Санкт-Петербургский государственный университет

Группа 23.Б12-мм

Алгоритмы Ахо-Корасик и Кнута-Морриса-Пратта

Брайченко Евдокия Тарасовна

Студенческий проект

Преподаватель:
ассистент кафедры теоретической и прикладной механики, Г.А.Нестерчук

Санкт-Петербург
2024

Оглавление

Введение

В данном проекте мы углубимся в реализацию алгоритмов Ахо-Корасика и Кнутта-Морриса-Пратта для поиска подстроки в заданном тексте на языке C++. Алгоритм Ахо-Корасика, разработанный Альфредом Ахо и Маргарет Корасик в 1975 году, основан на построении конечного автомата и отличается высокой производительностью, особенно при поиске нескольких паттернов одновременно. Алгоритм Кнутта-Морриса-Пратта, созданный Дональдом Кнуттом, Джеймсом Моррисом и Василием Праттом в 1977 году, использует префикс-функцию для быстрого сопоставления паттернов и текста, что делает его подходящим для поиска одиночных паттернов.

Реализация этих алгоритмов на C++ позволит нам изучить их преимущества и недостатки в различных сценариях поиска подстрок. Во-первых, мы создадим конечный автомат для алгоритма Ахо-Корасика, используя структуры данных и указатели. Во-вторых, мы реализуем префикс-функцию для алгоритма КМП.

1. Описание алгоритма Кнутта-Морисса - Пратта

Алгоритм Кнута-Морриса-Пратта предназначен для поиска вхождения шаблона в заданной строке. Он эффективен в задачах, где требуется найти все вхождения шаблона в строке. Прежде чем перейти к описанию алгоритма, необходимо рассмотреть понятие префикс-функции.

Префикс-функция строки π(S,i) – это длина наибольшего префикса строки S[1..i], который не совпадает с этой строкой и одновременно является ее суффиксом. Проще говоря, это длина наиболее длинного начала строки, являющегося также и ее концом. Для строки S удобно представлять префикс функцию в виде вектора длиной |S|-1. Можно рассматривать префикс-функцию длины |S|, положив π(S,1)=0.

Предположим, что π(S,i)=k. Отметим следующие свойства префикс-функции:

1. Если S[i+1]=S[k+1], то π(S,i+1)=k+1.
2. S[1..π(S,k)] является суффиксом строки S[1..i]. Действительно, если строка S[1..i] оканчивается строкой S[1... π(S,i)]=S[1..k], а строка S[1..k] оканчивается строкой S[1..π(S,k)], то и строка S[1..i] оканчивается строкой S[1..π(S,k)].
3. $\forall j \in (k, i)$, S[1..j] не является суффиксом строки S[1..i].

В противном случае было бы неверным предположение π(S,i)=k, так как j>k

Рассмотренные свойства позволяют получить алгоритм вычисления префикс-функции. Пусть π(S,i)=k. Необходимо вычислить π(S,i+1).

1. Если S[i+1]=S[k+1], то π(S,i+1)=k+1.
2. Иначе, если k=0, то π(S,i+1)=0.
3. Иначе положить k:=π(S,k) и перейти к шагу 1.

Ключевым моментом для понимания сути алгоритма является тот факт, что если найденный на предыдущем шаге суффикс не может быть расширен на следующую позицию, то мы пытаемся рассматривать меньшие суффиксы до тех пор, пока это возможно. Рассмотрим алгоритм Кнута-Морриса-Пратта, основанный на использовании префикс-функции. Как и в примитивном алгоритме поиска подстроки, образец «перемещается» по строке слева направо с целью обнаружения совпадения. Однако ключевым отличием является то, что при помощи префикс-функции мы можем избежать заведомо бесполезных сдвигов.

Пусть S[0..m–1] – образец, T[0..n–1] – строка, в которой ведется поиск. Рассмотрим сравнение строк на позиции i, то есть образец S[0..m–1] сопоставляется с частью строки T[i..i+m–1]. Предположим, первое несовпадение произошло между символами S[j] и T[i+j], где $i < j < m$. Обозначим P = S[0..j–1] = T[i..i+j–1]. При сдвиге можно ожидать, что префикс S сойдется с каким-либо суффиксом строки P. Поскольку длина наиболее длинного префикса, являющегося одновременно суффиксом, есть префикс-функция от строки S для индекса j, приходим к следующему алгоритму:

1. Построить префикс-функцию образца S, обозначим ее F.
2. Положить $k = 0$, $i = 0$.
3. Сравнить символы S[k] и T[i]. Если символы равны, увеличить k на 1. Если при этом k стало равно длине образца, то вхождение образца S в строку T найдено, индекс вхождения равен $i - |S| + 1$. Алгоритм завершается. Если символы не равны, используем префикс-функцию для оптимизации сдвигов. Пока $k > 0$, присвоим k = F[k–1] и перейдем в начало шага 3.
4. Пока $i < |T|$, увеличиваем i на 1 и переходим в шаг 3.

