**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»**

**(СПбГУТ)**

**Факультет инфоРМАЦИОННЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ (ИТПИ)**

**кафедра программной инженерии и вычислительной техники (пиивт)**

Дисциплина: «Алгоритмы и структуры данных»

Лабораторная работа №2

**Тема: «Методы поиска»**

Отчёт

Выполнили:

Студенты группы ИКПИ-33

Коньков М. Д.

Семенихин А. Р.

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Принял:

Дагаев А. В.

Подпись \_\_\_\_\_\_\_\_\_\_\_\_\_\_

Санкт-Петербург

2025г.

**Общая формулировка:**

Целью данной работы является **разработка на языке Python и тщательное исследование нескольких алгоритмов поиска подстроки в тексте**. В ходе исследования планируется **измерить время, затрачиваемое на поиск в текстовых данных разного объема** для каждого алгоритма, с последующим **сравнением их быстродействия**. Полученные **экспериментальные результаты** будут использованы для **оценки практической применимости и анализа теоретической сложности** рассматриваемых методов поиска.

### Общий алгоритм решения

* **Генерация текстовых данных случайной длины** для проведения испытаний.
* **Разработка на языке Python программных реализаций следующих алгоритмов поиска подстроки**
* **Проведение серии измерений времени выполнения** для каждого алгоритма на текстовых данных различной длины.
* **Оценка среднего времени поиска** для каждого алгоритма и каждой размерности текста.
* **Сохранение результатов измерений в файл формата XLSX (Excel)** для удобства дальнейшего анализа и обработки данных.
* **Сравнительный анализ практической производительности реализованных алгоритмов** на основе экспериментально полученных данных о времени выполнения.

### Описание алгоритмов поиска

**Алгоритм последовательного поиска:**

Последовательный (линейный) поиск – это один из самых простых алгоритмов для нахождения подстроки в строке. Алгоритм последовательно сравнивает каждый символ основной строки с первым символом подстроки. Если символы совпадают, начинается проверка остальных символов подстроки. Если вся подстрока совпадает с соответствующей частью строки, поиск завершается. В противном случае процесс продолжается до конца строки.

### ****Пример работы:****

#### ****1. Пошаговый поиск подстроки в строке****

Ищем подстроку **ABC** в строке **ABCAFDBCABC**.

1. **Первый шаг**
   * Начинаем с первого символа строки.
   * Сравниваем **ABC** и **ABC**.
   * Все символы совпадают → подстрока найдена на позиции 0.
2. **Второй шаг** (если продолжаем поиск)
   * Продолжаем проверку с четвертого символа строки.
   * Сравниваем **CAF** и **ABC**.
   * Символ C ≠ A → несовпадение, продолжаем поиск.
3. **Третий шаг**
   * Сравниваем **AFD** и **ABC**.
   * Символ A = A, но F ≠ B → несовпадение, продолжаем поиск.
4. **Четвертый шаг**
   * Сравниваем **FDB** и **ABC**.
   * Символ F ≠ A → несовпадение, продолжаем поиск.
5. **Пятый шаг**
   * Сравниваем **DBC** и **ABC**.
   * Символ D ≠ A → несовпадение, продолжаем поиск.
6. **Шестой шаг**
   * Сравниваем **BCA** и **ABC**.
   * Символ B ≠ A → несовпадение, продолжаем поиск.
7. **Седьмой шаг**
   * Сравниваем **CAB** и **ABC**.
   * Символ C ≠ A → несовпадение, продолжаем поиск.
8. **Восьмой шаг**
   * Сравниваем **ABC** и **ABC**.
   * Все символы совпали → подстрока найдена на позиции 8.

Алгоритм завершает работу, так как подстрока найдена дважды в строке.

**Алгоритм Бойера-Мура:**

Алгоритм Бойера-Мура считается одним из самых быстрых методов поиска подстроки в строке. В начале работы строится таблица смещений, после чего осуществляется процесс поиска. Сравнение символов начинается с конца подстроки и идет справа налево. Если последний символ не совпадает с символом основной строки, производится сдвиг на значение из таблицы смещений. В случае несовпадения другого символа подстрока сдвигается на одну позицию вправо. Если все символы совпали – подстрока найдена.

### ****Пример работы:****

#### ****1. Построение таблицы смещений****

Для построения таблицы смещений определяется удаленность каждого символа от конца подстроки. Используются следующие правила:

* Если символ встречается несколько раз, берется расстояние от последнего вхождения.
* Если символ отсутствует в подстроке, ему присваивается значение, равное длине подстроки.

