключение: Асимптотический анализ позволяет оценивать эффективность алгоритмов на больших входных данных и является основным инструментом теории алгоритмов.

15. Структуры данных. Линейные и нелинейные структуры данных

Структура данных — это способ организации и хранения данных в памяти, обеспечивающий эффективный доступ и обработку информации.

Линейные структуры характеризуются последовательным расположением элементов.

К линейным структурам относятся:

- массивы;
- связные списки;
- стек;
- очередь;
- дек.

Особенности:

- каждый элемент имеет не более одного предшественника и одного последователя;
- операции вставки и удаления зависят от конкретной реализации.

Нелинейные структуры данных

Нелинейные структуры характеризуются иерархическими или сетевыми связями между элементами.

К нелинейным структурам относятся:

- деревья;
- графы;
- кучи;
- хеш-таблицы.

Особенности:

- элементы могут иметь несколько связей;
- используются для представления сложных отношений.

Роль структур данных

Выбор структуры данных:

- напрямую влияет на эффективность алгоритмов;
- определяет сложность основных операций;
- является ключевым этапом проектирования алгоритмов.

Заключение: Асимптотический анализ и структуры данных являются фундаментальными понятиями теории алгоритмов. Их совместное использование позволяет разрабатывать корректные и эффективные программные решения.

16. Метод грубой силы. Алгоритмы сортировки выбором и пузырьковой сортировки. Алгоритм последовательного поиска

Метод грубой силы заключается в прямом переборе всех возможных вариантов решения задачи без использования дополнительных предположений или оптимизаций.

Характерные особенности:

- простота реализации;
- универсальность;
- высокая вычислительная сложность.

Метод применяется как базовый и используется для получения корректного, но неэффективного решения.

Сортировка выбором

Алгоритм основан на последовательном выборе минимального элемента из неотсортированной части массива и его перемещении в начало.

Этапы:

1. поиск минимального элемента;
2. обмен с первым элементом неотсортированной части;
3. повтор до полного упорядочивания.

Сложность:

- время: **O(n²)**;
- память: **O(1)**.

Пузырьковая сортировка

Алгоритм выполняет многократные проходы по массиву, попарно сравнивая соседние элементы и меняя их местами при необходимости.

Особенности:

- на каждом проходе максимальный элемент «всплывает» в конец массива;
- может быть оптимизирован досрочным завершением.

Сложность:

- худший и средний случай: **O(n²)**;
- лучший случай: **O(n)** (при оптимизации).

Последовательный поиск

Алгоритм поиска элемента путём последовательного просмотра элементов массива.

Сложность:

- худший случай: **O(n)**;
- лучший случай: **O(1)**.

Применяется для неотсортированных массивов.

17. Метод декомпозиции. Алгоритмы сортировки слиянием и быстрого поиска

Метод декомпозиции (разделяй и властвуй)

Метод основан на разбиении исходной задачи на несколько меньших подзадач, их независимом решении и объединении результатов.

Этапы:

1. разбиение задачи;
2. рекурсивное решение подзадач;
3. объединение решений.

Сортировка слиянием

Массив рекурсивно делится на две части, каждая часть сортируется, затем отсортированные части сливаются.

Сложность:

- время: **O(n log n)** во всех случаях;
- память: **O(n)**.

Алгоритм устойчив и эффективен для больших объёмов данных.

Алгоритм быстрого поиска (QuickSearch / QuickSort-подход)

Основан на выборе опорного элемента и разбиении массива относительно него.

Сложность:

- средний случай: **O(n log n)**;
- худший случай: **O(n²)**.

Эффективен на практике за счёт малых констант.

18. Метод декомпозиции. Алгоритм бинарного поиска. Алгоритм быстрого выбора

Бинарный поиск

Алгоритм поиска в **отсортированном массиве**, основанный на последовательном делении диапазона поиска пополам.

Этапы:

1. сравнение искомого элемента с серединой массива;
2. выбор левой или правой половины;
3. повтор до нахождения элемента или исчерпания диапазона.

Сложность:

- время: **O(log n)**;
- память: **O(1)** (итеративная реализация).

Алгоритм быстрого выбора (Quickselect)

Используется для нахождения k-го порядкового элемента без полной сортировки массива.

Основан на тех же принципах, что и быстрая сортировка:

- выбор опорного элемента;
- разбиение массива;
- рекурсивный вызов только для одной части.

Сложность:

- средний случай: **O(n)**;
- худший случай: **$O(n^2)$**.

Вывод: Метод грубой силы применяется для простых и универсальных решений, но обладает высокой сложностью.

Метод декомпозиции позволяет строить эффективные алгоритмы сортировки и поиска, существенно снижая вычислительные затраты за счёт рекурсивного разбиения задачи

19. Метод декомпозиции. Алгоритм Штрассена

Метод декомпозиции (разделяй и властвуй)

Метод декомпозиции заключается в разбиении исходной задачи на несколько подзадач меньшего размера, их рекурсивном решении и последующем объединении результатов. Эффективность метода достигается за счёт уменьшения сложности подзадач.

Алгоритм Штрассена — это алгоритм умножения квадратных матриц, основанный на методе декомпозиции.

