

# **Курсовая работа по курсу дискретного анализа: "Текстовый поиск"**

Выполнил студент группы М8О-306Б-21 МАИ *Оружеский Вячеслав Русланович*.

## **Условие**

Ваша программа должна читать входные данные из стандартного потока ввода и выводить ответ на стандартный поток вывода. Реализуйте инвертированный индекс, затем проведите поиск текстов содержащих заданные наборы слов.

### **Формат ввода**

В первой строке входного файла вам дано число  $n$  — количество текстов. В следующих  $n$  строках даны тексты, по одному тексту в строке, представленные наборами слов разделённые пробелами. Далее задано  $t$  — количество запросов в файле. В следующих  $t$  строках вам даны запросы по одному в строке представленные наборами слов разделённых пробелами.

### **Формат вывода**

В ответ на каждый запрос выведите список подходящих документов в виде:  $c_i$  — количество подходящих под запрос документов и далее номера текстов в которых встречались все слова из  $i$ -го запроса. Нумерация документов начинается с 0.

## **Теория**

### **Текстовый поиск**

Текстовый поиск - это процесс поиска слов или множества слов внутри текстов или текстовых файлов. В случае поставленной задачи, о порядке слов или их взаимоположении речи не идёт, требуется лишь найти текст, в котором будут содержаться все слова, включенные в запрос. Примером того, где может пригодиться подобный поиск, является веб-поиск, таргетированная реклама (чтобы найти страницы, содержащие определенные ключевые слова). Также это зачастую используется при поиске в больших базах данных или наборе текстовых файлов, когда нужно найти часть информации, содержащую какие-то конкретные сведения (например, все документы связанные с определённым ФИО). Поэтому, текстовый поиск является неотъемлемым решением во многих системах, связанных с поиском данных (веб, ведение реестров, базы данных и т.д.)

### **Инвертированный (обратный) индекс**

Инвертированный индекс - это структура данных, которая используется для поиска по тексту. В этой структуре данных каждому уникальному слову из набора текстов в соот-

вествии ставится список параметров, в данной задаче - индексов, которые указывают на то, в каком тексте было встречено данное слово. Зачастую инвертированный индекс может быть дополнен дополнительной информацией о вхождении слова (позиция в тексте/строке, падеж и т.д). В случае, если в параметрах слова указывается падеж, то в самом индексе хранится не целое слово, а лишь его основа (без окончания). Данная структура данных применяется при поиске по тексту, если запрос состоит из одного слова, то ответ - список, соответствующий данному слову, если же слов несколько, то ответом является пересечение списков для каждого из этих слов.

## **Словарь**

Словарь - это структура данных, специальным образом хранящая элементы в формате пары "ключ:значение". В словарях, ключ всегда должен быть уникален, данные могут дублироваться, в значении могут находиться самые различные структуры. Наиболее распространённой реализацией является хэш-таблица, в которой доступ к элементу (поиск, удаление, добавление) производится за  $O(1)$ , не считая времени на разрешение коллизии.

## **Очередь**

Очередь - структура данных, которая имеет тип FIFO (first in, first out), то есть первый добавленный элемент уходит первым. По сути, данная структура данных является аналогом бытовой очереди, когда человек приходит к конец очереди, а уходит из её начала (если он смог достоять). Поэтому, добавление элемента возможно лишь в конец, взятие элемента - только из начала, при этом выбранный элемент из очереди удаляется. Так как, мы постоянно оперируем с концами нашей структуры данных, то для реализации чаще всего используются структуры, напоминающие связные списки

## **Метод решения**

В данной задаче для решения поиска множества слов в данных текстах, будет использоваться инвертированный индекс. Для его построения будем использовать хэш-таблицу (`std::unordered_map`), которая будет хранить пары <строка:очередь>. В качестве очереди также используем решение из STL - `std::queue`. По сути, очередь будет представлять из себя множество номером текстов, в которых встретилось слово. Почему используется очередь? Потому что нас просят выводить индексы текстов в порядке возрастания, так как мы идём по текстам по порядку, то и в инвертированный индекс номера текстов будут добавляться в порядке возрастания. Более того, для очереди есть довольно очевидная реализация пересечения двух множеств, которая работает за линейное время.

### **Построение инвертированного индекса**

1. Считываем количество текстов;

2. Для каждого текста:
- Считываем текст;
  - Делим текст на слова;
  - Если это слово ещё не встречалось в данном тексте, то добавляем номер данного текста в очередь для этого слова в инвертированном индексе.

## Обработка запросов

- Считываем количество запросов;
- Для каждого запроса:
  - Считываем запрос;
  - Делим запрос на слова;
  - Ищем по словам нужные нам списки индексов
  - Ищем пересечения между взятыми списками
  - Выводим результат.

