**IT-Колледж “Сириус”**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**ДОКЛАД**

по дисциплине “Введение в специальность”

на тему “Алгоритмы”

Выполнил:  
Студент группы

1.9.7.1  
Аргун Владислав Гарриевич

Принял:

Старший преподаватель  
Тенигин Альберт Андреевич

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

IT-Колледж “Сириус”  
2022

Содержание

[**1. Введение** 3](#_Toc117193691)

[**1.1.** Каким должен быть алгоритм? 3](#_Toc117193692)

[**1.2.** Основная цель 4](#_Toc117193693)

[**1.3.** Разработка алгоритма 4](#_Toc117193694)

[**1.4.** Реализация алгоритма для конкретного исполнителя 4](#_Toc117193695)

[**1.5.** Способы представления алгоритма 4](#_Toc117193696)

[**1.** **Виды Алгоритмов** 5](#_Toc117193697)

[**1.1.** Бинарный поиск 5](#_Toc117193698)

[2.1.1. Что такое бинарный поиск? 5](#_Toc117193699)

[2.1.2. Принцип работы алгоритма бинарного поиска 5](#_Toc117193700)

[2.1.3. Реализация бинарного поиска 6](#_Toc117193701)

[2.1.4. В каких случаях используют бинарный поиск? 6](#_Toc117193702)

[2.1.5. Разновидности алгоритма 6](#_Toc117193703)

[**2.2.** Сортировка выбором 7](#_Toc117193704)

[2.2.1. Что такое сортировка выбором? 7](#_Toc117193705)

[2.2.2. В чём идея сортировок выбором? 7](#_Toc117193706)

[2.2.3. Двухсторонняя сортировка выбором 7](#_Toc117193707)

[2.2.4. Отличие сортировок выбором от сортировок вставками 8](#_Toc117193708)

[**2.3.** Рекурсия 8](#_Toc117193709)

[2.3.1. Что такое рекурсия? 8](#_Toc117193710)

[2.3.2. Рекурсия в коде и в реальной жизни 8](#_Toc117193711)

[2.3.3. Рекурсия простыми словами 9](#_Toc117193712)

[**2.4.** Алгоритм хеширования 10](#_Toc117193713)

[2.4.1. Что такое хеширование? 11](#_Toc117193714)

[2.4.2. Зачем использовать хеширование? 11](#_Toc117193715)

[2.4.3. Основные операции 12](#_Toc117193716)

[2.4.4. Компоненты хэширования 12](#_Toc117193717)

[**3. Заключение** 13](#_Toc117193718)

**1. Введение**

До недавнего времени алгоритмы были прерогативой программистов. Но теперь они вошли в нашу жизнь и становятся всепроникающими. Алгоритм больше не иностранное слово. Алгоритмическая торговля, алгоритмический уклон, алгоритм Facebook, даже алгоритмическая война — все эти термины стали частью нашего лексикона в последние годы. Некоторые доходят до того, что утверждают, что мы живем в новую эпоху: эпоху алгоритмов. Но алгоритмы не такие уж новые. Мы использовали их, сознательно или нет, сотни и тысячи лет. Алгоритмы — это просто конкретные описания пошаговых действий, которые необходимо выполнить для достижения определенного результата. Они являются одним из наиболее распространенных инструментов обмена знаниями.

Практически любой процесс обучения кого-то тому, как что-то делать, использует алгоритмы.

Однако некоторые аспекты алгоритмов изменились за последние несколько десятилетий. В частности, появление компьютеров означает, что многие алгоритмы сегодня значительно сложнее, чем мы могли себе представить в прошлом.

**Алгоритм** - это совокупность четко определенных действий, выполнение которых ведет к решению задачи.

**1.1.** Каким должен быть алгоритм?

Критерии идеального алгоритма представлены ниже:  
**Дискретность**

Алгоритм должен состоять из последовательно выполняющихся элементарных действий. Пока не закончится выполнение одного действия, следующее не начинается.  
**Детерминированность**  
Все действия в алгоритме должны быть строго определены. Ни одно из них не может двояко трактоваться исполнителем.  
**Конечность**

Алгоритм должен выполняться за определенное количество шагов.  
**Массовость**  
Алгоритм должен вести к результату при любых исходных данных.