Идея алгоритма:

- матрицы разбиваются на блоки;
- вместо стандартных 8 умножений блоков выполняется **7 умножений**;
- результат собирается из полученных промежуточных матриц.

Сложность:

- стандартное умножение матриц: **$O(n^3)$**;
- алгоритм Штрассена: **$O(n^{\log_2 7}) \approx O(n^{2.81})$**.

Особенности:

- эффективен для больших размеров матриц;
- увеличивает количество сложений;
- менее устойчив численно по сравнению с классическим алгоритмом.

20. Метод уменьшения размера задачи. Сортировка вставкой

Метод уменьшения размера задачи заключается в сведении задачи размера n к задаче размера $n-1$ или меньшего размера, после чего решение достраивается до исходной задачи.

Сортировка вставкой

Алгоритм сортировки вставкой строит отсортированную последовательность, последовательно вставляя каждый новый элемент в уже отсортированную часть массива.

Этапы:

1. первый элемент считается отсортированным;
2. очередной элемент вставляется в правильную позицию;
3. процесс повторяется до конца массива.

Сложность:

- худший и средний случай: $\mathbf{O(n^2)}$;
- лучший случай (отсортированный массив): $\mathbf{O(n)}$.

Особенности:

- сортировка на месте;
- устойчива;
- эффективна для малых и почти отсортированных массивов.

21. Метод уменьшения размера задачи. Алгоритм умножения «по-русски»

Алгоритм умножения по-русски — это алгоритм умножения двух чисел, основанный на последовательном **уменьшении одного аргумента вдвое** и удвоении другого.

Идея алгоритма:

1. одно число делится на 2 (целочисленно);
2. второе число умножается на 2;
3. если первое число нечётное — второе добавляется к результату;
4. процесс повторяется, пока первое число не станет равным нулю.

Связь с методом уменьшения размера задачи

На каждом шаге задача умножения сводится к задаче меньшего размера (вдвое меньшему числу).

Сложность:

- время: **O(log n)**;
- память: **O(1)** (итеративная реализация).

Заключение: Метод декомпозиции позволяет ускорить решение задач путём рекурсивного разбиения, что демонстрирует алгоритм Штрассена.

Метод уменьшения размера задачи применяется в сортировке вставкой и алгоритме умножения по-русски, последовательно сводя исходную задачу к более простой и обеспечивая эффективное решение

22. Алгоритм умножения по-русски. Задачи умножения числа x на заданное число n и возведения числа x в степень n

Алгоритм умножения по-русски основан на **методе уменьшения размера задачи** и использует операции деления на 2 и умножения на 2.

Идея алгоритма:

- одно число последовательно делится на 2 (целочисленно);
- второе число последовательно удваивается;
- если делимое число нечётное, удвоенное значение прибавляется к результату;
- процесс продолжается до тех пор, пока делимое не станет равным нулю.

Сложность:

- временная сложность — **O(log n)**;
- память — **O(1)**.

Умножение числа x на заданное число n

Задача умножения x*n решается этим алгоритмом за счёт представления числа n в двоичной форме и суммирования соответствующих степеней двойки числа x.

Алгоритм эффективен, так как число шагов пропорционально числу бит в записи n.

Возведение числа x в степень n

Задача возведения в степень решается аналогично, с использованием уменьшения размера задачи:

- если n чётное:

  x^n = (x^2)^{n/2}

- если n нечётное:

  x^n = x \cdot x^{n-1}

Этот алгоритм известен как **быстрое возведение в степень**.

Сложность:

- временная сложность — **O(log n)**;
- память — **O(log n)** при рекурсивной реализации.

23. Метод уменьшения размера задачи. Быстрая сортировка

Метод уменьшения размера задачи сводит исходную задачу размера n к задачам меньшего размера, как правило n-1 или n/2, после чего решение достраивается до исходного.

Быстрая сортировка (QuickSort)

Алгоритм быстрой сортировки основан на выборе **опорного элемента** и разбиении массива на две части:

- элементы меньше опорного;
- элементы больше опорного.

После разбиения каждая часть сортируется рекурсивно.

Сложность:

- средний случай — **O(n log n)**;
- худший случай — **O(n²)** (неудачный выбор опорного элемента);
- память — **O(log n)**.

Особенности:

- сортировка на месте;
- высокая практическая эффективность;
- неустойчива.

24. Алгоритмы поиска в глубину и в ширину

Поиск в глубину (DFS)

Поиск в глубину — алгоритм обхода графа, при котором сначала максимально углубляются по одному пути, прежде чем переходить к другим вершинам.

Характеристики:

- используется стек или рекурсия;
- применяется для проверки связности, поиска циклов, топологической сортировки.

Сложность:

- время — **O(|V| + |E|)**;
- память — **O(|V|)**.

Поиск в ширину (BFS)

Поиск в ширину — алгоритм обхода графа, при котором сначала посещаются все вершины одного уровня, затем следующего.

Характеристики:

- используется очередь;
- находит кратчайшие пути в невзвешенных графах.

Сложность:

- время — **O(|V| + |E|)**;
- память — **O(|V|)**.