## Описание программы

- Реализуем функцию пересечения двух множеств. Идём по каждому элементу, если совпадает - оставляем, если нет, выбрасываем.
- Реализуем разделение строк по словам через объект istream, который работает с потоком ввода
- Считываем кол-во текстов, считываем их, делим по словам, строим инвертированный индекс.
- Игнорируем все символы в потоке ввода до новой строки, чтобы после getline можно было использовать считывание числа
- Повторяем п.3 для запросов
- К каждому списку индексов применяем последовательно пересечение с последующим

Представление кода на языке C++:

---

```
#include <bits/stdc++.h>

//  
std::queue<int> Intersection(std::queue<int> left, std::queue<int>  
right) {
```

```

std::queue<int> result;

while (!left.empty() and !right.empty()) {
    if (left.front() > right.front()) {
        right.pop();
    } else if (left.front() < right.front()) {
        left.pop();
    } else { //
        result.push(left.front());
        left.pop();
        right.pop();
    }
}

return result;
}

// Separation
std::vector<std::string> Separation(std::string &&text) {
    std::vector<std::string> res;
    std::istringstream iss(text);
    while (iss >> text) {
        res.push_back(text);
    }
    return res;
}

int main() {
    // std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);

    int texts, requests;
    std::unordered_map<std::string, std::queue<int>> index_words; // ( - )
    std::cin >> texts;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    //

    for (int i(0); i < texts; ++i) {
        std::string text;
        getline(std::cin, text);

```

```

        for (const auto &word: Separation(std::move(text))) {
            if (index_words[word].empty() || (index_words[word].back() != i)) {
                index_words[word].push(i);
            }
        }
    }

    std::cin >> requests;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');

    for (int idx(0); idx < requests; ++idx) {
        std::string tmp;
        getline(std::cin, tmp);
        const auto words = Separation(std::move(tmp));
        std::queue<int> answer = index_words[words[0]];

        for (int i(1); i < words.size(); ++i) {
            answer = Intersection(answer, index_words[words[i]]);
            if (answer.empty()) { // break;
            }
        }
    }

    std::cout << answer.size() << ',';

    while (!answer.empty()) {
        std::cout << answer.front() << ',';
        answer.pop();
    }
    std::cout << '\n';
}
return 0;
}

