Алгоритмы и структуры данных с примерами (в основном) на C

Иван Овчинников

20 июля 2022 г.

Оглавление

Глава 1

Базовые знания

1.1 Введение

От автора

Приветствую, коллеги! Начиная работу над этим документом я в первую очередь думал о структурировании собственных знаний в части алгоритмов и структур данных. Мои студенты и просто знакомые программисты при упоминании об алгоритмах представляют себе что-то неимоверно сложное и недоступное пониманию простого человека. Приняв решение развеять этот устоявшийся стереотип я начал изучать вопрос и пришёл к двум взаимоисключающим выводам: «не так всё и страшно» и «алгоритмы это бесконечность». Первое справедливо, потому что достаточно просто хорошо разобраться в вопросе, а второе справедливо, потому что это постоянно эволюционирующая и пополняющаяся область знаний. Изучив некоторые фундаментальные труды умных мужчин и женщин я понял, почему тема алгоритмов кажется такой сложной: не нашлось ни одного материала, написанного понятным и простым языком. В этом документе я постараюсь не повторить этой досадной ошибки именитых авторов. Конечно, нельзя не сказать, что подавляющее большинство авторов относятся к собственным творениям как к озарению свыше, поэтому считают свои произведения если не истиной в последней инстанции, то текстом, который нужно обязательно прочесть от ”корки до корки”. Этот документ не является ничем подобным.

1.1.1 Основные понятия

Алгоритмы Прежде, чем начинать изучать алгоритмы и структуры данных, нам нужно получить базовое понимание, что это такое. В этом разделе мы познакомимся с понятием алгоритма, требованиями, предъявляемыми к алгоритмам, способами записи алгоритмов. Поговорим о базовых структурах данных, их различиях и наиболее часто встречающемся применении.

> **Алгоритм** - это набор инструкций, описывающих порядок действий исполнителя для достижения некоторого результата. Алгоритмы пишут как для приложений, так и для станков, роботов, автомобилей, искусственного интеллекта и так далее.

Часто в качестве исполнителя алгоритма выступает компьютер, но понятие алгоритма не обязательно относится к компьютерным программам, так, например, чётко, пошагово описанный рецепт приготовления блюда также является алгоритмом, в таком случае исполнителем является человек.

Само слово «алгоритм» происходит от имени хорезмского учёного аль-Хорезми. Около 825-го года н.э. он написал сочинение («Книга о сложении и вычитании»), где аль-Хорезми сформулировал правила вычислений в десятичной системе счисления и, вероятно, впервые использовал цифру ”0”(ноль) для обозначения пропущенной позиции в записи числа. Итак, первоначально, алгоритм - это было это искусство счёта с помощью цифр. Постепенно значение слова расширялось. Учёные начинали применять его не только к сугубо вычислительным, но и к другим математическим процедурам. Одновременно с развитием понятия алгоритма постепенно происходила и его экспансия из чистой математики в другие сферы. Здесь, конечно нельзя не сказать про графиню Аду Лавлейс, которая известна прежде всего созданием описания вычислительной машины. Составила первую в мире программу. Ввела в употребление термины «цикл» и «рабочая ячейка», считается первым программистом в истории. Затем были Норберт Винер с его трудом о кибернетике, и Алан Тьюринг с его, абстрактной вычислительной «Машиной Тьюринга», которую можно считать моделью компьютера общего назначения. Она позволила формализовать понятие алгоритма и до сих пор используется во множестве теоретических и практических исследований. А идея теста Тьюринга до сих пор используется при определении способности компьютера мыслить более-менее схожим с человеком образом. Затем было появление персональных компьютеров, благодаря которым слово «алгоритм» вошло во все школьные учебники информатики и обрело новую жизнь. Различные определения алгоритма в явной или неявной форме содержат

следующий ряд общих требований:

1. Дискретность — алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых шагов. При этом для выполнения каждого шага алгоритма требуется конечный отрезок времени;

2. Детерминированность (определённость). В каждый момент времени следующий шаг работы однозначно определяется текущим состоянием системы. Таким образом, алгоритм выдаёт один и тот же результат (ответ) для одних и тех же исходных данных;

3. Понятность — алгоритм должен включать только те команды, которые доступны исполнителю и входят в его систему команд;

4. Завершаемость (конечность) — в более узком понимании алгоритма как математической функции, при правильно заданных начальных данных алгоритм должен завершать работу и выдавать результат за определённое число шагов;

5. Массовость (универсальность) - алгоритм должен быть применим к разным наборам начальных данных;

6. Результативность — подразумевает обязательное завершение алгоритма определёнными результатами.

Как бы это ни было грустно, чаще всего на практике мы видим, что последние два пункта не всегда соблюдаются как начинающими, так и продолжающими программистами, аналитиками и алгоритмистами. Словесная форма используется обычно для описания алгоритмов, предназначенных исполнителю – человеку. Команды записываются на обычном языке и выполняются по порядку. В командах могут использоваться формулы, специальные обозначения, но каждая команда должна обязательно быть понятна исполнителю. Естественный (последовательный) порядок команд может быть нарушен (если требуется, например, переход к предыдущей команде или требуется обойти очередную команду при каком-то условии), в этом случае команды можно нумеровать и указывать команду, к которой требуется перейти. Например, «перейти к п.3» или «повторить с п.4».

