

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**"МИРЭА - Российский технологический университет"**

РТУ МИРЭА

Институт информационных технологий (ИИТ)

Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

**ОТЧЕТ**  
**ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 6.2**

**«Поиск образца в тексте»**

**по дисциплине**

**«**Структуры и алгоритмы обработки данных**»**

|  |  |  |
| --- | --- | --- |
| Выполнил студент группы ИКБО-11-24 |  | Авраменко А.А. |
|  |  |  |
| Принял доцент каф. МОСИТ |  | Сартаков М.В. |

Практическую работу выполнил «\_\_»\_\_\_\_\_\_\_2025 г.

«Зачтено» «\_\_»\_\_\_\_\_\_\_2025 г.

Москва 2025

**Цель работы:** освоить приёмы реализации алгоритмов поиска образца в тексте.

**Практическое задание:**

Разработайте приложения в соответствии с заданиями в индивидуальном варианте (п.2).

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 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).

**Решение:**

1. Математическая модель решения (описание алгоритма)

Дан текст T длины n и образец P длины m (m ≤ n). Требуется найти индекс первого вхождения P в T или определить, что образец в тексте отсутствует.

Формально: необходимо найти такое i из диапазона [0, n – m], что T[i … i+m–1] = P[0 … m–1].

* 1. Линейный поиск

Алгоритм перебирает все возможные позиции сдвига i от 0 до n–m. Для каждой позиции последовательно сравниваются символы текста и образца.

Листинг 1.1 – псевдокод алгоритма линейного поиска

для i от 0 до n–m:  
 j = 0  
 пока j < m и T[i+j] = P[j]:  
 j++  
 если j = m → найдено совпадение, вернуть i

Сложность:

Худший случай — O(n \* m), когда почти весь образец совпадает, но отличается последний символ.

Средний случай (при случайных данных и большом алфавите) — близок к O(n + m).

* 1. Алгоритм Бойера–Мура–Хорспула (BMH).

Этот алгоритм использует идею сравнения символов образца и текста с конца. Если найдено несовпадение, окно сдвигается так, чтобы несовпавший символ текста совпал с его последним появлением в образце. Если символа нет в образце, сдвиг равен m.

Правило сдвига:

Ecли символ встречается в образце (кроме последнего), то shift(c) = m – 1 – позиция последнего вхождения c в образце.

Eсли символа нет в образце, shift(c) = m.

Листинг 1.2 – псевдокод алгоритма

i = 0  
пока i <= n – m:  
 j = m – 1  
 пока j >= 0 и T[i+j] = P[j]:  
 j--  
 если j < 0 → найдено совпадение, вернуть i  
 i = i + shift(T[i+m–1])

Сложность:

Худший случай — O(n \* m) (например, если текст и образец состоят из одинаковых символов).

Средний случай — O(n / m), так как обычно делаются большие сдвиги и количество сравнений уменьшается.

Сравнение алгоритмов:

Таблица 1.1 – сравнение алгоритмов поиска подстроки

|  |  |  |
| --- | --- | --- |
| Алгоритм | Худший случай | Средний случай |
| Линейный | O(n \* m) | O(n + m) |
| BMH | O(n\*m) | O(n / m) |

Из сравнения видно, что оба алгоритма имеют одинаковую теоретическую сложность в худшем случае — O(n \* m). Однако на практике алгоритм Бойера–Мура–Хорспула значительно быстрее линейного за счёт больших сдвигов окна поиска. Это особенно заметно при увеличении длины текста и при использовании случайных данных на большом алфавите. Линейный поиск проще и предсказуем, но BMH показывает лучшую среднюю производительность.

