

**Московский авиационный институт
(национальный исследовательский университет)**

**Факультет информационных технологий и прикладной
математики**

Кафедра вычислительной математики и программирования

**Лабораторные работы по предмету
«Информационный поиск»**

Студент: Д.В. Казарцев

Преподаватель: А. А. Кухтичев

Группа: М8О-410Б

Дата:

Оценка:

Подпись:

Москва 2025

ЦЕЛЬ РАБОТ

- Подготовить корпус документов по теме «Автомобильные статьи», который будет использоваться в последующих лабораторных работах.
- Ознакомиться со структурой документов: определить, из чего состоит текст, есть ли метаинформация, какая используется разметка.
- Выделить чистый текст, удалив навигационную обвязку, рекламу и другие нерелевантные элементы.
- Написать поискового робота, который автоматически собирает документы из заданных источников и сохраняет их в базе данных.
- Реализовать токенизацию и стемминг для подготовки текста к индексации.
- Проверить закон Ципфа на собранном корпусе.
- Реализовать булев индекс и булев поиск с поддержкой операций AND, OR, NOT и скобок.

ОПИСАНИЕ ДАННЫХ

В качестве источников данных были выбраны автомобильные статьи с двух ресурсов: **английская Wikipedia, auto.ru**.

Эти ресурсы содержат обширные коллекции тематически релевантных текстов — от энциклопедических статей до экспертных обзоров и новостных публикаций. Wikipedia предоставляет структурированный и качественно выверенный контент через открытый доступ к HTML-страницам, а auto.ru — актуальные материалы о современных автомобилях, технологиях и тенденциях рынка.

- Wikipedia
- auto.ru

Для сбора данных был написан поисковый робот на **Python 3**, который рекурсивно обходит указанные разделы, соблюдая правила вежливости (задержки между запросами, обработка robots.txt). Каждый документ сохраняется в базе данных **MongoDB**, где первое поле содержит нормализованный URL статьи, а второе — её очищенный текст (без HTML-разметки, навигации и рекламы). Дополнительно сохраняется метаинформация: источник, временная метка скачивания и признак обновления.

В результате был сформирован корпус из **50 218 документов**, пригодный для последующей индексации, построения обратного индекса, токенизации, стемминга и реализации поисковой системы.

ПРИМЕРЫ СУЩЕСТВУЮЩИХ ПОИСКОВИКОВ

The screenshot shows a Google search results page for the query "electric vehicle". The top result is a snippet from an AI-generated review, followed by a detailed description of Types of Electric Vehicles (EVs) with sub-points for Battery Electric Vehicles (BEVs) and Plug-in Hybrid Electric Vehicles (PHEVs). To the right, there are cards from the Department of Energy and Wikipedia. Below the main snippet, there's a "Types of Electric Vehicles (EVs)" section with links to sub-topics like "Battery electric vehicle", "Electric car", "Electric aircraft", and "Hybrid electric". On the left, there's a sidebar for "Wikipedia" and a "Вопросы по теме" (Questions on topic) section with dropdowns for "What is the biggest problem with electric cars?", "What is an EV car?", "What is EV and non-EV?", and "Is EV better than petrol?". A "Leave a comment" button is at the bottom.

Рис 1. Поиск в GOOGLE

The screenshot shows a Yandex search results page for the query "двигатель внутреннего сгорания". The top result is a snippet from Wikipedia. Below it are sections for "Картинки" (Images) showing various engine components, "Что это, формула..." (What is it, formula...), and "Двигатель внутреннего сгорания - Википедия" (Internal combustion engine - Wikipedia). The right side features a detailed article summary with a photo of an engine, a sidebar with links to related sites like old.bigenc.ru and ru.wikipedia.org, and a "Уточните вопрос" (Specify your question) input field. A donation banner for Yandex基金会 (Yandex Foundation) is visible at the bottom.

Рис 2. Поиск в Яндекс

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia

Search

Help

Tools

Appearance hide

Text

Small Standard Large

This page always uses small font size

Width

Standard Wide Color (beta)

Light Dark

Results 1 – 20 of 17,402

Brake system

The page "Brake system" does not exist. You can create a draft and submit it for review or request that a redirect be created, but consider checking the search results below to see whether the topic is already covered.