Графическая форма нужна для визуализации более сложных алгоритмов, обычно содержащих множества условных переходов или зацикливаний. Алгоритмы в этом случае представляются в виде блок-схем. Существуют специальные стандарты для построения блок-схем, где определяются графические изображения блоков. Команды алгоритмов

записываются внутри блоков на обычном языке, на так называемом «псевдокоде» (чтобы облегчить работу программиста) или с использованием математических формул. Блоки соединяются по определенным правилам линиями связи, которые показывают порядок выполнения команд. Как мы помним, алгоритм должен быть сформулирован на языке, понятном исполнителю. Если алгоритм разработан для решения задачи на ЭВМ, то для того, чтобы он мог выполниться исполнителем – ЭВМ, его необходимо записать на языке, понятном ЭВМ. Для этого разработано множество языков программирования. Запись алгоритма с их использованием называется программой. Так, язык программирования это базовая форма общения с ЭВМ.

Структуры данных Поверхностно обсудив алгоритмы, как часть, отвечающую за команды, можно переходить к структурам данных, как к части, отвечающей за хранение данных. Таким образом мы приходим к тому, что описание алгоритмов и структур данных есть ни что иное, как описание фон-Неймановой модели вычислительной машины. Вычислитель фон-Неймана устроен, в общих чертах, так: существует поток данных, который надо принять по шине данных из памяти или с устройств ввода, обработать по нужному алгоритму, и вывести результаты на устройства вывода или в память. Для реализации алгоритмов обработки данных другой поток - поток команд - запускается из памяти машины через шину команд. Именно поток команд является главным, он управляет, в том числе, началом и завершением, а также очередностью выбора данных для обработки. То есть поток команд - это каким-либо образом запрограммированный алгоритм обработки данных.

> **Структура данных** (англ. data structure) — программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных в вычислительной технике.

Структурировать данные обычно необходимо для ускорения работы с ними или оптимизации хранения. Для добавления, поиска, изменения и удаления данных структура данных обычно предоставляет некоторый набор функций, составляющих её интерфейс. То есть осуществлять вышеописанные действия мы можем только теми способами, которые определены самой структурой данных. Это позволяет не только структурировать данные ради структурирования, но и предоставлять максимально отлаженные, оптимальные способы взаимодействия со структурой.

Существует несколько основных видов структур данных: массивы, списки, деревья, графы. Действия над этими структурами можно условно разбить на две большие категории:

1. запросы, которые возвращают некоторую информацию о множестве;

2. модифицирующие операции, изменяющие множество.

У каждого из видов есть свои специфические операции, например, обход для графа, доступ по индексу для массива или балансировка для дерева, но при этом у большинства структур есть и некоторый общий набор действий, таких как поиск, вставка, возврат и удаление узла. **Массивы** - это базовая структура данных которая есть практически в любом языке программирования, внутри массивов зачастую хранятся и все остальные структуры данных. **Графы** - это мощная абстракция, описывающая множество реальных проблем, например, транспортные системы, компьютерные сети и World Wide Web. Это лишь небольшое количество реальных систем, которые легко можно представить в форме графов. **Деревья** это именно то, что вы себе представляете - структура состоящая из корня, ветвей и листьев. Деревья помогают структурировать иерархические данные, такие, как, например, файловые системы. Все вот эти папки и подпапки - это дерево. **Списки** - это такие структуры, в которых легко хранить и работать с очередями, буферизацией, собственно списками. **Хэш-таблицы** - довольно специфичная структура. Если говорить об их применении - это супер-быстрый поиск. Все же знают про Twitter и Instagram? Twitter придумал, а инстаграм ввёл в широкий оборот хэштеги. Они дают отличное понимание того, как устроены хэш-таблицы. Есть некоторое простое, захэшированное значение каких-то данных (в случае Instagram это хэштег у фотографии) и есть цепочка данных, у которых получился такой-же хэш (серия фотографий, которые были подписаны тем-же тегом). Сколько сейчас фотографий в Instagram? Сотни миллиардов? А по хэшу ищутся почти моментально.

1.1.2 Инструментарий создания алгоритмов

Изучая алгоритмы, было бы хорошо их визуализировать. Поговорим о способах и инструментах визуализации алгоритмов и структур данных. Стандарте де-факто для графической визуализации, языке UML, а именно его второй версии, а также программном обеспечении, используемом для визуализации алгоритмов и структур данных.

Мотивация Визуализация. С её помощью мы лучше понимаем важные абстрактные процессы и другие вещи. Проще говоря, зрение помогает нам думать. Более того, если кто-то хочет сам разобраться в работе того или иного алгоритма, то лучше всего попробовать сделать визуализацию самостоятельно, потому что попытка объяснить нечто — это лучший способ разобраться в предмете. А в процессе визуализации мы не только поясняем сами себе как работает алгоритм, но и фиксируем детали, которые имеют

неприятное свойство забываться в процессе, например, устного повествования. К тому же, графическое представление проблемы помогает найти ошибки в реализации алгоритма, да и вообще, часто оказывается, что это интересно само по себе.

Общие сведения Для визуализации алгоритмов обычно используют текстовое описание, блок-схемы и другие диаграммы. Все диаграммы делятся на две большие категории:

1. структурные диаграммы (диаграмма классов, компонентов, составной структуры, профилей и прочие);
2. диаграммы поведения (диаграмма деятельности, состояний, вариантов использования, разного рода диаграммы взаимодействия:
   (a) диаграмма коммуникации;
   (b) обзора взаимодействия;
   (c) последовательности

   и другие).

