# ООП

Лекция 2 | Алгоритмы и их сложность

Лектор: Киселев Станислав Дмитриевич

РЕКЛАМНАЯ ПАУЗА

# Intern Sprint: ботаем алгоритмы и готовимся к стажам

**Команда ACM MISIS подготовила крутые активности**

**До 23 февраля (вс)** будет открыт онлайн-контест, специально для которого нашли задачи с алгосекций бигтехов

**26 февраля** тренеры яндексоиды проведут разбор задач и ответят на вопросы про стажировки

Среди ТОП-10 участников, разыграем возможность поучаствовать в **тестовом собеседовании**

QR на пост в сообществе ITAM

**Больше информации и ссылки на чаты в посте от ITAM**

# Как выбрать язык программирования? Продолжение

**Языки программирования** – это инструменты для решения разных задач

**Параметры для выбора, на которые стоит обратить внимание:**

1. Задача проекта
2. Ваш опыт
3. Сообщество
4. Доступность ресурсов
5. Производительность
6. Ключевые моменты каждого языка

# Rust

## ! Безопасность и производительность

Rust стремится обеспечить безопасность памяти и потоков на уровне компиляции без потери производительности, характерной для C++

**Высокая производительность**

Сравнима с C++, отсутствие сборщика мусора обеспечивает предсказуемое время выполнения

**Высочайшая безопасность памяти**

благодаря системе владения и заимствования

Безопасная многопоточность

**Надежность**

Строгая система типов и компилятор помогают выявлять ошибки на этапе компиляции

«Zero-Cost» Шаблоны

**Современный язык**

Современный синтаксис, удобные инструменты сборки и управления зависимостями

# Rust | Применимость

- **Системное программирование**
  (операционные системы, драйверы устройств)
- **Высокопроизводительные сетевые сервисы**
- **Встраиваемые системы и разработка для ограниченных ресурсов**
- **Приложения, где безопасность и надежность критически важны**
  (финансовые системы, криптография)
- **Backend веб-разработка с акцентом на производительность и безопасность**

# Python

## ! Простота, читаемость кода и скорость разработки

Python известен своим лаконичным синтаксисом и богатой стандартной библиотекой, что ускоряет разработку

**Простота изучения и использования**

Легкий синтаксис, отлично подходит для начинающих

**Скорость разработки**

Высокая продуктивность благодаря динамической типизации и большой экосистеме библиотек

**Большая экосистема**

Огромное количество библиотек и фреймворков для различных задач

**Кросс-платформенность**

Работает на большинстве операционных систем

# Python | Применимость

- **Разработка на основе библиотек языков «низкого уровня», например C++, C, Rust, Zig**
- **Наука о данных и машинное обучение**
  (библиотеки NumPy, Pandas, Scikit-learn, TensorFlow, PyTorch)
- **Скрипты автоматизации и утилиты**
- **Прототипирование и быстрое создание MVP (Minimum Viable Product)**
- **Образование и изучение программирования**

# Golang

**! Производительность, конкурентность и простота развертывания**

Go разработан Google для создания эффективных и масштабируемых сетевых приложений

**Хорошая производительность,**
хотя ниже, чем у C++ и Rust

Компилируемый язык, эффективная работа с памятью (сборщик мусора)

**Конкурентность**

Встроенная поддержка легковесного исполнения фоновых задач делает конкурентное программирование простым и эффективным

**Простота и читаемость**

Простой синтаксис, легко читать и поддерживать код

**Очень быстрая компиляция,**
что ускоряет итерации разработки

# Golang | Применимость

- **Сетевое программирование, где нет ограничений в ресурсах**
  (веб-серверы, API, микросервисы)
- **Облачные вычисления и инфраструктурное программное обеспечение**
- **Инструменты командной строки и утилиты**

# C++

**! Максимальная производительность и контроль**

C++ предлагает прямой доступ к аппаратному обеспечению и тонкий контроль над ресурсами, что делает его идеальным для требовательных к производительности приложений

**Наивысшая возможная производительность**

Прямой доступ к памяти, ручное управление памятью

**Контроль**

Полный контроль над аппаратным обеспечением и ресурсами системы

**Зрелая экосистема**

Много библиотек и фреймворков, хотя и менее интегрированных, чем в Python или Go

**Совместимость со старым кодом на C**

Важно для поддержки и расширения унаследованных систем

# C++ | Применимость

- **Системное программирование**
  (операционные системы, драйверы, компиляторы)
- **Разработка игр**
- **Высокопроизводительные приложения**
  (финансовые системы, научные вычисления)
- **Встраиваемые системы с жесткими ограничениями по ресурсам**
- **Приложения, где критически важна каждая доля производительности и контроль над ресурсами**

# Zig

**! Безопасность, контроль и простота, «C done right»**

Zig стремится заменить C как лучший язык для системного программирования, но предлагая ручное управление памятью с улучшенной безопасностью и современными инструментами

**Производительность и безопасность**

Ручное управление памятью, механизмы для обнаружения ошибок на этапе компиляции и во время выполнения

**Простота и читаемость**

Более простой синтаксис, чем C++ и Rust, нацеленный на уменьшение сложности

**Субъективное мнение в сравнении**

**Интроспекция и метапрограммирование**

Возможности, позволяющие генерировать код и проверять типы

Быстрая компиляция, кросс-компиляция

**C-совместимость**

Хорошая совместимость с C кодом и библиотеками, возможность постепенной миграции с C

# Zig | Применимость

- **Системное программирование**
  (операционные системы, загрузчики, гипервизоры)
- **Встраиваемые системы и программирование для «микрокода»**
- **Высокопроизводительные приложения, где требуется ручное управление памятью и контроль**
- **Разработка игр и графических движков**
- **Замена C в проектах, где нужна большая безопасность и современность**

!

## Введение в STL

Компоненты стандартной библиотеки и ее важность

Мы здесь

## Нотация Big O

Распределение, примеры и применение

## Контейнеры

Описание основных типов и их особенностей

## Алгоритмы

Описание и примеры основных алгоритмов

## Оценка алгоритма

И общее заключение

# Структура библиотеки STL

Стандарт языка программирования C++ состоит из двух частей:

**Core Guidelines –** ядро языка, которое включает в себя указания по поводу того, каким должен быть корректный код и как он должен работать

**Стандартная библиотека шаблонов (англ. Standard Template Library, STL) –** набор классов и функций в C++, которые по умолчанию поставляются вместе с компилятором и решают большинство задач

**Примеры разделов стандартной библиотеки:**

| | | | | |
|---|---|---|---|---|
| **Контейнеры** | **Алгоритмы** | **Итераторы** | **Функциональные объекты** | **Утилиты** |

# Почему STL важно?

1. **«Middle»:** де-факто стандарт для всех разработчиков, имеющих опыт в индустрии
2. **Унификация кода:** весь код написан в одном стиле, который легко и удобно читать
   Внимание акцентируется на логике работы приложения, без привязки к конкретной реализации
3. **Использование готовых алгоритмов:** зачастую алгоритмы в стандарте языка имеют высокую производительность, сохраняя легкость их модификации и настройки под свои задачи
4. **Кросс-платформенность:** стандарт языка предполагает компиляцию на любой платформе
5. **Производительность «из коробки»:** начиная с C++20, алгоритмы над контейнерами поддерживают несколько видов параллельной обработки данных

# Контейнеры и алгоритмы STL

**Контейнеры** в C++ – структуры данных, которые нужны для хранения и организации элементов определенного типа

**Алгоритмы** в C++ – функции, которые могут выполнять различные базовые операции

Большинство алгоритмов STL подходит для любых контейнеров,
так как имеют одинаковый интерфейс

# Контейнеры STL

| | | | |
|---|---|---|---|
| vector | deque | list | map |
| multimap | unordered_map | set | unordered_set |
| stack | queue | priority_queue | |

# Алгоритмы STL

## Алгоритмы поиска

**Простой поиск**
find, find_if, find_if_not, adjacent_find

**Бинарный поиск (для отсортированных диапазонов)**
binary_search, lower_bound, upper_bound, equal_range

**Поиск подстроки**
search, find_end, find_first_of

**Поиск мин или макс элемента**
min_element, max_element, minmax_element

## Алгоритмы сравнения

**Сравнение диапазонов**
equal, mismatch, lexicographical_compare

## Алгоритмы копирования и перемещения

**Копирование**
copy, copy_n, copy_if, copy_backward

**Перемещение**
move, move_backward

# Алгоритмы STL

## Алгоритмы заполнения и генерации

**Заполнение значениями**
fill, fill_n

**Генерация значений**
generate, generate_n, iota

## Алгоритмы преобразования

**Преобразование элементов**
transform

## Алгоритмы удаления и замены

**Удаление по значению или условию**
remove, remove_if, remove_copy, remove_copy_if

**Удаление дубликатов (в отсортированном диапазоне)**
unique, unique_copy

**Замена значений**
replace, replace_if, replace_copy, replace_copy_if

# Алгоритмы STL

## Алгоритмы перестановки и изменения порядка

**Изменение порядка следования**
reverse, rotate, shuffle (перемешивание)

**Перестановки**
next_permutation, prev_permutation

## Алгоритмы сортировки

**Полная сортировка**
sort, stable_sort

**N-ый элемент**
nth_element (после частичной сортировки)

**Частичная сортировка**
partial_sort, partial_sort_copy

## Минимум и максимум

**Нахождение минимума и максимума**
min, max, minmax

**Сравнение и выбор наименьшего или наибольшего**
clamp (ограничение значения диапазоном)

# Алгоритмы STL

| Алгоритмы для работы с кучей (heap) | |
|---|---|
| **Создание кучи**<br>make_heap | **Добавление элемента в кучу**<br>push_heap |
| **Удаление элемента из кучи**<br>pop_heap | **Сортировка кучи**<br>sort_heap |
| **Проверка, является ли диапазон кучей**<br>is_heap, is_heap_until | |

# Алгоритмы STL

| Численные алгоритмы (из <numeric>) | |
|---|---|
| **Суммирование**<br>accumulate | **Внутреннее произведение**<br>inner_product |
| **Вычисление разностей**<br>adjacent_difference | **Частичные суммы**<br>partial_sum |
| **Операции сведения (reduction):**<br>reduce (C++17) | **Эксклюзивная и инклюзивная префиксная сумма**<br>exclusive_scan, inclusive_scan (C++17) |
| **Трансформированное сведение**<br>transform_reduce (C++17) | **Трансформированная префиксная сумма**<br>transform_exclusive_scan, transform_inclusive_scan (C++17) |

