МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №8

Поиск в тексте образца. Алгоритмы. Эффективность алгоритмов
по дисциплине
«СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ»

Выполнил студент Иолович Е.А.

группа ИНБО-03-22

Москва 2023

СОДЕРЖАНИЕ

1 ОТВЕТЫ НА ВОПРОСЫ ... 3
2 ПОСТАНОВКА И УСЛОВИЕ ЗАДАЧИ ... 8
2.1 Условие задачи и требования ... 8
2.2 Постановка задачи ... 9
3 ЗАДАНИЕ 1 ... 10
3.1 Код задачи ... 10
3.2 Сводная таблица результатов ... 11
3.3 График зависимости ... 11
3.4 Вывод программы ... 11
3.5 Анализ алгоритма ... 12
4 ЗАДАНИЕ 2 ... 13
4.1 Код задачи ... 13
4.2 Сводная таблица результатов ... 13
4.3 График зависимости ... 15
4.4 Вывод программы ... 15
4.5 Анализ алгоритма ... 15
5 ЗАДАНИЕ 3 ... 16
5.1 Код задачи ... 16
5.2 Сводная таблица результатов ... 17
5.3 Вывод программы ... 17
5.4 Анализ алгоритма ... 17
6 ВЫВОДЫ ... 18
7 СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ ... 19

1 ОТВЕТЫ НА ВОПРОСЫ

1. Что называют, строкой?

Строка — это последовательность символов. Эти символы могут быть любыми —буквами, цифрами, специальными символами и т.д.

2. Что называют префиксом строки?

Префиксом строки называют подстроку, которая находится в начале данной строки. Например, для строки "Hello, World!" ее префиксами будут: "H", "He", "Hel", "Hell", "Hello". Префикс может быть пустым для пустой строки или состоять из одного символа для строки из одного символа.

3. Что называют суффиксом строки?

Суффиксом строки называют подстроку, которая находится в конце данной строки. Например, для строки "Hello, World!" ее суффиксами будут: "!", "d!", "ld!", "rld!", "orld!", "World!", и т.д.

4. Асимптотическая сложность последовательного поиска подстроки в строке?

Асимптотическая сложность последовательного поиска подстроки в строке равна O(n*m), где n - длина строки, а m - длина подстроки.

5. В чем особенность поиска образца алгоритмом Бойера –Мура?

Особенностью алгоритма Бойера-Мура для поиска образца в строке является использование эвристических подходов для предотвращения бесполезных сравнений. Ключевым моментом в алгоритме Бойера-Мура является подсчет смещений для каждого символа в образце, которые позволяют "прыгать" через несколько символов при несовпадении.

6. Приведите асимптотическую сложность алгоритма Бойера – Мура поиска подстроки в строке по времени и памяти.

Асимптотическая сложность алгоритма Бойера-Мура поиска подстроки в строке по времени составляет O(n+m), где n - длина исходной строки, m - длина подстроки. Это происходит благодаря использованию таблицы сдвигов и правила пропуска, которые уменьшают количество операций сравнения символов.

7. Приведите пример входных данных для реализации эффективного метода прямого поиска подстроки в строке.

Строка: «abcabb»; Подстрока «abb»

8. Приведите пример строки, для которой поиск подстроки "aaabaaa" будет более эффективным, если делать его методом Кнута, Морриса и Пратта, чем, если делать его методом Бойера-Мура. И наоборот.

Метод Бойера-Мура обычно более эффективен в случаях, когда в строке есть много различных символов, а подстрока содержит несколько повторяющихся символов. Например, для строки "aabaacaadaabaaba" и подстроки "aaba" метод Бойера-Мура будет более эффективен, чем метод Кнута, Морриса и Пратта. С другой стороны, метод Кнута, Морриса и Пратта может быть более эффективен в случаях, когда в строке есть много повторяющихся символов, а подстрока содержит несколько различных символов. Например, для строки 4 "aaaaaaaaaaaaaab" и подстроки "aab" метод Кнута, Морриса и Пратта будет более эффективен, чем метод Бойера-Мура.

9. Объясните, как влияет размер таблицы кодов в алгоритме Бойера и Мура на скорость поиска.

Размер таблицы смещений зависит от количества возможных символов в алфавите. Если алфавит маленький, то таблица будет меньше, что может ускорить поиск, так как меньше информации нужно обрабатывать для каждого символа. Однако, в противном случае таблица может стать очень большой, что затруднит как хранение, так и обработку. В таком случае, использование алгоритма Кнута-Морриса-Пратта может быть более эффективным, так как он не использует таблицу смещений.

10. За счет чего в алгоритме Бойера и Мура поиск оптимален в большинстве случаев?

При несовпадении символа шаблона и символа текста алгоритм сдвигает окно на максимально возможное расстояние, основываясь на таблице

смещений. Это позволяет "пропустить" большое количество символов, не сравнивая их с шаблоном.