**Результативность**

Работа алгоритма всегда должна приводить к ожидаемому результату.

**1.2.** Основная цель

Основная цель алгоритмизации - составление алгоритма для определенного класса задач, решаемых исполнителем.   
Приведем несколько примеров алгоритмов из повседневной жизни:  
Инструкция по эксплуатации бытового прибора - это по сути алгоритм правильного пользования устройством.

Правила дорожного движения (ПДД) - однозначно регламентирует движение всех участников движения.  
Когда водитель соблюдает правила, он действует по определенному алгоритму. Чтобы запустить массовое промышленное производство, нужно определить порядок или алгоритм сборки продукции на конвейере.   
**Алгоритм сборки** - это набор действий, в результате выполнения которого мы получаем продукт.

**1.3.** Разработка алгоритма

Разработку алгоритма можно условно разбить на 3 этапа:   
1. Обдумывание общей идеи алгоритма.

2. Формализация идей.

3. Написание программного кода.

**1.4.** Реализация алгоритма для конкретного исполнителя

На каждом этапе алгоритм принимает разную форму: сначала мы описываем ход решения задачи словами - это “словесное представление”. Затем мы постепенно конкретизируем алгоритм, создаем псевдокод, блок-схемы, и в итоге приходим к цельной конкретизированной реализации алгоритма.

**1.5.** Способы представления алгоритма

**Словесный**  
Описываем алгоритм с помощью алгоритмического языка. Словесная форма представления алгоритма происходит на естественном языке, на том, на котором говорим мы с вами.

**Псевдокод** - это описание алгоритма на искусственном языке с использованием зарезервированных ключевых слов.   
В псевдокоде отсутствует несущественные для понимания алгоритма человеком деталей (написание переменных, системно зависимый код, и т.д.).

**Графический** - способ представления алгоритма с помощью общепринятых графических фигур, называемых блок-схемами, каждая из которых описывает один или несколько шагов алгоритма. Внутри блока записывается описание команд или условий.

**Блок-схема** - это графическое представление алгоритма.

**Программный** способ записи алгоритма – это запись алгоритма на языке программирования, позволяющем на основе строго определенных правил формировать последовательность предписаний, однозначно отражающих смысл и содержание алгоритма, с целью его последующего исполнения на компьютере.

**Программа** - это алгоритм, написанный на языке программирования.

**2. Виды Алгоритмов**

На самом деле есть очень много различных алгоритмов, будут представлены наиболее интересные и популярные.

**1.1.** Бинарный поиск

2.1.1. Что такое бинарный поиск?

Предположим, вы ищете фамилию человека в телефонной книге (не в устройствах). Она начинается с буквы «К». В принципе, можно начать с самого начала и перелистывать страницы, пока вы не доберетесь до буквы «К». Но скорее всего для ускорения поиска лучше раскрыть книгу на середине: ведь буква «К» должна находиться где-то ближе к середине телефонной книги. Или предположим, что вы ищете слово в словаре, и оно начинается с буквы «О». И снова лучше начать с середины.  
  
Следовательно **бинарный поиск** — это тип поискового алгоритма, который последовательно делит пополам заранее отсортированный массив данных, чтобы обнаружить нужный элемент. Другие его названия — двоичный поиск, метод половинного деления, дихотомия.

### 2.1.2. Принцип работы алгоритма бинарного поиска

Принцип работы алгоритма бинарного поиска можно описать следующим образом:  
1. Сортируем массив.   
2. Делим массив пополам и находим середину.   
3. Сравниваем срединный элемент с заданным искомым элементом.   
4. Если искомое число больше среднего, тогда мы продолжаем поиск в правой части массива (т. е. если он отсортирован по возрастанию):   
делим ее пополам, повторяя пункт 3.   
5. Если же заданное число меньше — алгоритм продолжит поиск в левой части массива, снова возвращаясь к пункту 3.