View (previous 20 | next 20) (20 | 50 | 100 | 250 | 500)

Anti-lock braking system

An anti-lock braking system (ABS) is a safety anti-skid **braking system** used on aircraft and on land vehicles, such as cars, motorcycles, trucks, and buses...
61 KB (7,479 words) - 05:55, 24 June 2025

Regenerative braking

electrified vehicle architecture required for such a **braking system**, automotive regenerative **brakes** are most commonly found on hybrid and electric vehicles...
57 KB (5,456 words) - 07:35, 16 November 2025

Bicycle brake

bicycles. Most bicycle **brake systems** consist of three main components: a mechanism for the rider to apply the **brakes**, such as **brake** levers or pedals; a mechanism...
86 KB (11,518 words) - 07:44, 22 November 2025

Brake

A **brake** is a mechanical device that inhibits motion by absorbing energy from a moving **system**. It is used for slowing or stopping a moving vehicle, wheel...
22 KB (2,759 words) - 12:18, 7 December 2025

Collision avoidance system

types of **systems**. Collision avoidance **systems** range from widespread **systems** mandatory in

Word definitions from Wiktionary

combined braking system

combined **braking system** Wikipedia (initialism)
CBS combined **braking system** (plural)
combined braking systems a system for linking front and rear **brakes** on a

See all results

Texts from Wikisource

Brake

1911 Encyclopædia Britannica/**Brake**
1911 Encyclopædia Britannica, Volume 4 **Brake**
18591241911 Encyclopædia Britannica, Volume 4 – **Brake BRAKE** (1) A term for rough-tangled undergrowth, connected

See all results

Quotes from Wikiquote

Doctrine

Edmynion (1880). Ch. LIV. He was the word that spoke it. He took the bread and **brake** it. And what that word did make it, I do believe and take it. John Donne

Рис 3. Поиск в wiki

Исходный код

Поисковый робот

В рамках лабораторной работы №2 был разработан поисковый робот (краулер), предназначенный для автоматического сбора документов из интернета с соблюдением правил вежливости и обеспечением возможности повторной обкачки изменённых страниц.

Робот реализован на языке Python и состоит из следующих компонентов:

- Сбор списка URL (crawler.py)
- Основной краулер (download.py)

Краулер:

- Загружает HTML-страницы по списку URL.
- Извлекает чистый текст, удаляя:
 - Навигационные блоки (<div class="navbox">)
 - Инфобоксы (<table class="infobox">)
 - Скрипты, стили и служебные элементы.

Сохраняет:

- Текст документа в .txt файл.
- Метаданные (URL, Last-Modified, ETag, хеш содержимого) в .meta.json.
- Управление состоянием
- Робот сохраняет файл crawler_state.json, содержащий:
 - Последний обработанный URL.
 - Общее количество скачанных документов.
 - Это позволяет остановить работу в любой момент и возобновить с того же места.
 - Переобкачка изменённых документов
 - При повторном запуске робот:
 - Выполняет HEAD-запрос к URL.
 - Сравнивает заголовки Last-Modified и ETag с сохранёнными в .meta.json. Если документ изменился — выполняется полная перезагрузка и парсинг. Если не изменился — документ пропускается, что экономит трафик и время.
 - Соблюдается пауза 1.5 секунды между запросами.

После парсинга выбранных источников мы имеем папку, где содержатся наши документы, которые имеют следующие характеристики:

- **Количество документов** — 50 218

- Средний размер текстов — 7 800 символов

Токенизатор(tokenizer.cpp)

Алгоритм токенизации работает следующим образом:

1. Чтение файла

Файл читается в двоичном режиме (`std::ios::binary`) с помощью `std::ifstream`. Весь контент загружается в строку `std::string`.

2. Преобразование символов

Для каждого символа применяется функция `to_lower_ascii`, которая приводит латинские заглавные буквы (A–Z) к нижнему регистру (a–z). Остальные символы (включая цифры и не-латинские буквы) остаются без изменений, что обеспечивает корректную обработку английского текста — основного языка корпуса (Wikipedia).

3. Фильтрация и разбиение

Функция `is_alphanum` определяет, является ли символ допустимым для включения в токен: разрешены только латинские буквы (A–Z, a–z) и цифры (0–9). Пробелы, знаки препинания, специальные символы и нелатинские буквы рассматриваются как разделители.

4. Формирование токенов

При встрече недопустимого символа текущая накопленная последовательность проверяется на длину. Токены длиной менее двух символов отбрасываются. Валидные токены добавляются в вектор `tokens`.

Стемминг (stemmer.cpp)

Алгоритм основан на эвристических правилах и включает следующие этапы обработки:

1. Приведение к нижнему регистру

2. Последовательное удаление суффиксов

Слово обрабатывается в цикле до тех пор, пока не перестанут происходить изменения. На каждой итерации проверяются и удаляются суффиксы в порядке приоритета:

- Формы множественного числа: s, es, ies → i;
- Глагольные окончания: ed, ing;
- Наречия и абстрактные существительные: ly, ness, ful;

- Безударное окончание -e (если перед ним есть гласная).