##### ****Пример: Подстрока ABCABD, строка ABCAFDFABCABD****

Алфавит: A, B, C, D, F

***ч***2. Пошаговый поиск подстроки в строке****

Ищем подстроку ABCABD в строке ABCAFDFABCABD.

1. **Первый шаг**
   * Сравниваем ABCABD и ABCAFD.
   * Символ D ≠ F, но F отсутствует в подстроке.
   * Сдвигаем подстроку на всю её длину (6 символов).
2. **Второй шаг**
   * Сравниваем ABCABD и DFABCA.
   * Символ D ≠ A, A есть в подстроке, его сдвиг – 2.
   * Сдвигаем подстроку на 2 символа.
3. **Третий шаг**
   * Сравниваем ABCABD и ABCABD.
   * Все символы совпали.
   * Подстрока найдена.

Алгоритм завершает работу, так как найдено точное совпадение.

**Алгоритм Кнута-Морриса-Пратта:**

Алгоритм Кнута-Морриса-Пратта (КМП) – в начале работы строит префикс-функцияю(таблица длин наибольших совпадающих префиксов и суффиксов подстроки). Затем происходит сравнение символов строки и подстроки. При несовпадении поиск продолжается не с начала подстроки, а с позиции, определяемой префикс-функцией, что ускоряет процесс.

#### **Пример построения таблицы LPS (префикс-функции)**

**Дано:**

* **Строка (text):** "ABABDABACDABABCABAB"
* **Подстрока (pattern):** "ABABCABAB"

#### **Шаг 1: Генерация таблицы LPS для подстроки "ABABCABAB"**

Таблица LPS (Longest Prefix Suffix) определяет длину наибольшего собственного суффикса, который совпадает с префиксом.

**Алгоритм построения LPS:**

1. Инициализируем массив lps нулями, length = 0, i = 1.
2. Если pattern[i] == pattern[length], то lps[i] = length + 1, увеличиваем length и i.
3. Если нет, то:
   * Если length != 0, то length = lps[length - 1] (возвращаемся назад).
   * Иначе lps[i] = 0 и увеличиваем i.

**Пошаговое заполнение LPS для "ABABCABAB":**

| Индекс (i) | Символ | LPS[i] | Объяснение |
| --- | --- | --- | --- |
| 0 | A | 0 | Всегда 0 (нет префикса/суффикса) |
| 1 | B | 0 | B ≠ A → lps[1] = 0 |
| 2 | A | 1 | A == A → lps[2] = 1 (length = 1) |
| 3 | B | 2 | B == B → lps[3] = 2 (length = 2) |
| 4 | C | 0 | C ≠ A → length = lps[1] = 0 → lps[4] = 0 |
| 5 | A | 1 | A == A → lps[5] = 1 (length = 1) |
| 6 | B | 2 | B == B → lps[6] = 2 (length = 2) |
| 7 | A | 3 | A == A → lps[7] = 3 (length = 3) |
| 8 | B | 4 | B == B → lps[8] = 4 (length = 4) |

**Итоговая таблица LPS:**

Индекс: [0, 1, 2, 3, 4, 5, 6, 7, 8]

Символы: A B A B C A B A B

LPS: [0, 0, 1, 2, 0, 1, 2, 3, 4]

#### **Шаг 2: Поиск подстроки в тексте с использованием LPS**

**Текст:** "ABABDABACDABABCABAB"  
**Подстрока:** "ABABCABAB"  
**LPS:** [0, 0, 1, 2, 0, 1, 2, 3, 4]

****Процесс поиска:****

1. ****Сравнение символов:****
   * Совпадают ABAB (первые 4 символа), затем C ≠ D → несовпадение.
   * lps[3] = 2 → сдвигаем подстроку на j - lps[j-1] = 4 - 2 = 2 позиции.
2. ****Новое сравнение:****
   * Подстрока начинается с позиции 2: "ABABCABAB" vs "ABDABACD...".
   * Совпадает AB (первые 2 символа), затем A ≠ D → lps[1] = 0 → сдвиг на 1.
3. ****Нахождение подстроки:****
   * После нескольких сдвигов подстрока "ABABCABAB" полностью совпадает с частью текста "ABABCABAB" на позиции **10**.

****Визуализация:****

Текст: A B A B D A B A C D A B A B C A B A B

Подстр.: A B A B C A B A B

**Сравнение:**

1. ABAB (совпадение), затем C ≠ D → сдвиг на 2.

2. AB (совпадение), затем A ≠ D → сдвиг на 1.

3. Найдено на позиции 10!