Заключение: Алгоритмы умножения по-русски и быстрого возведения в степень демонстрируют эффективность метода уменьшения размера задачи. Быстрая сортировка использует тот же принцип для упорядочивания данных. Алгоритмы поиска в глубину и ширину являются базовыми методами обхода графов и применяются при решении широкого класса задач теории графов и алгоритмов.

25. Динамическое программирование. Задача о рюкзаке

Динамическое программирование — это метод разработки алгоритмов, применяемый к задачам, обладающим свойствами:

- **оптимальной подструктуры**;
- **перекрывающихся подзадач**.

Метод заключается в разбиении задачи на подзадачи, решении каждой подзадачи **один раз** и сохранении результатов для повторного использования.

Задача о рюкзаке

Классическая задача о рюкзаке формулируется следующим образом:

дан набор предметов, каждый из которых имеет вес и стоимость, и рюкзак ограниченной вместимости. Требуется выбрать набор предметов с максимальной суммарной стоимостью, не превышая допустимый вес.

Решение методом динамического программирования

Строится таблица, где:

- строки соответствуют количеству рассматриваемых предметов;
- столбцы — допустимой вместимости рюкзака.

Каждая ячейка содержит максимальную стоимость, достижимую при данных условиях.

Сложность:

- время — **O(nW)**;
- память — **O(nW)**, где n — число предметов, W — вместимость рюкзака.

Особенности

- гарантирует оптимальное решение;
- применим только при целочисленных весах;
- ресурсоёмок при больших значениях W.

26. Динамическое программирование. Алгоритм Флойда. Алгоритм Уоршелла

Алгоритм Флойда используется для нахождения кратчайших путей между **всеми парами вершин** взвешенного графа.

Идея алгоритма:

- последовательно допускаются промежуточные вершины;
- на каждом шаге улучшается оценка кратчайшего пути.

Сложность:

- время — $\mathbf{O(n^3)}$;
- память — $\mathbf{O(n^2)}$.

Алгоритм Уоршелла предназначен для нахождения **транзитивного замыкания** ориентированного графа.

Идея:

- поэтапно добавляются возможные пути через промежуточные вершины;
- определяется достижимость вершин друг из друга.

Сложность:

- время — $\mathbf{O(n^3)}$;
- память — $\mathbf{O(n^2)}$.

Связь с динамическим программированием

Алгоритмы Флойда и Уоршелла:

- используют принцип динамического программирования;
- строят решение на основе ранее полученных промежуточных результатов;
- демонстрируют оптимальную подструктуру задач на графах.

Заключение: Динамическое программирование является мощным методом решения оптимизационных задач. Задача о рюкзаке иллюстрирует его применение к комбинаторным задачам, а алгоритмы Флойда и Уоршелла демонстрируют использование метода для анализа графов и нахождения оптимальных и достижимых путей.

27. Жадные методы. Алгоритмы Прима, Крускала и Дейкстры

Жадные алгоритмы — это методы разработки алгоритмов, при которых на каждом шаге выбирается локально оптимальное решение в надежде получить глобально оптимальный результат.

Основные свойства жадных алгоритмов:

- простота реализации;
- высокая скорость работы;
- корректность гарантируется не для всех задач.

Алгоритм Прима

Алгоритм Прима используется для построения **минимального остовного дерева** связного взвешенного графа.

Идея алгоритма:

- начинается с произвольной вершины;
- на каждом шаге добавляется минимальное ребро, соединяющее построенное дерево с новой вершиной.

Сложность:

- при использовании матрицы смежности — $\mathbf{O(n^2)}$;
- при использовании очереди с приоритетом — $\mathbf{O(E \log V)}$.

Алгоритм Крускала

Алгоритм Крускала также строит минимальное остовное дерево, но:

- сортирует все рёбра по весу;
- последовательно добавляет рёбра минимального веса, избегая циклов.

Сложность:

- сортировка рёбер — $\mathbf{O(E \log E)}$;
- эффективен при разреженных графах.

Алгоритм Дейкстры

Алгоритм Дейкстры предназначен для поиска кратчайших путей от одной вершины до всех остальных во взвешенном графе с **неотрицательными весами**.

Идея:

- на каждом шаге выбирается вершина с минимальной текущей оценкой расстояния;
- оценки расстояний уточняются для соседних вершин.

Сложность:

- **$O(n^2)$** или **$O(E \log V)$** при использовании очереди с приоритетом.

28. Информированный и неинформированный поиск

Неинформированный поиск не использует дополнительной информации о цели, кроме описания пространства состояний.

Примеры:

- поиск в ширину;
- поиск в глубину;
- равномерный поиск по стоимости.

Характеристики:

- универсальность;
- высокая вычислительная сложность;
- отсутствие направленности к цели.

Информированный поиск

Информированный поиск использует **эвристическую функцию**, оценивающую расстояние до цели.

Характеристики:

- направленность поиска;
- повышение эффективности;
- зависит от качества эвристики.

Примеры:

- жадный поиск;
- алгоритм A*.

29. Алгоритмы поиска на сложных пространствах

Поиск на сложных пространствах состояний применяется к задачам с большим числом возможных состояний и переходов между ними.

Характерные особенности:

- экспоненциальный рост пространства состояний;
- невозможность полного перебора;
- необходимость эвристик и приближённых методов.