# Алгоритмы STL

| Операции с множествами (для отсортированных диапазонов) | |
|---|---|
| **Объединение**<br>set_union | **Пересечение**<br>set_intersection |
| **Разность**<br>set_difference | **Симметрическая разность**<br>set_symmetric_difference |
| **Включение**<br>includes | |

# Boost (Альтернативные алгоритмы)

### Общие утилиты:
- Boost.SmartPtr (умные указатели)
- Boost.NonCopyable (запрет копирования)
- Boost.TypeTraits (интроспекция типов)

### Строки и текст:
- Boost.Regex (регулярные выражения)
- Boost.StringAlgo (алгоритмы для строк)
- Boost.Format (форматирование строк)

### Контейнеры:
- Boost.Container (альтернативные STL контейнеры)
- Boost.Unordered (хэш-таблицы)
- Boost.Intrusive (intrusive контейнеры)
- Boost.MultiIndex (контейнеры с несколькими индексами)

### Математика и численные расчеты:
- Boost.Math (математические функции)
- Boost.Rational (рациональные числа)
- Boost.Multiprecision (числа произвольной точности)

### Алгоритмы:
- Boost.Algorithm (расширения алгоритмов)
- Boost.Sort (алгоритмы сортировки)

### Дата и время:
- Boost.DateTime (дата и время)
- Boost.Chrono (длительности и моменты времени)

### Параллелизм и конкурентность:
- Boost.Thread (потоки)
- Boost.Asio (асинхронный ввод-вывод)

### Тестирование:
- Boost.Test (фреймворк для тестирования)

### Графы и метапрограммирование:
- Boost.Graph (графы и алгоритмы)
- Boost.MPL (библиотека метапрограммирования)

## Введение в STL

Компоненты стандартной библиотеки и ее важность

## **Нотация Big O**

Распределение, примеры и применение

Мы здесь

## Контейнеры

Описание основных типов и их особенностей

## Алгоритмы

Описание и примеры основных алгоритмов

## Оценка алгоритма

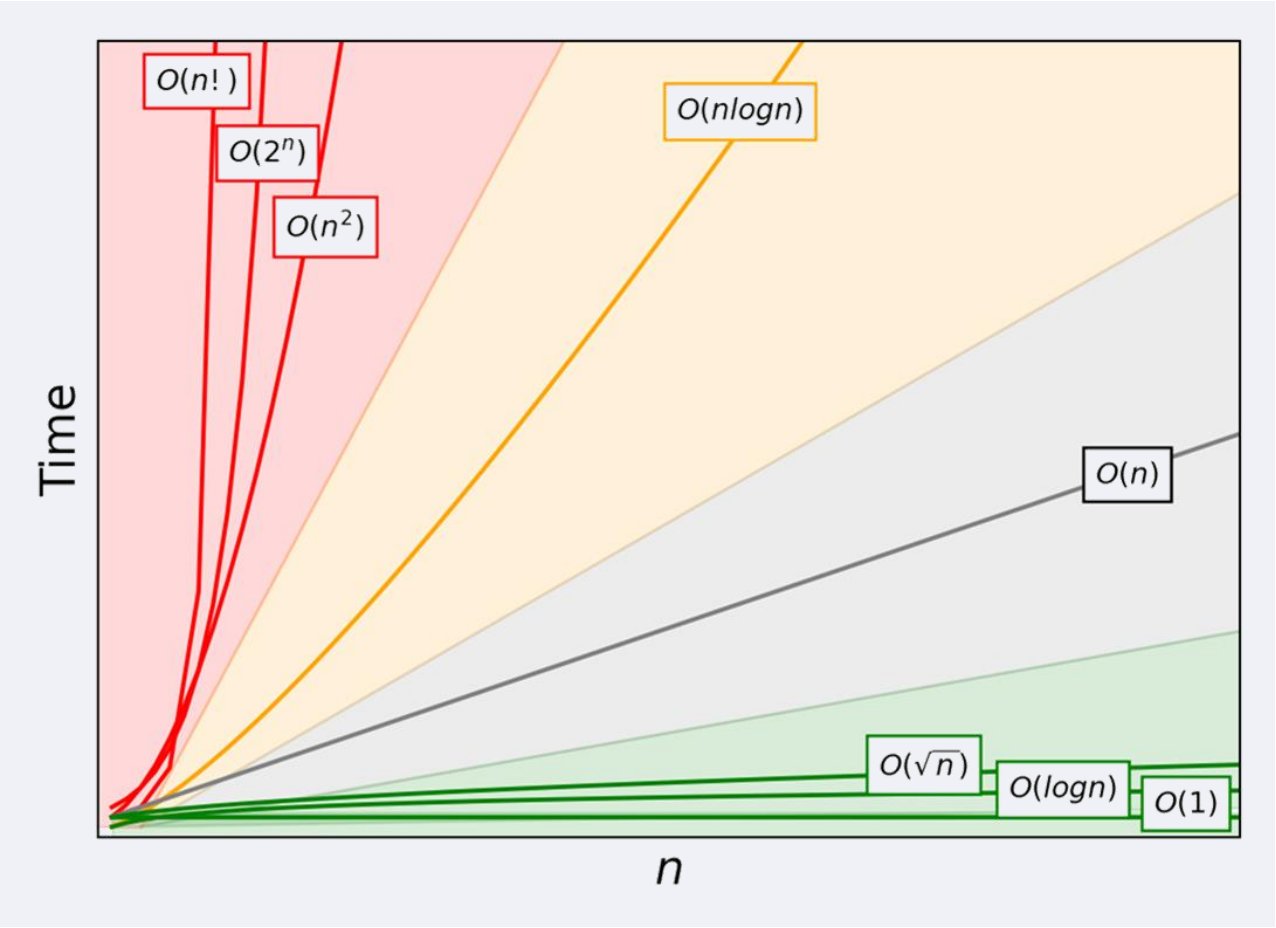
И общее заключение

# Big O

**Big O нотация –** способ описать, как **скорость роста** времени выполнения или памяти меняется в зависимости от размера входа

Big O **не измеряет** точное время в секундах или миллисекундах

А представляет собой **математически выраженную** сложность алгоритма

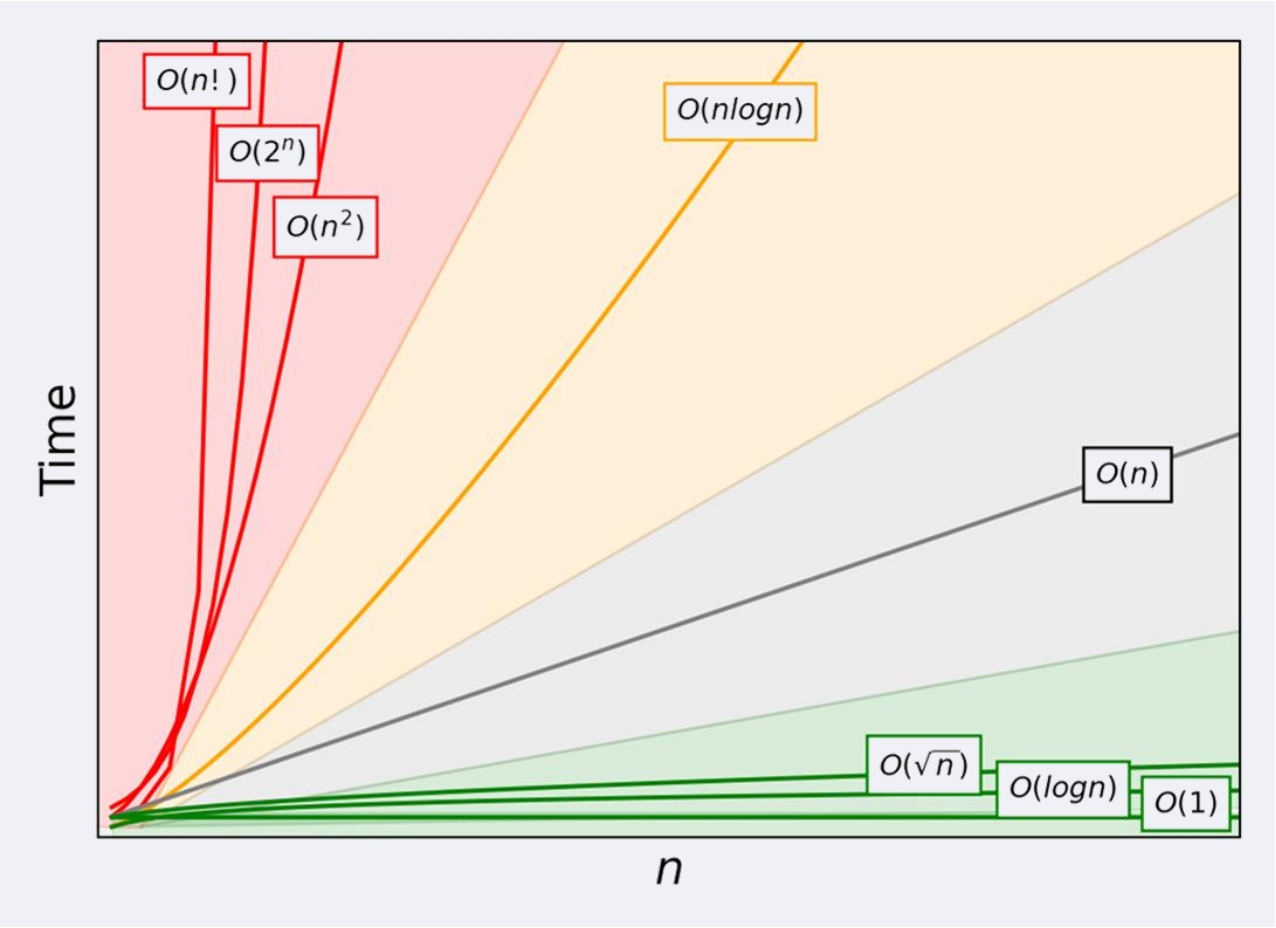


# Big O

**Big O –** асимптотическая нотация

Описывает поведение алгоритма
при очень больших размерах входных
данных (стремящихся к бесконечности)