Алгоритмический язык в общем случае представляет собой систему обозначений и правил для единообразной и точной записи алгоритмов и исполнения их. Примером такой системы является псевдокод – описание структуры алгоритма на естественном, частично формализованном языке. В псевдокоде применяются формальные конструкции и математические символы. При записи алгоритма с помощью псевдокода применяются служебные слова и символы. Каждая команда записывается с новой строки. Иногда серию команд, составляющих тело какой-либо единой алгоритмической конструкции, например, цикла, рекомендуется заключать в операторные скобки.

```
АЛГ – АЛГоритм;
НАЧ – НАЧало алгоритма;
КОН – КОНец алгоритма;
НЦ – Начало Цикла;
КЦ – Конец Цикла;
ЕСЛИ - ветвление по условию
:= – присвоить значение;
// – комментарий.
```

После служебного слова АЛГ обычно указывается название алгоритма. Между названием и началом обычно пишут условия для входных данных, и т.д. Основная мысль такого описания в том, чтобы сделать его понятным реализующему, например, программисту.

Одним из наиболее наглядных способов записи алгоритма является изображение его в виде графической схемы. То есть, при помощи последовательности блоков, выполняющих определенные действия, и связей между ними, указывающих однозначный порядок выполнения отдельных инструкций. Рисовать графические схемы можно как в разных офисных пакетах, так и в онлайн редакторах. Также существуют плагины для подавляющего большинства популярных сред разработки, которые так или иначе умеют работать с визуализацией алгоритмов, структур и моделей данных. О программном обеспечении мы поговорим чуть позже.

Язык моделирования UML Это язык широкого профиля, который был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода, в том числе автоматизированная, для многих популярных языков программирования. Это открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой UML-моделью. UML позволяет разработчикам программного обеспечения достигнуть соглашения в графических обозначениях для представления общих понятий и сконцентрироваться на проектировании и архитектуре.

> **UML** объектно-ориентирован, в результате чего методы описания результатов анализа и проектирования семантически близки к методам программирования на современных объектно-ориентированных языках.

UML позволяет описать систему практически со всех возможных точек зрения и продемонстрировать разные аспекты поведения системы. Диаграммы UML сравнительно просты для чтения: UML расширяет и позволяет вводить собственные текстовые и графические стереотипы, что способствует его применению не только в сфере разработки программного обеспечения. UML получил широкое распространение и динамично развивается. Казалось бы, почему именно UML, а не любой другой инструмент, позволяющий рисовать схемы? Ответ прост: UML - промышленный стандарт для проектирования больших информационных систем, а также это язык, то есть все диаграммы описываются текстом, и затем специальными интерпретаторами генерируются в картинку. Отсюда простота версионирования и коллективной работы над диаграммой. Вот, например, самые популярные диаграммы, используемые для визуализации алгоритмов и структур данных:

1. **Диаграмма классов** - диаграмма, описывающая структуру системы, демонстрирующая классы, их атрибуты, методы и зависимости. Несмотря на то, многие язы-

ки программирования не содержит инструментов для работы с объектно-ориентированным программированием, и многие алгоритмы абсолютно не объектно-ориентированы, не могу не упомянуть диаграммы для визуализации классов, как чрезвычайно часто используемые. Существуют разные точки зрения на построение диаграмм классов в зависимости от целей их применения:

   - концептуальная точка зрения описывает модель предметной области, в ней присутствуют только классы прикладных объектов;
   - точка зрения спецификации применяется при проектировании информационных систем;
   - точка зрения реализации содержит классы, используемые непосредственно в программном коде (при использовании объектно-ориентированных языков программирования).

2. **Диаграмма конечного автомата или диаграмма состояний** - это диаграмма, на которой представлен конечный автомат с простыми состояниями, переходами и композитными состояниями. Конечный автомат это последовательность состояний, через которые проходит объект, а также ответные действия объекта на эти события. Конечные автоматы часто описывают всю систему целиком, а в диаграммах объектно-ориентированных приложений конечные автоматы часто прикреплены к классу и служат для определения поведения его экземпляров.

3. **Диаграмма деятельности (активности)** — диаграмма, на которой показано разложение некоторой деятельности на составные части. Под деятельностью понимаются исполняемые команды в виде согласованного последовательного или параллельного выполнения отдельных действий. Диаграммы деятельности используются при моделировании бизнес-процессов, технологических процессов, последовательных и параллельных вычислений, описания ролевых моделей процессов, происходящих внутри приложения.

Помимо перечисленных существуют и многие другие, и используются они незначительно реже, но для перечисления вообще всех типов диаграмм с развёрнутыми описаниями есть немало специализированных ресурсов.

1.1.3 Инструментарий реализации алгоритмов

Данный документ не ставит себе целью изучение языков программирования, скорее наоборот, целью является демонстрация универсальности алгоритмов и структур данных. Поэтому языки программирования и находятся здесь, в небольшом подразделе об

инструментарии реализации алгоритмов. Так автор хочет лишний раз подчеркнуть, что разницы на каком языке программирования реализовывать алгоритм - нет.

Язык программирования C/C++

Язык программирования Java

Язык программирования Python

1.2 Сложность алгоритма

1.2.1 Мотивация

Мы рассуждаем об алгоритмах. Алгоритмы - это план действий, который мы собираемся произвести, будь то в реальной жизни или запрограммированный в компьютерном коде. Пришло время научиться понимать, насколько эффективны алгоритмы, которые мы изучаем и насколько хорош план наших действий. В этом подразделе мы рассмотрим такое понятие, как асимптотическая сложность алгоритма, зачем её считать, и как анализировать существующие алгоритмы на предмет их сложности. Многие разработчики думают, что прекрасно обходятся без знания алгоритмов, но это верно лишь отчасти. Внутри каждой программы так или иначе присутствуют алгоритмы, как минимум те, которые мы сами составляем, как последовательность действий для нашей программы.