11. Поясните влияние префикс-функции в алгоритме Кнута, Морриса и Пратта (КМП) на организацию поиска подстроки в строке. Префикс-функция позволяет находить максимальный суффикс строки, который является её же префиксом. Эта функция используется в алгоритме КМП для определения длины наибольшего префикса подстроки, который совпадает с её суффиксом. В процессе поиска подстроки в строке алгоритм КМП с помощью префикс-функции выполняет сравнение символов только один раз, что делает его очень быстрым в большинстве случаев. После того, как префикс-функция вычислена, алгоритм КМП может осуществлять поиск с линейной временной сложностью O(n), где n - длина строки, что является оптимальным значением для алгоритма поиска подстроки. Таким образом, использование префикс-функции в алгоритме КМП позволяет существенно повысить эффективность поиска подстроки в строке.

12. Приведите пример префикс-функции для поиска образца в тексте для алгоритма КМП.

Префикс-функция для образца «abcabc» будет выглядеть следующим образом:

	a	b	c	a	b	c
P	0	0	0	1	2	3

13. В чем особенность поиска образца алгоритмом Рабина и Карпа?

Особенностью алгоритма Рабина-Карпа является то, что он позволяет осуществлять поиск подстроки в строке со сложностью O(n+m), где n – длина строки, а m – длина подстроки. Алгоритм использует хеширование для быстрого определения равенства строк, что позволяет сократить количество необходимых сравнений.

14. Приведите асимптотическую сложность алгоритма Рабина и Карпа поиска подстроки в строке.

Асимптотическая сложность алгоритма Рабина-Карпа для поиска подстроки в строке составляет O(n+m), где n - длина строки, а m - длина подстроки. Это означает, что время работы алгоритма зависит от суммарной длины строки и подстроки, а не от их отношения.

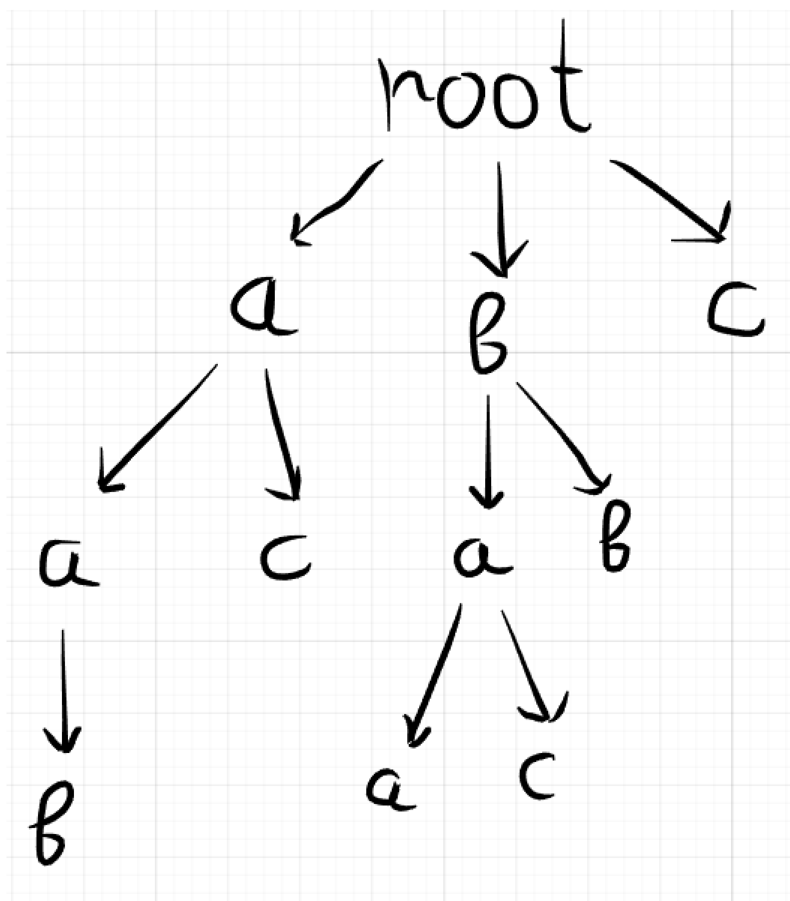
15. Что такое бор?

Бор (также известен как Trie, префиксное дерево) — это структура данных, которая используется для хранения множества строк или последовательностей символов в виде дерева. Это дерево, в котором каждый узел представляет префикс некоторой строки, и каждый лист соответствует полной строке. В отличие от обычного дерева, где каждый узел имеет максимум двух потомков, каждый узел в боре может иметь любое количество потомков, равное числу уникальных символов в алфавите, используемом для строк.

16. Какие структуры хранения данных используются для реализации простого бора?

Для реализации простого бора используется массив указателей на узлы бора.

17. Приведите пример бора и реализуйте его одним из способов. Объясните алгоритм поиска образца с использованием бора.



1)aab 2)ac 3)baa 4)baa 5)bac 6)bb 7)c

18. Поясните применение алгоритма Ахо – Корасик. Приведите его вычислительную и емкостную сложность.

Алгоритм Ахо – Корасик используется для эффективного поиска множества строк (например, ключевых слов) в тексте. Он является усовершенствованным вариантом алгоритма Трие или бора.

Время работы алгоритма составляет O(n + m + z), где n - длина текста, m – суммарная длина всех строк, z - количество вхождений строк в текст.

