МИНОБРНАУКИ РОССИИ

*Федеральное государственное бюджетное образовательное учреждение высшего образования*

***«МИРЭА – Российский технологический университет»***

**РТУ МИРЭА**

---

Отчет по практической работе №6.2

**Тема:**

**«Поиск образца в тексте»**

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнила студент группы ИКБО-42-23 Туляшева А.Т.

Принял ассистент Муравьева Е.А.

Москва 2024

**Содержание**

## ЦЕЛЬ РАБОТЫ

Освоить приемы реализации алгоритмов поиска образца в тексте.

## ЗАДАНИЕ 1

Необходимо ответить на вопросы:

1. Что такое строка, ее префикс и сумма?

Строка — это последовательность символов, которая может представлять текстовые данные. В программировании строки часто используются для хранения и манипуляции текстом.

Префикс строки — начальные символы последовательности (с начального по максимум предпоследний). Например, для строки "abc" префиксами будут: "", "a", "ab" и "abc".

Суффикс строки — это окончание последовательности символов. Например, для строки "abc" суффиксами будут: "abc", "bc", "c" и "" (пустая строка).

2. Объясните идею алгоритма последовательного (наивного) поиска шаблона в строке. Какова асимптотическая сложность наивного поиска подстроки в строке?

Наивный поиск шаблона заключается в том, что мы последовательно проверяем, начинается ли подстрока (шаблон) с каждой позиции в строке. Алгоритм работает так:

1. Начинаем с первой позиции в строке и сравниваем символы строки и шаблона.

2. Если все символы совпадают, мы нашли подстроку.

3. Если не совпадают, продолжаем проверку, сдвигаясь на одну позицию вправо.

Асимптотическая сложность наивного поиска составляет O(n  m), где n — длина строки, а m — длина шаблона. Это потому, что в худшем случае мы можем сравнить каждый символ строки с каждым символом шаблона.

3. В чём идея поиска образца алгоритмом Бойера–Мура?

Алгоритм Бойера–Мура использует недостатки за счет предобработки шаблона. Основная идея заключается в том, чтобы сравнивать шаблон и строку с конца и применять два метода сдвига:

1. Сдвиг по плохому символу: если символ не совпадает, сдвигаем шаблон так, чтобы последний совпадающий символ совпадал с текущим символом строки, или, если такой символ отсутствует в шаблоне, сдвигаем шаблон за пределы текущей позиции.

2. Сдвиг по хорошему суффиксу: если часть шаблона совпала, но последующий символ — нет, сдвигаем шаблон так, чтобы следующая возможная позиция совпадала (если такая существует).

Эти методы позволяют избегать лишних сравнений, что обычно увеличивает скорость поиска.

4. В чём идея поиска образца алгоритмом Рабина-Карпа?

Алгоритм Рабина-Карпа использует метод хеширования для поиска шаблона в строке. Основная идея заключается в следующем:

1. Вычисляем хеш-значение для шаблона.

2. Затем для каждой подстроки в строке длины шаблона также вычисляем хеш-значение.

3. Сравниваем хеш-значение подстроки с хеш-значением шаблона. Если они совпадают, дополнительно проводим проверку по символам (для предотвращения коллизий).

Асимптотическая сложность в среднем случае составляет O(n + m), но в худшем случае может достигнуть O(n  m) из-за возможных коллизий.

5. Назовите асимптотическую сложность алгоритма Бойера–Мура, Рабина-Карпа и Кнута-Мориса-Пратта поиска подстроки в строке по времени и памяти. Что лучше, в каком случае и почему?

Алгоритм Бойера–Мура:

- Временная сложность: O(n/m) в среднем случае, O(nm) в худшем.
- Память: O(m).

Алгоритм Рабина-Карпа:

- Временная сложность: O(n + m) в среднем, O(nm) в худшем.
- Память: O(1) или O(m) (если хеш-таблицы используются).