1. Реализация:

Листинг 1.3 – файл main.cpp

#include <string>  
#include <iostream>  
#include <unordered\_map>  
#include <fstream>  
using namespace std;  
  
std::ofstream out("/Users/artem/Desktop/Универ/2 курс/3 сем/СИАОД/siaod-rtu-mirea/practice6/6.2/results.csv");  
  
int substrLinearSearch(string &s, const string &pattern, long long &comparisons) {  
 int n = s.size();  
 int m = pattern.size();  
 if (m > n) return -1;  
  
 comparisons = 0;  
 for (int i = 0; i <= n - m; i++) {  
 int j = 0;  
 while (j < m) {  
 comparisons++;  
 if (s[i + j] != pattern[j]) break;  
 j++;  
 }  
 if (j == m) {  
 return i;  
 }  
 }  
 return -1;  
}  
  
int boyerMooreHorspool(const std::string& text, const std::string& pattern, int start, long long &comparisons, long long &shifts) {  
 std::unordered\_map<char, int> table;  
 int n = text.size();  
 int m = pattern.size();  
  
 if (m > n) return -1;  
  
 shifts = 0;  
  
 for (int i = 0; i < m - 1; i++) {  
 table[pattern[i]] = m - i - 1;  
 }  
  
 int i = start;  
 while (i <= n - m) {  
 int j = m - 1;  
  
 while (j >= 0) {  
 comparisons++;  
 if ( pattern[j] != text[i + j]) {  
 break;  
 }  
 j--;  
 }

Продолжение листинга 1.3

if (j < 0) {  
 return i;  
 }  
  
 char c = text[i + m - 1];  
 int shift = table.count(c) ? table[c] : m;  
 i += shift;  
 shifts++;  
 }  
  
 return -1;  
}  
  
int findLast(string &text, string &pattern) {  
 int lastIndex = -1;  
 long long cmp = 0, sh = 0;  
 int pos = boyerMooreHorspool(text, pattern, 0, cmp, sh);  
 while (pos != -1) {  
 lastIndex = pos;  
 pos = boyerMooreHorspool(text, pattern, pos + 1, cmp, sh);  
 }  
 return lastIndex;  
}  
  
#include <chrono>  
#include <random>  
  
string makeString(size\_t n, const string &alphabet = "abcdefghijklmnopqrstuvwxyz") {  
 std::mt19937\_64 rng(1234567);  
 std::uniform\_int\_distribution<int> dist(0, (int)alphabet.size() - 1);  
 string s; s.reserve(n);  
 for (size\_t i = 0; i < n; ++i) s.push\_back(alphabet[dist(rng)]);  
 return s;  
}  
  
void printHeader() {  
 out << "algorithm,n\_text,m\_pattern,case,found\_index,success,comparisons,shifts,duration\_us" << '\n';  
}  
  
void runLinear(const string &text, const string &pattern, const string &caseName) {  
 long long cmp = 0;  
 string t = text; *// API expects non-const reference* auto t0 = std::chrono::high\_resolution\_clock::*now*();  
 int pos = substrLinearSearch(t, pattern, cmp);  
 auto t1 = std::chrono::high\_resolution\_clock::*now*();  
 auto us = std::chrono::duration\_cast<std::chrono::microseconds>(t1 - t0).count();  
 out << "linear" << ',' << text.size() << ',' << pattern.size() << ',' << caseName << ','  
 << pos << ',' << (pos != -1) << ',' << cmp << ',' << 0 << ',' << us << '\n';  
}

Продолжение листинга 1.3

void runBMH(const string &text, const string &pattern, const string &caseName) {  
 long long cmp = 0, sh = 0;  
 auto t0 = std::chrono::high\_resolution\_clock::now();  
 int pos = boyerMooreHorspool(text, pattern, 0, cmp, sh);  
 auto t1 = std::chrono::high\_resolution\_clock::now();  
 auto us = std::chrono::duration\_cast<std::chrono::microseconds>(t1 - t0).count();  
 out << "bmh" << ',' << text.size() << ',' << pattern.size() << ',' << caseName << ','  
 << pos << ',' << (pos != -1) << ',' << cmp << ',' << sh << ',' << us << '\n';  
}  
  
int main() {  
 ios::sync\_with\_stdio(false);  
 cin.tie(nullptr);  
  
 printHeader();  
  
 // Small cases  
 string small = "sdifjhsdf";  
 runLinear(small, "fj", "small\_mid\_match");  
 runBMH(small, "fj", "small\_mid\_match");  
 runLinear(small, "sd", "small\_start\_match");  
 runBMH(small, "sd", "small\_start\_match");  
 runLinear(small, "df", "small\_end\_match");  
 runBMH(small, "df", "small\_end\_match");  
 runLinear(small, "zz", "small\_no\_match");  
 runBMH(small, "zz", "small\_no\_match");  
  