2 ПОСТАНОВКА И УСЛОВИЕ ЗАДАЧИ

2.1 Условие задачи и требования

1. Выполнить разработку программы, выполняя все этапы разработки.

2. Включить в этап «Описание модели (подход к решению)» описание алгоритма рассматриваемого метода. Разобрать алгоритм на примере. Подсчитать количество сравнений для успешного поиска первого вхождения образца в текст и безуспешного поиска. Определить функцию (или несколько функций) для реализации алгоритма. Определить предусловие и постусловие.

3. Сформировать таблицу тестов с указанием успешного и неуспешного поиска, используя большой и небольшой по объему текст, и образец различного объема. Включить ее в этап тестирование.

4. Разработать и реализовать программу тестирования алгоритма.

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

2.2 Постановка задачи

Получить знания и навыки применения алгоритмов поиска в тексте подстрок (образца).

Номер индивидуального варианта: 10%15 + 1 = 16, но 16 нет, поэтому 1.

Номер теста	Задачи варианта
1	1) Линейный поиск первого вхождения подстроки в строку. 2) Используя алгоритм Бойера-Мура. Найти первое вхождение подстроки в строку. 3) Поиск по бору. Создать словарь слов исходного текста на основе цифрового поиска с применением бора, реализовав операцию добавления нового слова, если слово не было найдено в словаре.

3 ЗАДАНИЕ 1

3.1 Код задачи

```
#include <iostream>
#include <string>
#include <random>
#include <chrono>
using namespace std;
using namespace chrono;


string generateString(int length)
{
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution <int> charDistribution(65, 90);

    string str;
    for (int i = 0; i < length; ++i)
    {
        char randomChar = static_cast<char>(charDistribution(gen));
        str.push_back(randomChar);
    }

    return str;
}

int linearSearch(const string& str, const string& substr)
{
    int n = str.length();
    int m = substr.length();

    for (int i = 0; i <= n - m; ++i)
    {
        int j;

        for (j = 0; j < m; ++j)
        {
            if (str[i + j] != substr[j])
                break;
        }
        if (j == m)
        {
            return i;  //подстрока найдена, возвращаем индекс первого символа.
        }
    }
    return -1;  //подстрока не найдена.
}

int countComparisons(const string& str, const string& substr)
{
    int n = str.length();
    int m = substr.length();
    int comparisons = 0;

    for (int i = 0; i <= n - m; ++i)
    {
        int j;

        for (j = 0; j < m; ++j)
        {
            comparisons++;
            if (str[i + j] != substr[j])
                break;
        }
```

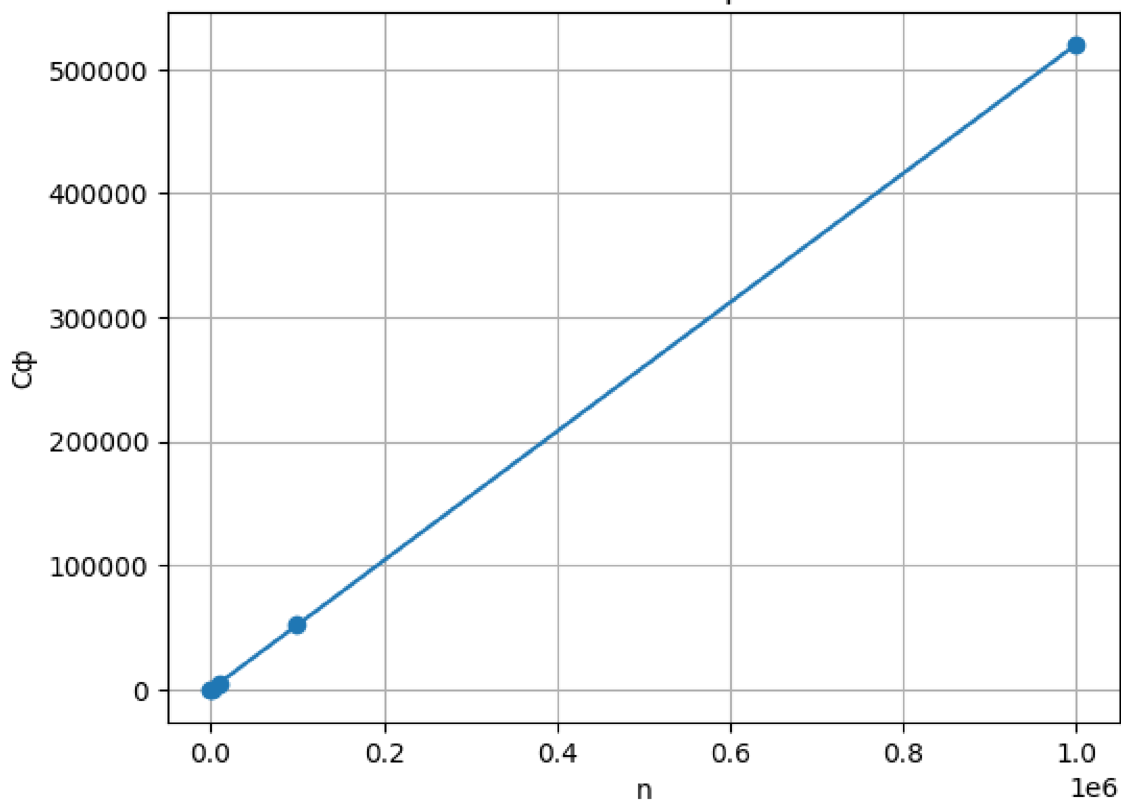
```
        if (j == m)
        {
            break;
        }
    }

    return comparisons;
}


int main()
{
    setlocale(LC_ALL, "Russian");

   /* string str = "Hello, world!";
    string substr = "world"; */

    int length;
    char character;

    cout << "Введите длину строки: ";
    cin >> length;

    string str;
    string substr;

    do
    {
        str = generateString(length);
        int substrLength = 5; //желаемая длина подстроки
        int substrStart = (length - substrLength) / 2; //индекс начала подстроки в
середине строки
        substr = str.substr(substrStart, substrLength);


    } while (linearSearch(str, substr) == -1);

    ////////////////////////////////////////////////////////////////////index search.
    int index;
    auto start1 = chrono::steady_clock::now();
    index = linearSearch(str, substr);
    auto end1 = chrono::steady_clock::now();
    ///////////////////////////////////////////////////////////////////compare search.
    int comparisons = countComparisons(str, substr);
    cout << endl;
    cout << "Кол-во сравнений: " << comparisons;
    //////////////////////////////////////////////////////////////////time search.
    cout << "\n\nЗатраченное время в микросекунд для поиска:\n" <<
duration_cast<chrono::microseconds>(end1 - start1).count() << "
мкс.\n";///////////////////////\\
    //////////////////////////////////////////////////////////////////

    cout << "Сгенерированная строка: " << str << endl;
    cout << "Искомая подстрока: " << substr << endl;

    if (index != -1)
    {
        cout << "Подстрока найдена на позиции: " << index << endl;
    }
    else
    {
        cout << "Подстрока не найдена" << endl;
    }

    return 0;
}
```