### 2.1.3. Реализация бинарного поиска

Существуют два способа реализации бинарного поиска:  
1.) **Итерационный**  
При таком подходе используется цикл, тело которого повторяется, пока не найдется заданный элемент либо не будет установлено, что его нет в массиве. Пример: в Python для этой цели удобно использовать цикл while.  
  
2.) **Рекурсивный**  
В этом случае пишется функция, которая вызывает сама себя (рекурсивно), пока не будет найден искомый элемент в массиве.

### 2.1.4. В каких случаях используют бинарный поиск?

Бинарный поиск подходит для нахождения позиций элемента в упорядоченном списке: в этом случае он эффективнее линейного, поскольку массив данных на каждом шаге разделяется надвое и одна половина сразу отбрасывается.   
Последовательная сложность двоичного метода в худшем и среднем случаях равна O(log n), в лучшем — O(1) (если обнаруживаем искомый элемент на первой итерации).   
Для сравнения: вычислительная сложность линейного поиска равна O(n) (обычный проход по всем элементам в поисках нужного).

### 2.1.5. Разновидности алгоритма

На основе бинарного поиска разработано несколько дополнительных разновидностей поисковых алгоритмов:

* однородный бинарный поиск, при котором аргумент поиска А сравнивается с ключом Ki, где i — золотое сечение интервала (оно выбирается так, чтобы отношение длины большего отрезка к длине всего интервала равнялось отношению длины меньшего отрезка к длине большего отрезка);
* троичный поиск, когда интервал делится на три части вместо двух. Обычно применяется для поиска положения экстремума функции;
* интерполирующий поиск, который предсказывает позицию нужного элемента на основе разницы значений. Эффективен, если элементы распределены достаточно равномерно;
* дробный спуск — применяется для ускорения двоичного поиска в многомерных массивах данных, и другие.

## **2.2.** Сортировка выбором

### 2.2.1. Что такое сортировка выбором?

Сортировка выбором - это простой алгоритм сортировки данных со сложностью O(n2), где n — количество элементов для сортировки.

### 2.2.2. В чём идея сортировок выбором?

1. Проходим по массиву в поисках максимального элемента.   
2. Найденный максимум меняем местами с последним элементом.   
3. Неотсортированная часть массива уменьшилась на один элемент (не включает последний элемент, куда мы переставили найденный максимум). К ней применяем те же действия — находим максимум и ставим его на последнее место в неотсортированной части массива. И продолжаем до тех пор, пока неотсортированная часть массива не уменьшится до одного элемента.

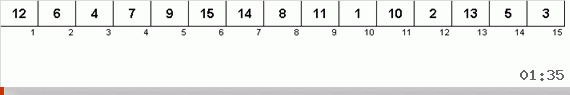


Рисунок 1. Сортировка простым выбором представляет из себя грубый двойной перебор.

Рассмотрим несколько модификаций (Рисунок 1).

### 2.2.3. Двухсторонняя сортировка выбором

Похожая идея используется в шейкерной сортировке [3], которая является вариантом пузырьковой сортировки. Проходя по неотсортированной части массива, мы кроме максимума также попутно находим и минимум. Минимум ставим на первое место, а максимум на последнее. Таким образом, неотсортированная часть при каждой итерации уменьшается на два элемента.

На первый взгляд кажется, что это ускоряет алгоритм в 2 раза — после каждого прохода неотсортированный подмассив уменьшается не с одной, а сразу с двух сторон. Но при этом в 2 раза увеличилось количество сравнений, а число свопов осталось неизменным. Двойной выбор лишь незначительно увеличивает скорость алгоритма, а на некоторых языках даже почему-то работает медленнее

### 2.2.4. Отличие сортировок выбором от сортировок вставками

Может показаться, что сортировки выбором и сортировки вставками — это суть одно и то же, общий класс алгоритмов. Ну, или сортировки вставками — разновидность сортировок выбором. Или сортировки выбором — частный случай сортировок вставками. И там и там мы по очереди из неотсортированной части массива извлекаем элементы и перенаправляем их в отсортированную область.