Таким образом программа сама по себе - это алгоритм, преобразующий какие-то данные или пользовательский ввод в какие-то другие данные, или вывод пользователю. Внутри программы также используются алгоритмы, уже реализованные в тех или иных подключаемых библиотеках используемого языка программирования: от простейших - поиска минимальных или максимальных значений до сложных - поиска с возвратом или кратчайших маршрутов обхода графа. Написав что-то более сложное, чем базовый калькулятор, встаёт вопрос о расчёте эффективности написанного участка кода, ведь один и тот же результат может быть достигнут множеством способов. И способы эти часто отличаются как раз эффективностью. Поэтому, чтобы не замерять ”на глазок”мы будем учиться проводить формальный анализ сложности алгоритма.

Учитывая что языки программирования во многом изначально разрабатывались людьми знакомыми с математикой и активно использующими её при написании программного обеспечения, давайте для начала попробуем разобрать само понятие - **асимптотическая сложность алгоритма**. Итак, обратимся к слову асимптота.

> **Асимптота** (от др.-греч. - не касающийся) — прямая, обладающая тем свойством, что расстояние от точки кривой до этой прямой стремится к нулю при удалении точки вдоль ветви в бесконечность.

Думаю что для людей не знакомых с математикой такое определение скажет мало. Но так как мы привыкли понимать то что читаем, а этот документ направлен на широкую категорию читателей, то попробуем ещё раз. Отойдём от математического определения и обратимся к определению для простых людей и начнём с примера. Предположим, есть коллекция из десяти элементов, и это будут входные данные. Пусть эта коллекция об-

рабатывается (не принципиально, что именно происходит, предположим, сортировка) одну секунду. Изменим входные данные. Сколько понадобится времени, чтобы точно также обработать коллекцию из тысячи элементов? Сто секунд? Тысяча или пять секунд? *Как изменится время исполнения и объём занятой памяти в зависимости от размера входящих данных?*. Это и есть центральный вопрос разработки алгоритмов. Это и есть вопрос эффективности и сложности алгоритмов.

Вполне очевидным является то, что в данном определении присутствуют три элемента интересующие нас: время исполнения, объём занятой памяти и размер вводимых данных. В первую очередь нас будут интересовать время исполнения и размер вводимых данных, потому что они, в конечном итоге, влияют на объём занимаемой памяти. Теперь можно дать чуть более простое но чуть более формализованное определение в том виде, который будет более понятен.

> Асимптотическая сложность (производительность) определяется функцией, которая указывает, насколько ухудшается работа алгоритма с увеличением объёма перерабатываемых алгоритмом данных, а именно, сколько будет затрачено времени.

При подсчёте сложности алгоритма чаще всего говорят о худшем случае, чтобы люди использующие алгоритм были уверены, что алгоритм совершенно точно не займёт больше времени или памяти. Такой худший случай работы алгоритма измеряют функцией, которая записывается через О-большое. Существуют также понятия Ω (омега-большое) и Θ (тета-большое). Например, доступ к ячейке массива описывается как $O(1)$, так как для доступа к ячейке потребуется всего одна операция, и её сложность не возрастает независимо от размера массива, Это называют константной сложностью. $O(N^2)$ означает, что, по мере увеличения количества входных данных, время работы алгоритма (использование памяти либо другой измеряемый параметр) возрастает квадратично. Если данных станет вдвое больше, производительность алгоритма замедлится приблизительно в четыре раза. При увеличении количества входных данных в три раза, она станет меньше в девять раз, и так далее. Важно понимать, что асимптотическая сложность даёт представление о теоретическом поведении алгоритма. Практические результаты могут, и скорее всего будут, отличаться.

Итак, как уже было упомянуто, помимо О-большого, описывающего худшее время выполнения алгоритмов, существуют также такие типы сложности алгоритмов как тета-большое(Θ) для среднего времени выполнения и омега-большое(Ω), говорящее о лучшем времени выполнения алгоритмов. Дадим чуть более развёрнутые определения для представленных случаев. Важно, что автор не преследует цель быть академически

точным и попытаться переформулировать выверенные формулировки из учебников, а хочет попытаться дать информацию так, чтобы появилось в первую очередь понимание того или иного явления.

- Время работы алгоритма в **наихудшем случае** - это верхний предел для любых входных данных. Располагая этим значением, мы точно знаем, что для выполнения алгоритма не потребуется большее количество времени. Не нужно будет делать каких-то сложных предположений о времени работы и надеяться, что на самом деле эта величина не будет превышена. Например, вы оптимизируете SQL-запрос и у вас на выбор 6 вариантов, один из которых в худшем случае дает $O(1)$, другой $O(n)$, третий $O(n^2)$. Тут ничего кроме O-большое и не понадобится.

- Характер поведения **усреднённого времени работы** часто ничем не лучше поведения времени работы для наихудшего случая. Предположим что у нас есть некая последовательность чисел 123_5678. Сколько времени понадобится чтобы определить, в какое место последовательности следует поместить число 4 чтобы последовательность осталась упорядоченной? В среднем половина элементов последовательности меньше четырёх, а другая половина больше. Таким образом, в среднем, нужно проверить половину элементов последовательности. В результате получается, что тета-большое, среднее время работы алгоритма, является линейной функцией от количества входных элементов, т.е. характер этой зависимости такой же, как и для времени работы в наихудшем случае. Оценить среднее время можно только многократно исполнив алгоритм и оценив его фактическую скорость при разных входящих данных.