2. Описание алгоритма Ахо-Корасик

Суть алгоритма заключена в использование структуры данных — бора и построения по нему конечного детерминированного автомата. Важно помнить, что задача поиска подстроки в строки тривиально реализуется за квадратичное время, поэтому для эффективной работы важно, чтоб все части Ахо-Корасика ассимптотически не превосходили линию относительно длинны строк. Мы вернемся к оценке сложности в конце, а пока поближе посмотрим на составляющие алгоритма.

Структура бора

Что же такое бор? Строго говоря, бор — это дерево, в котором каждая вершина обозначает какую-то строку (корень обозначает нулевую строку — ε). На ребрах между вершинами написана 1 буква, таким образом, добираясь по ребрам из корня в какую-нибудь вершину и контангенируя буквы из ребер в порядке обхода, мы получим строку, соответствующую этой вершине. Из определения бора как дерева вытекает также единственность пути между корнем и любой вершиной, следовательно — каждой вершине соответствует ровно одна строка (в дальнейшем будем отождествлять вершину и строку, которую она обозначает).

Строить бор будем последовательным добавление исходных строк. Изначально у нас есть 1 вершина, корень (root) — пустая строка. Добавление строки происходит так: начиная в корне, двигаемся по нашему дереве, выбирая каждый раз ребро, соответствующее очередной букве строки. Если такого ребра нет, то мы создаем его вместе с вершиной.

Построение автомата по бору

Наша задача — построить конечный детерминированный автомат. Конечный детерминированный автомат (КДА) - это математическая модель, которая представляет процесс принятия решения. В нашем случае КДА используется для поиска подстрок в тексте.

Вкратце, состояние автомата — это какая-то вершина бора. Переход из состояний осуществляется по 2 параметрам — текущей вершине v и символу ch. по котороромy нам надо сдвинуться из этой вершины. Поконкретнее, необходимо найти вершину u, которая обозначает

наидлиннейшую строку, состоящую из суффикса строки v (возможно нулевого) + символа ch. Если такого в боре нет, то идем в корень.

Зачем это нам надо? Предположим, что мы можем вычислить такую вершину быстро, за константное время. Пусть, мы стоим в некой вершине бора, соответствующей подстроке [i..j] строки s, вхождения в которую мы ищем. Теперь найдем все строки бора, суффиксы s[i..j]. Утверждается, что их можно искать быстро . После этого, просто перейдем из состояния автомата v в состояние u по символу s[j+1] и продолжим поиск.Для реализации автомата нам понадобится понятие суффиксной ссылки из вершины.

Суффиксные ссылки

Назовем суффиксной ссылкой вершины v указатель на вершину u, такую что строка u — наибольший собственный суффикс строки v, или, если такой вершины нет в боре, то указатель на корень. В частности, ссылка из корня ведет в него же.

Реализация автомата

Введем вычисляемую функцию для перехода (v,ch). Идея тут вот в чем: если из текущей вершины есть ребро с символом ch, то пройдем по нему, в обратном случаем пройдем по суффиксной ссылке и запустимся рекурсивно от новой вершины. Почему это работает, догадаться не трудно. Вопрос лишь в корректном получение суф. ссылки от вершины. В этой задаче тоже можно использовать ленивую динамику. Эвристика заключена в следующем: для получения суф. ссылки вершины v (строки s[i..j]) спустимся до ее предка par, пройдем по суф. ссылке par и запустим переход от текущей вершины t по символу symb, который написан на ребре от par до v. Очевидно, что сначала мы попадем в наибольший суфикс s[i..j-1] такой что, он имеет ребро с символом symb, потом пройдем по этому ребру. По определению, получившаяся вершина и есть суффикная ссылка из вершин v. Итак, видно, что функции получения суффиксной ссылки и перехода из состояния автомата взаимосвязаны. Их удобная реализация представляет 2 функции, каждая из которых рекурсивно вызывает другую. База обоих рекурсий — суф. ссылка из корня или из сына корня ведет в корень.