Для малых входных данных различия в Big O
могут быть не так заметны



# Применение Big O

1. **Сравнение эффективности разных алгоритмов, в том числе оценка Open Source кода**
   Что позволяет выбрать алгоритм с лучшим Big O для больших наборов данных

2. **Прогнозирование производительности программы при увеличении входных данных**
   Это важно для оценки масштабируемости вашего решения

3. **Выявление «узких мест» в коде**
   Чтобы производительность не стала проблемой при больших объемах данных

4. **Выявление направление для оптимизации**
   Так как алгоритмы и части кода с худшим Big O оказывают наибольшее влияние на производительность

5. **«Общение о производительности» и прохождение технических собеседований**
   Big O - это стандартный язык для обсуждения производительности алгоритмов в информатике

# Big O – это язык эффективности

**Просто замерить скорость работы алгоритма недостаточно!**

На эффективность может влиять:

**«Железо»**
Локально на ноутбуке код может работать за 0.01 сек, а на сервере с гигабайтами данных – 10 часов

**Нерепрезентативность данных**
На маленьких данных все алгоритмы быстрые

**Случайность**
Время зависит от ОС, фоновых процессов, оптимизаций компилятора

# Примеры нотаций Big O

**O(1)**
Константное время

**O(log n)**
Логарифмическое время

**O(n)**
Линейное время

**O(n log n)**
Линеарифметическое время (или квазилинейное)

**$O(n^2)$**
Квадратичное время

**$O(2^n)$**
Экспоненциальное время

# O(1) – Константное время

Good

**Время выполнения не зависит от размера входных данных**

Алгоритм всегда выполняется за одно и то же время, независимо от того, сколько данных ему передано

**Пример алгоритмов:**

- **Доступ к элементу массива по индексу (array[index])**

# O(log n) – Логарифмическое время

Good

**С каждым удвоением размера входа, время выполнения увеличивается на константную величину**

Обычно встречается в алгоритмах, которые при обходе или поиске в массиве делят его пополам, выбирают нужную часть, делят уже ее снова на две части и так далее

Пока не доходят до нужного элемента

Алгоритмы подобной сложности легко встретить в древовидных отсортированных структурах, когда количество пройденных элементов каждый раз существенно уменьшается

**Пример алгоритмов:**

- **Бинарный поиск в отсортированном массиве**
  Каждый шаг поиска отбрасывает половину оставшейся области поиска
- **Поиск элемента в сбалансированном двоичном дереве поиска**

Normal

# O(n) – Линейное время:

**Время выполнения растет прямо пропорционально размеру входных данных**

Т.е. если входные данные увеличиваются в два раза, время выполнения тоже примерно удваивается

> Чтобы прочитать книгу полностью необходимо последовательно изучать каждую страницу от начала до конца
> Таким образом время, которое будет потрачено, прямо пропорционально количеству страниц в книге

**Пример алгоритмов:**

- **Линейный поиск в массиве (проверка каждого элемента по очереди)**
- **Нахождение максимального или минимального элемента в массиве**
- **Перебор всех элементов в std::vector или std::list в цикле**

# O(n log n) – Линеарифметическое время

So so

**Удвоение размера входных данных увеличит время выполнения чуть более, чем вдвое**

Получается в случаях, когда линейно повторяется «поиск по дереву»

Например, для перетасовки колоды карт методом слияния необходимо разделить колоду на части, отсортировать **каждую часть**, а затем соединить все вместе

**Пример алгоритмов:**

- **Эффективные алгоритмы сортировки, такие как std::sort**
  (обычно IntroSort, гибрид QuickSort, HeapSort, InsertionSort), Merge Sort, Heap Sort

# $O(n^2)$ – Квадратичное время

Bad

**Время выполнения растет пропорционально квадрату размера входных данных**

Часто встречается в алгоритмах со вложенными циклами, где для каждого элемента входных данных выполняется еще один проход по подмножеству данных

Предположим, надо найти количество дубликатов всех карточек в стопке – если алгоритм будет просто по одной сравнивать карточки со всеми остальными, не убирая дубликаты, то для выполнения такого алгоритма будет затрачено $n^2$ операций

**Пример алгоритмов:**

- **Простые, но неэффективные алгоритмы сортировки**
  (Пузырьковая сортировка, Сортировка вставками, Сортировка выбором)
- **Вложенные циклы, перебирающие все элементы двумерного массива**

# $O(2^n)$ – Экспоненциальное время:

Bad

**Время выполнения растет экспоненциально с увеличением размера входных данных**

Обычно встречается в алгоритмах, которые перебирают все возможные комбинации или перестановки входных данных

Примерно такая эффективность будет, если попытаться открыть чемодан с кодовым замком простым перебором всех возможных комбинаций цифр по порядку

**Пример алгоритмов:**

- **Поиск всех подмножеств множества**

## Введение в STL

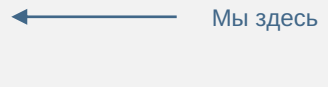
Компоненты стандартной библиотеки и ее важность

## Нотация Big O

Распределение, примеры и применение

## Контейнеры

Описание основных типов и их особенностей




## Алгоритмы

Описание и примеры основных алгоритмов

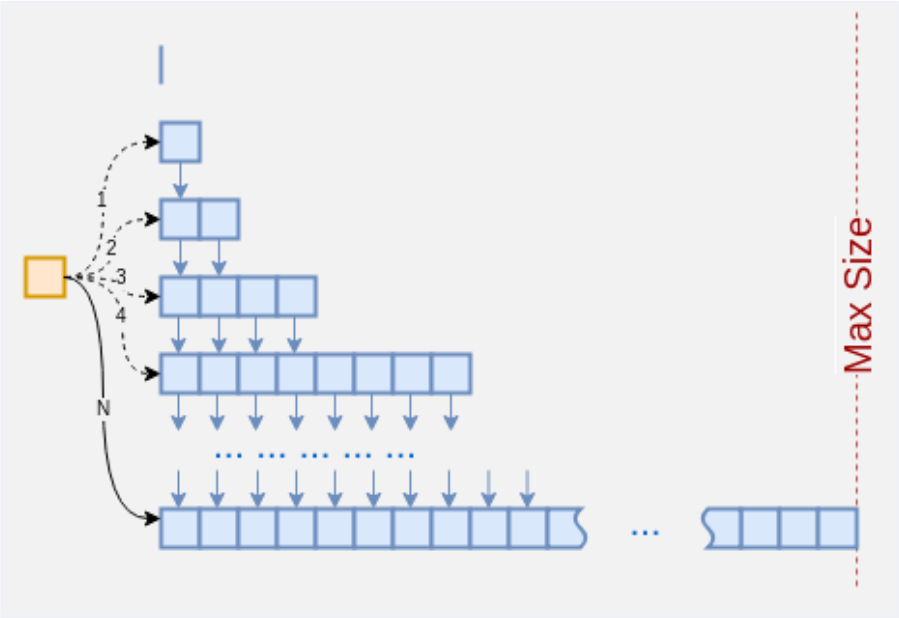
## Оценка алгоритма

И общее заключение

# Vector

**std::vector** в C++ представляет собой **динамический массив,** который обеспечивает эффективное хранение и доступ к элементам

Элементы **std::vector** хранятся в **непрерывном блоке памяти,** подобно обычному массиву C-style
Это означает, что в памяти элементы располагаются последовательно, но не подряд



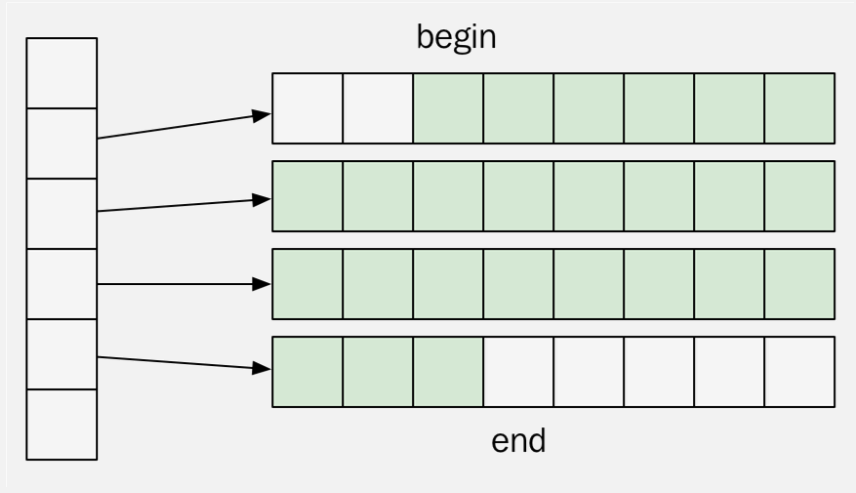

## Основные особенности:

- Динамические массивы
- Хранение последовательностей данных
- Реализация других структур данных:
  vector можно использовать как основу для реализации других структур данных, например, стека (через push_back и pop_back) или очереди (хотя для очереди лучше подходит std::deque)
- Возврат коллекций из функций

# Deque

**std::deque** – контейнер в C++ STL, который представляет собой **двустороннюю очередь**
Разработан для эффективной вставки и удаления элементов как в начале, так и в конце последовательности

**Сегментированное хранение** – не гарантирует непрерывное хранение всех элементов в одном блоке памяти
Вместо этого обычно реализуется как массив массивов




## Основные особенности:

- Состоит из нескольких отдельных блоков памяти (сегментов или «маленьких массивов»)
- Эти блоки не обязательно располагаются в памяти непрерывно друг за другом – std::deque управляет блоками с помощью индексной структуры
  Например, массива указателей, который позволяет отслеживать, где начинаются и заканчиваются блоки, и в каком блоке находится нужный элемент

# Deque | Использование

### Двусторонняя очередь
Когда необходимо эффективно добавлять и удалять элементы с обоих концов последовательности

### Очереди сообщений
В системах обмена сообщениями, где сообщения могут добавляться в конец очереди для обработки и извлекаться из начала очереди для отправки

### История операций (Undo и Redo)
Для реализации функциональности «отмены» и «повтора» действий пользователя
Новые операции добавляются в конец, а отмена операций может потребовать удаления с конца или начала истории