- В некоторых алгоритмах наихудший случай встречается достаточно часто. Например, если в базе данных происходит поиск информации, то наихудшему случаю соответствует ситуация, когда нужная информация в базе данных отсутствует, то есть программа будет искать вообще по всей базе и не найдёт искомое. В некоторых приложениях поиск отсутствующей информации может происходить довольно часто. Знать наилучший случай для алгоритма полезно, несмотря даже на то, что такая ситуация крайне редко встречается на практике. Во многих случаях она обеспечивает понимание оптимальных условий для работы алгоритма. Например, мы решаем логистическую задачу, как грузовику в девять утра по пробкам в Москве объехать пятнадцать магазинов, магазины открываются раз в год и всё очень плохо... тут-то нам и пригодится омега-большое, чтобы просчитать в какой момент и по какому маршруту ехать.

1.2.2 Классификация

Прежде, чем продолжить чтение этого материала я рекомендую вам коротко ознакомиться с математическими понятиями логарифма и экспоненты. Рассмотрим классификацию сложности алгоритмов, упорядоченную по убыванию эффективности, т.е. по увеличению времени выполнения алгоритмов:

1. Константной сложностью алгоритма называется сложность, при которой время затраченное на выполнение алгоритма не зависит от размера входных данных;

2. Алгоритмы, работающие за логарифмическое время, обычно встречаются при операциях с двоичными деревьями или при использовании двоичного поиска;

3. Линейное время как правило означает, что для достаточно большого размера входных данных время работы увеличивается в прямой пропорции;

4. Линейно-логарифмическая сложность во многих случаях является просто результатом выполнения операций с линейной сложностью несколько раз;

5. Наверное наиболее типичным случаем квадратичной сложности алгоритмов является поиск какой-то подходящей пары элементов, например, ближайших точек на координатной плоскости;

6. Экспоненциальные алгоритмы чаще всего возникают в результате подхода именуемого ”метод грубой силы”или брутфорс (от англ. - brute force).

Анализируя производительность алгоритмов, можно утверждать, что некоторые примитивные операции обеспечивают **константную производительность**. Очевидно, что это утверждение не говорит о том, что будет выполнена именно один оператор кода, или действие будет выполнено за один процессорный такт, так как при подсчёте сложности мы никак не затрагиваем аппаратное обеспечение. Константная производительность означает, что количество фактически произведённых операций не зависит от входящих значений. Например, сравнение двух 32-разрядных чисел *x* и *y* на равенство должно иметь одинаковую производительность независимо от фактических значений *x* и *y*. Константная операция определяется как имеющая производительность $O(1)$.

Логарифмические алгоритмы чрезвычайно эффективны, потому что быстро сходятся к решению. Эти алгоритмы уменьшают размер задачи на определенный множитель при каждой итерации (чаще всего, примерно, пополам). Самый яркий пример — бинарный поиск. Его мы, конечно же изучим, но в общем - если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение дихотомией, методом

деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет, потому что там все элементы точно больше. Если же меньше, то наоборот — отбросим первую половину, там все элементы точно меньше. И так будем продолжать делить пополам, в итоге проверим $O(\log_n)$ элементов.

Алгоритм с временем $O(n)$, или **линейным временем**, обладает одним очень естественным свойством: его время выполнения не превышает размера входных данных умноженного на константу. Чтобы обеспечить линейное время выполнения, проще всего обработать входные данные за один проход, с постоянными затратами времени на обработку каждого элемента (например, проверкой элемента на равенство искомому значению). Другие алгоритмы достигают линейной границы времени выполнения по менее очевидным причинам. Такой сложностью обладает, например, алгоритм поиска элемента в не отсортированном массиве. Нам придётся пройтись по всем *n* элементам массива, чтобы гарантированно найти или не найти искомый. В некоторых ситуациях встречается время выполнения, асимптотически меньшее линейного. Так как для простого чтения входных данных потребуется линейное время, такие ситуации обычно встречаются в модели вычислений с косвенными «проверками» входных данных вместо их полного чтения, а целью является минимизация количества необходимых проверок. Алгоритмы сублинейного времени, как правило, вероятностны и дают лишь приближённое решение, поэтому и их чаще всего причисляют к простой линейной сложности.

Линейно-логарифмическая сложность алгоритмов тоже встречается очень часто, одной из главных причин его распространенности является время выполнения любого алгоритма, который разбивает входные данные на две части одинакового размера, рекурсивно обрабатывает каждую часть, а затем объединяет два решения за линейное время. Пожалуй, классическим примером задачи, которая может решаться подобным образом, является сортировка. Так, алгоритм сортировки слиянием разбивает входной набор чисел на две части одинакового размера, рекурсивно сортирует каждую часть, а затем объединяет две отсортированные половины в один отсортированный выходной список.

Типичная задача: заданы *n* точек на плоскости, определяемых координатами *x* и *y*; требуется найти пару точек, расположенных ближе всего друг к другу. Естественный алгоритм «грубого перебора» для такой задачи рассматривает все возможные пары точек, вычисляет расстояния между каждой парой, а затем выбирает пару, для которой это расстояние окажется наименьшим. Проще говоря, количество пар ограничивается $O(n^2)$ потому, что количество способов выбора первого элемента пары (не больше *n*)

умножается на количество способов выбора второго элемента пары (тоже, не больше *n*). В этом примере представлена очень распространенная ситуация, в которой встречается **квадратичное время выполнения** $O(n^2)$: перебор всех пар входных элементов с постоянными затратами времени на каждую пару. Квадратичное время так же естественно возникает в парах вложенных циклов: алгоритм состоит из цикла с $O(n)$ итерациями, и каждая итерация цикла запускает внутренний цикл с временем $O(n)$. Произведение двух множителей *n* дает квадратичную сложность о которой мы говорим

Экспоненциальные алгоритмы практичны только для очень малых значений *n*. Некоторые алгоритмы могут иметь экспоненциальное поведение в наихудшем случае и при этом активно использоваться на практике из-за их поведения в среднем случае. Хорошим примером является симплекс-метод для решения задач линейного программирования.

