|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 2.4** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Поиск образца в тексте»** | |
|  | |
| Выполнил студент группы ИКБО-30-22 | Сенькевич Г.Д. |
| Принял преподаватель | Красников С.А. |

Москва 2023

# **Цель работы**

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

1. **Ход работы**

# **. Формулировка задачи**

Для каждой задачи варианта:

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 большие и небольшие по объему текст и образец и

включить ее в этап тестирование.

4. Разработать программу

5. Оценить практическую сложность алгоритма в зависимости от длины

текста и длины образца и отобразить результаты в таблицу (для

отчета).

6. Сравнить время поиска различных методов на одних и тех же наборах

данных.

Вариант №5:

1) Дано предложение, состоящее из слов, разделенных одним пробелом, удалить из него слова, встретившиеся более одного раза.

2) Дано предложение, состоящее из слов, разделенных одним пробелом. Удалить из предложения все вхождения заданного слова, применяя для поиска слова в тексте метод Кнута-Мориса-Пратта.

Вариант №27. Условие задания:

|  |  |
| --- | --- |
| Тип значения узла | Символ |
| Тип дерева | Бинарное дерево поиска |
| Вставка элемента | + |
| Обратный обход | + |
| Симметричный обход | + |
| Найти сумму значений листьев | + |
| Найти высоту дерева | + |

* 1. **. Описание модели (подход к решению)**

Алгоритм Кнута-Морриса-Пратта (KMP) — это эффективный алгоритм для поиска подстроки в строке. Он был разработан Дональдом Кнутом, Джеймсом Моррисом и Вацлавом Праттом. Основная идея алгоритма - избегать многократных сравнений символов, которые уже были сравнены.

Теоретическое обоснование:

1. Подготовка префикс-функции (также известной как "префикс-таблица") для образца (ищем подстроку) позволяет нам определить, как далеко можно "перепрыгнуть" при несовпадении символов, так как мы уже знаем некоторую информацию о совпавших символах.

2. Главная идея заключается в том, что когда мы сравниваем символы образца и текста, и обнаруживаем несовпадение, мы можем использовать информацию из префикс-таблицы, чтобы определить, сколько символов можно пропустить при смещении образца.

3. Префикс-таблица строится для образца, и она представляет собой массив, в котором каждому элементу соответствует длина максимального суффикса образца (начиная с начала), который также является его префиксом.

4. Время работы KMP алгоритма составляет O(m + n), где m - длина текста, а n - длина образца. Это существенное улучшение по сравнению с наивным алгоритмом, который работает за O(m \* n).

Пример работы алгоритма KMP:

Пусть у нас есть строка текста "ABCABCDABABCDABCDABDE" и мы ищем образец "ABCDABD". Мы сначала создаем префикс-таблицу для образца:

Образец: ABCDABD

Префикс-таблица: [0, 0, 0, 0, 1, 2, 0]

Теперь мы начинаем сравнивать образец с текстом. Мы сравниваем символы слева направо и, при несовпадении, используем информацию из префикс-таблицы, чтобы определить, на сколько символов мы можем сдвинуть образец.

1. Сравниваем "A" из образца и "A" из текста - совпадение.

2. Сравниваем "B" из образца и "B" из текста - совпадение.

3. Сравниваем "C" из образца и "C" из текста - совпадение.

4. Сравниваем "D" из образца и "A" из текста – несовпадение. (Здесь мы используем префикс-таблицу: сдвигаем образец на 1 символ влево.)

5. Сравниваем "B" из образца и "B" из текста - совпадение.

6. Сравниваем "D" из образца и "A" из текста - несовпадение. (Здесь мы снова используем префикс-таблицу: сдвигаем образец на 2 символа влево.)

7. Сравниваем "B" из образца и "B" из текста - совпадение.

Таким образом, мы нашли первое вхождение образца "ABCDABD" в тексте "ABCABCDABABCDABCDABDE". Этот алгоритм позволяет нам сдвигать образец на минимально возможное количество символов при несовпадении, что делает его эффективным для поиска подстрок в тексте.