Главное отличие: в сортировке вставками мы извлекаем из неотсортированной части массива любой элемент и вставляем его на своё место в отсортированной части. В сортировке выбором мы целенаправленно ищем максимальный элемент (или минимальный), которым дополняем отсортированную часть массива. Во вставках мы ищем куда вставить очередной элемент, а в выборе — мы заранее уже знаем в какое место поставим, но при этом требуется найти элемент, этому месту соответствующий.

Это делает оба класса алгоритмов совершенно отличными друг от друга по своей сути и применяемым методам.

## **2.3.** Рекурсия

### 2.3.1. Что такое рекурсия?

Рекурсия — это способ решения проблем с помощью уменьшенных версий одной и той же задачи. То есть основная задача делится на подзадачи и уже их мы решаем при помощи рекурсии. Рекурсией можно назвать функцию, вызывающую саму себя, но с другими значениями параметров. Мы решаем задачу с помощью меньших подзадач до тех пор, пока не достигнем тривиальной версии задачи, то есть базового случая.

В программировании рекурсия тесно связана с функциями, точнее именно благодаря функциям в программировании существует такое понятие как рекурсия или рекурсивная функция.

### 2.3.2. Рекурсия в коде и в реальной жизни

Предположим, вы стоите в длинной очереди людей. Сколько людей находится прямо за вами в очереди?

Правила:

* Один человек может видеть только человека, стоящего прямо впереди и сзади. Таким образом, нельзя просто оглянуться назад и посчитать.
* Каждому человеку разрешается задавать вопросы человеку, стоящему впереди или сзади. Как мы можем решить эту проблему рекурсивно?

Решение:

* Вы смотрите сзади и видите, есть ли там человек. Если нет, то можно вернуть ответ «0». Если есть человек, повторите этот шаг, и дождитесь ответа от человека, стоящего сзади.
* Как только человек получает ответ, он добавляет 1 для человека, стоящего за ним, и отвечает человеку, который его спросил, или человеку, стоящему перед ним.

