**Поиск подстроки в строке – отчёт**

Романенко Михаил, ФТ-203

**Задача:**

Реализовать несколько алгоритмов поиска подстроки в строке, сравнить их эффективность по времени выполнения и используемой памяти. Написать отчёт о выполненной работе с подведением итогов. Обосновать итог.

**Параметры вычислительного узла:**

1. ЦП: AMD Ryzen 7 5800H
2. ОЗУ: 16 Гб
3. ОС: Windows 10 Home x64

**Используемые алгоритмы:**

1. Алгоритм поиска грубой силы:

Мы идём по проверяемой строке А и ищем в ней вхождение первого символа искомой строки Х. Когда находим, делаем гипотезу, что это и есть то самое искомое вхождение. Затем остается проверять по очереди все последующие символы шаблона на совпадение с соответствующими символами строки A. Если они все совпали — значит вот оно, прямо перед нами. Но вот если какой-то из символов не совпал, то ничего не остается, как признать нашу гипотезу неверной, что возвращает нас к символу, следующему за вхождением первого символа из X.

Сложность – O (m \* n), где m – количество символов искомой строки, n – количество символов данной строке

1. Алгоритм Кнута-Мориса-Пратта (стандартный и продвинутый)

Формируем массив (префикс-функцию, это длина наибольшего префикса строки S[1..i], который не совпадает с этой строкой и одновременно является ее суффиксом. Проще говоря, это длина наиболее длинного начала строки, являющегося также и ее концом.), который будет использоваться при сдвиге образа вдоль строки. Как и в примитивном алгоритме поиска подстроки, образец «перемещается» по строке слева направо с целью обнаружения совпадения. Однако ключевым отличием является то, что при помощи префикс-функции мы можем избежать заведомо бесполезных сдвигов. Ищем образ в тексте, при несовпадении, сдвигаем весь образ на то количество символов, которое было вычислено в префикс-функции.

Сложность – O (m + n), где m – количество символов искомой строки, n – количество символов данной строке

Модифицированная версия отличается тем, что после нахождения первого вхождения искомой строки, алгоритм не прекращает свою работу, а продолжает до тех пор, пока не проверит всю данную строку и вернёт массив чисел, который будет содержать индексы всех вхождений искомой строки в данной.

1. Алгоритм Рабина-Карпа

Вычисляем хэш искомой строки. Вычисляем хэш всех подстрок данной строки, длиной искомой строки. Затем сравнивается хэш подстроки текста с хэшем искомой строки. Если они совпадают, то сравниваются отдельные символы для выявления точного совпадения двух строк, если они совпали, мы нашли ответ. Если же нет, переходим к следующему хэшу следующей подстроки данной строки и так до конца.

Сложность – O (m \* n), где m – количество символов искомой строки, n – количество символов в данной строке

1. Алгоритм Бойера-Мура

Основная идея алгоритм - начать поиск не с начала, а с конца подстроки. Наткнувшись на несовпадение, мы просто смещаем подстроку до самого правого вхождения данного символа.

Сложность – O (m + n), где m – количество символов искомой строки, n – количество символов в данной строке

1. Двусторонний алгоритм

Строка x разбивается на две части u и v так, что x=uv. Затем фаза поиска в двустороннем алгоритме состоит в сравнении символов v слева направо, и затем, если на первом этапе не происходит несовпадений, в сравнении символов u справа налево (второй этап). Фаза предобработки, таким образом, заключается в поиске подходящего разбиения (u,v).

Сложность – O (m + n), где m – количество символов искомой строки, n – количество символов в данной строке

**План тестирования:**

1. Подготовка к тестированию
2. Случайная генерация входных данных
3. Тестирование
4. Обработка результатов
5. Вывод

**Методика тестирования:**

1. **На время**

При тестирование на время используется специально написанная функция в файлу benchmark.py, которая с помощью библиотеки time вычисляется текущее время. Потом функция через вложенный цикл вычисляет значение для всех входных данных, после из текущего времени вычисляется стартовое время и результат записывается в текстовый документ, для дальнейшей обработки.

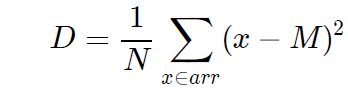
1. **На память**

Для тестирования на память используется метод asizeof. asizeof () из библиотеки pympler. Он выводит количество байт, используемых всеми объектами в тестируемой функции. Данные так же записываются в файл для дальнейшей обработки. Тестирование на память происходило отдельно, так как оно могло занимать дополнительное время.