### ****Сравнение алгоритмов поиска****

***Таблица 3***

| ****Алгоритм**** | ****Плюсы**** | ****Минусы**** |
| --- | --- | --- |
| **Последовательный поиск** | Прост в реализации, не требует дополнительной памяти | Медленный при больших объемах данных |
| **Бойера-Мура** | Быстро на длинных строках, использует эвристики для оптимизации | Может быть неэффективным на коротких строках или при частых совпадениях |
| **Кнута-Морриса-Пратта** | Использует предварительную обработку для ускорения поиска | Требует дополнительной памяти для таблицы префиксов |

В ходе выполнения сравнения трех алгоритмов поиска подстроки мы измеряли время выполнения для каждого из них при различных размерах строк, повторяя измерения 10 000 раз для минимизации влияния случайных колебаний. Это позволяет более точно оценить производительность каждого алгоритма и исключить случайные отклонения, вызванные внешними факторами.

Проанализировав результаты, можно выделить несколько ключевых наблюдений:

* **Последовательный поиск** показывает среднюю производительность среди всех алгоритмов. Время выполнения растет линейно с увеличением размера строки, что делает его малопригодным для работы с большими объемами данных.
* **Алгоритм Бойера-Мура** продемонстрировал наибольшее преимущество при работе с длинными строками, особенно когда текст содержит разнообразный алфавит. Эвристики "плохого символа" и "хорошего суффикса" позволяют значительно сократить количество проверок, что ускоряет поиск в случае больших текстов.
* **Алгоритм Кнута-Морриса-Пратта** в сравнении с другими показал более низкую производительность, что связано с особенностями генерации строк. Поскольку строка была случайной и не содержала явных паттернов, время на предобработку данных не давало значительного выигрыша.
* При **малых размерах строк** различия между алгоритмами не так заметны, однако с увеличением длины строки преимущества Бойера-Мура становятся более выраженными, а недостатки алгоритма Кнута-Морриса-Пратта усиливаются.
* На больших строках **алгоритм Бойера-Мура** показывает значительное превосходство в скорости (в несколько десятков раз быстрее, чем последовательный поиск), а **Кнут-Моррис-Пратт** стабильно работает при наличии повторяющихся фрагментов, но не достигает такой же скорости.

Таким образом, выбор алгоритма поиска зависит от размера данных и структуры текста: для небольших строк разница в производительности не так значительна, но для длинных строк алгоритм Бойера-Мура является наиболее эффективным.

**Описание программы**

Данная программа, написанная на языке Python (версии 3.13.2), предназначена для оценки и сравнения производительности различных алгоритмов поиска подстроки в строке. В процессе работы программа генерирует случайные тексты различной длины и случайный ключ (подстроку).

Целью программы является проведение сравнительного анализа производительности данных алгоритмов, акцентируя внимание на времени их выполнения в зависимости от длины текста.