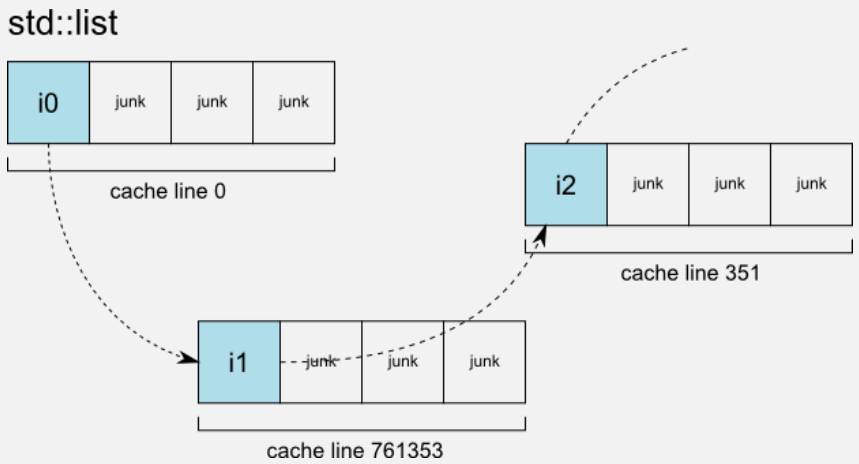
### Задачи, требующие быстрого добавления или удаления с обеих сторон
Например, в некоторых алгоритмах обработки графов или в задачах потоковой обработки данных

### Реализация адаптеров контейнеров std::queue и std::stack в STL
std::queue и std::stack: std::deque часто используется как базовый контейнер, так как он обеспечивает необходимые операции для очередей и стеков

# List

**std::list** в C++ – это контейнер, оптимизированный для эффективной вставки и удаления элементов в любом месте списка, но с менее эффективным доступом по индексу по сравнению с vector или deque

Хранится в памяти как **двусвязный список,** непрерывность не гарантируется




## Основные особенности:

- Связное хранение:
  Каждый элемент std::list хранится в отдельном узле, который содержит:
  - Данные элемента.
  - Указатель на следующий узел
  - Указатель на предыдущий узел
- Двусвязность:
  Каждый узел хранит указатели как на следующий, так и на предыдущий элемент. Это позволяет эффективно перемещаться по списку в обоих направлениях (вперед и назад)

# List | Особенности

### Производительность вставки и удаления
Вставка и удаление элементов в любой позиции (после нахождения позиции) выполняется за константное время **O(1)**

### Неэффективный доступ по индексу
Для доступа к элементу по индексу необходимо последовательно пройтись по списку от начала до нужной позиции, что занимает линейное время **O(n)**

### std::list не предоставляет оператор [ ] для прямого доступа по индексу
Для доступа к элементам обычно используются итераторы и последовательный обход.

### Больше накладных расходов на память
Из-за необходимости хранения указателей на следующий и предыдущий узлы для каждого элемента, std::list обычно требует больше памяти на элемент, чем std::vector или std::deque

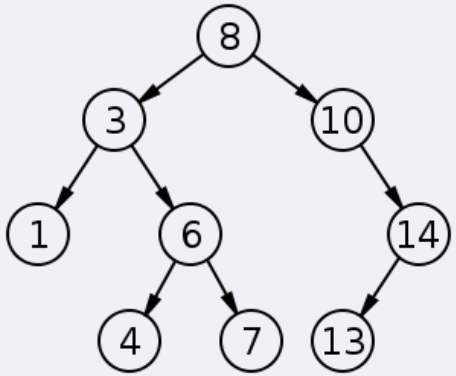
### std::list может быть использован для реализации различных абстрактных типов данных, где важна эффективность операций вставки и удаления в произвольных позициях

# Бинарное дерево поиска

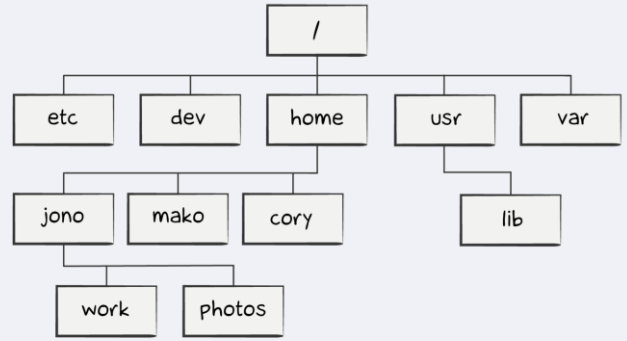
**Бинарное дерево поиска** – структура данных для работы с отсортированным множеством

Бинарное дерево поиска обладает следующим свойством:
если x – узел бинарного дерева с ключом k, то все узлы в левом поддереве должны иметь ключи, меньшие k, а в правом поддереве большие k

## Математическое представление дерева




## Пример из жизни (файловая система)



# Бинарное дерево поиска на C++

```
// Node structure for a Binary Search Tree
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Function to create a new Node
Node* createNode(int data)
{
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = nullptr;
    return newNode;
}
```

```
// Function to insert a node in the BST
Node* insertNode(Node* root, int data)
{
    if (root == nullptr) { // If the tree is empty, return a
                           // new node
        return createNode(data);
    }

    // Otherwise, recur down the tree
    if (data < root->data) {
        root->left = insertNode(root->left, data);
    }
    else if (data > root->data) {
        root->right = insertNode(root->right, data);
    }

    // return the (unchanged) node pointer
    return root;
}
```

# Бинарное дерево поиска на C++

```
// Function to do inorder traversal of BST
void inorderTraversal(Node* root)
{
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}
```

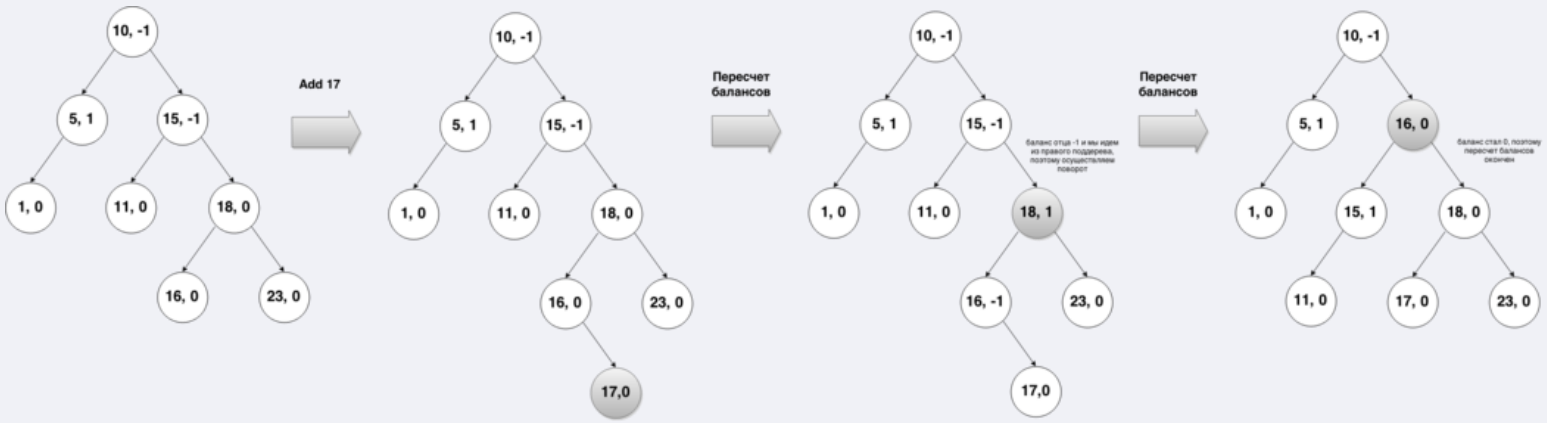
```
// Main function to demonstrate the operations of BST
auto main() -> int
{

    Node* root = nullptr;
    // create a BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);

    // Print the inorder traversal of a BST
    cout << "Inorder traversal of the given Binary Search "
            "Tree is: ";
    inorderTraversal(root);
    cout << endl;

    return EXIT_SUCCES;
}
```

Inorder traversal of the given Binary Search Tree is:
20 30 40 50 70

# AVL Дерево

**АВЛ-дерево** – сбалансированное, в котором поддерживается следующее свойство:
для каждой его вершины высота её двух поддеревьев различается не более чем на 1

Подобное дерево позволяет всегда иметь доступ к вершине **O(log n)**

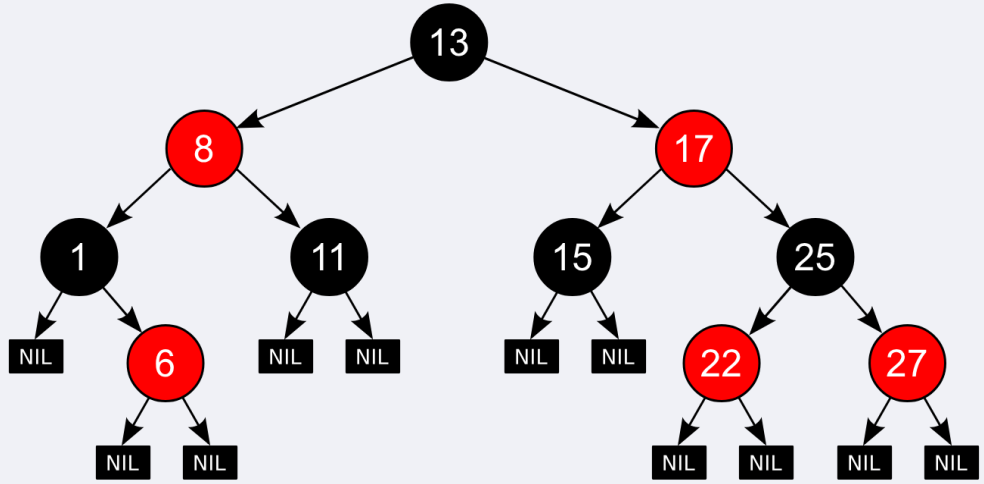



```
struct node
{
    int key;
    unsigned char height;
    node* left;
    node* right;
    node(int k) {
        key = k;
        left = right = 0;
        height = 1;
    }
};
```