Применяемые подходы:

- поиск с эвристикой;
- локальный поиск;
- генетические алгоритмы;
- алгоритмы с отсечениями.

Заключение: Жадные методы обеспечивают эффективные решения для задач оптимизации при выполнении определённых условий. Информированный и неинформированный поиск представляют различные подходы к исследованию пространства состояний, а алгоритмы поиска на сложных пространствах используются для решения задач, где прямой перебор невозможен из-за высокой размерности.

30. Генетические алгоритмы

Генетические алгоритмы — это эвристические методы поиска и оптимизации, основанные на моделировании процессов естественного отбора и эволюции.

Основные понятия

- **популяция** — множество возможных решений задачи;
- **особь** — отдельное решение;
- **хромосома** — кодированное представление решения;
- **функция приспособленности** — количественная оценка качества решения.

Основные этапы генетического алгоритма

1. **Инициализация популяции** - Формирование начального набора случайных решений.
2. **Оценка приспособленности** -Вычисление значения функции приспособленности для каждой особи.
3. **Селекция** - Отбор лучших особей для размножения.
4. **Скрещивание (кроссовер)** - Обмен генетическим материалом между особями.
5. **Мутация** - Случайное изменение отдельных генов.
6. **Формирование нового поколения** - Замена старой популяции новой

Процесс повторяется до достижения критерия остановки.

Характеристики

- не гарантируют нахождение точного оптимума;
- хорошо работают в задачах с большим пространством решений;
- устойчивы к локальным минимумам.

Области применения

- задачи оптимизации;
- планирование;
- машинное обучение;
- задачи поиска на сложных пространствах.

31. Метод преобразования. Задача проверки единственности элементов массива

Метод преобразования заключается в предварительном изменении представления задачи или входных данных с целью упрощения или ускорения её решения.

Задача проверки единственности элементов массива

Требуется определить, все ли элементы массива различны.

Решение без преобразования

Прямой перебор всех пар элементов массива.

Сложность:

- время — **$O(n^2)$**;
- память — **$O(1)$**.

Решение с использованием метода преобразования

Выполняется **преобразование массива путём сортировки**:

1. массив сортируется;
2. проверяются только соседние элементы.

Сложность после преобразования

- сортировка — **$O(n \log n)$**;
- проверка — **$O(n)$**;
- итоговая сложность — **$O(n \log n)$**.

Смысл преобразования

- уменьшение количества сравнений;
- упрощение проверки условия;
- выигрыш во времени за счёт предварительного преобразования данных.

Заключение: Генетические алгоритмы являются эвристическими методами решения задач оптимизации на сложных пространствах. Метод преобразования позволяет существенно повысить эффективность алгоритмов, что наглядно демонстрируется на задаче проверки единственности элементов массива.

32. Метод преобразования. Метод исключения Гаусса

Метод преобразования — это метод разработки алгоритмов, при котором исходная задача предварительно преобразуется к эквивалентной, но более простой или удобной для решения форме. Целью преобразования является снижение вычислительной сложности, упрощение алгоритма или повышение численной устойчивости.

Метод исключения Гаусса — это алгоритм решения системы линейных алгебраических уравнений, основанный на преобразовании исходной системы к эквивалентной системе с верхнетреугольной матрицей коэффициентов.

Идея метода

Система $Ax=b$ с помощью элементарных преобразований строк приводится к треугольному виду, после чего решение находится методом обратной подстановки.

Этапы алгоритма

1. **Прямой ход** — последовательное исключение неизвестных путём обнуления элементов под главной диагональю.
2. **Обратный ход** — вычисление значений неизвестных, начиная с последнего уравнения.

Выбор ведущего элемента

Для повышения численной устойчивости используется **выбор ведущего элемента**, то есть перестановка строк для выбора максимального по модулю коэффициента.

Сложность

- временная сложность — $\mathbf{O(n^3)}$;
- пространственная сложность — $\mathbf{O(n^2)}$.

33. Метод преобразования. LU-разложение матрицы

LU-разложение — это метод преобразования матрицы коэффициентов A в произведение двух треугольных матриц:

A = L * U,

где

L — нижнетреугольная матрица,

U — верхнетреугольная матрица.

Смысл преобразования

Исходная задача решения системы Ax=b сводится к решению двух более простых систем:

1. Ly=b — прямая подстановка;
2. Ux=y — обратная подстановка.

Преимущества

- разложение выполняется один раз;
- при изменении правой части b решение находится быстрее;
- эффективно при многократном решении систем с одной и той же матрицей.

Сложность

- построение LU-разложения — **$O(n^3)$**;
- решение системы после разложения — **$O(n^2)$**.

34. Задача вычисления значения полинома в заданной точке

Постановка задачи

Требуется вычислить значение полинома
$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$
в заданной точке x_0.

Наивный способ

Прямое вычисление степеней x_0^k и умножение на коэффициенты.

Сложность:

- время — **O(n²)**.

Метод преобразования. Схема Горнера

Полином преобразуется к вложенной форме:

$$P(x) = (\ldots((a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_0.$$

После преобразования значение полинома вычисляется последовательным циклом.

Преимущества схемы Горнера

- снижение числа операций;
- упрощение вычислений;
- повышение эффективности.

Сложность:

- время — **O(n)**;
- память — **O(1)**.