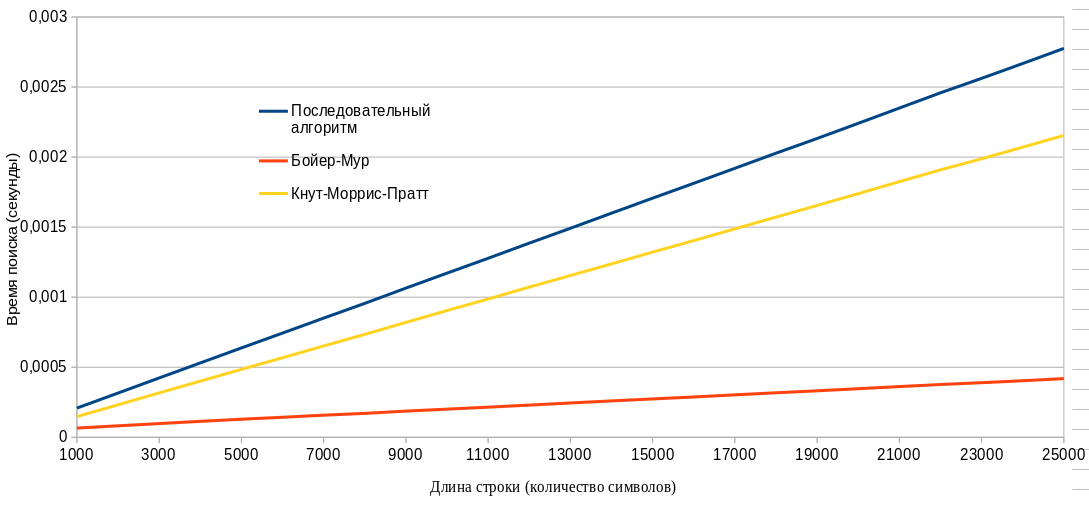
**Основные функции программы:**

• **generate\_key(length=15)** – создает случайную строку-ключ заданной длины, используя буквы и цифры.  
• **generate\_text(length, key)** – формирует случайный текст указанной длины, встраивая в него несколько вхождений заданного ключа.  
• **sequential\_search(text, key)** – реализует последовательный алгоритм поиска подстроки для подсчета вхождений ключа в тексте.  
• **BoyerMoore.search(text)** – метод класса BoyerMoore, реализует алгоритм Бойера-Мура для поиска вхождений ключа, используя эвристики плохого символа и хорошего суффикса.  
• **KMP.search(text)** – метод класса KMP, реализует алгоритм Кнута-Морриса-Пратта, применяя предварительно вычисленный массив наибольших суффиксов-префиксов (LPS) для эффективного поиска.  
• Основная часть скрипта выполняет итерации по различным длинам текста и множеству повторений, замеряя время выполнения каждого алгоритма поиска и усредняя результаты.  
• Программа сохраняет усредненные значения времени выполнения для каждого алгоритма и длины текста в файл Excel.  
• В заключение, программа выводит в консоль усредненные значения времени выполнения для каждого алгоритма и длины текста.

**Дополнительные компоненты и технологии:**

• **random.choices()** – используется для генерации случайных строк и ключей.  
• **string.ascii\_letters + string.digits + ' '** – задает набор символов (буквы, цифры и пробел), используемый при генерации случайных строк.  
• **time.time()** – применяется для точного измерения времени выполнения алгоритмов.  
• **pandas.DataFrame** – библиотека для создания и манипуляции табличными данными, используемая для хранения и экспорта результатов в Excel.  
• **collections.defaultdict** – используется для хранения накапливаемых значений времени выполнения для каждого алгоритма и длины текста.

**Результаты работы**



****Рис. 1.** Графическое сравнение последовательного поиска, алгоритма Бойера-Мура, алгоритма Кнута-Морриса-Пратта**

**Синим цветом показана работа наивного поиска, красным — алгоритма Бойера-Мура, а жёлтым — Кнута-Морриса-Прата.**

◦ Последовательный алгоритм продемонстрировал предсказуемую линейную зависимость времени выполнения от длины строки. Несмотря на простоту, его производительность оказалась значительно ниже по сравнению с другими методами.

◦ Алгоритм Бойера-Мура показал наилучшие результаты по скорости поиска благодаря возможности пропуска частей строки при сравнении. Его преимущество особенно заметно при длинных строках.

**◦ Алгоритм Кнута-Морриса-Пратта продемонстрировал устойчивую производительность, опережая последовательный метод, но уступая алгоритму Бойера-Мура. Подготовка вспомогательного массива (lps) снижает количество сравнений, но на случайных строках его эффективность не достигает максимума.**

**Вывод по работе**

**В ходе лабораторной работы:**

1. **Была разработана программа** на Python, реализующая три алгоритма поиска подстроки:
   * Последовательный поиск
   * Алгоритм Бойера-Мура
   * Алгоритм Кнута-Морриса-Пратта
2. **Были освоены** ключевые аспекты работы алгоритмов:
   * Принцип полного перебора (последовательный поиск)
   * Эвристики плохого символа и хорошего суффикса (Бойер-Мур)
   * Префикс-функция и её построение (КМП)
3. **Было проведено тестирование** производительности на текстах разной длины (1000-25000 символов) с 10000 итераций для каждого алгоритма.
4. **Были сделаны выводы** о практической применимости методов:
   * Бойер-Мур показал наилучшую скорость на больших текстах
   * КМП эффективен при наличии повторяющихся паттернов
   * Последовательный поиск актуален только для коротких строк

Таким образом, в ходе работы были успешно реализованы, исследованы и сравнены различные методы поиска подстрок, что позволило на практике закрепить понимание их работы и особенностей применения.

****Приложение 1****

****Файл «main.py»:****

import random

import string

import time

import pandas as pd

from collections import defaultdict

def generate\_key(length=15):

return ''.join(random.choices(string.ascii\_letters + string.digits, k=length))

def generate\_text(length, key):

text = ''.join(random.choices(string.ascii\_letters + string.digits + ' ', k=length))

num\_insertions = random.randint(1, 100)

positions = random.sample(range(length), num\_insertions)

for pos in positions:

text = text[:pos] + key + text[pos:]

return text

def sequential\_search(text, key):

count = 0

n = len(text)

m = len(key)

for i in range(n - m + 1):

match = True

for j in range(m):

if text[i + j] != key[j]:

match = False

break

if match:

count += 1

return count

class BoyerMoore:

def \_\_init\_\_(self, pattern):