# Красно-Черное дерево

**Красно-чёрное** – бинарное поисковое дерево, усложненный вариант АВЛ-дерева, у которого каждому узлу сопоставлен дополнительный атрибут – цвет

Все операции имеют логарифмическую сложность **O(log n)**, но быстрее, чем в АВЛ при условии частых изменений (из-за меньшего количества поворотов при вставке и удалении)



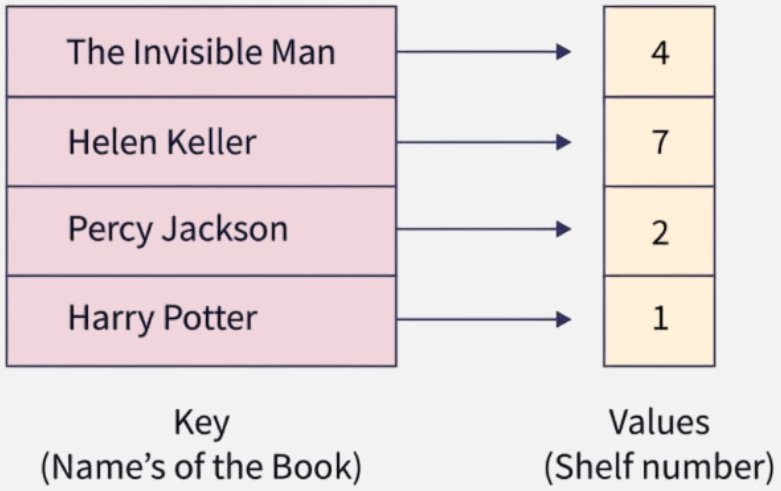

**Применяется в:**

- std::map, std::set в STL
- Базы данных (например, PostgreSQL B-Tree основаны на RBT)
- Кэширование (например, Linux-кернел использует RBT для таймеров)

# Map

**std::map** в C++ – ассоциативный контейнер, обеспечивает эффективный поиск, вставку и удаление элементов на основе ключей, а также поддерживает упорядоченное хранение элементов по ключам

Хранит элементы в виде **ключ-значение пар**, где каждый ключ является уникальным




## Основные особенности:

- В std::map не может быть двух элементов с одинаковыми ключами
- Элементы в std::map автоматически упорядочиваются по ключам (< Operator)
- Реализация на основе дерева:
  Внутри std::map обычно реализован с использованием красно-черного дерева
- Есть похожий контейнер – std::multimap, который позволяет хранить несколько элементов с одинаковыми ключами

# Map | Вставка элементов

**insert (std::pair<KeyType, ValueType> (key, value) )**
Вставляет новую пару ключ-значение (если такой ключ уже есть, то итератор просто указывает на существующий элемент)
**Сложность: O(log n)**

**operator [ ] (key) = value**
Вставляет или обновляет значение для заданного ключа (если ключ не существует, он будет вставлен с указанным значением)
**Сложность: O(log n)**

**emplace (key, value)**
Вставляет новую пару ключ-значение на месте (по возможности без копирования или перемещения) – иногда эффективнее insert
**Сложность: O(log n)**

# Map | Поиск элементов:

**find (key)**
Ищет элемент с заданным ключом и возвращает итератор на найденный элемент (или map.end() итератор, если ключ не найден)
**Сложность: O(log n)**

**count (key)**
Возвращает количество элементов с заданным ключом (для std::map (с уникальными ключами) результат всегда либо 0, либо 1)
**Сложность: O(log n)**

**lower_bound (key)**
Возвращает итератор на первый элемент, чей ключ *не меньше* заданного (или подходящую позицию, если ключ не найден)
**Сложность: O(log n)**

**upper_bound (key)**
Возвращает итератор на первый элемент, чей ключ *больше* заданного (или подходящую позицию, если ключ не найден)
**Сложность: O(log n)**

**equal_range (key)**
Возвращает пару итераторов, определяющих диапазон элементов с ключом, равным заданному (для std::map будет 0 или 1)
**Сложность: O(log n)**

# Map | Удаление элементов

**erase (key)**
Удаляет элемент с заданным ключом и возвращает количество удаленных элементов
**Сложность: O(log n)**

**erase (iterator)**
Удаляет элемент, на который указывает итератор (итератор должен быть валидным и указывать на элемент в std::map)
**Сложность: O(log n)**

**clear ()**
Удаляет все элементы из std::map
**Сложность: O(n)**

# Map | Размер и емкость:

**size ()**
Возвращает количество элементов в std::map
**Сложность: O(1)**

**empty()**
Проверяет, пуст ли std::map
**Сложность: O(1)**

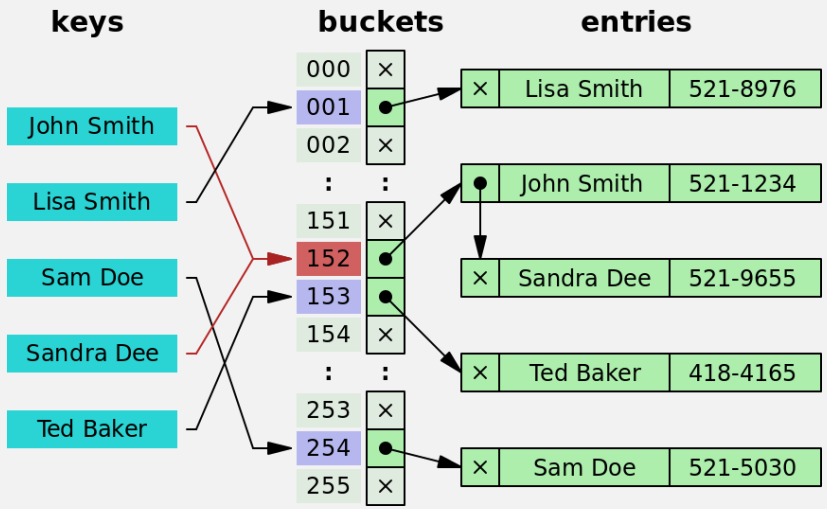
# Map | Пример работы

```
auto main() -> int {
    std::map<std::string, std::string> dictionary;
    dictionary["apple"] = "яблоко";
    dictionary["banana"] = "банан";
    dictionary["cherry"] = "вишня";

    std::string word = "banana";
    auto it = dictionary.find(word);
    if (it != dictionary.end()) {
        std::cout << "Перевод слова '" << word << "': " << it->second << std::endl;
    } else {
        std::cout << "Слово '" << word << "' не найдено в словаре." << std::endl;
    }

    return EXIT_SUCESS;
}
```

# Unordered_map

**std::unordered_map** – ассоциативный контейнер, оптимизирован для максимально быстрого доступа к элементам в среднем за счет использования хеширования

Реализует **хэш-таблицу**, хранит элементы в виде пар ключ-значение, но, в отличие от std::map, не гарантирует упорядоченность элементов




## Основные особенности:

- В среднем константное время O(1) для операций поиска, вставки и удаления
- Худшее время доступа – O(n)
  В худшем случае, время доступа может деградировать до линейного времени O(n), где n - количество элементов в контейнере
- Оператор == (равенство):
  Необходим для разрешения коллизий, когда разные ключи имеют одинаковый хэш-код, используется для проверки фактического равенства ключей в случае коллизии

# Unordered_map | Использование

### Реализация кэшей для хранения результатов ресурсоемких вычислений

Ключом может быть набор входных параметров функции, а значением – результат
При повторном вызове функции с теми же параметрами, результат быстро извлекается из кэша вместо повторного вычисления

### Быстрый поиск по ключу

Когда требуется очень быстрый поиск данных по ключу, и порядок хранения не важен
Например, индексация данных, таблицы символов в компиляторах, реализация словарей и т.д.

### Индексация данных

Создание индексов для быстрого доступа к записям в больших объемах данных по определенным ключам
Например, индексация строк по их хэш-значениям для быстрого поиска дубликатов

# Set

**std::set** – ассоциативный контейнер, обеспечивает эффективный поиск, вставку и удаление, а также поддержание элементов в упорядоченном виде (реализует математическую концепцию **множества**)

Хранит **уникальные элементы** в отсортированном порядке


## Основные особенности:

- Элементы автоматически упорядочиваются в соответствии с заданным критерием сравнения
  (по умолчанию используется оператор <)
- Реализация на основе дерева (красно-черное дерево):
  std::set обычно реализован с использованием самобалансирующегося двоичного дерева поиска
- Логарифмическая сложность для основных операций
  Обеспечивается за счет внутренней древовидной структуры

# Set | Использование

**Упорядоченное хранение уникальных элементов**
Удаление дубликатов, представление множеств в математическом смысле, перебор элементов в отсортированном порядке, поиск диапазонов значений

**Проверка принадлежности элемента множеству**
Быстрая проверка, присутствует ли определенный элемент в наборе данных

**Представление наборов уникальных идентификаторов, категорий, состояний и т.п.**

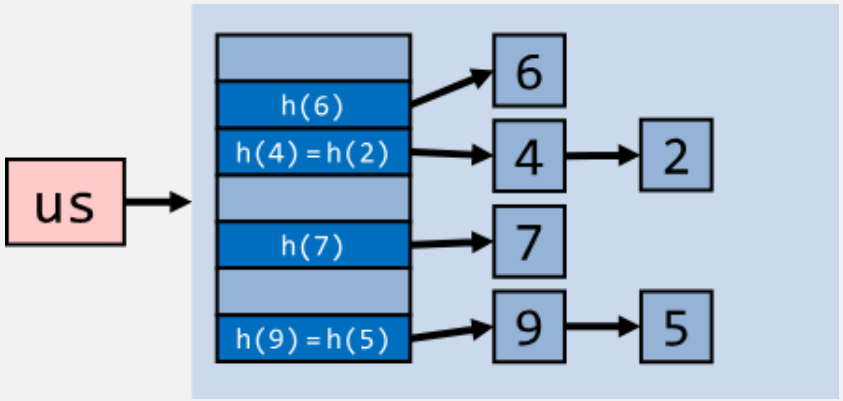
# Set | Пример работы

```
int main() {
    std::set<int> userIDs;
    userIDs.insert(123);
    userIDs.insert(456);
    userIDs.insert(123); // Дубликат, не будет вставлен

    std::cout << "Unique user IDs:" << std::endl;
    for (int id : userIDs) {
        std::cout << id << std::endl;
    }

    if (userIDs.count(456)) {
        std::cout << "User ID 456 exists in the set." << std::endl;
    }

    return 0;
}
```