3. Работа кода

3.1. Класс StringSearch

```
class StringSearch {
    private:
    BohrVertex *bohr;
    std::string *patterns;
    int bohrSize;
    int patternsSize;
    int bohrCapacity;
    int patternsCapacity;

    void initializeBohr();
    void ensureBohrCapacity();
    void ensurePatternsCapacity();
    void addStringToBohr(const std::string &s);
    int getAutoMove(int vertex, char ch);
    int getSuffixLink(int vertex);
    int getSuffixFlink(int vertex);
    void check(int vertex, int i, std::ostream &outStream);
    void findAllPositions(const std::string &text, std::ostream &outStream);
    int *computePrefixFunction(const std::string &pattern);
    void findKMP(const std::string &text, const std::string &pattern, std::ostrean

    public:
    StringSearch();
    //Конструктор по умолчанию
    ~StringSearch();
    //Деструктор
    bool getInputFromFile(const std::string &filename, std::string &text); // Ten
    void findPatterns(const std::string &text,
                      const std::string &pattern,
                      int algorithmChoice,
                      std::ostream &outStream);
};
```

3.2. Структура BohrVertex

```
#ifndef BOHRVERTEX_H
#define BOHRVERTEX_H

#include <unordered_map>

struct BohrVertex
{
    std::unordered_map<char, int> nextVertices, autoMoves; // Переходы и автомати
    int patternNum;                                 // Номер шаблона
    int suffixLink;                                 // Суффиксная ссылка
    int parent;                                     // Родительская вершина
    int suffixFlink;                                // Сжатая суффиксная ссылка
    bool isPatternEnd;                              // Является ли конец шаблона
    char symbol;                                    // Символ

    BohrVertex(int p = -1, char c = '$');     //Конструктор по умолчанию
};

#endif                                          // BOHRVERTEX_H
```

3.3. Инициализация структуры данных Автомат Бора для поиска подстроки

```
void StringSearch::initializeBohr()
{
    // Удаляем ранее созданные массивы
    delete[] bohr;
    delete[] patterns;
    // Устанавливаем начальную вместимость массивов
    bohrCapacity = 1000;          // Начальная вместимость массива вершин
    patternsCapacity = 100;       // Начальная вместимость массива
    ↪ шаблонов
    // Выделяем память для массивов
    bohr = new BohrVertex[bohrCapacity];
    patterns = new std::string[patternsCapacity];
    // Устанавливаем размер автомата Бора и создаем корневую вершину
    bohrSize = 1;
```

```
14      bohr[0] = BohrVertex(0, '$'); // Добавляем корневую вершину
15      patternsSize = 0;
16  }
```

3.4. Функция гарантирует, что буфер всегда имеет достаточный размер для размещения всех вершин, необходимых для выполнения поиска.

```
1   void StringSearch::ensureBohrCapacity()
2   {
3       // Проверяем, превышен ли текущий размер буфера Бора по отношению к
        ↪  его емкости
4       if (bohrSize >= bohrCapacity)
5       {
6           // Удваиваем емкость буфера
7           bohrCapacity *= 2;
8
9           // Создаем новый массив вершин Бора с увеличенной емкостью
10          BohrVertex *newBohr = new BohrVertex[bohrCapacity];
11
12          // Копируем существующие вершины Бора из старого массива в новый
            ↪  массив
13          for (int i = 0; i < bohrSize; ++i)
14          {
15              newBohr[i] = bohr[i];
16          }
17
18          // Удаляем старый массив вершин Бора
19          delete[] bohr;
20
21          // Устанавливаем новый массив вершин Бора в качестве текущего
            ↪  буфера
22          bohr = newBohr;
23      }
24  }
```

3.5. Функция для обеспечения достаточной емкости массива образцов

```
void StringSearch::ensurePatternsCapacity()
{
    // Проверяем, достаточно ли текущей емкости для размещения всех
    ↪ образцов
    if (patternsSize >= patternsCapacity)
    {
        // Удваиваем емкость массива образцов
        patternsCapacity *= 2;

        // Выделяем новый массив с увеличенной емкостью
        std::string *newPatterns = new std::string[patternsCapacity];

        // Копируем существующие образцы в новый массив
        for (int i = 0; i < patternsSize; ++i)
        {
            newPatterns[i] = patterns[i];
        }

        // Освобождаем старый массив образцов
        delete[] patterns;

        // Назначаем новый массив образцов
        patterns = newPatterns;
    }
}
```