Приведенный список классификации сложности алгоритмов не является полным, однако является вполне достаточным для работы связанной с программированием.

1.3 Простейшие алгоритмы

1.3.1 Мотивация

В этом подразделе мы ещё немного поговорим об общих моментах, касающихся алгоритмов и попробуем начать писать простейшие примеры. Изучение простейших алгоритмов нужно не с академической целью (не ради изучения как такового), а ради введения в область знаний, практики на простых примерах, развития навыка алгоритмического мышления. Также важной частью изучения простейших примеров является сопоставление формального описания (диаграмм и блок-схем) с описанием на языке программирования.

1.3.2 Классификация

Все алгоритмы можно разделить на несколько категорий - линейный, ветвящийся и циклический. Все три эти категории мы рассмотрим на примере простейшего калькулятора.

Линейный алгоритм считается наиболее простым в информатике считается. Он предполагает простую последовательность выполнения действий. Приведем наиболее простой пример алгоритма такого вида. Научим наш будущий калькулятор складывать два введённых пользователем числа.

```
НАЧАЛО
ОБЪЯВИТЬ результат
ЗАПРОС ЧИСЛА а
ЗАПРОС ЧИСЛА б
результат ПРИСВОИТЬ а СУММА б
ВЫВОД результат
КОНЕЦ
```

Как видно, абстрактное описание алгоритма человекочитаемо, но и не привязано к реализации на каком-либо языке программирования. То есть, по сути, является связующим звеном между бизнес-логикой и программированием.

Блок схема этого алгоритма будет иметь прямолинейный вид. Такие алгоритмы не только просты, но и весьма распространены. Если рассматривать любую программу в глобальном смысле - обычно она реализует такой алгоритм: последовательно выполняет какие-то действия. Даже если внутри будут содержаться какие-то другие алгоритмы - общий вид любой программы будет именно такой. На языке программирования такие простые алгоритмы выглядят весьма похожими на блок схему. Инициализируем переменные для хранения, запрашиваем числа, суммируем, выводим результат.

```c
// C/C++ example
int num1;
int num2;
int rslt;
printf("Первое: ");
scanf("%d", &num1);
printf("Второе: ");
scanf("%d", &num2);
rslt = num1 + num2;
printf("Итог: %d\n", rslt);
```

```python
# Python example


rslt = 0
num1 = int(input("Первое: "))

num2 = int(input("Второе: "))

rslt = num1 + num2
print("Итог: " + rslt)
```

На этом, простейшем примере, можно также обратить внимание на схожесть реализаций для разных языков программирования. Действительно, хорошо описанный алгоритм является абсолютно отвязанным от языка программирования и программисту с минимальными навыками в любом языке программирования не сложно его реализовать привычными для себя инструментами, будь то C/C++, Python или Java Теперь обратимся к ветвящемуся или разветвлённому алгоритму. Такие алгоритмы, предполагают наличие условия, при котором в случае его истинности мы выполним одни действия, а в случае ложности - другие. В нашем прекрасном во всех отношениях калькуляторе до этого момента не хватало предупреждения пользователя о невозможности делить на ноль и выбора действия.

```
НАЧАЛО
ЗАПРОС ЧИСЛА
ЗАПРОС ЗНАКА
ЗАПРОС ЧИСЛА
ЕСЛИ (НЕ(ЗНАК == / И ВТОРОЕ ЧИСЛО !=0))
ВЫБОР ДЕЙСТВИЯ
ВЫВОД РЕЗУЛЬТАТА
ИНАЧЕ
ВЫВОД НЕКОРРЕКТНЫЙ ВВОД
КОНЕЦ
```

Допишем наш простейший калькулятор до более-менее полезного:

```c
char sign;
printf("%s", "Введите знак: ");
fseek(stdin, 0, SEEK_END);
scanf("%c", &sign);
```

```
  if (!(sign == '/' && num2 == 0)) {
    switch (sign) {
      case '/':
        rslt = num1 / num2; break;
      case '+':
        rslt = num1 + num2; break;
      case '-':
        rslt = num1 - num2; break;
      case '*':
        rslt = num1 * num2; break;
      default: break;
    }
    printf("Результат: %f\n", rslt);
  } else {
    printf("%s\n", "Некорректный ввод: деление на ноль");
```

Циклические алгоритмы Такие алгоритмы предполагают наличие участка вычислений или действий, который будет повторяться до выполнения определенного условия. Текущий вариант калькулятора завершает свою работу, если пользователь решил осуществить деление, и вторым операндом операции выбрал ноль. Выходить из программы в случае деления на ноль - не очень хорошо, потому мы будем переспрашивать у пользователя нужное количество раз, пока он не введёт правильный делитель.

```
НАЧАЛО
ЗАПРОС ЧИСЛА
ЗАПРОС ЗНАКА
ЕСЛИ ЗНАК == /
 ДЕЛАТЬ:
   ЗАПРОС ЧИСЛА
 ПОКА ЧИСЛО == 0
ИНАЧЕ
 ЗАПРОС ЧИСЛА
ВЫБОР ДЕЙСТВИЯ
ВЫВОД РЕЗУЛЬТАТА
КОНЕЦ
```