Алгоритм Кнута-Мориса-Пратта:

- Временная сложность: O(n + m) в среднем и худшем случае.
- Память: O(m).

На практике алгоритм Кнута-Мориса-Пратта и алгоритм Бойера-Мура чаще оказываются более эффективными для больших текстов и паттернов, особенно в случае больших объемов данных, поскольку их временные сложности не зависят от вероятных коллизий, как в случае Рабина-Карпа.

6. Объясните идею алгоритма Ахо–Корасика. Приведите его вычислительную и ёмкостную сложность.

Алгоритм Ахо–Корасика предназначен для поиска множества шаблонов в тексте одновременно. Основная идея алгоритма:

1. Построение префиксного дерева (Trie) для всех шаблонов.

2. Добавление функции перехода к каждому узлу префиксного дерева (prefix link), что облегчает переход к следующему узлу при несоответствии.

3. Поиск происходит с помощью прохода по тексту, где для каждого символа мы либо переходим по узлу, либо следуем по ссылке на префикс.

Вычислительная сложность:

- Построение префиксного дерева: O(m) для всех шаблонов, где m — общее количество символов всех шаблонов.

- Поиск в тексте: O(n), где n — длина текста.

- Таким образом, общая сложность: O(n + m).

**Емкостная сложность**:

- O(m * k), где k — количество шаблонов или символов в них (в зависимости от структуры префиксного дерева).

Алгоритм Ахо–Корасика обычно подходит для задач, где необходимо искать множество шаблонов одновременно, что делает его очень эффективным для таких приложений, как фильтрация текста или антивирусные программы.

## ЗАДАНИЕ 2

**Формулировка задачи:**

Разработайте приложения в соответствии с заданиями в индивидуальном варианте, указанными в таблице 1.

В отчёте в разделе «Математическая модель решения (описание алгоритма)» разобрать алгоритм поиска на примере. Подсчитать количество сравнений для успешного поиска первого вхождения образца в текст и безуспешного поиска.

Определить функцию (или несколько функций) для реализации алгоритма поиска. Определить делить предусловие и постусловие.

Сформировать таблицу тестов с указанием успешного и неуспешного поиска, используя большие и небольшие по объему текст и образец, провести на её основе этап тестирования.

Оценить практическую сложность алгоритма в зависимости от длины текста и длины образца и отобразить результаты в таблицу (для отчета).

Задание в соответствии с индивидуальным вариантом (1 вариант):

1. Линейный поиск первого вхождения подстроки в строку.

2. Используя алгоритм Бойера-Мура-Хорспула, найти последнее вхождение подстроки в строку.

**Математическая модель**

Линейный алгоритм поиска перебирает все возможные позиции в строке, чтобы найти первое вхождение подстроки (шаблона). Линейный поиск хорош для небольших строк, но его эффективность уменьшается с увеличением длины текста и шаблона.

Алгоритм:

1. Пройти по строке с индекса от 0 до (n - m), где n — длина строки, а m — длина подстроки.

2. На каждой позиции сравнить символы подстроки со строкой.

3. Если все символы совпали, вернуть текущий индекс.

4. Если пройдены все символы, вернуть -1.

Пример:

Строка: `abcabcabcd`

Подстрока: `abc`

- Сравниваем `abc` с `abc` на первой позиции (индекс 0):

- Сравнения: `a == a`, `b == b`, `c == c` – > Совпадает.

- Первое вхождение найдено на индексе 0.

Сравнения (успешный поиск): 3 (= длине подстроки)

Для безуспешного поиска можно использовать строку `abcd` и подстроку `xyz`.

- Сравниваем на первой позиции (индекс 0): `a != x`.

- Следующая проверка на индексе 1: `b != x`.

- Индекс 2: `c != x`.

- Индекс 3: `d != x`.

Сравнения (безуспешный поиск): 4 (= длине строки)

Алгоритм Бойера–Мура–Хорспула улучшает наивный поиск за счет использования двух правил сдвига: плохого символа и хорошего суффикса. Основное внимание уделяется сдвигу уже в процессе сравнения, что позволяет избежать лишних сравнений.