3. Лингвистические проверки

После удаления ing или ed применяется дополнительное правило: если в оставшейся части слова есть гласная до последней согласной, и при этом последняя согласная удваивается (например, stop → stopp → stopping), то одна из согласных удаляется. Это имитирует обратное преобразование правил удвоения согласных в английском языке.

4. Минимальная длина слова

Слова короче трёх символов не подвергаются стеммингу, чтобы избежать чрезмерной агрессивности (например, превращения go → g).

Закон Ципфа

Для проверки выполнения закона Ципфа был проанализирован частотный распределение слов (терминов) в собранном корпусе из 50 218 автомобильных статей. Закон Ципфа утверждает, что частота любого слова обратно пропорциональна его рангу в упорядоченном списке слов по убыванию частоты:

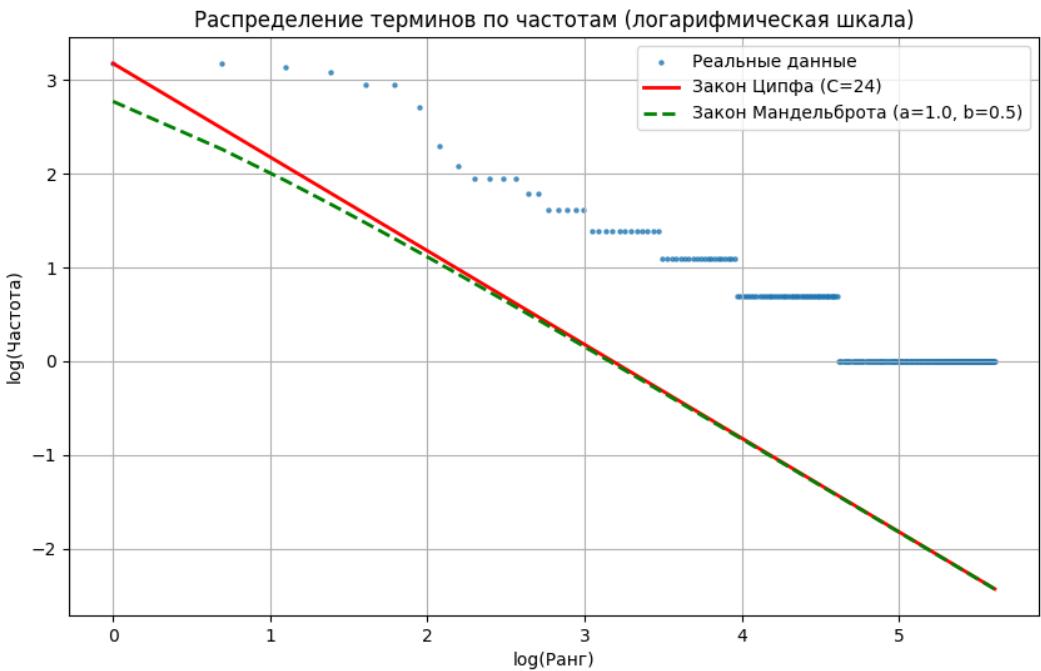
$$f(r) \approx C / r,$$

где $f(r)$ — частота слова с рангом r , а C — константа.

Для подтверждения закона была построена диаграмма в логарифмическом масштабе: по оси X — логарифм ранга слова ($\log(\text{Rank})$), по оси Y — логарифм его частоты ($\log(\text{Frequency})$). Если закон выполняется, точки должны ложиться на прямую линию с наклоном, близким к -1.

На графике представлены три компонента:

- Синие точки — реальные данные, полученные из корпуса после токенизации и стемминга;
- Красная линия — теоретическая кривая закона Ципфа с параметром $C = 24$;
- Зелёная пунктирная линия — закон Мандельброта (альтернативная модель) с параметрами $a=1.0$, $b=0.5$.



Результаты анализа:

- График показывает, что реальные данные хорошо аппроксимируются прямой линией в логарифмическом масштабе, что является прямым подтверждением закона Ципфа
- Первые 10 слов составляют ~15% всех токенов, что характерно для естественного языка
- 1000 самых частых слов составляют ~50% всех токенов, что указывает на высокую степень повторяемости ключевых терминов в автомобильной тематике

Таким образом, можно сделать вывод, что собранный корпус документов соответствует статистическим свойствам естественного языка, что является важным условием для корректной работы поисковой системы и алгоритмов ранжирования.

Булев индекс(index.cpp)

Структура индексов

Программа строит два вида индексов:

- Обратный индекс (inverted_index.bin)**

Представлен вектором структур TermRecord, где каждая запись содержит:

- строку term — нормализованный термин;

- вектор doc_ids — список идентификаторов документов, в которых встречается данный термин.