3.2 Сводная таблица результатов

n – количество символов строки

n	T, микросекунд (средний случай, когда искомая подстрока находится в середине строки)	Тэп = f(C) - функция	Сф - количество
100	3	O(n)	53
1000	27	O(n)	518
10000	263	O(n)	5217
100000	3372	O(n)	51938
1000000	25196	O(n)	519983

3.3 График зависимости



3.4 Вывод программы

```
Введите длину строки: 100

Кол-во сравнений: 55

Затраченное время в микросекунд для поиска:
3 мкс.
Сгенерированная строка: UBPWTNNLXDGRHUBXYWJICPWWSWAGGRFWPRZFNCMXSIEOFKUXWJCVDVIXKXYAXRSPJXAUHKNOBFWUNKMLLAZPYSDMDFASKZKV
WQPQ
Искомая подстрока: XWJCV
Подстрока найдена на позиции: 47
```

3.5 Анализ алгоритма

Линейный поиск первого вхождения подстроки в строку имеет следующие особенности в зависимости от случаев, когда подстрока находится и когда не находится:

Случай, когда подстрока найдена:

Лучший случай: Подстрока находится в самом начале строки. В этом случае, сложность алгоритма будет O(1), так как достаточно будет выполнить всего одно сравнение.

Худший случай: Подстрока находится в самом конце строки или отсутствует вообще. В этом случае, алгоритм будет выполнен за O(n), где n - длина строки. Потому что алгоритм будет проверять каждый символ строки до конца.

Случай, когда подстрока не найдена:

Алгоритм завершится после проверки всех символов строки, и подстрока не будет найдена. В этом случае, сложность алгоритма также будет O(n), так как нужно пройти по всей строке.

Линейный поиск первого вхождения подстроки в строку является простым, но не самым эффективным алгоритмом для этой задачи. Его сложность зависит от положения подстроки в строке, и в среднем и худшем случаях требуется просмотреть все символы строки.