3.6. Функция добавляет заданную строку s в структуру данных префиксного дерева Бора

```
void StringSearch::addStringToBohr(const std::string &s) {
    // Начинаем с корневой вершины
    int num = 0;

    // Перебираем символы в строке s
    for (char ch : s) {
        // Проверяем, содержит ли текущая вершина переход по символу ch
```

```
        if (bohr[num].nextVertices.find(ch) ==
        ↪  bohr[num].nextVertices.end()) {
            // Если перехода нет, добавляем новую вершину
            ensureBohrCapacity();
            bohr[bohrSize] = BohrVertex(num, ch);  // Добавляем новую
            ↪  вершину
            bohr[num].nextVertices[ch] = bohrSize; // Обновляем переход
            ++bohrSize;
        }
        num = bohr[num].nextVertices[ch];          // Переходим к
        ↪  следующей вершине
    }

    bohr[num].isPatternEnd = true;                 // Отмечаем конец
    ↪  шаблона

    ensurePatternsCapacity();
    patterns[patternsSize] = s;                    // Добавляем шаблон в
    ↪  массив шаблонов

    bohr[num].patternNum = patternsSize;           // Обновляем номер
    ↪  шаблона
    ++patternsSize;
}
```

3.7. Функция в дереве Бора вычисляет автоматический переход для данной вершины и символа

```
int StringSearch::getAutoMove(int vertex, char ch)
{
    //Есть ли автоматический переход для данного символа ch из данной
    ↪  вершины vertex в словаре автоматических переходов
    if (bohr[vertex].autoMoves.find(ch) == bohr[vertex].autoMoves.end())
    {
        if (bohr[vertex].nextVertices.find(ch) !=
        ↪  bohr[vertex].nextVertices.end())
        // Существует ребро для ch из вершины vertex
        {
```

```
9               bohr[vertex].autoMoves[ch] = bohr[vertex].nextVertices[ch]; //
                ↪ Автоматический переход
10          }
11          else
12          {
13              bohr[vertex].autoMoves[ch] =
14                  (vertex == 0 ? 0 : getAutoMove(getSuffixLink(vertex),
                    ↪ ch)); // Суффиксная ссылка
15          }
16      }
17      return bohr[vertex].autoMoves[ch];
18  }
```

3.8. Функция возвращает суффиксную ссылку для данной вершины vertex в боре

```
1   int StringSearch::getSuffixLink(int vertex) {
2       // Если суффиксная ссылка для данной вершины еще не была вычислена.
3       if (bohr[vertex].suffixLink == -1) {
4           // Для корня и его детей суффиксная ссылка всегда равна 0.
5           if (vertex == 0 || bohr[vertex].parent == 0) {
6               bohr[vertex].suffixLink = 0;
7           } else {
8               // Для других вершин суффиксная ссылка вычисляется путем
                ↪ перехода из родительской вершины по символу данной
                ↪ вершины.
9               bohr[vertex].suffixLink =
                ↪ getAutoMove(getSuffixLink(bohr[vertex].parent),
                ↪ bohr[vertex].symbol);
10          }
11      }
12      // Возвращаем суффиксную ссылку для данной вершины.
13      return bohr[vertex].suffixLink;
14  }
```

3.9. Функция вычисляет сжатую суффиксную ссылку для данной вершины в боре

```
int StringSearch::getSuffixFlink(int vertex)
{
    if (bohr[vertex].suffixFlink == -1)
    {
        int suffLink = getSuffixLink(vertex);
        if (suffLink == 0)
        {
            bohr[vertex].suffixFlink = 0; // Сжатая суффиксная ссылка для
            ↪ корня
        }
        else
        {
            bohr[vertex].suffixFlink = bohr[suffLink].isPatternEnd ?
            ↪ suffLink : getSuffixFlink(suffLink);
        }
    }
    return bohr[vertex].suffixFlink;
}
```

3.10. Функция check ищет совпадения паттернов в подстроке, начиная с позиции i, и выводит найденные совпадения в поток outStream

```
void StringSearch::check(int vertex, int i, std::ostream &outStream)
{
    for (int v = vertex; v != 0; v = getSuffixFlink(v))
    {
        if (bohr[v].isPatternEnd)
        {
            outStream << "Позиция: " << i -
            ↪ patterns[bohr[v].patternNum].length() + 1
                    << " Шаблон: " << patterns[bohr[v].patternNum] <<
                    ↪ std::endl;
        }
    }
```

```
}
```