* 1. **. Код программы**

Для решения поставленных задач были написаны следующие функции:

* pre\_kmp – функция, строящая префикс-таблицу для алгоритма Кнута-Морриса-Пратта;
* kmp – функция, реализующая алгоритм Кнута-Морриса-Пратта;
* simple\_search – функция, реализующая «наивный» поиск (то есть алгоритм, в котором образец последовательно подставляется на все возможные позиции в строке и проверяется на совпадение с этой частью строки);
* remove\_duplicates – функция, реализующая решение 1 задания варианта (удаляет дубликаты в тексте);
* remove\_word – функция, реализующая решение 2 задания варианта (удаляет все вхождения заданного слова из текста);
* random\_string – функция, генерирующая строку заданной длины, состоящую из случайных символов;
* run\_tests – функция, тестирующая КМП и «наивный» алгоритм на строках и образцах различной длины (сгенерированных функцией random\_string) и выводящая результат тестирования на экран;
* main – основная функция программы, реализует текстовый интерфейс пользователя, позволяющий выбрать между функциями, описанными выше.

Листинг 1 ­— Код программы.

|  |
| --- |
| #include <iostream>  #include <string>  #include <vector>  #include <random>  #include <ctime>  #include <chrono>  // Строит префикс-таблицу для КМП  std::vector<int> pre\_kmp(std::string pattern)  {  std::vector<int> pie(pattern.length(), 0);  int k = 0;  for (int i = 1; i < pattern.length(); i++)  {  while (k > 0 && pattern[k] != pattern[i])  {  k = pie[k - 1];  }  if (pattern[k] == pattern[i])  {  k++;  }  pie[i] = k;  }    return pie;  }  // Реализует КМП (алгоритм Кнута-Морриса-Пратта)  std::vector<int> kmp(std::string text, std::string pattern, int& comps)  {  std::vector<int> res;  std::vector<int> pie = pre\_kmp(pattern);  int k = 0;  for (int i = 0; i < text.length(); i++)  {  while (k > 0 && pattern[k] != text[i])  {  comps++;  k = pie[k - 1];  }  comps++;  if (pattern[k] == text[i])  {  k++;  }  if (k == pattern.length())  {  res.push\_back(i - pattern.length() + 1);  k = pie[k - 1];  }  }  return res;  }  // "Наивный" поиск, то есть простой перебор всех вариантов  std::vector<int> simple\_search(std::string text, std::string pattern, int& comps)  {  std::vector<int> res;  for (int i = 0; i < text.length() - pattern.length() + 1; i++)  {  bool fits = true;  for (int j = 0; j < pattern.length(); j++)  {  comps++;  if (pattern[j] != text[i + j])  {  fits = false;  break;  }  }  if (fits)  {  res.push\_back(i);  }  }  return res;  }  // Решение 1 задания, удаляет слова-дубликаты из текста  std::string remove\_duplicates(std::string text, int& comps)  {  std::string res;  std::string current\_word;  text.push\_back(' ');  for (int i = 0; i < text.length(); i++)  {  current\_word.push\_back(text[i]);  if (text[i] == ' ')  {  std::vector<int> search\_res = kmp(res, current\_word, comps);  if (search\_res.size() == 0)  {  res += current\_word;  }  current\_word.clear();  }  }  res.pop\_back();  return res;  }  // Решение 2 задания, удаляет все вхождения заданного слова  std::string remove\_word(std::string text, std::string word, int& comps)  {  text.push\_back(' ');  word.push\_back(' ');  std::vector<int> search\_res = kmp(text, word, comps);  for (int i = 0; i < search\_res.size(); i++)  {  int begin\_ind = search\_res[i] - i \* word.length();  int end\_ind = begin\_ind + word.length();  text.erase(text.begin() + begin\_ind, text.begin() + end\_ind);  }  text.pop\_back();  return text;  }  // Генерирует строку из случайных символов заданной длины  std::string random\_string(int length) {  const char charset[] =  "0123456789"  "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  "abcdefghijklmnopqrstuvwxyz";  std::string res;  for (int i = 0; i < length; i++)  {  res += charset[std::rand() % (sizeof(charset) - 1)];  }    return res;  }  // Тестирует производительность КМП и "наивного" алгоритма  // на строках и образцах различной длины,  // генерируемых функцией random\_string  void run\_tests()  {  const int min\_text\_size = 1000;  const int max\_text\_size = 1000000;  const int text\_size\_mult = 10;  const int runs\_per\_size = 5;  std::cout << "TEXT\tPATTERN\tFOUND\tCOMPS\tTIME\tCOMPS\tTIME\n";  for (int text\_size = min\_text\_size; text\_size <= max\_text\_size; text\_size \*= text\_size\_mult)  {  int pattern\_size\_step = text\_size / runs\_per\_size;  for (int pattern\_size = pattern\_size\_step;  pattern\_size <= text\_size;  pattern\_size += pattern\_size\_step)  {  std::string text = random\_string(text\_size);  std::string pattern;  int comps = 0;  int pattern\_start\_ind = std::rand() % (text\_size - pattern\_size + 1);  for (int i = 0; i < pattern\_size; i++)  {  pattern += text[pattern\_start\_ind + i];  }  // Запуск КМП на образце, который взят из строки  auto start = std::chrono::high\_resolution\_clock().now();  std::vector<int> kmp\_res = kmp(text, pattern, comps);  auto stop = std::chrono::high\_resolution\_clock().now();  auto kmp\_duration = std::chrono::duration\_cast<std::chrono::milliseconds>(stop - start).count();  int kmp\_comps = comps;    // Запуск "наивного" поиска на образце, который взят из строки  comps = 0;  start = std::chrono::high\_resolution\_clock().now();  std::vector<int> simple\_res = simple\_search(text, pattern, comps);  stop = std::chrono::high\_resolution\_clock().now();  auto simple\_duration = std::chrono::duration\_cast<std::chrono::milliseconds>(stop - start).count();  int simple\_comps = comps;  // Вывод результатов  std::cout << text\_size << '\t' << pattern\_size << '\t' << (int)(kmp\_res.size() != 0)  << '\t' << kmp\_comps << '\t' << kmp\_duration  << '\t' << simple\_comps << '\t' << simple\_duration << '\n';  // Образец - случайная строка (скроее всего не будет найдена)  pattern = random\_string(pattern\_size);  comps = 0;  // Запуск КМП на случайном образце  start = std::chrono::high\_resolution\_clock().now();  kmp\_res = kmp(text, pattern, comps);  stop = std::chrono::high\_resolution\_clock().now();  kmp\_duration = std::chrono::duration\_cast<std::chrono::milliseconds>(stop - start).count();  kmp\_comps = comps;  // Запуск "наивного" поиска на случайном образце  comps = 0;  start = std::chrono::high\_resolution\_clock().now();  simple\_res = simple\_search(text, pattern, comps);  stop = std::chrono::high\_resolution\_clock().now();  simple\_duration = std::chrono::duration\_cast<std::chrono::milliseconds>(stop - start).count();  simple\_comps = comps;  // Вывод результатов  std::cout << text\_size << '\t' << pattern\_size << '\t' << (int)(kmp\_res.size() != 0)  << '\t' << kmp\_comps << '\t' << kmp\_duration  << '\t' << simple\_comps << '\t' << simple\_duration << '\n';  }  }  }  // Основная функция программы  int main()  {  std::srand(time(NULL));  // Текстовый интерфейс пользователя  while (true)  {  // Меню  std::cout << "1 - Удалить из предложения все слова, встретившиеся более одного раза\n"  << "2 - Удалить из предложения все вхождения заданного слова\n"  << "3 - Запустить тестирование\n"  << "4 - Выход\n";  int choice = 0;  std::cin >> choice;  std::cin.get();  if (choice == 1) // Удалить из предложения все слова, встретившиеся более одного раза  {  std::cout << "Введите предложение: ";  std::string text;  std::getline(std::cin, text);  int comps = 0;  std::cout << "Результат: " << remove\_duplicates(text, comps) << '\n'  << "Количество сравнений: " << comps << "\n\n";  }  else if (choice == 2) // Удалить из предложения все вхождения заданного слова  {  std::string text, word;  std::cout << "Введите предложение: ";  std::getline(std::cin, text);  std::cout << "Введите слово: ";  std::getline(std::cin, word);  int comps = 0;  std::cout << "Результат: " << remove\_word(text, word, comps) << '\n'  << "Количество сравнений: " << comps << "\n\n";  }  else if (choice == 3) // Запустить тестирование  {  run\_tests();  }  else if (choice == 4) // Выход  {  break;  }  else  {  std::cout << "Некорректный ввод!\n\n";  }  }  return 0;  } |