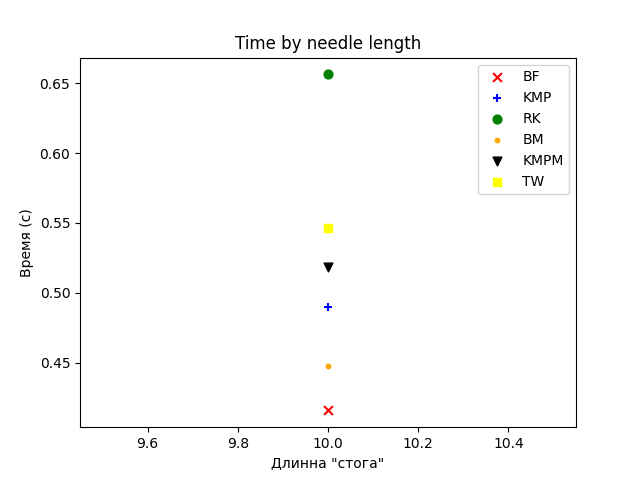
**Корректность тестирования:**

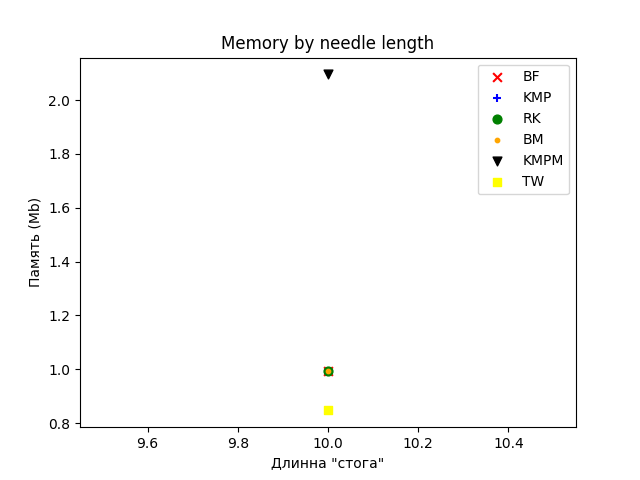
При выполнение самой функции поиска подстроки нет ничего лишнего, все методики тестирования проходят отдельно. Так же перед тестированием идёт «холостой» прогон функции. Время и используемая память считается для случайно сгенерированного массива данных разных размеров, а также для хороших, средних и плохих данных.

По полученным данным строились графики и высчитывалось квадратичное отклонение от среднего по формуле:

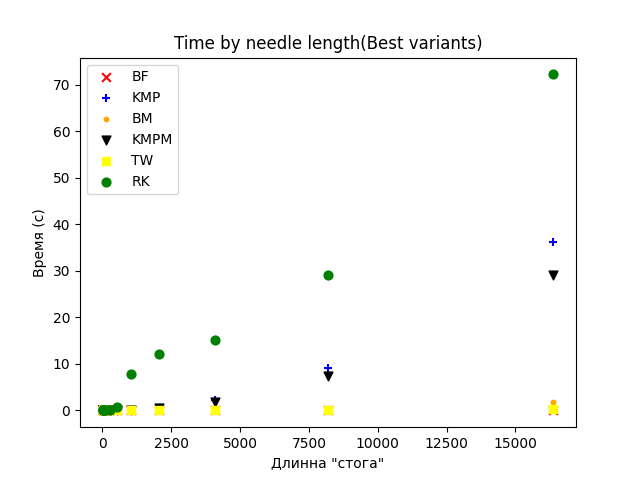


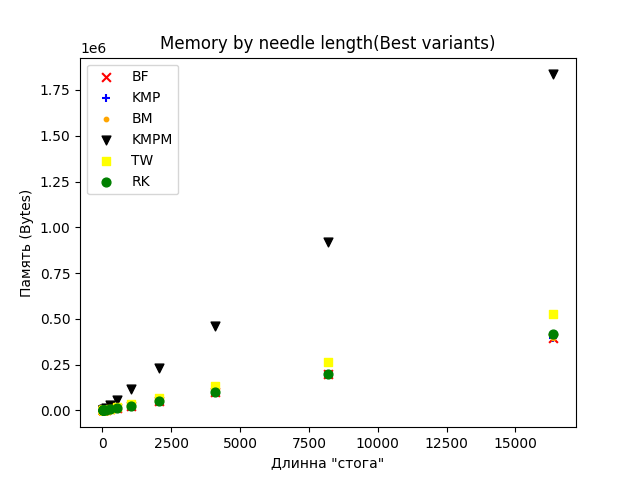
**Результат тестирования:**

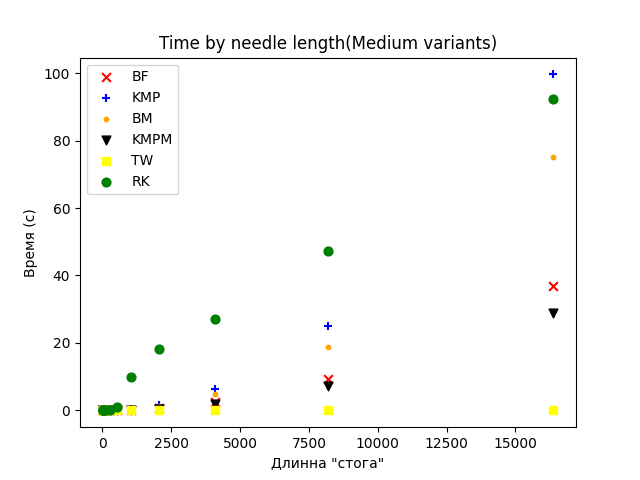
**Тестирование на случайном наборе данных:**