Чтобы сделать наш калькулятор полностью функциональным консольным приложением с некоторой защитой от нерадивого пользователя - запишем запрос делителя внутрь цикла, проверяющего что этот делитель не будет равен нулю

```
  float num1; float num2; char sign; float rslt; printf("%s", "Введите
    первое число: "); scanf("%f", &num1); printf("%s", "Введите знак: ");
```

```
fseek(stdin, 0, SEEK_END); scanf("%c", &sign); if (sign == '/') { do {
printf("%s", "Введите второе число: "); scanf("%f", &num2); } while (
num2 == 0); } else { printf("%s", "Введите второе число: "); scanf("%f",
 &num2); } switch (sign) { case '/': rslt = num1 / num2; break; case '+'
: rslt = num1 + num2; break; case '-': rslt = num1 - num2; break; case '
*': rslt = num1 * num2; break; default: break; } printf("Результат: %f\n
", rslt);
```

На этом занятии мы рассмотрели понятия линейного, ветвящегося и циклического алгоритмов, написали короткие примеры, демонстрирующий каждый из типов, и в итоге объединили все три типа алгоритмов в одной программе. до встречи на следующем уроке

В этом видео будем говорить о простых, а именно об одних из самых старых придуманных человечеством алгоритмах Для начала обратим внимание на алгоритм придуманный древнегреческим математиком Евклидом. Данный алгоритм нужен для нахождения наибольшего общего делителя двух целых чисел. Это один из старейших численных алгоритмов, используемых в наше время. Следом нас ожидает расширенный и ускоренный алгоритмы Евклида. И в конце занятия мы рассмотрим так называемое - решето Эратосфена Перед тем как начать рассмотрение алгоритма Евклида нам нужно обратиться к такому математическому понятию как - ”Наибольший общий делитель”. Нам это пригодится для дальнейшего понимания работы алгоритма Евклида. Итак - НОД двух и более чисел — это самое большее натуральное число, на которое эти числа делятся без остатка. Например, НОД 8 и 4 = 4, потому что и 8 и 4 делятся на 4 без остатка и 4 больше, чем другой общий делитель этих чисел - двойка. В самом простом случае алгоритм Евклида применяется к паре положительных целых чисел. Евклид предложил алгоритм только для натуральных чисел и геометрических величин. Для его реализации нужно последовательно вычитать из большего числа меньшее, пока они не сравняются. Равенство чисел в конце алгоритма будет означать, что они являются НОД исходных. Для данного алгоритма существует множество теоретических и практических применений. В частности, он является основой для криптографического алгоритма с открытым ключом RSA, широко распространённого в электронной коммерции Перейдём к рассмотрению реализации алгоритма Евклида. Для начала мы делаем запрос на ввод двух чисел для которых будем проводить поиск НОД. Проверок на ввод осуществлять не будем, для простоты, и предположим, что пользователь знает, что необходимо ввести целые положительные числа. Для полноценной обработки введенных чисел нам понадобится цикл, условием завершения которого станет равенство обрабатываемых чисел. В самом цикле собственно и будет реализовываться алгоритм. Какое число на текущей итерации цикла будет больше, из того и будем проводить вычитание второго

числа. Таким образом, постепенно уменьшая оба исходных числа, мы придём к их равенству, после чего цикл завершится и полученное значение НОД можно будет вывести в консоль. Сложность данного алгоритма O(н) поскольку увеличение исходных чисел увеличит время работы цикла в прямой пропорции.

```
int euclidusSimple(int a, int b) {while (a != b)
  if (a > b) { a = a - b;
  } else { b = b - a;
  }return a;}
int main (int argc, const char * argv [])
int a; int b;
printf("%s", "Input a: "); scanf("%d", &a);
printf("%s", "Input b: "); scanf("%d", &b);
int r = euclidusSimple(a, b);printf ("GCD: %d\n" , r);return 0;}
```

Ещё одним вариантом нахождения НОД является вычисление с использованием математической операции получения остатка от деления. Иногда его называют ускоренный алгоритм Евклида Рассмотрим на примере кода. В начале создадим функцию gcd, для поиска НОД, и передадим ей на вход два числа, для которых и будем проводить операцию поиска НОД. В теле функции определим промежуточную переменную c, которая будет содержать в себе результат деления по модулю одного входного числа на другое. После этого организуем цикл. Условием завершения его работы станет равенство второго значения нулю. Далее, собственно происходит выполнение алгоритма. Во временную переменную на каждой итерации цикла будет записываться результат деления входных чисел по модулю. После чего первой переменной будет присвоено значение второй, а второй переменной будет присвоено значение временной. В итоге вернём результат работы функции и в методе main запустим её на выполнение для двух пар значений. В результате, для чисел 54 и 26 мы получим НОД равный двум. Ускорение состоит в том, что получение остатка от деления будет сокращать исходные числа гораздо быстрее, поэтому получится сложность O(логН)

```
long euclidusModulus(long a, long b) {
  long c;
  while (b) {
    c = a % b;a = b;b = c;}
    return a; }
  int main (int argc, const char * argv [])
  printf("GCD: %d\n", euclidusModulus(54, 26));
}
```

Рассмотрев простой пример алгоритма Евклида. мы можем говорить о том, как расширить его. За основу расширенного алгоритма Евклида возьмём его простой аналог. В