# Unordered_set

**std::unordered_set** – ассоциативный контейнер, который также хранит уникальные элементы, но не гарантирует упорядоченность и **использует хэш-таблицу для реализации**, обеспечивая быстрый средний доступ

Хранит **уникальные элементы** в хэш-таблице




## Основные особенности:

- Скорость поиска и вставки или удаления является критически важной
  Если вам нужен максимально быстрый доступ к элементам (в среднем), std::unordered_set будет быстрее, чем std::set, особенно для больших наборов данных
- Применим, если порядок элементов не важен
- Тип элементов должен быть хешируемым (или вы должны предоставить свою хэш-функцию)

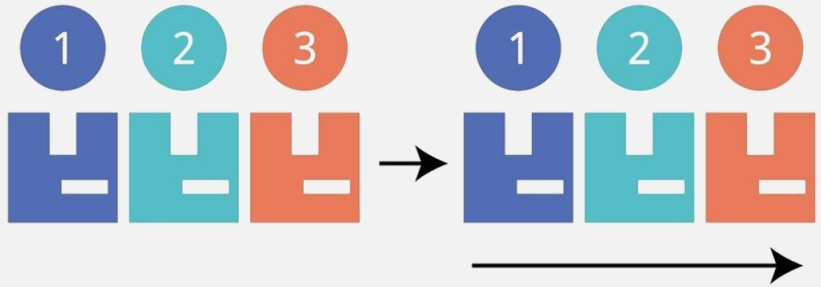
# Unordered_set | Пример работы

```cpp
#include <iostream>
#include <unordered_set>
#include <string>

int main() {
    std::unordered_set<std::string> stopWords = {"a", "an", "the", "is", "are", "was", "were"};
    std::string word = "the";

    if (stopWords.count(word)) { // Быстрая проверка, является ли слово стоп-словом
        std::cout << "'" << word << "' is a stop word." << std::endl;
    } else {
        std::cout << "'" << word << "' is not a stop word." << std::endl;
    }
    return 0;
}
```

# Queue

**std::queue** – еще один адаптер контейнера, который реализует структуру данных очередь (First-In, First-Out или «первым пришел – первым ушел»)

Не хранит данные напрямую, а **адаптирует** другой контейнер (по умолчанию std::deque, но можно использовать std::list) для предоставления интерфейса очереди




## Операции с std::queue и их производительность

- push (element): Добавляет элемент в конец очереди
  Сложность: обычно O(1) амортизированное или O(1)
- pop (): Удаляет элемент из начала очереди
  Сложность: O(1) амортизированное или O(1)
- front () и back (): Возвращает ссылку на элемент в начале и конце
  Сложность O(1)
- empty (): Проверяет, пуста ли очередь
  Сложность O(1)
- size (): Возвращает количество элементов в очереди
  Сложность O(1)

# Queue | Использование

### Очереди задач (Task Queues)
Очереди используются для хранения задач, ожидающих выполнения, в многопоточных приложениях и системах
Потоки-обработчики извлекают задачи из начала очереди и выполняют их

### Очереди сообщений (Message Queues)
В системах обмена сообщениями очереди используются для передачи сообщений между компонентами системы
Отправители помещают сообщения в конец очереди, получатели извлекают сообщения из начала

### Буферизация данных
Очереди могут использоваться для буферизации данных между производителем и потребителем
Например, при чтении данных из файла или сети и их последующей обработке

### Алгоритмы обхода графов (BFS - Breadth-First Search):
Очереди играют ключевую роль в алгоритме поиска в ширину для обхода графов

### Моделирование реальных очередей
Симуляция очередей людей, запросов, событий в различных моделях и системах

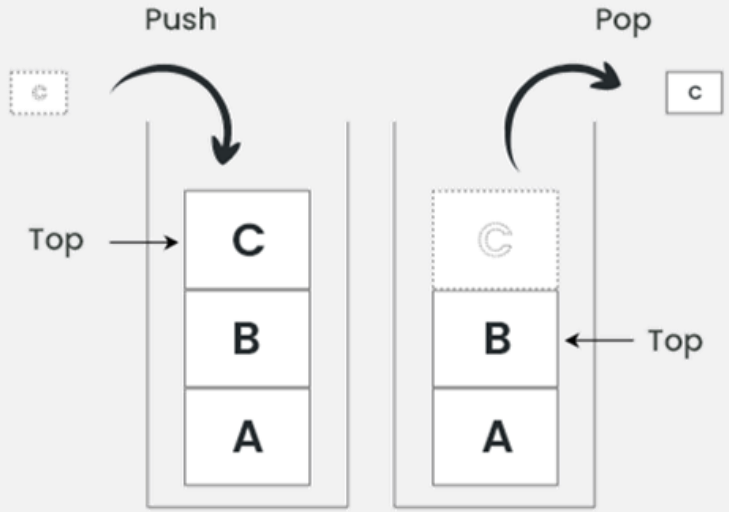
# Queue | Пример реализации Task Queues

```
void processTask(const std::string& task) {
    std::cout << "Processing task: " << task << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1)); // Имитация работы
    std::cout << "Task '" << task << "' completed." << std::endl;
}

int main() {
    std::queue<std::string> taskQueue;
    taskQueue.push("Task A");
    taskQueue.push("Task B");
    taskQueue.push("Task C");

    while (!taskQueue.empty()) {
        std::string currentTask = taskQueue.front(); taskQueue.pop();
        processTask(currentTask);
    }
    std::cout << "All tasks processed." << std::endl;
    return 0;
}
```

# Stack

**std::stack** – этоне самостоятельный контейнер, а **адаптер для другого контейнера** (по умолчанию std::deque)**,** который реализует интерфейс стека (Last-In, First-Out или «последним пришел – первым ушел»)

**Не хранит данные напрямую**, но использует внутренний контейнер для хранения элементов и предоставляет ограниченный интерфейс, который соответствует операциям стека




## Операции с std::stack и их производительность

- push (element): Добавляет элемент на вершину стека
  Сложность: обычно O(1) амортизированное или O(1)
- pop (): Удаляет элемент с вершины стека
  Сложность: O(1) амортизированное или O(1)
- top (): Возвращает ссылку на элемент на вершине стека
  Сложность O(1)
- empty (): Проверяет, пуст ли стек
  Сложность O(1)
- size (): Возвращает количество элементов в стеке
  Сложность O(1)

# Stack | Использование

## Функциональный стек вызовов

Компиляторы и интерпретаторы используют стеки для управления вызовами функций – при вызове функции информация (аргументы, локальные переменные, адрес возврата) помещается в стек, а при завершении – извлекается из стека

## Undo и Redo механизмы

Стек можно использовать для хранения последовательности операций, выполненных пользователем
Для «отмены» последняя операция извлекается из стека, а для «повтора» может быть взята из «стека повтора»

## Вычисление выражений

Играют важную роль в алгоритмах разбора и вычисления математических выражений, а также используются для проверки правильности расстановки скобок в выражениях (в коде или математических формулах)

## Алгоритмы обхода графов (DFS - Depth-First Search)

Используются в алгоритме поиска в глубину для отслеживания пути обхода графа

## Обработка HTML/XML

Стеки могут использоваться для разбора HTML или XML документов, чтобы проверять правильность вложенности тегов

# В чем польза контейнеров?

**Готовые структуры данных:**

Контейнеры STL предоставляют широкий выбор уже реализованных и оптимизированных структур данных,

Не нужно писать их с нуля, что экономит время и усилия

**Управление памятью:**

Контейнеры автоматически управляют памятью для хранения элементов

Не нужно беспокоиться о выделении и освобождении памяти вручную, что снижает риск ошибок и утечек памяти

**Стандартизированный интерфейс:**

Все контейнеры STL имеют общий и предсказуемый интерфейс для основных операций

Это упрощает использование разных контейнеров

**Эффективность:**

Контейнеры STL реализованы с акцентом на производительность

Они используют эффективные алгоритмы и структуры данных, что обеспечивает хорошую скорость работы

**Обобщенное программирование:**

Контейнеры работают с любыми типами данных, которые соответствуют определенным требованиям

Это делает их универсальными и применимыми в разных ситуациях

# Итераторы в STL

**Итераторы в STL** – это обобщённые указатели для перебора элементов в контейнерах

Предоставляют **единый** способ доступа к элементам различных типов контейнеров (независимо от того, используется ли vector, list или map)

- Скрывают детали внутреннего устройства контейнеров
- Позволяют выполнять разнообразные операции над элементами контейнеров: перебор, чтение, запись, перемещение и т.д.

Итераторы позволяют алгоритмам STL работать с любыми контейнерами, не завися от их внутренней реализации

Алгоритм, например, sort, может сортировать элементы как в vector, так и в list благодаря итераторам

# Использование пользовательских типов

STL контейнеры и алгоритмы разработаны для хранения элементов различных типов, включая пользовательские

**Оператор <**
для сортировки

Для контейнеров, которые хранят элементы в упорядоченном виде, а также для алгоритмов сортировки необходимо, чтобы ваш пользовательский тип поддерживал оператор <

```
class MyClass {
public:
    int value;
    MyClass(int v) : value(v) {}

    bool operator<(const MyClass& other) const {
        return value < other.value;
    }
};
```

**Хэш-функция**
для неупорядоченных ассоциативных контейнеров

Для вычисления хэш-кода объектов, необходимого для быстрого их распределения по «корзинам» в хэш-таблице
Обеспечивает в среднем константное время доступа к элементам

```
namespace std {
    template <>
    struct hash<MyClass> {
        std::size_t operator()(const MyClass& obj) const {
            return std::hash<int>()(obj.value);
        }
    };
}

struct MyClassHash {
    std::size_t operator()(const MyClass& obj) const {
        return std::hash<int>()(obj.value); // Используем стандартную хэш-функцию для int
    }
};
```