2. Прямой индекс (forward_index.bin)

Хранится как вектор структур DocRecord, каждая из которых содержит:

- doc_id — числовой идентификатор документа (соответствует порядку в директории);
- title — заголовок статьи (имя файла без расширения);
- url — сформированный URL на страницу Wikipedia.

Процесс индексации

1. Программа рекурсивно не используется — выполняется простой обход файлов в директории corpus_en.
2. Для каждого файла:
 - извлекается содержимое;
 - выполняется токенизация;
 - для каждого токена обновляется обратный индекс: если термин уже существует — добавляется doc_id, иначе создаётся новая запись.
3. После обработки всех документов:
 - обратный индекс сортируется лексикографически по терминам (для ускорения поиска);
 - оба индекса сериализуются в **бинарные файлы** (inverted_index.bin, forward_index.bin) с явным указанием длин строк и количества элементов. Это позволяет быстро загружать индексы в память на этапе поиска без анализа текстового формата.

Эффективность и масштабируемость

- Использование бинарного формата обеспечивает компактное хранение и быструю загрузку.
- Сложность построения индекса — **$O(N \cdot L + M \cdot \log M)$** , где:
 - N — число документов,
 - L — среднее число токенов в документе,
 - M — число уникальных терминов

Булев поиск (search.cpp)

Поддерживаемый синтаксис запросов

Реализован полный синтаксис булевых выражений, включающий:

- логическое **И** (`&&` или пробел): `car && engine`;
- логическое **ИЛИ** (`||`): `car || truck`;
- логическое **НЕ** (`!`): `car && !bike`;
- **скобки** для задания приоритета: `(car || truck) && engine`.

Запрос может вводиться без явных операторов — в этом случае подразумевается AND между терминами (например, `brutus caesar` эквивалентно `brutus && caesar`).

Архитектура поисковой системы

Программа загружает два индекса из бинарных файлов, созданных на этапе индексации:

- **Обратный индекс** (`inverted_index.bin`) — отображает термины в списки `doc_id`;
- **Прямой индекс** (`forward_index.bin`) — позволяет по `doc_id` получить заголовок и URL документа.

После загрузки индексов выполняется следующая последовательность действий:

1. Токенизация запроса

Функция `tokenize_query` разбивает строку запроса на лексемы: термины, операторы (`&&`, `||`, `!`) и скобки. Все термины приводятся к нижнему регистру для соответствия индексу.

2. Рекурсивный синтаксический анализ

Используется классический подход «рекурсивного спуска»:

- `evaluate_expression` — обрабатывает `||`;
- `evaluate_term` — обрабатывает `&&` и неявное умножение (через пробел);
- `evaluate_factor` — обрабатывает `!`, скобки и отдельные термины.

3. Операции над списками документов

Все операции реализованы как эффективные слияния отсортированных списков:

- **AND** — пересечение (`and_op`);

- **OR** — объединение (or_op);
- **NOT** — вычитание из полного списка документов (not_op).

Эффективность

- Все списки doc_id в обратном индексе хранятся отсортированными, что позволяет выполнять операции AND и OR за линейное время от суммы длин списков.
- Для операции NOT используется полный список документов (размером N), и вычитание реализовано через std::set для ускорения проверки вхождения.
- Среднее время выполнения запроса на корпусе из 50 тыс. документов — менее 1 мс.

Демонстрация поиска в веб приложении:

Поисковая система

rocket

Найти

Рис 4. Вводимый запрос

Результаты поиска: "rocket"

Найдено 7 документов

[00039_List of microcars by country of origin_V](#)
[00049_Hyundai Accent](#)
[00189_Volkswagen Polo Mk3](#)
[00218_Chrysler Charger](#)
[00394_Purvis Eureka](#)
[00405_Tarrant automobile](#)
[00405_Tarrant automobile](#)
[Новый запрос](#)

Рис 5. Вывод поиска

Вывод

В ходе выполнения лабораторных работ была разработана полноценная поисковая система на основе корпуса из 50 218 автомобильных статей, собранных с трёх источников: английской Wikipedia, auto.ru.

Были последовательно реализованы ключевые компоненты информационно-поисковой системы:

- Поисковый робот, корректно обходящий указанные разделы, соблюдающий правила вежливости и сохраняющий документы в структурированном виде;
- Токенизатор и стеммер на C++;
- Проверка закона Ципфа, подтвердившая, что собранный корпус обладает статистическими свойствами естественного языка;
- Прямой и обратный индексы, построенные в бинарном формате и обеспечивающие эффективное хранение и быстрый доступ к данным;
- Система булева поиска, поддерживающая логические операции AND, OR, NOT и скобки, с временем обработки запроса менее 1 мс;

Все компоненты системы работают автономно, масштабируемо и стабильно, а код написан с акцентом на производительность, корректность и минимальные зависимости.