Заключение: Метод преобразования позволяет существенно повысить эффективность алгоритмов. Методы исключения Гаусса и LU-разложение демонстрируют преобразование задач линейной алгебры, а схема Горнера — преобразование формы представления полинома для эффективного вычисления его значения.

35. Пирамидальная сортировка

Пирамидальная сортировка (Heap Sort)

Пирамидальная сортировка — это алгоритм сортировки, основанный на использовании структуры данных **пирамида (куча)** и относящийся к методам сортировки на основе выбора.

Идея алгоритма

Массив преобразуется в **двоичную кучу** (обычно максимальную), после чего:

- на каждом шаге максимальный элемент (корень кучи) помещается в конец массива;
- размер кучи уменьшается;
- свойство кучи восстанавливается.

Этапы алгоритма

1. **Построение пирамиды** из исходного массива.
2. **Многократное извлечение максимального элемента** и восстановление пирамиды.

Сложность

- построение пирамиды — **O(n)**;
- сортировка — **O(n log n)**;
- общая временная сложность — **O(n log n)**;
- дополнительная память — **O(1)**.

Особенности

- сортировка на месте;
- неустойчива;
- гарантированная сложность **O(n log n)** в худшем случае.

Заключение: пирамидальная сортировка является эффективным алгоритмом с гарантированной асимптотической сложностью и широко применяется, когда требуется сортировка без использования дополнительной памяти.

36. Методы локального поиска

Методы локального поиска — это эвристические методы поиска, применяемые для решения задач оптимизации на больших и сложных пространствах состояний.

Основная идея

- поиск начинается с некоторого начального решения;
- на каждом шаге выполняется переход к **соседнему состоянию**;
- выбор следующего состояния основан на локальном улучшении целевой функции.

Характерные особенности

- используется **одно текущее решение**, а не дерево поиска;
- память используется эффективно;
- возможна остановка в **локальном оптимуме**.

Примеры методов локального поиска

- подъём на холм;
- случайный локальный поиск;
- поиск с возвратами.

(В лекциях упоминаются как общий класс методов без углубления в реализацию.)

Преимущества

- применимы к задачам с огромным пространством решений;
- просты в реализации;
- работают быстрее полного перебора.

Недостатки

- отсутствие гарантии нахождения глобального оптимума;
- зависимость от начального состояния.

Заключение: Методы локального поиска используются для приближённого решения задач оптимизации, когда полный перебор невозможен. Они являются важным инструментом поиска на сложных пространствах состояний.

37. Поиск с применением эвристики

Поиск с применением эвристики — это метод поиска решений, при котором используется **эвристическая функция**, оценивающая близость текущего состояния к целевому. Эвристика направляет поиск и позволяет существенно сократить количество рассматриваемых состояний.

Эвристическая функция

Эвристическая функция h(n) — это приближённая оценка стоимости пути от текущего состояния n до цели.

Свойства эвристики:

- не является точной;
- вычисляется быстро;
- отражает «близость» к цели.

Особенности эвристического поиска

- поиск становится **направленным**;
- снижается число рассматриваемых состояний;
- эффективность зависит от качества эвристики.

Преимущества и недостатки

Преимущества:

- высокая практическая эффективность;
- применимость к задачам с большим пространством состояний.

Недостатки:

- отсутствие универсальной гарантии оптимальности;
- сильная зависимость от выбора эвристики.

Заключение: Поиск с применением эвристики используется в задачах, где полный перебор невозможен, и позволяет получать решения за приемлемое время за счёт направленного поиска.

38. Алгоритм A* и его модификации

Алгоритм A* — это алгоритм информированного поиска, сочетающий стоимость пути от начальной вершины и эвристическую оценку расстояния до цели.

Используется оценочная функция:

$f(n) = g(n) + h(n)$,

где

$g(n)$ — стоимость пути от начальной вершины до вершины n,

$h(n)$ — эвристическая оценка расстояния до цели.

Идея алгоритма

- на каждом шаге выбирается вершина с минимальным значением $f(n)$;
- поиск направляется к цели с учётом уже пройденного пути;
- используется при поиске кратчайших путей.

Свойства

- при допустимой эвристике алгоритм находит **оптимальный путь**;
- эффективнее неинформированных методов;
- требует хранения множества вершин в памяти.

Модификации алгоритма A*

В лекциях упоминаются модификации A*, направленные на:

- уменьшение потребления памяти;
- ускорение работы;
- ослабление требований к оптимальности.

(Конкретные реализации приводятся без углубления в детали.)

Заключение: Алгоритм A* является базовым алгоритмом эвристического поиска и широко применяется для решения задач поиска путей. Его модификации позволяют адаптировать алгоритм под различные ограничения по времени и памяти.

39. Деревья решений

Дерево решений — это иерархическая модель принятия решений, представленная в виде ориентированного дерева, в котором:

- внутренние вершины соответствуют проверке условия;
- рёбра — возможным результатам проверки;
- листья — итоговым решениям или ответам.

Назначение деревьев решений

Деревья решений используются для:

- принятия решений;
- классификации;
- выбора действий в зависимости от условий.

Особенности

- путь от корня к листу соответствует последовательности решений;
- каждая вершина обрабатывается один раз;
- дерево легко интерпретируется.