self.pattern = pattern

self.bad\_char = self.bad\_character\_table(pattern)

self.good\_suffix = self.good\_suffix\_table(pattern)

def bad\_character\_table(self, pattern):

table = {}

for i in range(len(pattern) - 1):

table[pattern[i]] = len(pattern) - 1 - i

return table

def good\_suffix\_table(self, pattern):

m = len(pattern)

suffix\_table = [0] \* m

last\_prefix\_pos = m

for i in range(m - 1, -1, -1):

if self.is\_prefix(pattern, i + 1):

last\_prefix\_pos = i + 1

suffix\_table[m - 1 - i] = last\_prefix\_pos - i

prefix\_table = [0] \* m

for i in range(m):

prefix\_table[i] = last\_prefix\_pos

for i in range(m - 1):

slen = self.suffix\_length(pattern, i)

prefix\_table[slen] = m - 1 - i

return prefix\_table

def is\_prefix(self, pattern, p):

m = len(pattern)

for i in range(p):

if pattern[i] != pattern[m - p + i]:

return False

return True

def suffix\_length(self, pattern, i):

length = 0

m = len(pattern)

for j in range(i, -1, -1):

if pattern[j] == pattern[m - 1 - i + j]:

length += 1

else:

break

return length

def search(self, text):

m, n = len(self.pattern), len(text)

if m > n:

return 0

bad\_char = self.bad\_char

good\_suffix = self.good\_suffix

count = 0

i = 0

while i <= n - m:

j = m - 1

while j >= 0 and self.pattern[j] == text[i + j]:

j -= 1

if j < 0:

count += 1

i += good\_suffix[0] if good\_suffix[0] != 0 else 1

else:

char = text[i + j]

bad\_char\_shift = bad\_char.get(char, m) - (m - 1 - j) if char in bad\_char else m

good\_suffix\_shift = good\_suffix[j + 1] if j + 1 < m else 1

i += max(bad\_char\_shift, good\_suffix\_shift)

return count

class KMP:

def \_\_init\_\_(self, pattern):

self.pattern = pattern

self.lps = self.compute\_lps(pattern)

def compute\_lps(self, pattern):

m = len(pattern)

lps = [0] \* m

length = 0

i = 1

while i < m:

if pattern[i] == pattern[length]:

length += 1

lps[i] = length

i += 1

else:

if length != 0:

length = lps[length - 1]

else:

lps[i] = 0

i += 1

return lps

def search(self, text):

m, n = len(self.pattern), len(text)

lps = self.lps

i = j = 0

count = 0

while i < n:

if self.pattern[j] == text[i]:

i += 1

j += 1

if j == m:

count += 1

j = lps[j - 1]

elif i < n and self.pattern[j] != text[i]:

if j != 0:

j = lps[j - 1]

else:

i += 1

return count

num\_iterations = 10000

text\_lengths = list(range(1000, 25001, 1000))

results = defaultdict(lambda: {'sequential': 0, 'boyer\_moore': 0, 'kmp': 0})

for \_ in range(num\_iterations):

key = generate\_key()

for length in text\_lengths:

text = generate\_text(length, key)

start = time.time()

sequential\_search(text, key)

time\_sequential = time.time() - start

bm = BoyerMoore(key)

start = time.time()

bm.search(text)

time\_bm = time.time() - start

kmp = KMP(key)

start = time.time()

kmp.search(text)

time\_kmp = time.time() - start

results[length]['sequential'] += time\_sequential

results[length]['boyer\_moore'] += time\_bm

results[length]['kmp'] += time\_kmp

data = {

"Длина строки": [],

"Последовательный алгоритм": [],

"Бойер-Мур": [],

"Кнут-Моррис-Пратт": []

}

for length in text\_lengths:

data["Длина строки"].append(length)

data["Последовательный алгоритм"].append(round(results[length]['sequential'] / num\_iterations, 6))

data["Бойер-Мур"].append(round(results[length]['boyer\_moore'] / num\_iterations, 6))

data["Кнут-Моррис-Пратт"].append(round(results[length]['kmp'] / num\_iterations, 6))

df = pd.DataFrame(data)

df.to\_excel("search\_results.xlsx", index=False, float\_format="%.6f")

for length in text\_lengths:

print(f"Длина строки: {length}")

print(f" Последоватльный алгоритм: {results[length]['sequential'] / num\_iterations:.6f} сек")

print(f" Бойер-Мур: {results[length]['boyer\_moore'] / num\_iterations:.6f} сек")

print(f" Кнут-Моррис-Пратт: {results[length]['kmp'] / num\_iterations:.6f} сек")

print("-")