то время как ”обычный”алгоритм Евклида просто находит наибольший общий делитель двух чисел a и b, расширенный алгоритм Евклида находит помимо НОД коэффициенты x и y. Сумма исходных чисел, умноженных на эти коэффициенты даст нам собственно НОД. Расширенный алгоритм эвклида подразумевает множество внутренних вычислений, поэтому его реализация включает в себя довольно большое количество внутренних переменных для хранения временных результатов вычислений. Начинается алгоритм с простейшего варианта - если число б равно нулю - в таком случае коэффициенты будут 1 и 0 соответственно, а НОД числа а - будет само это число. в противном случае нужно будет начинать совершать сложные манипуляции, а именно: заполнить первую временную переменную результатом целочисленного деления а на б она нужна будет для подсчёта коэффициентов, а вторую пересчитать по формуле а минус первая переменная, умноженная на б. она будет участвовать в дальнейшем пересчёте НОД. подсчитаем коэффициенты согласно текущим значениям временных переменных, а также переприсвоим значения, чтобы правильно считать и впредь. По окончании работы цикла запишем значения корректных коэффициентов и НОД в соответствующие переменные и вернём из функции результат.

```
int ext_gcd(int a, int b, int& x, int& y){
  int q, r, x1, x2, y1, y2,d;
  if (b == 0) {
    d = a, x = 1, y = 0;
    return d;
  }
  x2 = 1, x1 = 0, y2 = 0, y1 = 1;
  while (b > 0) {
    q = a / b;
    r = a - q * b;
    x = x2 - q * x1;
    y = y2 - q * y1;
    a = b, b = r;
    x2 = x1, x1 = x;
    y2 = y1, y1 = y;
  }
  d = a, x = x2, y = y2;
  return d;
}
```

Далее рассмотрим алгоритм - Решето Эратосфена. Название «решето» метод получил потому, что, согласно легенде, Эратосфен писал числа на дощечке, покрытой воском, и прокалывал дырочки в тех местах, где были написаны составные числа. Поэтому дощечка являлась неким подобием решета, через которое «просеивались» все состав-

ные числа, а оставались только числа простые. Т.е. алгоритм Эратосфена используется для поиска всех простых чисел на основе выбранного диапазона. Эратосфен создал таблицу простых чисел до 1000. Для реализации алгоритма Эратосфена нам нужно выполнить пять простых шагов. В первом шаге нам понадобится выбрать из некоего диапазона n целых чисел, начиная от 2 и до n. Вторым шагом надо определить некоторую переменную, например p, которой мы на старте присвоим значение два, т.е. первое значение взятое из списка целых чисел. На третьем шаге нам понадобится исключить из выбранного диапазона числа начиная от 2p до n. Как результат мы будем исключать числа с шагом равным нашей переменной p. На четвёртом шаге нужно найти первое неисключённое из списка число, большее переменной p, в текущем случае, число большее 2. После чего, найденное число вы присвоим нашей переменной p. На пятом шаге нам нужно будет повторить шаги 3 и 4, пока это будет возможно. В результате, все не исключённые числа в искомом диапазоне будут являться простыми. На практике, алгоритм Эратосфена часто улучшают. Например, на третьем шаге, в котором мы исключаем числа из выбранного диапазона, можно исключать числа начиная сразу с числа p**2, потому что все составные числа меньше него уже будут зачеркнуты к этому времени. И, соответственно, останавливать алгоритм можно, когда p**2 станет больше, чем n. Также, все простые числа (кроме 2) — нечётные числа, и поэтому для них можно считать шагами по 2p, начиная с p**2.

Для решения задачи Решето эратосфена мы воспользуемся массивом. о том что такое массив и как он хранит свои значения мы узнаем на следующих уроках. пока что важно только то, что массив может сохранять для нас какие-то значения. например, значения, найденные алгоритмом решета эратосфена. Я предлагаю рассмотреть вариант решета Эратосфена, который не вычёркивает составные числа из готового массива, а выписывает простые числа в отдельный массив. В начале создадим одномерный массив в котором будут храниться простые числа, размером, например, 255 чисел. Инициализируем массив первыми двумя простыми числами. Далее, добавим переменную capacity, которая поможет нам следить за заполнением массива. Организуем цикл в котором будем перебирать все числа подряд. Для этого нам понадобится переменная isPrime, которая будет указывать делится ли текущее число на какое-то из ранее найденных простых. Во вложенном цикле, который будет заниматься перебором всех найденных простых чисел, разместим условие: если текущее число из внешнего цикла делится на текущее значение из внутреннего цикла без остатка, то текущее число внешнего цикла составное. При этом флажок isPrime устанавливается в значение ноль и происходит выход из цикла. По завершении работы внутреннего цикла, при условии что переменная isPrime осталась единицей, что значит что мы прошли по всем простым числам и ни

на одно не смогли поделить текущее проверяемое число, добавляем его к массиву с простыми числами. Также при этой операции увеличивается счётчик заполнения массива capacity. В результате работы программы мы получим массив заполненный простыми числами. После выполнения цикла - можем вывести получившийся массив на экран. Решето Эратосфена один раз просматривает все числа заданного диапазона, что даёт нам сложность O(н) и внутри каждой итерации внешнего цикла идёт с проверкой по массиву с простыми числами, но их количество настолько мало, в сравнении с количеством начальных чисел, что их можно либо не учитывать, либо оставить какую то очень маленькую расчётную сложность. строго говоря, получим O(H*(лог(логH)))

```
  void eratosphen() {
    int arr[255] = {2, 3};
    int capacity = 2;
    for ( int i = 4; i < 255; ++i ) {
      int isPrime = 1;
      for ( int j = 0; j < capacity; ++j ) {
        if (i % arr[j] == 0) {
        isPrime = 0;
        break;
      }
    }
    if (isPrime == 1)
    arr[capacity++] = i;
  }
  for ( int k = 0; k < capacity; ++k ) {
    printf("%d ", arr[k]);
  }
}

int main (int argc, const char * argv []) {
  eratosphen();
  return 0;
}
```

На этом занятии мы программно описали алгоритм Евклида и расширенный алгоритм Евклида, а также технику поиска простых чисел ”решето”Эратосфена. До встречи на следующих уроках.