Шаги алгоритма:

1. Построение таблиц сдвигов (на основе шаблона).

- Таблица плохого символа: хранит сдвиги для символов, не найденных в шаблоне.

- Таблица хорошего суффикса: хранит информацию о подстроках, совпадающих с окончанием шаблона.

2. Начинаем проверку с последнего символа шаблона и движемся влево.

3. Если символы не совпадают, используем таблицы для определения сдвига шаблона.

Пример:

Строка: `abcabcabcd`

Подстрока: `abc`

1. Построим таблицу плохого символа:

- Для буквы `a`: сдвиг = 2 (т.к. последний символ `b`)

- Для буквы `b`: сдвиг = 1

- Для буквы `c`: сдвиг = 0

- Для всех остальных символов: сдвиг = m (3)

2. Начинаем сравнение:

- Начинаем с индекса 7: `d != b`. Сдвигаем на 3.

- Индекс 4: `b != b`. Совпадает. Сравниваем дальше.

- Индекс 0: `a == a`, `b == b`, `c == c` → Совпадение обнаружено.

Сравнения (успешный поиск): 5

Теперь для безуспешного поиска (например, строка `abcd` и подстрока `xyz`):

- Начальное сравнение: `d != x` → сдвиг на 3.

- Индекс 0: `abcd` по аналогии → все символы не совпадают.

Сравнения (безуспешный поиск): 4

**Код программы**

Для реализации линейного поиска была написана функция lin_search (рис. 1).

```
//линейный поиск первого вхождения подстроки в строку
int lin_search(string& str, string& substr) {
    int len = str.length();
    int s_len = substr.length();
    for (int i = 0; i <= len - s_len; i++) {
        if (str.substr(i, s_len) == substr) { //совпадает ли подстрока с i длиной s_len
            return i; //возвращаем индекс первого вхождения
        }
    }
    return -1; //если не найдено
}
```

Рисунок 1 – Линейный поиск

Для реализации алгоритма Бойера-Мура-Хорспула для поиска последнего вхождения подстроки были написаны две функции: preprocess и bhm_search. Функция preprocess – функция для предварительной обработки подстроки и заполнения таблицы, в которой будут последние индексы символов строки (рис. 2). Функция bhm_search – реализация самого поиска (рис. 3).

```
//алгоритм Бойера-Мура-Хорспула для поиска последнего вхождения подстроки
//last_ind - таблица с последними индексами символом строки
void preprocess(const string& substr, vector<int>& last_ind) { //ф. для обработки подстроки и заполнения таблицы
    int n = substr.length(); //длина подстроки
    for (int i = 0; i < 256; i++) last_ind[i] = -1; //инициализация
    for (int i = 0; i < n; i++) {
        last_ind[(int)substr[i]] = i; //заполнение таблицы
    }
}
```

Рисунок 2 – Алгоритма Бойера-Мура-Хорспула

```cpp
int bhm_search(const string& str, const string& substr) {
    int len = str.length();
    int s_len = substr.length();

    if (s_len > len) {
        cout << "Подстрока длиннее строки\n";
        return -1; //подстрока длиннее строки
    }

    vector<int> last_ind(256); //таблица, в которой хранятся последние индексы символов строки
    preprocess(substr, last_ind);
    int i = s_len - 1; //стартуем с конца подстроки
    int j = s_len - 1; //индекс символа в подстроке
    int index = -1; //для хранения последнего найденного индекса

    while (i < len) {
        if (str[i] == substr[j]) { //совпадают ли символы в текущих индексах строки и подстроки
            if (j == 0) { //значит нашли совпадение всей подстроки
                index = i; // обновляем последний найденный индекс
                i += s_len; // сдвигаем индекс для продолжения поиска
                j = s_len - 1; // обнуляем j, чтобы искать снова
            }
            else {  //если совпадений нет
                i--; //смещаемся влево
                j--;
            }
        }
        else { //если символы не совпадают, используем таблицу,
            i += s_len - min(j, 1 + last_ind[(int)str[i]]); //основываясь на последнем индексе текущего символа
            j = s_len - 1; //обнуляем j
        }
    }
    return index; // возвращаем последний найденный индекс
}
```