Преимущества и недостатки

Преимущества:

- наглядность;
- простота реализации;
- интерпретируемость.

Недостатки:

- возможный рост размера дерева;
- чувствительность к выбору условий.

Заключение: Деревья решений являются удобным способом представления логики принятия решений и применяются в задачах классификации и выбора.

40. Деревья

Дерево — это нелинейная структура данных, представляющая собой связный ациклический граф с выделенной вершиной — **корнем**.

Основные элементы дерева

- корень;
- вершины;
- рёбра;
- листья;
- поддеревья.

Свойства дерева

- между любыми двумя вершинами существует единственный путь;
- дерево с n вершинами содержит n-1 рёбер;
- отсутствие циклов.

Виды деревьев

- бинарные деревья;
- деревья поиска;
- сбалансированные деревья;
- пирамиды (кучи).

Обходы дерева

- прямой (preorder);
- симметричный (inorder);
- обратный (postorder).

Назначение деревьев

Деревья используются для:

- хранения и поиска данных;
- представления иерархий;
- реализации алгоритмов поиска и сортировки.

Заключение: Деревья являются одной из базовых нелинейных структур данных и широко применяются в алгоритмах и структурах данных для эффективной организации информации. Если хочешь, дальше могу продолжить **билеты 41–56** в том же экзаменационном формате.

41. Алгоритмы на графах

Граф — это структура данных, состоящая из множества вершин и множества рёбер, соединяющих пары вершин. Графы могут быть ориентированными и неориентированными, взвешенными и невзвешенными.

Основные задачи на графах

Алгоритмы на графах предназначены для решения следующих задач:

- обход графа;
- поиск путей;
- поиск кратчайших путей;
- построение остовных деревьев;
- анализ связности.

Основные алгоритмы

К базовым алгоритмам на графах относятся:

- поиск в глубину (DFS);
- поиск в ширину (BFS);
- алгоритм Дейкстры;
- алгоритмы Прима и Крускала;
- алгоритмы Флойда и Уоршелла.

Сложность

Сложность графовых алгоритмов зависит от:

- числа вершин V;
- числа рёбер E;
- способа представления графа.

Типичная сложность обхода графа:

O(V + E)

Заключение: Алгоритмы на графах являются основой для анализа сложных систем и широко применяются в задачах маршрутизации, сетевого анализа и оптимизации.

42. Эволюция графов

Эволюция графов изучает процессы изменения структуры графа во времени, такие как добавление и удаление вершин и рёбер.

Основные модели эволюции графов

В лекциях рассматриваются следующие модели:

- **случайные графы**;
- **модель малого мира**;
- **модель предпочтительного присоединения**.

Характерные свойства эволюционирующих графов

- рост числа вершин;
- неравномерное распределение степеней вершин;
- появление кластеров и сообществ;
- устойчивость структуры к случайным воздействиям.

Назначение моделей эволюции

Модели эволюции графов применяются для:

- анализа социальных сетей;
- моделирования информационных и коммуникационных систем;
- исследования сложных сетей.

Заключение: Эволюция графов позволяет понять закономерности формирования и развития сложных сетевых структур и является важной частью теории графов и анализа сложных систем.

43. Математический анализ нерекурсивных алгоритмов

Математический анализ нерекурсивных алгоритмов направлен на определение их **временной и пространственной сложности** на основе подсчёта количества выполняемых элементарных операций.

Основные приёмы анализа

- анализ последовательных операторов;
- анализ циклов;
- анализ вложенных циклов;
- использование сумм и оценок роста функций.

Анализ циклов

- один цикл с n итерациями → **O(n)**;
- вложенные циклы → произведение количества итераций;
- циклы с изменяющимся шагом (деление на 2, умножение на 2) → **O(log n)**.

Принципы анализа

- учитывается доминирующий член;
- отбрасываются константы и младшие члены;
- анализ проводится для худшего, среднего и лучшего случая.

Заключение: Математический анализ нерекурсивных алгоритмов позволяет формально оценить эффективность алгоритмов без использования рекурсии и служит основой для сравнения алгоритмов.

44. Математический анализ рекурсивных алгоритмов

Рекурсивные алгоритмы характеризуются самовызовом функции и описываются с помощью **рекуррентных соотношений**.

Рекуррентное соотношение

Временная сложность рекурсивного алгоритма задаётся в виде:

$T(n) = aT(n/b) + f(n)$,

где:

- a — число рекурсивных вызовов;
- n/b — размер подзадачи;
- f(n) — стоимость нерекурсивной части.

Методы анализа

- метод подстановки;
- метод итераций (развёртывания);
- метод рекурсивного дерева;
- основная теорема (Master Theorem).

Заключение: Математический анализ рекурсивных алгоритмов позволяет определить асимптотическую сложность алгоритмов, построенных по методу декомпозиции, и является важным инструментом теории алгоритмов.

45. Методы решения системы СЛАУ

Система линейных алгебраических уравнений (СЛАУ)

СЛАУ имеет вид:

$Ax = b$,

где A — матрица коэффициентов, x — вектор неизвестных, b — вектор свободных членов.