## Введение в STL

Компоненты стандартной библиотеки и ее важность

## Нотация Big O

Распределение, примеры и применение

## Контейнеры

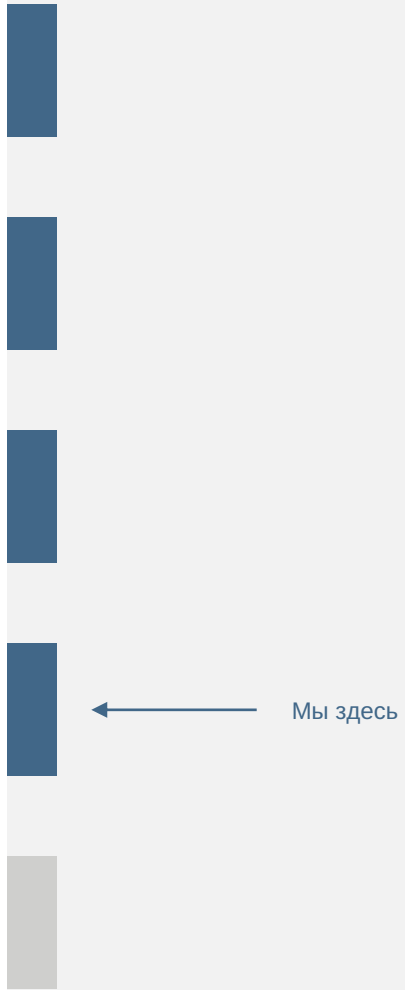
Описание основных типов и их особенностей

## Алгоритмы

Описание и примеры основных алгоритмов

## Оценка алгоритма

И общее заключение



# Классификация алгоритмов STL

**Неизменяющие:**

Операции чтения данных без изменения порядка или значений элементов

**Изменяющие:**

Операции, которые меняют порядок или значения элементов в контейнере

**Удаляющие:**

Операции для удаления элементов из контейнера (логически или физически)

**Мутирующие:**

Операции, изменяющие значения элементов, но не их порядок

**Сортирующие:**

Алгоритмы для упорядочивания элементов в контейнере

**Числовые:**

Алгоритмы для выполнения численных операций над диапазонами

**Алгоритмы работы с множествами:**

Операции над отсортированными диапазонами, рассматриваемыми как множества

# Неизменяющие операции

> Это алгоритмы, которые не изменяют порядок или значения элементов в диапазоне
> Используются для поиска информации и обхода элементов

**Поиск:**

- find(begin, end, value): Ищет первое вхождение value в диапазоне
- binary_search(begin, end, value): Проверяет, есть ли value в отсортированном диапазоне (быстрый поиск)
- count(begin, end, value): Подсчитывает количество вхождений value в диапазоне

**Обход:**

- for_each(begin, end, function): Применяет функцию к каждому элементу диапазона

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::for_each(numbers.begin(), numbers.end(), [](int n){ // Лямбда для вывода квадрата числа
    std::cout << std::pow(n, 2) << " ";
}); // Выведет: 1 4 9 16 25
```

# Изменяющие операции

> Алгоритмы, которые изменяют порядок или значения элементов в диапазоне

**Сортировка:**

- sort(begin, end): Сортирует диапазон в порядке возрастания (быстрая сортировка)
- partial_sort(begin, middle, end): Частично сортирует диапазон: [begin, middle) будут содержать наименьшие элементы в отсортированном порядке
- stable_sort(begin, end): Стабильная сортировка (сохраняет относительный порядок равных элементов)

**Перестановка:**

- shuffle(begin, end, generator): Перемешивает элементы диапазона в случайном порядке
- reverse(begin, end): Переворачивает порядок элементов в диапазоне

**Преобразование:**

- transform(begin1, end1, begin2, function): Применяет функцию к каждому элементу из [begin1, end1) и записывает результат в [begin2, ...)
- copy(begin1, end1, begin2): Копирует элементы из [begin1, end1) в [begin2, ...)
- move(begin1, end1, begin2): Перемещает элементы из [begin1, end1) в [begin2, ...)

# Изменяющие операции

> Алгоритмы, которые изменяют порядок или значения элементов в диапазоне

**Удаление (логическое):**

- remove(begin, end, value): Перемещает элементы, не равные value, в начало диапазона и возвращает итератор на «логический конец» (не удаляет физически)
- unique(begin, end): Удаляет последовательные дубликаты в отсортированном диапазоне и возвращает итератор на «логический конец»

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::sort(numbers.begin(), numbers.end()); // Выведет: 1 4 9 16 25

std::sort(snumbers.begin(), numbers.end(), [](int a, int b) { return a > b; });
```

# Числовые алгоритмы

> Алгоритмы для выполнения численных операций над диапазонами
> Нужны для суммирования, произведения и других математических действий

- accumulate(begin, end, initial_value): Вычисляет сумму элементов диапазона, начиная с initial_value
- inner_product(begin1, end1, begin2, initial_value): Вычисляет скалярное произведение двух диапазонов, начиная с initial_value
- adjacent_difference(begin, end, result_begin): Вычисляет разности между соседними элементами в диапазоне и записывает их в [result_begin, ...)
- partial_sum(begin, end, result_begin): Вычисляет частичные суммы элементов в диапазоне и записывает их в [result_begin, ...)

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
auto sum = std::accumulate(numbers.begin(), numbers.end(), 0); // sum == 15
```

# Алгоритмы работы с множествами

Алгоритмы для операций над отсортированными диапазонами, рассматриваемыми как множества (объединение, пересечение, разность и т.д.)

- set_union(begin1, end1, begin2, end2, result_begin): Вычисляет объединение двух отсортированных диапазонов и записывает результат в [result_begin, ...)
- set_intersection(begin1, end1, begin2, end2, result_begin): Вычисляет пересечение двух отсортированных диапазонов
- set_difference(begin1, end1, begin2, end2, result_begin): Вычисляет разность множеств (элементы из первого диапазона, которых нет во втором)
- set_symmetric_difference(begin1, end1, begin2, end2, result_begin): Вычисляет симметрическую разность (элементы, которые есть только в одном из диапазонов)

# Алгоритмы работы с множествами

```
int main() {
    // Представим, что у нас есть два набора ID пользователей, участвовавших в разных мероприятиях
    std::set<int> event1_participants = {123, 456, 789, 101, 202}; // ID пользователей, участвовавших в мероприятии 1
    std::set<int> event2_participants = {789, 202, 303, 505, 909}; // ID пользователей, участвовавших в мероприятии 2
 -
    // 1. Пересечение множеств: Пользователи, участвовавшие в обоих мероприятиях
    std::set<int> intersection_participants;
    std::set_intersection(event1_participants.begin(), event1_participants.end(),
                          event2_participants.begin(), event2_participants.end(),
                          std::inserter(intersection_participants, intersection_participants.begin()));

    // 2. Объединение множеств: Пользователи, участвовавшие хотя бы в одном мероприятии
    std::set<int> union_participants;
    std::set_union(event1_participants.begin(), event1_participants.end(),
                   event2_participants.begin(), event2_participants.end(),
                   std::inserter(union_participants, union_participants.begin()));

    // 3. Разность множеств (event1 - event2): Пользователи, участвовавшие только в мероприятии 1
    std::set<int> difference1_participants;
    std::set_difference(event1_participants.begin(), event1_participants.end(),
                        event2_participants.begin(), event2_participants.end(),
                        std::inserter(difference1_participants, difference1_participants.begin()));
    // 4. Симметрическая разность множеств: Пользователи, участвовавшие только в ОДНОМ из мероприятий (но не в обоих)
    std::set<int> symmetric_difference_participants;
    std::set_symmetric_difference(event1_participants.begin(), event1_participants.end(),
                                  event2_participants.begin(), event2_participants.end(),
                                  std::inserter(symmetric_difference_participants, symmetric_difference_participants.begin()));
}
```

**Участники мероприятия 1:**
101 123 202 456 789

**Участники мероприятия 2:**
202 303 505 789 909

**Участвовали в обоих мероприятиях:**
202 789

**Участвовали хотя бы в одном мероприятии:**
101 123 202 303 456 505 789 909

**Участвовали только в мероприятии 1:**
101 123 456

**Участвовали только в ОДНОМ из мероприятий:**
101 123 303 456 505 909

# C++ 20 Ranges

> **C++20 Ranges**, также известная как **STL v2** – более эффективная замена существующих алгоритмов и технических средств STL

## Основные добавления:

- **Ленивые вычисления**
  Они выполняются только при необходимости (например, при итерации), что экономит ресурсы и позволяет работать с бесконечными последовательностями
- **Изменение подхода к работе с памятью**
  Адаптеры (например, views) не владеют данными и не копируют элементы, что снижает накладные расходы
- **Появление пайпов (|)**
  Они позволяют создавать цепочки операций, делая код более декларативным и похожим на «конвейер» данных (например, `ranges::views::filter | ranges::views::transform`)

# C++ 20 Ranges | Примеры

**std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};**

```
// Цепочка: Фильтрация четных и возведение в квадрат (композиция views)
   auto even_squared = numbers |
       std::views::filter([](int n){ return n % 2 == 0; }) |
       std::views::transform([](int n){ return n * n; });
```

```
// Генерация диапазона чисел на лету с помощью views::iota
и применение views::transform
   auto iota_transformed = std::views::iota(1, 6) | std::views::transform([]
(int n){ return n * 10; }); // Умножение на 10
```

```
// Взятие первых 5 элементов с помощью views::take
   auto first_five = numbers | std::views::take(5);
```

```
// Пропуск первых 3 элементов с помощью views::drop
   auto skip_three = numbers | std::views::drop(3);
```

```
// Сортировка диапазона (действие ranges::sort) – изменяет исходный
вектор
   std::vector<int> numbers_to_sort = {5, 2, 8, 1, 9, 4};

// ranges::sort возвращает void, изменяет numbers_to_sort напрямую
   auto sorted_range = numbers_to_sort | std::ranges::sort;

// auto sorted_range = std::ranges::sort(numbers_to_sort);
// Альтернативный синтаксис без pipe operator
```