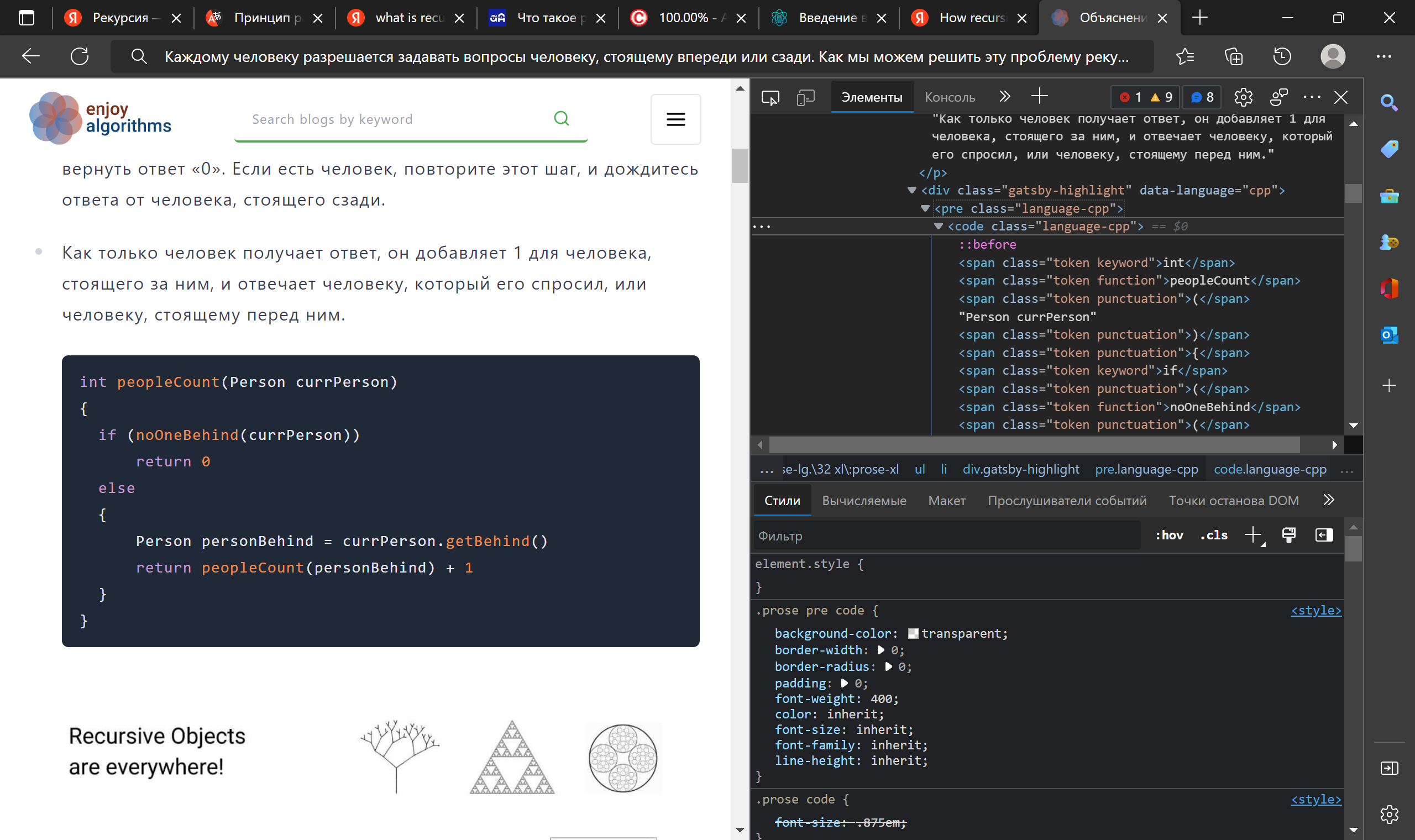


Рисунок 2. Решение задачи, реализованное в коде.

### 2.3.3. Рекурсия простыми словами

Если говорить простыми словами, то рекурсия является определением части функции или метода через саму себя, или же это функция, которая вызывает саму себя, непосредственно (в своём теле) или косвенно (через стороннюю функцию).

**\*** Функция может вызвать себя через промежуточные функции. Вот пример: Функция В вызывается функцией А, а потом снова вызывает функцию А.  
  
Цепочка вызовов не может являться бесконечной, она должна, прервавшись, вернуть ответ. Из-за этого обязательно возникает крайний случай (или несколько), когда функции уже не требует вызова себя с другими параметрами (то есть погружаться ещё глубже), а можно сразу выдать результат.

Стандартным примером рекурсивной функции будет являться вычисление факториала, иными словами произведения натуральных чисел от 1 и до числа n.

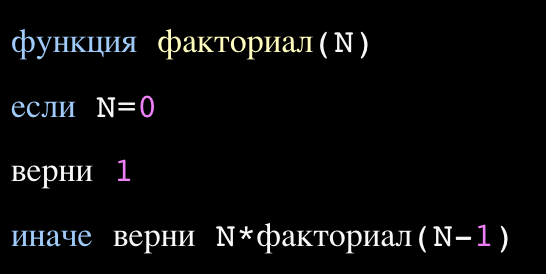


Рисунок 3. Стандартный пример рекурсивной функции.

Здесь **N** = 0 — это крайний случай: функция ничего не вызывает и сразу возвращает единицу (по определению, факториал нуля равен единице).

**(!) Рекурсия — не то же самое, что бесконечный цикл.**

## **2.4.** Алгоритм хеширования

Предположим, мы хотим спроектировать систему для хранения записей сотрудников с номерами телефонов (в виде ключей). И мы хотим, чтобы следующие запросы выполнялись эффективно:

1. Введите номер телефона и соответствующую информацию.
2. Найдите номер телефона и получите информацию.
3. Удалите номер телефона и связанную с ним информацию.