Основные методы решения СЛАУ

В лекциях рассматриваются следующие методы:

1. **Метод исключения Гаусса**

   Преобразует систему к треугольному виду с последующей обратной подстановкой.

2. **LU-разложение**

   Представляет матрицу в виде произведения $A = LU$ и позволяет эффективно решать систему при различных правых частях.

Характеристики методов

- прямые методы;
- имеют вычислительную сложность $\mathbf{O(n^3)}$;
- чувствительны к численной устойчивости.

Заключение: Методы решения СЛАУ, основанные на преобразовании матрицы, являются базовыми численными методами и широко применяются при решении инженерных и вычислительных задач.

46. Задача решения нелинейного уравнения

Постановка задачи

Требуется найти корень нелинейного уравнения

$f(x)=0$,

то есть значение x, при котором функция обращается в ноль.

Основные подходы решения

Задача решается **численными методами**, так как аналитическое решение часто невозможно.

В лекциях рассматриваются итерационные методы, основанные на последовательном уточнении приближений к корню.

Общие свойства методов

- используются начальные приближения;
- строится последовательность значений, сходящаяся к корню;
- важны условия сходимости.

Особенности

- решение является приближённым;
- точность зависит от шага и критерия остановки;
- возможна зависимость от начального приближения.

Заключение: Задача решения нелинейного уравнения относится к численным методам и решается путём итерационного приближения корня с заданной точностью

47. Задача нахождения площади под кривой

Постановка задачи

Требуется вычислить площадь фигуры, ограниченной графиком функции y=f(x), осью абсцисс и заданным интервалом [a,b].

Численный подход

Площадь вычисляется как приближённое значение определённого интеграла:

$$\int_a^b f(x)\,dx.$$

Используются численные методы интегрирования, основанные на разбиении интервала на малые отрезки.

Особенности

- точность зависит от числа разбиений;
- применяется приближённое вычисление;
- метод универсален для произвольных функций.

Заключение: Задача нахождения площади под кривой решается методами численного интегрирования и является типичной задачей вычислительной математики.

48. Задача сортировки

Постановка задачи

Требуется упорядочить элементы массива по заданному критерию (возрастание или убывание).

Основные подходы

В лекциях задача сортировки рассматривается как базовая алгоритмическая задача и решается различными методами:

- сортировка выбором;
- пузырьковая сортировка;
- сортировка вставкой;
- быстрая сортировка;
- пирамидальная сортировка;
- сортировка слиянием.

Критерии сравнения алгоритмов

- временная сложность;
- потребление памяти;
- устойчивость;
- простота реализации.

Заключение: Задача сортировки является фундаментальной задачей алгоритмов и используется как основа для сравнения эффективности различных методов.

49. Задача поиска

Постановка задачи

Требуется определить наличие элемента в структуре данных и, при необходимости, найти его позицию.

Основные методы поиска

В материалах рассматриваются:

- последовательный поиск;
- бинарный поиск;
- поиск на графах;
- поиск в пространстве состояний.

Особенности

- выбор метода зависит от структуры данных;
- для отсортированных данных применяются более эффективные методы;
- сложность поиска варьируется от **O(1)** до **O(n)** и выше.

Заключение: Задача поиска является одной из базовых задач алгоритмов. Эффективность её решения определяется выбором структуры данных и используемого алгоритма.

50. Задача о ферзях

Постановка задачи

Задача о ферзях заключается в размещении n ферзей на шахматной доске n * n так, чтобы ни один ферзь не бил другой.

Характер задачи

Задача относится к задачам **поиска на сложных пространствах состояний** и имеет комбинаторный характер.

Подход к решению

Решение задачи осуществляется методом **поиска с возвратом (backtracking)**:

- ферзи размещаются последовательно по строкам или столбцам;
- при нарушении условия допустимости выполняется возврат;
- перебираются допустимые варианты размещения.

Особенности

- перебор имеет экспоненциальную сложность;
- применяется отсечение недопустимых вариантов;
- решение не всегда единственно.

Заключение: Задача о ферзях является классическим примером задачи поиска с возвратом и используется для демонстрации методов перебора и отсечения.

51. Задача решения вырожденных линейных уравнений

Постановка задачи

Рассматривается система линейных алгебраических уравнений, для которой матрица коэффициентов является **вырожденной**, то есть её определитель равен нулю.

Особенности вырожденных систем

- система может не иметь решений;
- может иметь бесконечно много решений;
- возможна линейная зависимость уравнений.

Подход к решению

Для анализа вырожденных систем используется **метод преобразования**:

- приведение системы к ступенчатому виду;
- анализ ранга матрицы коэффициентов и расширенной матрицы;
- выявление совместности системы.

Результаты анализа

- если ранг матрицы коэффициентов меньше ранга расширенной матрицы — система несовместна;
- если ранги равны и меньше числа неизвестных — система имеет бесконечно много решений.

Заключение: Задача решения вырожденных линейных уравнений требует анализа структуры системы и использования преобразований матрицы для определения существования и множества решений.

52. Задача вырождения дерева

Вырождение дерева — это ситуация, когда (обычно бинарное дерево поиска) теряет «деревообразную» форму и превращается почти в **линейный список**, то есть его высота h становится близкой к n. Тогда операции поиска/вставки/удаления, которые зависят от высоты, ухудшаются до линейных.