3.11. Эта функция используется для поиска всех позиций, где заданный текст соответствует заданному шаблону

```
void StringSearch::findAllPositions(const std::string &text, std::ostream
    &outStream)
{
    int vertex = 0;
    for (int i = 0; i < text.length(); i++)
    {
        vertex = getAutoMove(vertex, text[i]);
        outStream << ''Промежуточный результат: '' << i << '' символ: \''' <<
            text[i] << ''\''' << std::endl;
        check(vertex, i + 1, outStream);
    }
}
```

3.12. Функция вычисляет префикс-функцию заданного шаблона

```
int *StringSearch::computePrefixFunction(const std::string &pattern)
{
    int m = pattern.length();
    int *pi = new int[m]();
    int k = 0;
    for (int i = 1; i < m; i++)
    {
        // Пока k > 0 и символы не совпадают, сдвигаем k назад
        while (k > 0 && pattern[k] != pattern[i])
        {
            k = pi[k - 1];
        }
        // Если символы совпали, увеличиваем k
        if (pattern[k] == pattern[i])
        {
            k++;
```

```
17          }
18          pi[i] = k;
19      }
20      return pi;
21  }
```

3.13. Данная функция реализует алгоритм Кнута-Морриса-Пратта для поиска подстроки в заданной строке

```
1  void StringSearch::findKMP(const std::string &text, const std::string
   ↪  &pattern, std::ostream &outStream)
2  {
3      int n = text.length(); // Длина исходной строки
4      int m = pattern.length(); // Длина паттерна
5      int *pi = computePrefixFunction(pattern); // Вычисляем префикс-функцию
       ↪  для паттерна
6
7      int q = 0; // Текущая позиция в паттерне
8      for (int i = 0; i < n; i++) // Итерируемся по исходной строке
9      {
10         while (q > 0 && pattern[q] != text[i]) // Если символ не
           ↪  совпадает, откатываемся к предыдущему префиксу
11         {
12             outStream << "Промежуточный результат: позиция: " << i << "
               ↪  символ: "" << text[i]
13                       << "" не совпадает, откат к: " << q - 1 <<
                         ↪  std::endl;
14             q = pi[q - 1];
15         }
16         if (pattern[q] == text[i]) // Если символ совпадает, продвигаемся
           ↪  дальше в паттерне
17         {
18             q++;
19             outStream << "Промежуточный результат: позиция: " << i << "
               ↪  символ: "" << text[i]
20                       << "" совпадает, длина текущего совпадения: " << q
                         ↪  << std::endl;
```

```
        }
        if (q == m) // Если длина текущего совпадения равна длине
        ↪  паттерна, паттерн найден
        {
            outStream << "Шаблон найден на позиции " << i - m + 1 <<
            ↪  std::endl;
            q = pi[q - 1]; // Откатываемся к предыдущему префиксу
        }
    }
    delete[] pi; // Освобождаем память, занятую префикс-функцией
}
```

3.14. Функция позволяет считывать текстовый файл и сохранять его содержимое в строке

```
bool StringSearch::getInputFromFile(const std::string &filename,
↪  std::string &text)
{
    // Пытается открыть файл
    std::ifstream inFile(filename);
    if (!inFile.is_open())
    {
        std::cerr << "Не удалось открыть файл: " << filename << std::endl;
        return false;
    }

    // Создает буфер потока для чтения файла
    std::stringstream buffer;

    // Считывает весь файл в буфер
    buffer << inFile.rdbuf();

    // Извлекает содержимое из буфера и сохраняет в text
    text = buffer.str();

    // Закрывает файл
    inFile.close();

    // Возвращает true, если файл был успешно считан
```

```
24      return true;
25  }
```

3.15. Функция поиска шаблонов в тексте

```
1   void StringSearch::findPatterns(const std::string &text,
2                                   const std::string &pattern,
3                                   int algorithmChoice,
4                                   std::ostream &outStream)
5   {
6       if (algorithmChoice == 1)
7       {
8           findKMP(text, pattern, outStream);
9       }
10      else if (algorithmChoice == 2)
11      {
12          initializeBohr();
13          addStringToBohr(pattern);
14          findAllPositions(text, outStream);
15      }
16  }
```