4 ЗАДАНИЕ 2

4.1 Код задачи

```
5   #include <iostream>
6   #include <string>
7   #include <vector>
8   #include <cstdlib>
9   #include <ctime>
10  #include <chrono>
11  using namespace std;
12  using namespace chrono;
13
14  //функция для создания таблицы смещений.
15  vector <int> createShiftTable(const string& substr)
16  {
17      int m = substr.length();
18      vector<int> shiftTable(256, m); //инициализируем таблицу
    максимальным смещением.
19
20      for (int i = 0; i < m - 1; ++i)
21      {
22          shiftTable[substr[i]] = m - 1 - i;
23      }
24
25      return shiftTable;
26  }
27
28  int boyerMooreSearch(const string& str, const string& substr)
29  {
30      int n = str.length();
31      int m = substr.length();
32
33      vector<int> shiftTable = createShiftTable(substr);
34
35      int i = m - 1; //индекс для прохода по строке.
36      int j = m - 1; //индекс для прохода по подстроке.
37
38      while (i < n)
39      {
40          if (str[i] == substr[j])
41          {
42              if (j == 0)
43              {
44                  return i; //подстрока найдена, возвращаем индекс первого
    символа.
45              }
46              --i;
47              --j;
48          }
49          else
```

```
50              {
51                  i += shiftTable[str[i]]; //смещаем индекс в строке на основе 
таблицы смещений.
52                  j = m - 1; //возвращаем индекс для прохода по подстроке в 
конец.
53              }
54          }
55  
56          return -1; //подстрока не найдена.
57      }
58  
59      int countComparisons(const string& str, const string& substr)
60      {
61          int n = str.length();
62          int m = substr.length();
63  
64          vector<int> shiftTable = createShiftTable(substr);
65  
66          int i = m - 1; //индекс для прохода по строке.
67          int j = m - 1; //индекс для прохода по подстроке.
68  
69          int comparisons = 0; //счетчик количества сравнений.
70  
71          while (i < n)
72          {
73              if (str[i] == substr[j])
74              {
75                  if (j == 0)
76                  {
77                      return comparisons + 1; //возвращаем общее количество 
сравнений (включая текущее сравнение).
78                  }
79                  --i;
80                  --j;
81                  comparisons++; //увеличиваем счетчик сравнений.
82              }
83              else
84              {
85                  i += shiftTable[str[i]]; //смещаем индекс в строке на основе 
таблицы смещений.
86                  j = m - 1; //возвращаем индекс для прохода по подстроке в 
конец.
87                  comparisons++; //увеличиваем счетчик сравнений.
88              }
89          }
90  
91          return comparisons; //подстрока не найдена.
92      }
93  
94      int countTotalShifts(const string& str, const string& substr)
95      {
```

```
96          int n = str.length();
97          int m = substr.length();
98
99          vector<int> shiftTable = createShiftTable(substr);
100
101         int i = m - 1; //индекс для прохода по строке.
102         int j = m - 1; //индекс для прохода по подстроке.
103
104         int totalShifts = 0; //счетчик общего количества перемещений.
105
106         while (i < n)
107         {
108             if (str[i] == substr[j])
109             {
110                 if (j == 0)
111                 {
112                     return totalShifts; //возвращаем общее количество
        перемещений.
113                 }
114                 --i;
115                 --j;
116             }
117             else
118             {
119                 int shift = shiftTable[str[i]];
120                 totalShifts += j + 1; //увеличиваем счетчик общего
        количества перемещений.
121                 i += shift;
122                 j = m - 1;
123             }
124         }
125
126         return totalShifts; //подстрока не найдена.
127     }
128
129
130     int main()
131     {
132         setlocale(LC_ALL, "Russian");
133         srand(static_cast<unsigned int>(time(0)));
134
135         string str;
136         string substr;
137
138         //генерация основной строки.
139         int strLength = 100; //длина основной строки.
140         for (int i = 0; i < strLength; ++i)
141         {
142             char randomChar = 'a' + rand() % 26; //генерация случайной буквы
        от 'a' до 'z'.
```

```
143             str.push_back(randomChar);
144         }
145 
146         //генерация случайной подстроки из основной строки.
147         int desiredLength = 5; //желаемая длина подстроки.
148         int startIndex = (strLength - desiredLength) / 2; //индекс начала
    подстроки в середине начальной строки.
149         substr = str.substr(startIndex, desiredLength);
150 
151         cout << "Сгенерированная строка: " << str << endl;
152         cout << "Искомая подстрока: " << substr << endl;
153 
154 
155         auto start1 = chrono::steady_clock::now();
156         int index = boyerMooreSearch(str, substr); //функция поиска бойра-
    мура.
157         auto end1 = chrono::steady_clock::now();
158 
159         cout << "\n\nЗатраченное время в микросекунд для поиска:\n" <<
    duration_cast<chrono::microseconds>(end1 - start1).count() << "
    мкс.\n";//////////////////////\
160 
161         int shifts = countTotalShifts(str, substr);
162         cout << endl;
163         cout << "Кол-во перемещений: " << shifts;
164 
165         int comparisons = countComparisons(str, substr);
166         cout << endl;
167         cout << "Кол-во сравнений: " << comparisons;
168 
169         cout << endl;
170 
171         if (index != -1)
172         {
173             cout << "Подстрока найдена на позиции " << index << endl;
174         }
175         else
176         {
177             cout << "Подстрока не найдена" << endl;
178         }
179 
180         return 0;
181     }
```