 // Repeated-character edge cases  
 string repText = string(1000, 'a');  
 string repPat1 = string(10, 'a');  
 string repPat2 = string(10, 'b');  
 runLinear(repText, repPat1, "repeated\_all\_match");  
 runBMH(repText, repPat1, "repeated\_all\_match");  
 runLinear(repText, repPat2, "repeated\_no\_match");  
 runBMH(repText, repPat2, "repeated\_no\_match");  
  
 // Larger randomized texts  
 string text1e4 = makeString(10000);  
 string pat10 = makeString(10);  
 string pat50 = makeString(50);  
 string pat200 = makeString(200);  
  
 runLinear(text1e4, pat10, "n=1e4,m=10\_rand");  
 runBMH(text1e4, pat10, "n=1e4,m=10\_rand");  
 runLinear(text1e4, pat50, "n=1e4,m=50\_rand");  
 runBMH(text1e4, pat50, "n=1e4,m=50\_rand");  
 runLinear(text1e4, pat200, "n=1e4,m=200\_rand");  
 runBMH(text1e4, pat200, "n=1e4,m=200\_rand");

Продолжение листинга 1.3

string text1e5 = makeString(100000);  
 runLinear(text1e5, pat10, "n=1e5,m=10\_rand");  
 runBMH(text1e5, pat10, "n=1e5,m=10\_rand");  
 runLinear(text1e5, pat50, "n=1e5,m=50\_rand");  
 runBMH(text1e5, pat50, "n=1e5,m=50\_rand");  
  
 out.close();  
  
 return 0;  
}

1. Тестирование:



Таблица 1.2 – результаты тестирования алгоритмов

1. Маленький текст (n = 9, m = 2).

В случаях совпадения образца в начале, середине и конце строки оба алгоритма показали одинаковый результат. Количество сравнений варьировалось от 2 до 6, время выполнения незначительно. При отсутствии совпадения линейный поиск выполнил 8 сравнений, тогда как алгоритм Бойера–Мура–Хорспула (BMH) сократил их до 4 за счёт использования механизма сдвигов. Таким образом, для малых данных различия в производительности минимальны, но BMH демонстрирует более экономное использование сравнений при отсутствии совпадений.

1. Повторяющиеся символы (n = 1000, m = 10).

В случае полного совпадения образца с текстом (строки из одинаковых символов) оба алгоритма выполнили ровно m сравнений и завершились сразу после первого совпадения. При отсутствии совпадений линейный поиск выполнил 991 сравнение, тогда как BMH ограничился 100 сравнениями и 100 сдвигами. Это демонстрирует преимущество BMH в подобных сценариях, хотя оба алгоритма работают близко к худшему случаю.

1. Средние тексты (n = 10 000, m = 10, 50, 200).

Образец находился в начале текста. В этом случае оба алгоритма выполняли количество сравнений, равное длине образца (10, 50, 200 соответственно). Время выполнения оставалось минимальным. BMH при этом фиксировал наличие сдвигов, тогда как линейный поиск последовательно проверял позиции. Отличия в производительности при таких данных несущественны.

1. Большие тексты (n = 100 000, m = 10, 50).

При случайных данных и отсутствии совпадений линейный поиск перебирал все возможные позиции, выполняя сравнения на каждой из них. BMH выполнял меньше сравнений за счёт больших сдвигов окна поиска. Время выполнения обоих алгоритмов оставалось небольшим, однако BMH показал лучшие результаты по числу сравнений и смещений, что подтверждает его эффективность на больших объёмах данных.

Общий вывод.

Оба алгоритма имеют одинаковую асимптотическую сложность в худшем случае (O(n·m)), однако практические результаты показывают преимущество алгоритма Бойера–Мура–Хорспула при обработке больших и случайных текстов. Для малых входных данных различия несущественны. Линейный поиск отличается простотой реализации и предсказуемым поведением, но хуже масштабируется при увеличении размера текста.

**Вывод:**

В ходе работы были изучены алгоритмы поиска подстроки: линейный и Бойера–Мура–Хорспула. Проведённое тестирование показало, что при одинаковой теоретической сложности в худшем случае BMH на практике выполняет меньше сравнений и эффективнее обрабатывает большие тексты.