**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

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

**(СПбГУТ)**

**Факультет инфокоммуникационных Сетей и систем (иксс)**

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

**Лабораторная работа №2.**

**«Методы поиска»**

Дисциплина: «Алгоритмы и Структуры Данных»

Выполнили: Студент группы ИКПИ-33  
 Коньков Максим

Семенихин Арсений

Принял: к.т.н., ПииВТ Дагаев А.В.

## **1. Описание алгоритмов**

### **1.1 Наивный поиск**

Наивный алгоритм поиска подстроки заключается в последовательном переборе всех возможных позиций в строке, начиная с которых можно сравнить подстроку. На каждом шаге алгоритм сравнивает подстроку с соответствующей частью строки. Если совпадение найдено, возвращается индекс начала подстроки. Сложность алгоритма в худшем случае составляет O(n\*m), где n – длина строки, m – длина подстроки.

### **1.2 Алгоритм Кнута-Морриса-Пратта (KMP)**

Алгоритм KMP использует предварительную обработку подстроки, создавая массив наибольших префиксов, которые также являются суффиксами (LPS - Longest Prefix Suffix). Это позволяет избегать повторных сравнений, значительно ускоряя процесс поиска по сравнению с наивным методом. Сложность алгоритма O(n + m).

### **1.3 Алгоритм Рабина-Карпа**

Рабин-Карп использует хеш-функцию для быстрого сравнения подстроки с частями строки. Сначала вычисляется хеш подстроки, затем вычисляются хеши всех подстрок строки соответствующей длины. Если хеши совпадают, выполняется дополнительная проверка на точное совпадение. Средняя сложность O(n + m), в худшем случае O(n\*m).

## **2. Сравнение алгоритмов**

**Исходные данные:**

* Текст: T = "ABABDABACDABABCABAB"
* Подстрока: P = "ABABCABAB"

### **1. Наивный алгоритм (Brute-Force)**

**Принцип:** Последовательное сравнение всех возможных позиций.

**Пошаговая работа:**

1. **Старт:**
   * Сравниваем T[0:8] ("ABABDABA") и P[0:8] → не совпадает.
   * Сдвигаемся на 1 символ.
2. **Шаги 1–6:**
   * Последовательно сравниваем подстроки T[1:9], T[2:10], ..., T[6:14] с P.
   * Совпадений нет.
3. **Шаг 7:**
   * T[10:18] ("ABABCABAB") = P → **найдено на позиции 10**.

**Результат:**

* Число шагов: 11
* Сравнений символов: 35

### **2. Алгоритм Кнута-Морриса-Пратта (KMP)**

**Принцип:** Использование префикс-функции для оптимизации сдвигов.

**Префикс-функция (LPS) для** P**:**

| Индекс | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Символ | A | B | A | B | C | A | B | A | B |
| LPS | 0 | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 4 |

**Пошаговая работа:**

1. **Совпадение:**
   * Первые 4 символа совпадают ("ABAB").
   * При несовпадении T[4]='D' и P[4]='C' → сдвиг на LPS[3]=2.
2. **Продолжение:**
   * Повторная проверка с j=2.
   * Полное совпадение на позиции i=10, j=8.

**Результат:**

* Число шагов: 13
* Сравнений символов: 18

### **3. Алгоритм Рабина-Карпа (РП)**

**Принцип:** Хеширование подстрок для быстрого сравнения.

**Параметры:**

* Простое число q = 101
* Хеш P: 42

**Пошаговая работа:**

1. **Вычисление хешей:**
   * T[0:8] → хеш 38 ≠ 42 → сдвиг.
   * T[1:9] → хеш 31 → сдвиг.
   * ...
   * T[10:18] → хеш 42 → проверка посимвольно → **совпадение**.

**Результат:**

* Число шагов: 11
* Полных сравнений подстрок: 2

## 3. Пошаговое решение программы

### 3.1 Генерация данных:

1. Вызывается generate\_string\_and\_pattern(n, m), которая создаёт случайную строку длины n и подстроку длины m из символов A, C, G, T.
2. Строки передаются алгоритмам поиска.

### 3.2 Запуск алгоритмов:

1. Вызывается measure\_time(search\_func, text, pattern) для каждого алгоритма:
   * Засекается время перед вызовом функции.
   * Запускается алгоритм поиска.
   * Засекается время после выполнения.
   * Рассчитывается среднее время выполнения за 10 000 итераций.

### 3.3 Сохранение результатов:

1. Результаты записываются в список results.
2. Вызывается save\_to\_csv(results), которая создаёт CSV-файл с колонками Length, Naive Search, KMP Search, Rabin-Karp Search.

### 3.4 Вывод информации:

1. В консоли отображается длина строки и время выполнения каждого алгоритма.
2. По завершении программа сообщает о сохранении результатов.

## ****4. Описание программы:****

* **Язык:** Python
* **Среда разработки:** Vim в ОС Linux
* **Структура кода:**

### 4.1 Функции:

* naive\_search(text, pattern): Простой перебор подстроки.
* kmp\_search(text, pattern): Использует префикс-функцию.
* rabin\_karp\_search(text, pattern): Применяет хеширование.
* generate\_string\_and\_pattern(n, m): Генерирует случайные строки.
* measure\_time(search\_func, text, pattern): Засекает среднее время выполнения.
* save\_to\_csv(data): Записывает результаты в файл.