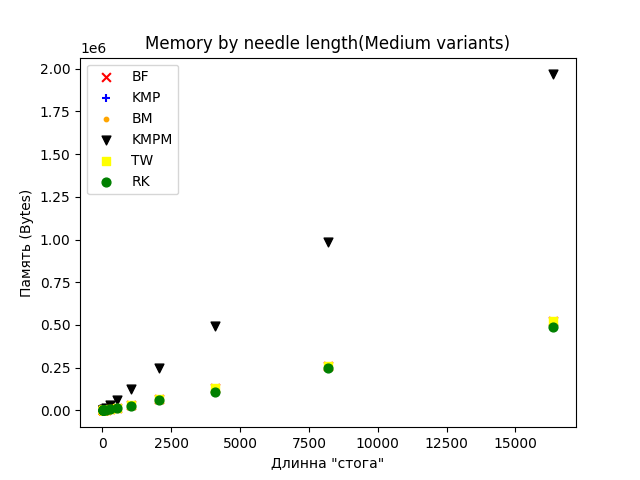
****

**Тестирование на лучшем наборе данных:**

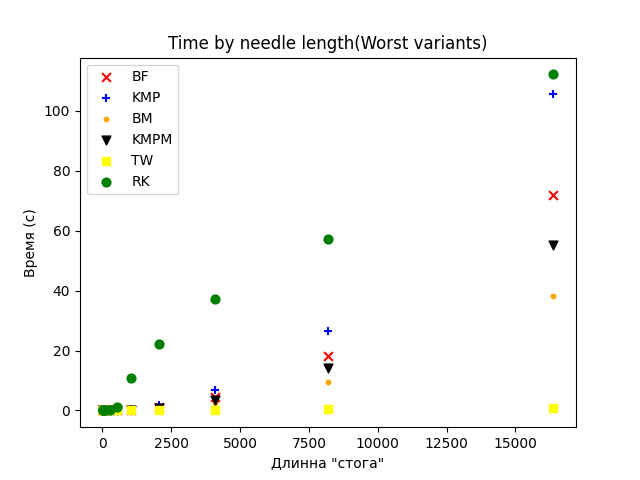
****

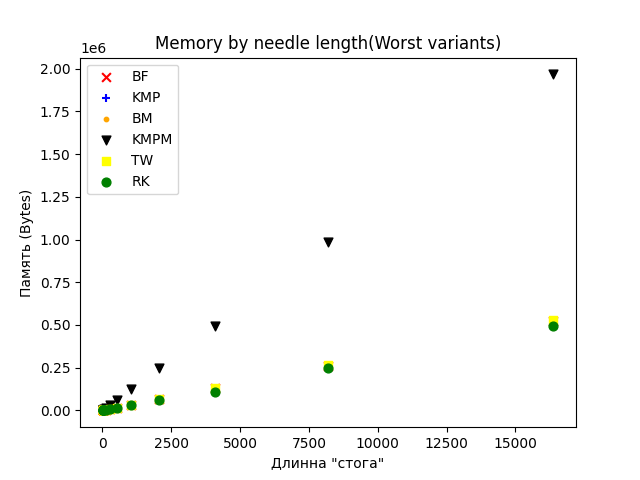
****

**Тестирование на среднем наборе данных: **

****

**Тестирование на худшем наборе данных:**

****

****

**Обоснование результата:**

Исследуя графики, можно сказать, что алгоритм Робина-Карпа оказался самым медленным, а двусторонний алгоритм оказался самым быстрым. Это связано с долгой предобработкой текста, такой как вычисления хэшей. По использованию памяти все алгоритмы равны, разница в большинстве случаев не превышает пары процентов, за исключением модифицированной версии алгоритма Кнута-Мориса-Пратта, так как она не только находит первое вхождение искомой строки, но также сохраняет все последующие вхождения.

**Вывод:**

Мы провели достаточно обширное исследование некоторых алгоритмов поиска подстроки в строке, объяснили полученные результаты, исходя из которых следуют следующие факты: если вам нужно найти все вхождения подстроки в строке, то для этого стоит использовать модифицированную версию алгоритма Кнута-Мориса-Пратта, так как этот алгоритм выдаёт не худшее время, при этом выполняя большую работу; при работе со случайными данными достаточно будет простого алгоритма грубой силы; двусторонний алгоритм пожалуй является лучшим алгоритмом, так как на хороших данных он работает примерно как и остальные алгоритмы, но на плохих выдаёт отличный результат, справляясь гораздо быстрее своих оппонентов.