В чём состоит задача

Задача вырождения дерева — **распознать и учитывать** этот худший случай, потому что:

- операции в BST имеют сложность **O(h)**,
- в лучшем случае h = O(\log n),
- в худшем (при вырождении) h = O(n).

Почему происходит вырождение

Типичный источник — **неудачный порядок вставок** (например, почти отсортированные данные), из-за чего вершины добавляются «в одну сторону», и дерево становится сильно несбалансированным (высота растёт). (Идея следует из зависимости операций от высоты h.)

Как предотвращают (по лекции)

Для предотвращения вырождения используются **сбалансированные деревья поиска**:

- балансировка (AVL, красно-чёрные, splay и т.п.),
- либо увеличение числа ключей в вершине (2-3-деревья, B-деревья и др.).

Вывод: Вырождение дерева опасно тем, что превращает эффективные операции BST из логарифмических в линейные, поэтому на практике применяют самобалансирующиеся структуры, чтобы сохранять малую высоту.

53. Линейное программирование как приведение задачи

Понятие линейного программирования

Линейное программирование — это метод решения оптимизационных задач, в которых:

- целевая функция является линейной;
- ограничения задаются системой линейных уравнений и неравенств.

Линейное программирование как приведение задач

В рамках курса линейное программирование рассматривается **как метод приведения сложной задачи** к стандартной форме оптимизационной задачи.

Суть подхода:

- исходная задача преобразуется;
- формулируется целевая функция;
- ограничения записываются в линейном виде;
- далее применяется стандартный метод решения.

Таким образом, сложная задача сводится к задаче линейного программирования, для которой существуют универсальные методы решения.

Назначение приведения

Приведение задачи к линейному программированию позволяет:

- использовать готовые алгоритмические методы;
- формально описывать условия и ограничения;
- получать оптимальные решения в рамках линейной модели.

Заключение: Линейное программирование выступает как универсальный инструмент приведения и решения задач оптимизации путём их преобразования к линейной форме.

54. Методы проектирования алгоритмов

Понятие метода проектирования алгоритмов

Методы проектирования алгоритмов — это обобщённые подходы к построению алгоритмов, определяющие способ перехода от постановки задачи к алгоритмическому решению.

Основные методы проектирования

В курсе рассматриваются следующие методы:

1. **Метод грубой силы**

   Полный перебор всех возможных вариантов.

2. **Метод декомпозиции (разделяй и властвуй)**

   Разбиение задачи на подзадачи меньшего размера.

3. **Метод уменьшения размера задачи**

   Последовательное сведение задачи к более простой.

4. **Динамическое программирование**

   Запоминание результатов подзадач.

5. **Жадные методы**

   Выбор локально оптимального решения.

6. **Метод преобразования**

   Предварительное изменение задачи или данных.

Роль методов проектирования

Методы проектирования алгоритмов:

- упрощают разработку алгоритмов;
- позволяют систематизировать решения;
- обеспечивают анализ эффективности и корректности.

Заключение: Методы проектирования алгоритмов являются фундаментальной частью теории алгоритмов и служат основой для построения корректных и эффективных

53. Линейное программирование как приведение задачи

Линейное программирование — это метод решения оптимизационных задач, в которых:

- целевая функция является линейной;
- ограничения задаются системой линейных уравнений и неравенств.

Линейное программирование как приведение задачи

В рамках курса линейное программирование рассматривается **как метод приведения сложной задачи** к стандартной форме оптимизационной задачи.

Суть подхода:

- исходная задача преобразуется;
- формулируется целевая функция;
- ограничения записываются в линейном виде;
- далее применяется стандартный метод решения.

Таким образом, сложная задача сводится к задаче линейного программирования, для которой существуют универсальные методы решения.

Назначение приведения

Приведение задачи к линейному программированию позволяет:

- использовать готовые алгоритмические методы;
- формально описывать условия и ограничения;
- получать оптимальные решения в рамках линейной модели.

Заключение: линейное программирование выступает как универсальный инструмент приведения и решения задач оптимизации путём их преобразования к линейной форме.

54. Методы проектирования алгоритмов

Методы проектирования алгоритмов — это обобщённые подходы к построению алгоритмов, определяющие способ перехода от постановки задачи к алгоритмическому решению.

Основные методы проектирования

В курсе рассматриваются следующие методы:

1. **Метод грубой силы**

   Полный перебор всех возможных вариантов.

2. **Метод декомпозиции (разделяй и властвуй)**

   Разбиение задачи на подзадачи меньшего размера.

3. **Метод уменьшения размера задачи**

   Последовательное сведение задачи к более простой.

4. **Динамическое программирование**

   Запоминание результатов подзадач.

5. **Жадные методы**

   Выбор локально оптимального решения.

6. **Метод преобразования**

   Предварительное изменение задачи или данных.

Роль методов проектирования

Методы проектирования алгоритмов:

- упрощают разработку алгоритмов;
- позволяют систематизировать решения;
- обеспечивают анализ эффективности и корректности.

Заключение: Методы проектирования алгоритмов являются фундаментальной частью теории алгоритмов и служат основой для построения корректных и эффективных алгоритмических решений.