3.16. Класс ConsoleHandler

```
1   class ConsoleHandler {
2       public:
3       void run();
4
5       private:
6       void correct(int &checkNumber);
7   };
```

3.17. Функция считывает верное ли число ввел пользователь

```
1
2   void ConsoleHandler::correct(int &checkNumber)
3   {
4       // Попытка считать число
```

```
    while (!(std::cin >> checkNumber))
    {
        // Если ввод некорректен, выводим сообщение
        std::cout << ”Введите другое число: ”;
        // Сбрасываем ошибочное состояние потока
        std::cin.clear();
        // Пропускаем максимальное количество символов до следующего
        ↪ перевода строки
        std::cin.ignore(10000, '\n'); // Здесь 10000 как пример большого
        ↪ числа
    }
}
```

3.18. Функция предоставляет интерактивный пользовательский интерфейс для поиска строк в тексте

```
void ConsoleHandler::run()
{
    StringSearch ss;
    int choice;
    while (true)
    {
        std::cout << ”Выберите опцию:\n1 - Ввод с консоли\n2 - Ввод из
        ↪ файла\n3 - Выход\n”;
        correct(choice);

        if (choice == 3)
        {
            break;
        }

        std::string text, pattern;
        if (choice == 1)
        {
            std::cin.ignore(); // Чтобы пропустить оставшийся символ новой
            ↪ строки

```

```
20              std::cout << ''Введите текст: '';
21              std::getline(std::cin, text);
22
23              std::cout << ''Введите шаблон: '';
24              std::getline(std::cin, pattern);
25          }
26          else if (choice == 2)
27          {
28              std::cin.ignore(); // Чтобы пропустить оставшийся символ новой
                ↪  строки
29
30              std::string filename;
31              std::cout << ''Введите имя файла: '';
32              std::getline(std::cin, filename);
33
34              if (!ss.getInputFromFile(filename, text))
35              {
36                  continue;
37              }
38
39              std::cout << ''Введите строку для поиска в файле: '';
40              std::getline(std::cin, pattern);
41          }
42          else
43          {
44              std::cout << ''Недопустимый выбор опции.\n'';
45              continue;
46          }
47
48          std::cout << ''Выберите алгоритм:\n1 - Кнута-Морриса-Пратта\n2 -
            ↪  Ахо-Корасик\n'';
49          correct(choice);
50          std::cin.ignore(); // Чтобы пропустить оставшийся символ новой
            ↪  строки
51
52          std::cout << ''Выберите опцию вывода:\n1 - Вывод в консоль\n2 -
            ↪  Вывод в файл\n'';
53          correct(choice);
54          std::cin.ignore(); // Чтобы пропустить оставшийся символ новой
            ↪  строки
```

```
55
56          std::ofstream outFile;
57          std::ostream *outStream = &std::cout;
58
59          if (choice == 2)
60          {
61              std::string outFilename;
62
63              std::cout << ''Введите имя файла для вывода: '';
64              std::getline(std::cin, outFilename);
65              outFile.open(outFilename);
66              if (!outFile.is_open())
67              {
68                  std::cerr << ''Не удалось открыть файл для вывода: '' <<
                    ↪  outFilename << std::endl;
69                  continue;
70              }
71              outStream = &outFile;
72          }
73
74          auto start = std::chrono::high_resolution_clock::now();
75          ss.findPatterns(text, pattern, choice, *outStream);
76          auto end = std::chrono::high_resolution_clock::now();
77          std::chrono::duration<double> duration = end - start;
78          *outStream << ''Время выполнения: '' << duration.count() << ''
            ↪  секунд.'' << std::endl;
79
80          if (outFile.is_open())
81          {
82              outFile.close();
83          }
84      }
85  }
```

Заключение

Реализация алгоритмов Кнута-Морриса-Пратта и Ахо-Корасика обеспечивает эффективный поиск подстрок в тексте. КМП использует таблицу префиксов и суффиксов, а Ахо-Корасик строит дерево конечных автоматов. Оба алгоритма обладают уникальными преимуществами, делающими их пригодными для различных сценариев поиска. Независимо от выбора алгоритма, разработчики могут воспользоваться преимуществами быстрой и надежной производительности, что делает КМП и Ахо-Корасика незаменимыми инструментами для решения сложных задач поиска подстрок в различных областях, включая обработку текста, поиск в базах данных и биоинформатику.