```

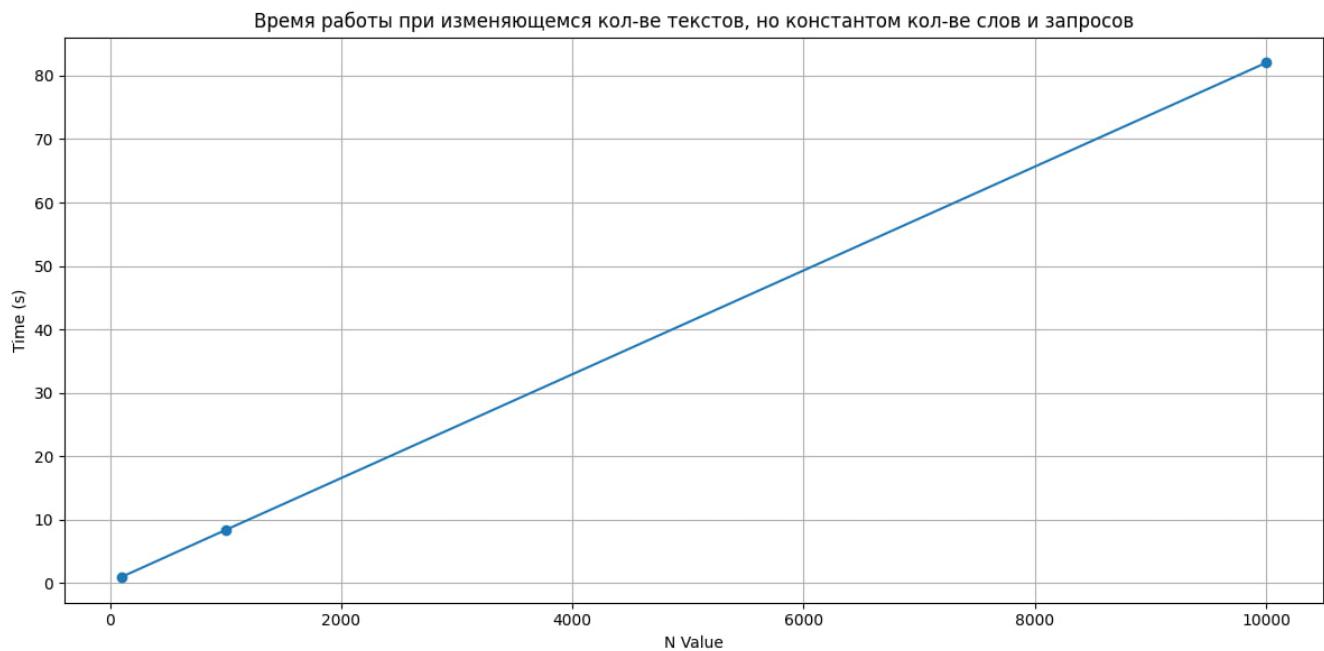
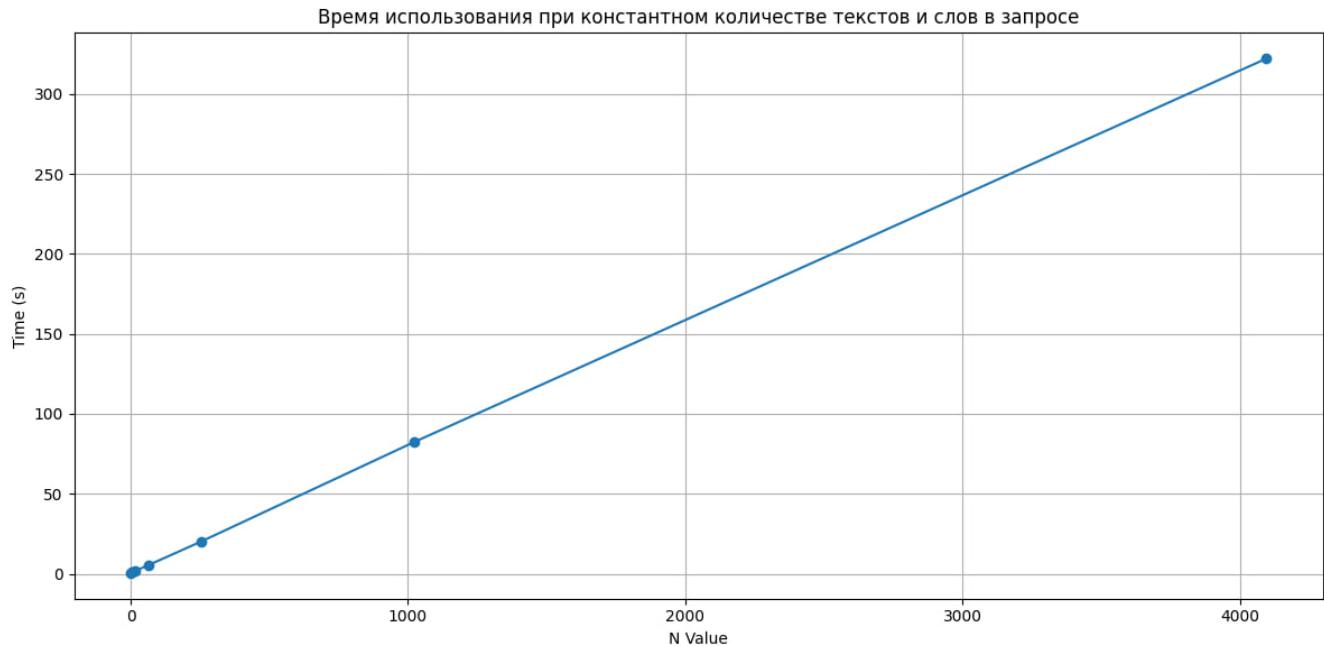
---

## Асимптотическая сложность

Так как, обратный индекс был реализован на хеш-таблице, где вставка работает за  $O(1)$ , за исключением ситуаций, когда будет вызвана обработка коллизий (что в нашем случае довольно большая редкость), то асимптотическая сложность построения индекса составляет  $O(NW)$ , где  $N$  - количество текстов, а  $W$  - наибольшее количество слов в тексте. Асимптотическая сложность обработки запроса составляет  $O(M(WN))$ , где  $M$  - количество запросов, а  $N$  - количество текстов. Таким образом, общая асимптотическая сложность программы составляет  $O(NW + M(WN))$ .

## Тест производительности

Из асимптотической сложности, видно, что по сути, сложность линейно зависит от каждого из двух параметров (кол-ва слов в строке, кол-ва строк), поэтому изменяя так или иначе один из параметров или сразу оба мы будем линейную зависимость, начиная с какого-либо значительного числа для параметров.



## **Выводы**

Во время выполнения курсовой работы, была изучена структура данных под названием инвертированный индекс, а также способ её использования при поиске слов в тексте. Была оценена асимптотическая сложность и проведена оценка реальной зависимости времени выполнения от входных данных. Реализовал алгоритм пересечения множеств на очередях, при чём предварительно множества должны быть отсортированы. Узнал, как можно искать какой-то набор слов среди большого кол-ва документов и где это может быть использовано.