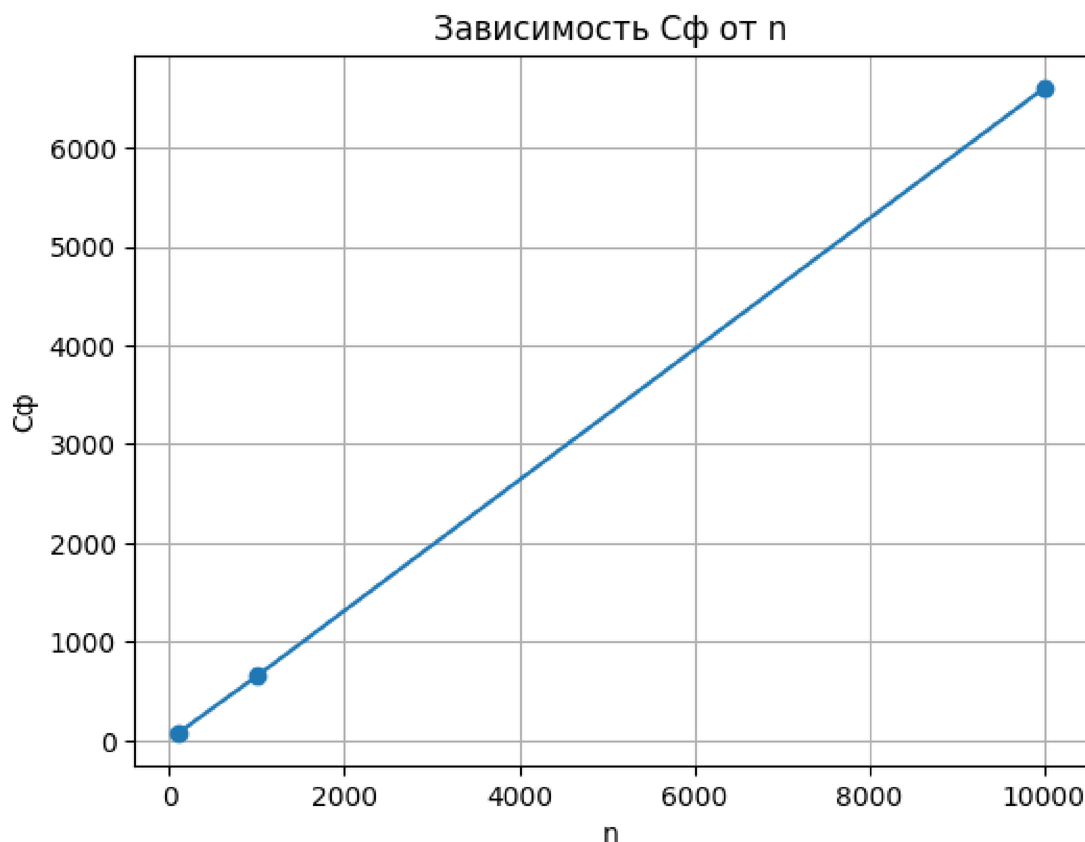
4.2 Сводная таблица результатов

m = 5 – количество символов подстроки

n – количество символов строки

n	**T, микросекунд (средний случай, когда искомая подстрока находится в середине строки)**	**Тэп = f(C+M) - функция**	**Сф - количество**
100	34	O(n\m)	59 + 18 = 77
1000	35	O(n\m)	538 + 115 = 653
10000	152	O(n\m)	5465 + 1140 = 6605

4.3 График зависимости



4.4 Вывод программы

```
Сгенерированная строка: uetuoxqgkggndahetdftdiyiuagwglikcqwrccsarrdsszdhdkhsqjzkdzfejswvrvdjhqiszvlyvldidentqevtwakbymvw
frxc
Искомая подстрока: hdkhs


Затраченное время в микросекунд для поиска:
34 мкс.

Кол-во перемещений: 59
Кол-во сравнений: 18
Подстрока найдена на позиции 47
```

4.5 Анализ алгоритма

Алгоритм Бойера-Мура имеет следующую сложность:

Лучший случай: если подстрока находится в самом начале строки, то алгоритм будет выполняться за O(m), где m - длина подстроки. В этом случае, число сравнений будет минимальным.

Худший случай: в худшем случае, когда подстрока не найдена или находится в конце строки, алгоритм Бойера-Мура будет выполняться за O(n), где n - длина строки. Это связано с тем, что мы можем пропустить несколько символов при каждом смещении.

Средний случай: в среднем случае, сложность алгоритма Бойера-Мура составляет O(n/m), где n - длина строки, m - длина подстроки. Это связано с тем, что в среднем мы можем пропускать большее количество символов при каждом

смещении.

Память: Дополнительная память, используемая алгоритмом, связана только с созданием таблицы смещений, которая занимает O(256) = O(1) пространства.

Если подстрока не найдена, то в случае алгоритма Бойера-Мура сложность будет линейной относительно длины строки, в которой производится поиск. В алгоритме Бойера-Мура, когда подстрока не найдена, выполняется смещение индекса i в строке на основе таблицы смещений и возвращаем индекс для прохода по подстроке в конец. Затем i увеличивается на shiftTable[str[i]] и процесс повторяется до тех пор, пока не будет достигнут конец строки. В этом случае сложность алгоритма Бойера-Мура будет **O(n)**, где **n - длина строки**, в которой выполняется поиск. Каждое смещение индекса i увеличивает его значение на значение из таблицы смещений, что приводит к линейному времени выполнения.

В общем случае, алгоритм Бойера-Мура работает эффективно и может быть значительно быстрее линейного поиска в больших строках или подстроках, особенно когда подстрока не найдена или находится в конце строки. Однако, его производительность может ухудшиться, когда подстрока имеет повторяющиеся символы или когда строка и подстрока имеют схожие структуры, что может привести к большому количеству сравнений и смещений.