### 4.2 Основной процесс (main())

1. Перебираются длины строк от 1000 до 3000.
2. Генерируются text и pattern.
3. Запускается тестирование всех алгоритмов.
4. Результаты сохраняются в CSV и выводятся в консоль.
5. Программа завершает работу.
6. Генерируем text и pattern.
7. Запускаем каждый алгоритм и измеряем время работы.
8. Сохраняем результаты.
9. Выводим данные в консоль и CSV.

**5. Результаты работы**

В ходе выполнения поставленной задачи были получены следующие результаты:

**Рис. 1. Сравнение временных затрат на поиски**: Наивного (Naive Search), Кнута-Морриса-Пратта (KMP Search), Рабин-карпа (Rabin-Karp Search).

На приведённом графике жёлтым цветом обозначена поиск алгоритмом Рабина-Карпа, оранжевым – КМП-поиск а синим – наивный поиск. **Наивный алгоритм** подходит для небольших текстов и единичных поисков. **KMP** эффективен при длинных строках и частом повторении шаблонов. **Рабин-Карп** удобен, если необходимо искать несколько разных подстрок за один проход.

**5. Вывод:**

На основании полученных данных можно сделать следующие выводы:

1. **Наивный поиск** – самый быстрый при небольших длинах строки, но его скорость увеличивается линейно с размером строки, что подтверждает его сложность O(n\*m). Однако он остаётся конкурентоспособным при небольших входных данных.
2. **KMP-поиск** – показывает стабильное время выполнения, увеличиваясь медленнее, чем наивный метод. Это объясняется использованием префикс-функции, которая позволяет избежать лишних сравнений. Однако начальная обработка шаблона делает его чуть медленнее на коротких строках.
3. **Рабин-Карп** – самый медленный алгоритм в среднем, что обусловлено пересчётом хешей и дополнительными проверками совпадений. Однако он может быть полезен при множественном поиске подстрок, где хеширование даёт преимущество.

**Приложение А: Листинг кода**

|  |
| --- |
| import random  import time  import csv  # Наивный поиск  def naive\_search(text, pattern):  n, m = len(text), len(pattern)  for i in range(n - m + 1):  if text[i:i + m] == pattern:  return i  return -1  # Алгоритм Кнута-Морриса-Пратта (KMP)  def kmp\_search(text, pattern):  def compute\_lps(pattern):  lps = [0] \* len(pattern)  length = 0  for i in range(1, len(pattern)):  while length > 0 and pattern[i] != pattern[length]:  length = lps[length - 1]  if pattern[i] == pattern[length]:  length += 1  lps[i] = length  return lps  lps = compute\_lps(pattern)  i, j = 0, 0  while i < len(text):  if text[i] == pattern[j]:  i += 1  j += 1  if j == len(pattern):  return i - j  else:  if j > 0:  j = lps[j - 1]  else:  i += 1  return -1  # Алгоритм Рабина-Карпа  def rabin\_karp\_search(text, pattern):  d = 256 # Размер алфавита  q = 101 # Простое число для хеширования  n, m = len(text), len(pattern)  h = pow(d, m - 1, q)  p\_hash, t\_hash = 0, 0  for i in range(m):  p\_hash = (d \* p\_hash + ord(pattern[i])) % q  t\_hash = (d \* t\_hash + ord(text[i])) % q  for i in range(n - m + 1):  if p\_hash == t\_hash:  if text[i:i + m] == pattern:  return i  if i < n - m:  t\_hash = (d \* (t\_hash - ord(text[i]) \* h) + ord(text[i + m])) % q  if t\_hash < 0:  t\_hash += q  return -1  # Генерация строки и подстроки  def generate\_string\_and\_pattern(n, m):  text = ''.join(random.choices('ACGT', k=n)) # Генерация строки из символов A, C, G, T  pattern = ''.join(random.choices('ACGT', k=m)) # Генерация подстроки  return text, pattern  # Функция для измерения времени выполнения  def measure\_time(search\_func, text, pattern, iterations=10\_000):  total\_time = 0  for \_ in range(iterations):  start\_time = time.time()  search\_func(text, pattern)  end\_time = time.time()  total\_time += (end\_time - start\_time)  return int((total\_time / iterations) \* 1\_000\_000) # Среднее время в микросекундах  # Сохранение результатов в CSV  def save\_to\_csv(data, filename="search\_results.csv"):  with open(filename, mode='w', newline='') as file:  writer = csv.writer(file)  writer.writerow(["Length", "Naive Search", "KMP Search", "Rabin-Karp Search"])  for row in data:  writer.writerow(row)  # Основная функция  def main():  results = []  for n in range(1000, 3001, 100): # Длина строки от 1000 до 3000 с шагом 100  m = 20 # Длина подстроки  text, pattern = generate\_string\_and\_pattern(n, m)  # Измерение времени для каждого алгоритма  naive\_time = measure\_time(naive\_search, text, pattern)  kmp\_time = measure\_time(kmp\_search, text, pattern)  rk\_time = measure\_time(rabin\_karp\_search, text, pattern)  # Сохранение результатов  results.append([n, naive\_time, kmp\_time, rk\_time])  print(f"Length: {n}, Naive: {naive\_time}, KMP: {kmp\_time}, Rabin-Karp: {rk\_time}")  # Сохранение в CSV  save\_to\_csv(results)  if \_\_name\_\_ == "\_\_main\_\_":  main() |