Рисунок 3 – Алгоритма Бойера-Мура-Хорспула

```cpp
int main() {
    setlocale(LC_ALL, "rus");
    string str;
    cout << "Введите строку: ";
    getline(cin, str);
    string substr;
    cout << "Введите подстроку, которую нужно найти: ";
    getline(cin, substr);

    int index1 = lin_search(str, substr);
    cout << "Первое вхождение подстроки начинается с индекса: " << index1 << endl;
    int index2 = bhm_search(str, substr);
    cout << "Последнее вхождение подстроки начинается с индекса: " << index2 << endl;
}
```

Рисунок 4 – Тестирование (main)

**Результаты тестирования**

Проведем тестирование на корректных вводных данных (рис. 5).

```
Введите строку: asdfghjkl asdfghsad asdjkl fgh fgh asdfgh a
Введите подстроку, которую нужно найти: fgh
Первое вхождение подстроки начинается с индекса: 3
Последнее вхождение подстроки начинается с индекса: 38
```

Рисунок 5 - Тестирование

Ситуация, когда длина подстроки больше длины самой строки (рис. 6).

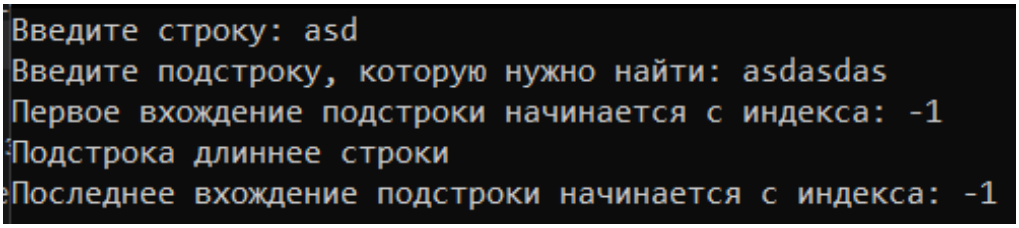


Рисунок 6 – Тестирование

Ситуация, когда поиск неуспешный на примере небольшого текста.

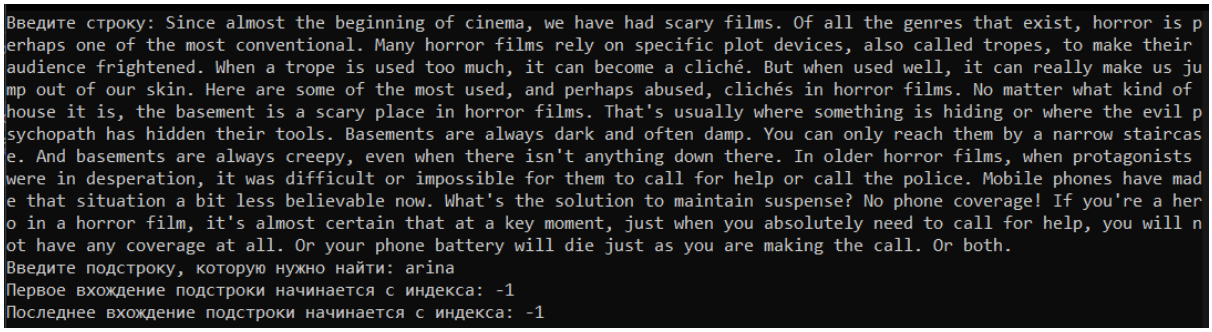


Рисунок 7 – Тестирование

Успешный поиск на примере небольшого текста (рис. 8).