5 ЗАДАНИЕ 3

5.1 Код задачи

```
6	#include <iostream>
7	#include <unordered_map> //используется в реализации бора для хранения
	дочерних узлов каждого узла. это контейнерное обобщение ассоциативного
	массива, где каждый элемент представляет пару ключ-значение.
8	#include <random>
9	#include <ctime>
10	#include <chrono>
11	using namespace std;
12	using namespace chrono;
13	
14	struct TrieNode //содержит информацию о дочерних узлах и флаге,
	указывающем на конец слова.
15	{
16	    unordered_map <char, TrieNode*> children; //контейнер unordered_map,
	который
17	    //хранит дочерние узлы текущего узла бора.
18	
19	    bool isEndOfWord; //флаг, который указывает, является ли текущий
	узел концом (последней буквой)
20	    //какого-либо слова в боре. если isEndOfWord установлен в true, это
	означает, что до текущего
21	    //узла можно обойти буквы и получить полное слово.
22	};
23	
24	TrieNode* createNode()
25	{
26	    TrieNode* newNode = new TrieNode; //создание нового узла с помощью
	оператора new.
27	    newNode->isEndOfWord = false; //инициализация флага isEndOfWord
	значением false.
28	    return newNode; //возвращение указателя на новый узел.
29	}
30	
31	void insert(TrieNode* root, const string& word)
32	{
33	    TrieNode* current = root; //инициализация текущего узла как
	корневого узла.
34	    for (char ch : word)  //цикл для каждого символа в слове.
35	    {
36	        if (current->children.find(ch) == current->children.end())
	//есть ли утекущего узла дочерний, соответствующий символу ch.
37	        {
38	            //если символ не является дочерним узлом текущего узла.
39	            current->children[ch] = createNode(); //создать узел для
	символа.
40	        }
41	        current = current->children[ch]; //перейти к дочернему узлу для
	символа.
42	    }
```

```
43      current->isEndOfWord = true; //пометить текущий узел как конец
    слова.
44  }
45
46  bool search(TrieNode* root, const string& word)
47  {
48      TrieNode* current = root; //инициализация текущего узла как
    корневого узла.
49      for (char ch : word) //цикл для каждого символа в слове.
50      {
51          if (current->children.find(ch) == current->children.end())
    //если символ не является дочерним узлом текущего узла.
52          {
53              return false; //слово не найдено.
54          }
55          current = current->children[ch]; //перейти к дочернему узлу для
    символа.
56      }
57      return current->isEndOfWord; //вернуть значение флага последнего
    узла.
58  }
59
60  vector <string> generateRandomWords(int count, int length)
61  {
62      vector <string> words;
63      string characters = "abcdefghijklmnopqrstuvwxyz"; //строка, из
    которой будут браться значения для генерации.
64
65      //строки кода инициализируют генератор случайных чисел gen с помощью
    случайных значений, полученных от random_device.
66      //затем создается распределение charDistribution, которое будет
    использоваться для генерации случайных индексов символов из строки
    characters, короче мега-имба.
67      random_device rd;
68      mt19937 gen(rd());
69      uniform_int_distribution <int> charDistribution(0,
    characters.length() - 1);
70
71      for (int i = 0; i < count; i++)
72      {
73          string word;
74          for (int j = 0; j < length; j++)
75          {
76              char randomChar = characters[charDistribution(gen)];
77              word += randomChar;
78          }
79          words.push_back(word);
80      }
81
82      return words;
83  }
84
```

```
85      int countComparisons(TrieNode* root, const string& word)
86      {
87          int comparisonCount = 0;
88          TrieNode* current = root; // инициализация текущего узла как
        корневого узла.
89          for (char ch : word) // цикл для каждого символа в слове.
90          {
91              if (current->children.find(ch) == current->children.end()) //
        если символ не является дочерним узлом текущего узла.
92              {
93                  comparisonCount++; // увеличение счетчика сравнений.
94                  return comparisonCount; // возврат количества сравнений.
95              }
96              current = current->children[ch]; // перейти к дочернему узлу для
        символа.
97              comparisonCount++; // увеличение счетчика сравнений.
98          }
99          comparisonCount++; // увеличение счетчика сравнений для последнего
        узла (флага конца слова).
100         return comparisonCount; // возврат количества сравнений.
101     }
102
103     int main()
104     {
105         setlocale(LC_ALL, "Russian");
106
107         TrieNode* root = createNode();
108
109         /*vector <string> dictionary = {"apple", "banana", "cat", "kit",
        "ybitemeni"};*/
110
111         int wordCount;
112         cout << "Введите количество случайных слов для словаря: ";
113         cin >> wordCount;
114
115         int wordLength;
116         cout << "Введите длину случайных слов: ";
117         cin >> wordLength;
118
119         vector <string> dictionary = generateRandomWords(wordCount,
        wordLength);
120
121         cout << endl;
122
123         cout << "Словарь:" << endl; //вывод вектора изначального словаря.
124         for (const string& word : dictionary)
125         {
126             cout << word << endl;
127         }
128         cout << endl;
129
```

```
130     for (const string& word : dictionary) //таким образом, данный цикл for перебирает каждое слово в векторе dictionary и вызывает функцию insert() для вставки каждого слова в бор.
131         //это позволяет построить бор, содержащий все слова из исходного словаря.
132     {
133         insert(root, word);
134     }
135 
136     //поиск слов в словаре, пользователь сам вводит слово.
137     cout << "Внимание! Если выводится 0, значит такого слова в словаре нет, если 1 - есть."  << endl;
138     string slowo = dictionary[wordCount / 2]; //wordCount - количество сгенерированных слов, wordCount / 2 - индекс серединного слова в списке
139     cout << "Искомое слово: " << slowo << endl;
140 
141     //cout << search(root, "apple") << endl; // нет
142     //cout << search(root, "dog") << endl; // нет
143 
144     auto start1 = chrono::steady_clock::now();
145     cout << "Результат работы поиска: " << search(root, slowo) << endl;
146     auto end1 = chrono::steady_clock::now();
147 
148     cout << endl;
149 
150     int comparisonCount = countComparisons(root, slowo);
151     cout << "Количество сравнений: " << comparisonCount << endl;
152 
153     cout << endl;
154 
155     cout << "\n\nЗатраченное время в микросекунд для поиска:\n" << duration_cast<chrono::microseconds>(end1 - start1).count() << " мкс.\n";
156 
157     cout << endl;
158 
159     //добавление нового слова.
160     cout << "Введите новое слово, которое необходимо добавить в словарь: " << endl;
161     string newWord;
162     cin >> newWord;
163     dictionary.push_back(newWord);
164     insert(root, newWord);
165 
166     cout << "Обновленный словарь:" << endl; //вывод обновленного варианта словаря.
167     for (const string& word : dictionary)
168     {
169         cout << word << endl;
170     }
171 
172     cout << endl;
173 
```

```
174         cout << "Проверка того, что слово было добавлено: " << endl;
175         cout << search(root, newWord) << endl; // есть
176 
177 
178         return 0;
179     }
180 
```