Мы можем подумать об использовании следующих структур данных для хранения информации о различных телефонных номерах.

1. Массив телефонных номеров и записей.
2. Связанный список телефонных номеров и записей.
3. Сбалансированное двоичное дерево поиска с номерами телефонов в качестве ключей.
4. Таблица прямого доступа.

Для массивов и связанных списков нам нужно искать линейно, что может быть дорогостоящим на практике. Если мы используем массивы и сохраняем данные отсортированными, то номер телефона можно искать во времени   
O(log n) с помощью двоичного поиска, но операции вставки и удаления становятся дорогостоящими, так как нам приходится поддерживать отсортированный порядок.

Сосбалансированным двоичным деревом поиска мы получаем умеренное время поиска, вставки и удаления. Все эти операции могут быть гарантированы во времени O(log n).

Еще одно решение, которое можно придумать, это использовать таблицу прямого доступа, где мы создаем большой массив и используем телефонные номера в качестве индекса в массиве. Запись в массиве равна NIL, если номер телефона отсутствует, в противном случае запись массива сохраняет указатель на записи, соответствующие номеру телефона. Временная сложность с точки зрения этого решения является лучшим среди всех, мы можем сделать все операции за O(1) время. Например, чтобы вставить номер телефона, мы создаем запись с деталями данного номера телефона, используем номер телефона в качестве индекса и сохраняем указатель на созданную запись в таблице.

Это решение имеет много практических ограничений. Первая проблема с этим решением заключается в том, что дополнительное пространство требуется огромно. Например, если номер телефона состоит из n цифр, нам нужно   
O(m \* 10n) для таблицы, где m — размер указателя для записи. Другая проблема заключается в том, что целое число в языке программирования может не хранить n цифр.

Из-за вышеуказанных ограничений таблица прямого доступа не всегда может быть использована.

### 2.4.1. Что такое хеширование?

**Хеширование** - это решение, которое может быть использовано почти во всех таких ситуациях и работает очень хорошо по сравнению с вышеуказанными структурами данных, такими как Array, Linked List, Balanced BST на практике. При хешировании мы получаем O(1) время поиска в среднем (при разумных предположениях) и O(n) в худшем случае. Теперь давайте разберемся, что такое хеширование.

Хэширование является популярным методом для хранения и извлечения данных как можно быстрее. Основная причина использования хеширования заключается в том, что оно дает оптимальные результаты, поскольку выполняет оптимальный поиск.

### 2.4.2. Зачем использовать хеширование?

Если вы внимательно наблюдаете, в сбалансированном двоичном дереве поиска, если мы пытаемся искать, вставлять или удалять какой-либо элемент, то сложность времени для того же самого равна O(log n). Теперь может возникнуть ситуация, когда наши приложения захотят выполнять те же операции быстрее, т.е. более оптимизированным способом, и здесь в игру вступает хеширование. При хешировании все вышеперечисленные операции могут быть выполнены в O(1), т.е. постоянном времени. Важно понимать, что наихудшая сложность времени для хэширования остается O(n), но средняя сложность времени случая равна O(1).

### 2.4.3. Основные операции

Давайте разберемся в нескольких основных операциях хеширования.

Основные операции:

* **HashTable:** Эта операция используется для создания новой хэш-таблицы.
* **Delete:**Эта операция используется для удаления определенной пары ключ-значение из хэш-таблицы.
* **Get:**Эта операция используется для поиска ключа внутри хэш-таблицы и возврата значения, связанного с этим ключом.
* **Put:**Эта операция используется для вставки новой пары ключ-значение в хэш-таблицу.
* **DeleteHashTable:**Эта операция используется для удаления хэш-таблицы

### 2.4.4. Компоненты хэширования

Компоненты хэширования бывают следующие:

**1) Hash Table** (Хэш-таблица):массив, в котором хранятся указатели на записи, соответствующие заданному номеру телефона. Запись в хэш-таблице равна NIL, если ни один из существующих телефонных номеров не имеет значения хэш-функции, равного индексу записи. Простыми словами можно сказать, что хэш-таблица является обобщением массива. Хэш-таблица предоставляет функциональность, в которой коллекция данных хранится таким образом, что при необходимости легко найти эти элементы позже. Это делает поиск элемента очень эффективным.