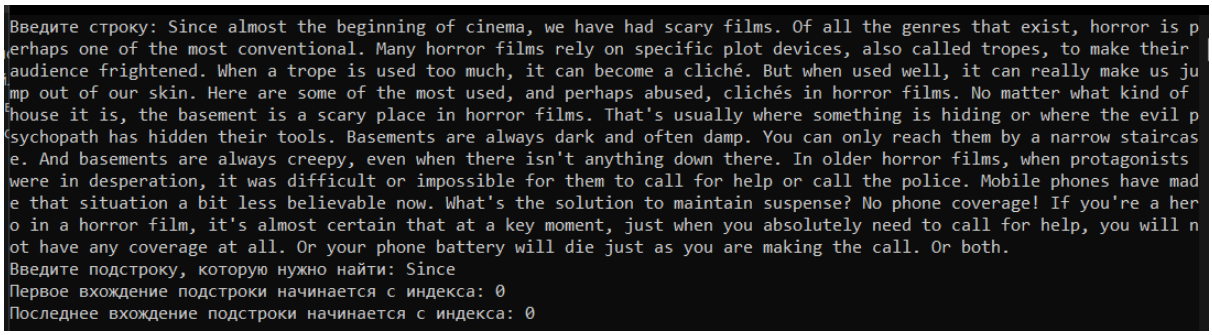


Рисунок 8 – Тестирование

Для проведения тестирований было добавлено отслеживание времени. Например, на рисунке 9 за 2мс было найдено самое последнее слово в тексте линейным алгоритмом и за приближенное к 0мс было найдено то же слово алгоритмом Бойера-Мура- Хорспула.

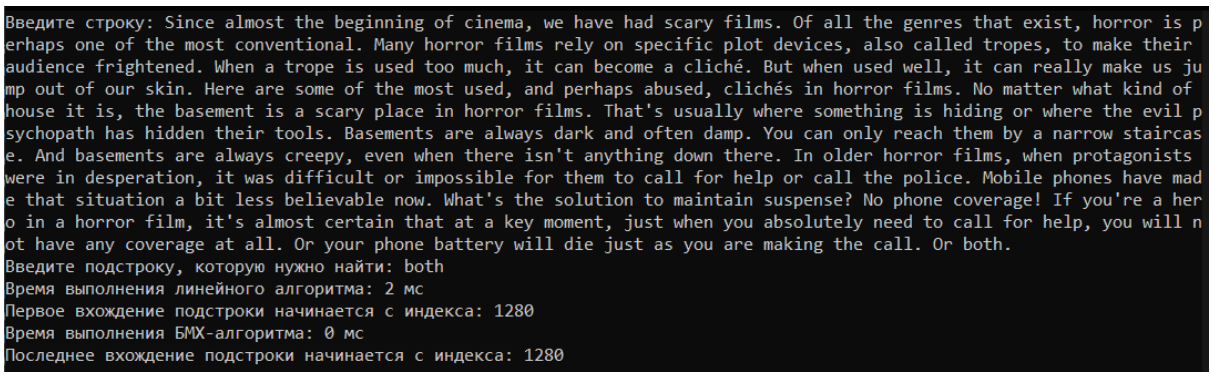


Рисунок 9 – Тестирование

Тестирование показало, что программа работает корректно.

# ВЫВОД

В ходе выполнения практической работы была успешно освоены приемы реализации алгоритмов поиска образца в тексте: линейный поиск и алгоритм Бойера-Мура-Хорспула.

Линейный поиск подходит для небольших строк, но его эффективность уменьшается с увеличением длины текста и шаблона. Алгоритм Бойера-Мура-Хорспула демонстрирует значительно лучшую производительность за счет предобработки, и его сложность в среднем случае ниже, что делает его предпочтительным выбором при работе с длинными строками и многообразными шаблонами.

Цель работы была достигнута.

## СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих, 2017. – С. 100-126.

2. Кормен Т.Х. и др. Алгоритмы. Построение и анализ, 2013. – С. 285-318.

3. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

4. Документация по языку C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 21.10.2024).

5. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 20.10.2024).