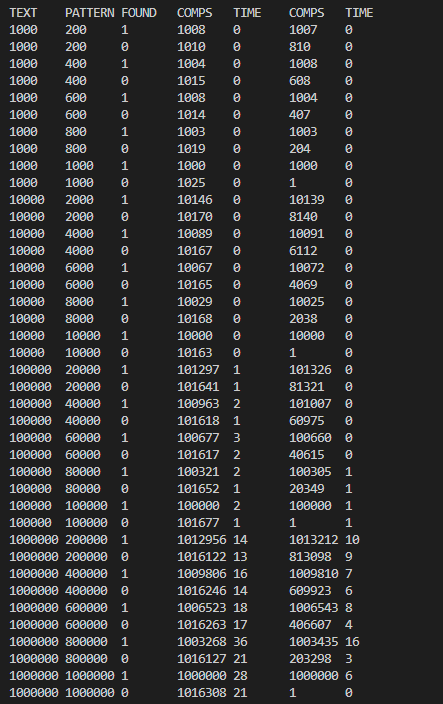
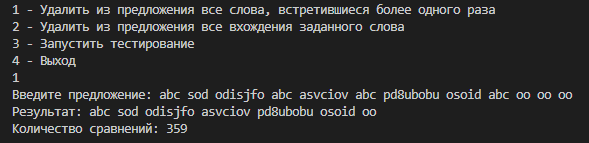
* 1. **. Результаты тестирования**

Рисунок 1 – Тестирование КМП и «наивного» алгоритма.

На рисунке 1 в первой колонке указана длина строки, во второй – длина образца, третьей – успешность поиска, в четвёртой – количество сравнений КМП, в пятой – время, затраченное КМП (в миллисекундах), в шестой – количество сравнений «наивного» алгоритма, в седьмой – время, затраченное «наивным» алгоритмом (в миллисекундах).

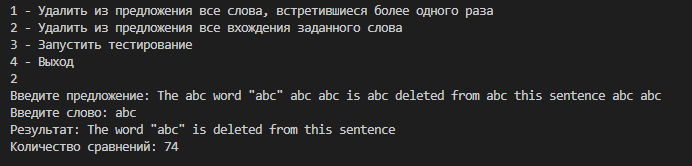
Рисунок 2 – Тестирование 1 задания.

Рисунок 3 – Тестирование 2 задания.

Тестирование показало, что программа работает правильно, корректно решая поставленные задачи.

1. **Выводы**

В результате выполнения работы я:

1. Познакомился с различными алгоритмами поиска подстроки в строке;
2. Реализовал наивный алгоритм поиска и алгоритм Кнута-Морриса-Пратта на C++.
3. **Список литературы**
4. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.
5. Документация по языку С++ [Электронный ресурс]. URL: https://docs.microsoft.com/ruru/cpp/cpp/ (дата обращения 01.09.2021).
6. Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2021).