**2)** **Hash Function**(Хэш-функция): функция, которая преобразует заданный большой номер телефона в небольшое практическое целое значение. Сопоставленное целое значение используется в качестве индекса в хэш-таблице. Итак, простыми словами можно сказать, что хэш-функция используется для преобразования заданного ключа в определенный индекс слота. Его основная задача состоит в том, чтобы сопоставить каждый возможный ключ в уникальный индекс слота. Если каждый ключ сопоставлен с уникальным индексом слота, то хэш-функция известна как идеальная хэш-функция. Создать идеальную хэш-функцию очень сложно, но наша задача как программиста состоит в том, чтобы создать такую хэш-функцию, с помощью которой количество коллизий как можно меньше. Столкновение обсуждается впереди.

**Хорошая хэш-функция** должна иметь следующие свойства:

1. Эффективно вычислимость.
2. Следует равномерно распределять ключи (каждая позиция стола одинаково вероятна для каждой).
3. Следует свести к минимуму столкновения.
4. Должен иметь низкий коэффициент загрузки (количество элементов в таблице, деленное на размер таблицы).

Например, для телефонных номеров плохая хэш-функция заключается в том, чтобы взять первые три цифры. Лучшей функцией является рассмотрение последних трех цифр. Обратите внимание, что это может быть не лучшая хэш-функция. Могут быть лучшие способы.

**3) Collision Handling** (Обработка коллизий): поскольку хэш-функция дает нам небольшое число для большого ключа, существует вероятность того, что два ключа приводят к одному и тому же значению. Ситуация, когда вновь вставленный ключ сопоставляется с уже занятым слотом в хэш-таблице, называется коллизией и должна обрабатываться с помощью некоторой техники обработки коллизий. Ниже приведены способы обработки коллизий:

* **Chaining** (Связывание): Идея состоит в том, чтобы каждая ячейка хэш-таблицы указывала на связанный список записей, которые имеют одно и то же значение хэш-функции. Сцепление простое, но требует дополнительной памяти за пределами таблицы.
* **Open Addressing** (Открытая адресация):При открытой адресации все элементы хранятся в самой хэш-таблице. Каждая запись таблицы содержит либо запись, либо NIL. При поиске элемента мы исследуем слоты таблицы один за другим, пока не будет найден нужный элемент или не станет ясно, что элемента нет в таблице.

# **3. Заключение**

В заключение можно сказать, что алгоритмы действительно являются простейшим описанием действий, которое записано согласно заданным правилам и требованиям. Очень часто для записи используется специальная интуитивно понятная терминология. Также они являются одним из наиболее распространенных инструментов обмена знаниями.

На основе работы алгоритмов строятся не только прикладные, повседневные задачи, но и трудные компьютерные программы, написание которых требует понимания выполнения каждой из задач в отдельности. Именно поэтому начало изучения программирования зачастую начинается со знакомства с основными алгоритмами, некоторые из которых я рассмотрел выше.

В данном докладе я разобрал классификацию алгоритмов и детально изучил некоторые из видов, их особенности и принципы работы. Во время его написания была подробно изучена новая информация, были получены новые знания, благодаря которым я смогу лучше понимать многие аспекты программирования.

**Интернет-ресурсы:**

1. <https://habr.com/ru/post/422085/>2. <https://habr.com/ru/post/275813/>

3. <https://habr.com/post/414653/#shaker>

4. <https://habr.com/post/415935/>

5. <https://skillbox.ru/media/code/rekursiya-vokrug-nas/>

6. <https://blog.skillfactory.ru/glossary/binarnyj-poisk/>