5.2 Сводная таблица результатов(поиска)

n – количество слов в словаре
m = 5 – длина искомого слова

n	T, микросекунд (если искомое слово найдено)	Тэп = f(C) - функция	Сф - количество
100	1915	O(m)	6
1000	1559	O(m)	6
10000	1314	O(m)	6
100000	1071	O(m)	6

5.3 Вывод программы

```
Введите количество случайных слов для словаря: 4
Введите длину случайных слов: 3

Словарь:
yue
qsd
euh
jct

Внимание! Если выводится 0, значит такого слова в словаре нет, если 1 - есть.
Введите искомое слово:
age
Результат работы поиска: 0



Затраченное время в микросекунд для поиска:
974 мкс.

Введите новое слово, которое необходимо добавить в словарь:
age
Обновленный словарь:
yue
qsd
euh
jct
age

Проверка того, что слово было добавлено:
1
```

5. 4 Анализ алгоритма

Прежде чем перейти к общему анализу алгоритма, нужно отметить, что время выполнения функции **search** уменьшается с увеличением количества слов в словаре, это связано с оптимизацией структуры данных бора и эффективностью поиска. Когда добавляются новые слова в бор, структура данных бора становится более заполненной и содержит больше информации о

словах. Это позволяет функции **search** быстрее определить наличие или отсутствие слова в словаре, так как она может сразу перейти к соответствующему поддереву бора, пропуская ненужные ветви. При поиске слова в боре, функция **search** последовательно сравнивает каждый символ слова с дочерними узлами текущего узла бора. Если символ не является дочерним узлом, поиск прекращается и возвращается false. Когда слово находится в боре, функция **search** успешно обходит все символы и возвращает true. С ростом размера словаря, структура бора остается той же, но количество узлов и связей в нем увеличивается. Это означает, что при поиске функция **search** может пропускать больше шагов и делать меньше сравнений, так как уже содержит информацию о ранее добавленных словах. Таким образом, увеличение количества слов в словаре может приводить к улучшению производительности функции **search**, так как более полный и информативный бор позволяет более эффективный поиск слов.

Сложность алгоритма в данном поиске зависит от длины искомого слова, обозначим ее как M, а также от общего количества слов в словаре, обозначим как N.

Вставка слов в бор (insert): при вставке каждого слова в бор, проходимся по каждому символу слова и выполняем операции вставки в хэш-таблицу для каждого символа. Если средняя длина слова равна M, а общее количество слов в словаре равно N, то сложность вставки будет O(M * N).

Поиск слова в боре(search): при поиске слова проходимся по каждому символу слова и выполняем операции поиска в хэш-таблице для каждого символа. Если длина искомого слова равна M, то сложность поиска будет O(M).

Итоговая сложность алгоритма для данного поиска состоит из суммы сложностей вставки, поиска: O(M * N) + O(M) = O(M * N)

Таким образом, сложность алгоритма в данном поиске будет O(M * N), где **M - длина искомого слова**, а **N - общее количество слов в словаре**.

6 ВЫВОДЫ

В ходе проведенной практической работы были изучены и проанализированы три алгоритма: линейный поиск, алгоритм Бойера-Мура и поиск по бору.

В заключении, можно сказать, что каждый из изученных алгоритмов имеет свои преимущества и недостатки в зависимости от задачи, которую необходимо решить.

7 СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Материалы по дисциплине (Сорокин А.В.).

2. Скворцова Л.А., Гусев К.В., Трушин С.М., Филатов А.С. Учебно-методическое пособие СиАОД.