```
 // Комбинация нескольких views и actions:
//  Генерация чисел от 1 до 20, фильтрация нечетных, возведение
в квадрат, взятие первых 3, сортировка
   std::vector<int> complex_result;
   std::views::iota(1, 21) |
   std::views::filter([](int n){ return n % 2 != 0; }) | // Фильтруем нечетные
   std::views::transform([](int n){ return n * n; }) | // Возводим в квадрат
   std::views::take(3) |        // Берем первые 3
   std::ranges::sort |          // Сортируем
   std::ranges::copy(std::back_inserter(complex_result));   // Копируем
результат
```

## Введение в STL

Компоненты стандартной библиотеки и ее важность

## Нотация Big O

Распределение, примеры и применение

## Контейнеры

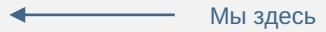
Описание основных типов и их особенностей

## Алгоритмы

Описание и примеры основных алгоритмов

## **Оценка алгоритма**

И общее заключение



# Шаги по оценке алгоритма

1. **Понять цель анализа**

   Определите, что именно вы хотите оценить: время выполнения, потребление памяти, или и то и другое

2.

3.

4.

5.

# Шаги по оценке алгоритма

1. **Выделить основные операции**

   Идентифицируйте операции, которые вносят наибольший вклад во время выполнения алгоритма (обычно циклы, рекурсивные вызовы, операции поиска / сортировки)
2. 
3. 
4. 
5. 

# Шаги по оценке алгоритма

1

2

3

4

5

**Проанализировать структуру алгоритма:**

- Последовательные операции: Суммируйте их сложности
- Вложенные циклы: Перемножьте сложности вложенных циклов
- Условные операторы (if/else): Выберите наихудший вариант из ветвей
- Рекурсия: Анализируйте дерево рекурсивных вызовов и количество операций на каждом уровне

# Шаги по оценке алгоритма

1. **Определить доминирующий член**

2. Из полученного выражения сложности, выберите член, который растет быстрее всего с увеличением размера входных данных

3. Отбросьте константы и  младшие члены

4.

5.

# Шаги по оценке алгоритма

1. **Записать Big O нотацию:**

2. Выразите сложность алгоритма в нотации Big O с доминирующим членом (например, O(n), O(n^2), O(log n))

3.

4.

5.

# Оценка «на глаз»

```
def example_function(data): # data - список размером n
    for x in data:          # Линейный цикл O(n)
        print(x)
    for i in range(len(data)): # Еще один линейный цикл O(n)
        for j in range(len(data)): # Вложенный линейный цикл O(n) - итого O(n^2)
            print(data[i] + data[j])
```

1. Первый цикл: **O(n)**
2. Второй (вложенный) цикл: **$O(n^2)$**
- Общая сложность: **$O(n) + O(n^2) = O(n^2)$**
(доминирует квадратичный член)

```
int some_function(int [] arr):
    n = arr.length
    for i in 0..n:
        for j in 0..n:
            action_A(arr) // log(n) operations
        action_B(arr) // n operations
    action_C() // 5 operations
```

- **$O(n * (n * log(n) + n) + 5) = O(n^2 * log(n) + n^2 + 5)$**

**$= O(n^2 * log(n))$**

# Другие варианты оценки

## Мастер-теорема

Слабое место прикидывания на глаз – рекурсия
Поэтому для оценки сложности рекурсивных алгоритмов широко используют **мастер-теорему**

По сути, это набор правил по оценке сложности:

Он учитывает, сколько новых ветвей рекурсии создаётся на каждом шаге и на сколько частей дробятся данные в каждом шаге рекурсии

## Метод Монте-Карло

Применяют довольно редко, часто с его помощью описывают производительность систем, состоящих из большого **множества алгоритмов**

Суть метода:

1. Берём алгоритм и гоняем его на случайных данных разного размера, замеряем время и память
2. Полученные измерения выкладываем на отдельные графики для памяти и времени
3. Затем автоматически вычисляется функция, которая лучше всего описывает полученное облако точек

# Рекурсия через мастер-теорему

**Рекурсия** – функции, которые вызывают сами себя

Сложность этих алгоритмов зависит не только от сложности внутренних циклов,
но и от количества итераций рекурсии

```
function Factorial(n: Word): integer;
   begin
      if n > 1 then
         Factorial:=n*Factorial(n-1)
      else
         Factorial:=1;
    end;
```

Эта процедура выполняется n раз, таким образом,
вычислительная сложность этого алгоритма равна O(n)

Рекурсивная процедура может выглядеть достаточно простой, но она может серьёзно усложнить программу,
многократно вызывая себя

# Многократная рекурсия

Рекурсивный алгоритм, который вызывает себя несколько раз, называется многократной рекурсией

Рабочий цикл функции : O(2N) = O(N)

```
procedure DoubleRecursive(N: integer);
   begin
      if N>0 then
         begin
            DoubleRecursive(N-1);
            DoubleRecursive(N-1);
         end;
   end;
```

Из-за многократной рекурсии количество вызовов будет $O(2^{n+1} – 1) = O(2^n)$

Необходимо быть аккуратными с применениями рекурсией!

# Метод Монте-Карло

**Метод Монте-Карло** – это подход, основанный на случайных выборках для приближённого решения задач, где точный **расчёт сложен или невозможен**

Он особенно полезен для анализа алгоритмов, когда нужно оценить их эффективность, время работы или вероятность ошибок

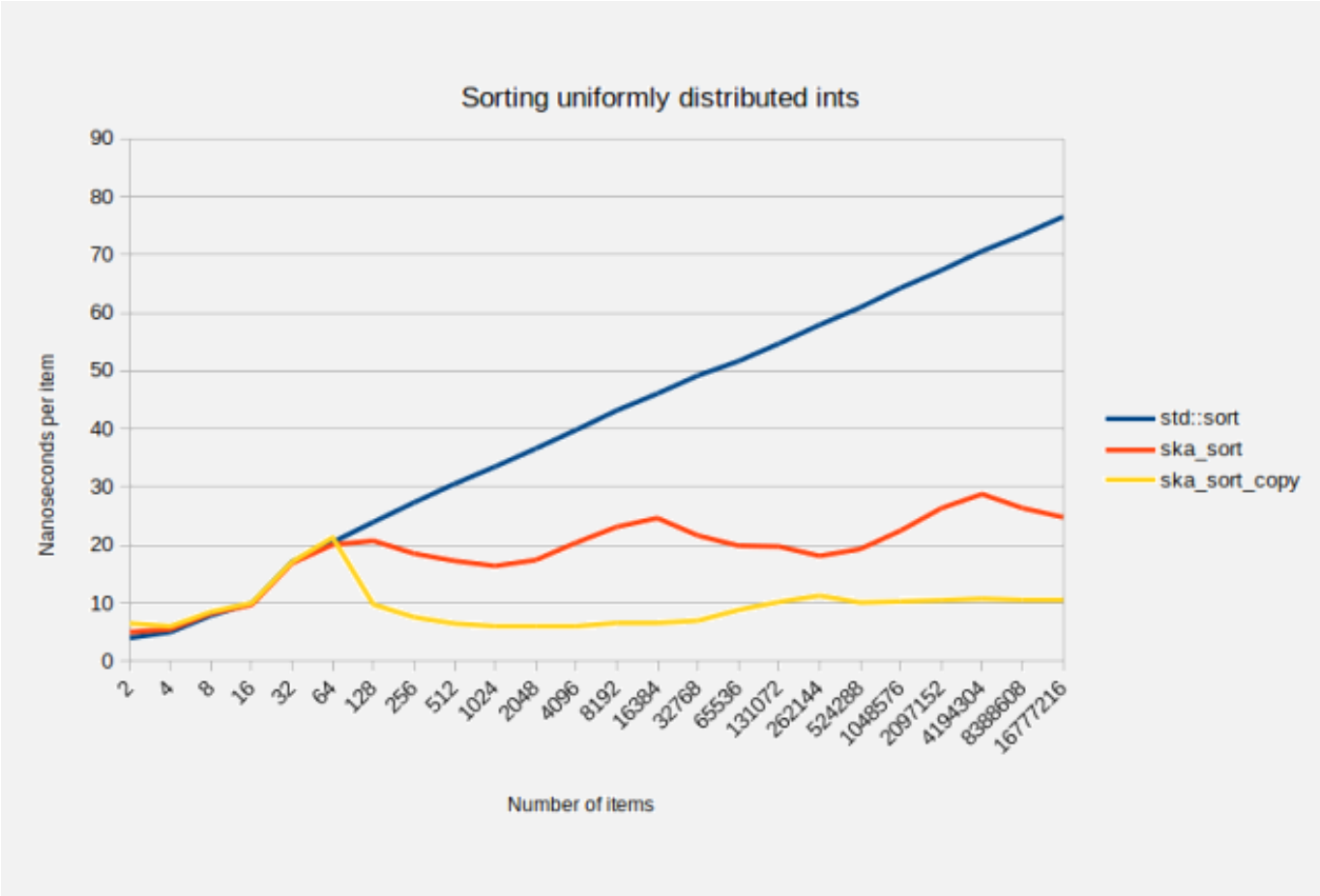
## Как это работает?

1. Создаётся множество случайных входных данных (например, массивы чисел для сортировки, графы)
2. Алгоритм запускается на каждом из сгенерированных входов
3. Результаты (время выполнения, количество операций, успешность) записываются
4. На основе статистики делаются выводы (среднее время работы, вероятность ошибки, зависимость от размера входных данных)

# Пример применения метода Монте-Карло

Допустим, мы хотим оценить среднее время сортировки случайного массива:

1. Генерируем 1000 случайных массивов разного размера
2. Запускаем алгоритм сортировки на каждом
3. Усредняем полученное время
4. Результат даст приближённую оценку производительности алгоритма



# Плюсы и минусы метода

## Плюсы

**Простота:**
Не требует сложной математики

**Универсальность:**
Применим к задачам с высокой вычислительной сложностью

**Параллелизм:**
Испытания можно проводить одновременно на разных ядрах процессора

## Минусы

**Неточность:**
Результат может быть приближённый (зависит от числа испытаний, выборки и учета выбросов)

**Громоздкое исполнение:**
Для точных и релевантных измерений необходимо выполнить много операций

# Советы на будущее

**Практика:** Решайте задачи на LeetCode с пометкой «Time Complexity»

**Анализируйте свой код:** «Как моя функция поведёт себя на 1 000 000 элементов?»

**Сравнивайте алгоритмы:** Например, ищите разницу между $O(n \log n)$ и $O(n^2)$

**Смотрите глубже:** Big O – это не только время, но и работа с памятью

# Спасибо за внимание!
# Ваши вопросы?

Оставьте, пожалуйста, обратную связь