Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**

КАФЕДРА **«ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

О Т Ч Е Т

по лабораторной работе № 2

Название: Трудоёмкость алгоритмов умножения матриц

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б (Группа) __________ (Подпись, дата) В.А. Иванов (И.О. Фамилия)

Преподаватель __________ (Подпись, дата) __________ (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

Трудоёмкость алгоритма - это зависимость стоимости операций от линейного размера входа[2].
Модель вычислений трудоёмкости учитывает следующие оценки:

- Оценка стоимости базовых операций. Операции $=$, $+$, - и т.д. имеют стоиость 1.

- Оценка циклов.

- Оценка условного оператора if.

Оценка характера трудоёмкости даётся по наиболее быстрорастущему слагаемому. Такая оценка играет важную роль в разработке и анализе алгоритмов, так как позволяет судить об оптимальности использования алгритма при тех или иных входных данных.

В данной лабораторной оценивается трудоёмкость классического алгоритма умножения матриц и алгоритма Винограда.

1. Аналитическая часть

Целью лабораторной работы является оценка трудоёмкости алгоритма умножения матриц и получение практического навыка оптимизации алгоритмов.

Выделены следующие задачи лабораторной работы:

- математическое описание операции умножения матриц;
- описание и реализация алгоритмов умножения матриц;
- описание применённых к алгоритму Винограда способов оптимизации;
- проведение замеров процессорного времени работы алгоритмов при различных размерах матриц (серия экспериментов для чётного размера и для нечётного);
- оценка трудоёмкости алгоритом;
- проведение сравнительного анализа алгоритмов на основании экспериментов.

Умножение матриц - операция над матрицами $[MxN]$ и $B[NxQ]$. Результатом операции является матрица С размерами $M * Q$, в которой элемент $c_{i,j}$ задаётся формулой

$$c_{i,j} = \sum_{k=1}^{N}(a_{i,k} \cdot b_{k,j}) \tag{1.1}$$

2. Конструкторская часть

Рассмотрим и произведём вычисление трудоёмкости для классического алгоритма и алгоритма Винограда для умножения матриц $[MxN]$ и $B[NxQ]$

2.1. Классический алгоритм умножения

Данный алгоритм непосредственно использует вышеприведённую формулу. Для вычисления каждого элемента матрицы С совершается циклический обход k элементов из таблиц А и В.

Схема алгоритма приведена на рисунке 2.1.

2.2. Алгоритм Винограда

Цель алгоритма заключается в сокращении доли умножений в самом трудоёмком участке кода. Основная идея заключается в следующем.

Пусть u, v - элементы матриц А, В соотв., участвующие в вычислении значения элемента матрицы С. Тогда данный элемент вычисляется как $u_1v_1 + u_2v_2 + u_3v_3 + u_4v_4$. Такое выражение можно представить как $(u_1 + v_2)(v_1 + u_2) + (u_3 + v_4)(v_3 + u_4) - u_1u_2 - u_3u_4 - v_1v_2 - v_3v_3)$. В этом выражении вычитаемые можно вычислить однократно и применить их для всех столбцов и строк, где они используются. Таким образом можно снизить трудоёмкость алгоритма за счёт снижения количества операций.

В случае, если матрица нечётный размер N, требуется производить дополнительные вычисления для крайних строк и столбцов. Таким образом, алгоритм наиболее эффективен в случае матриц, у которых N является чётным.

Схема алгоритма приведена на рисунках 2.2. и 2.3.

2.3. Требования к программному обеспечению

Для полноценной проверки и оценки алгоритмов необходимо выполнить следующее.

1. Обеспечить возможность консольного ввода двух матриц и выбора алгоритма для умножения. Программа должна вывести результирующую матрцу.

2. Реализовать функцию замера процессорного времени, затраченного функцией. Для этого также создать возможность ввода размера матрицы, на которых будет выполнен замер.

2.4. Заготовки тестов

При проверке алгоритмов необходимо будет использовать следующие классы тестов:

- матрицы размером 1x1;
- две или одна пустая матрица;
- квадратные матрицы;
- чётный и нечётный размер N;

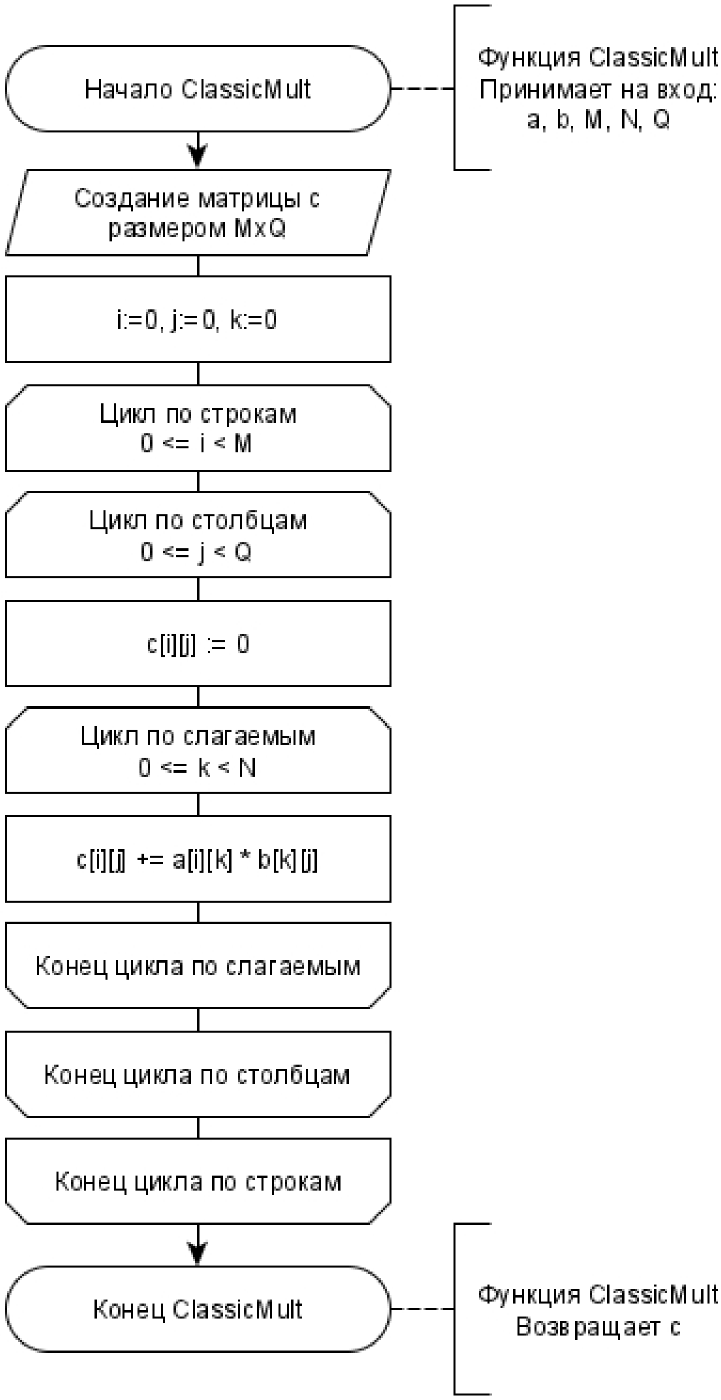

Рис. 2.1: Классический алгоритм умножения матриц

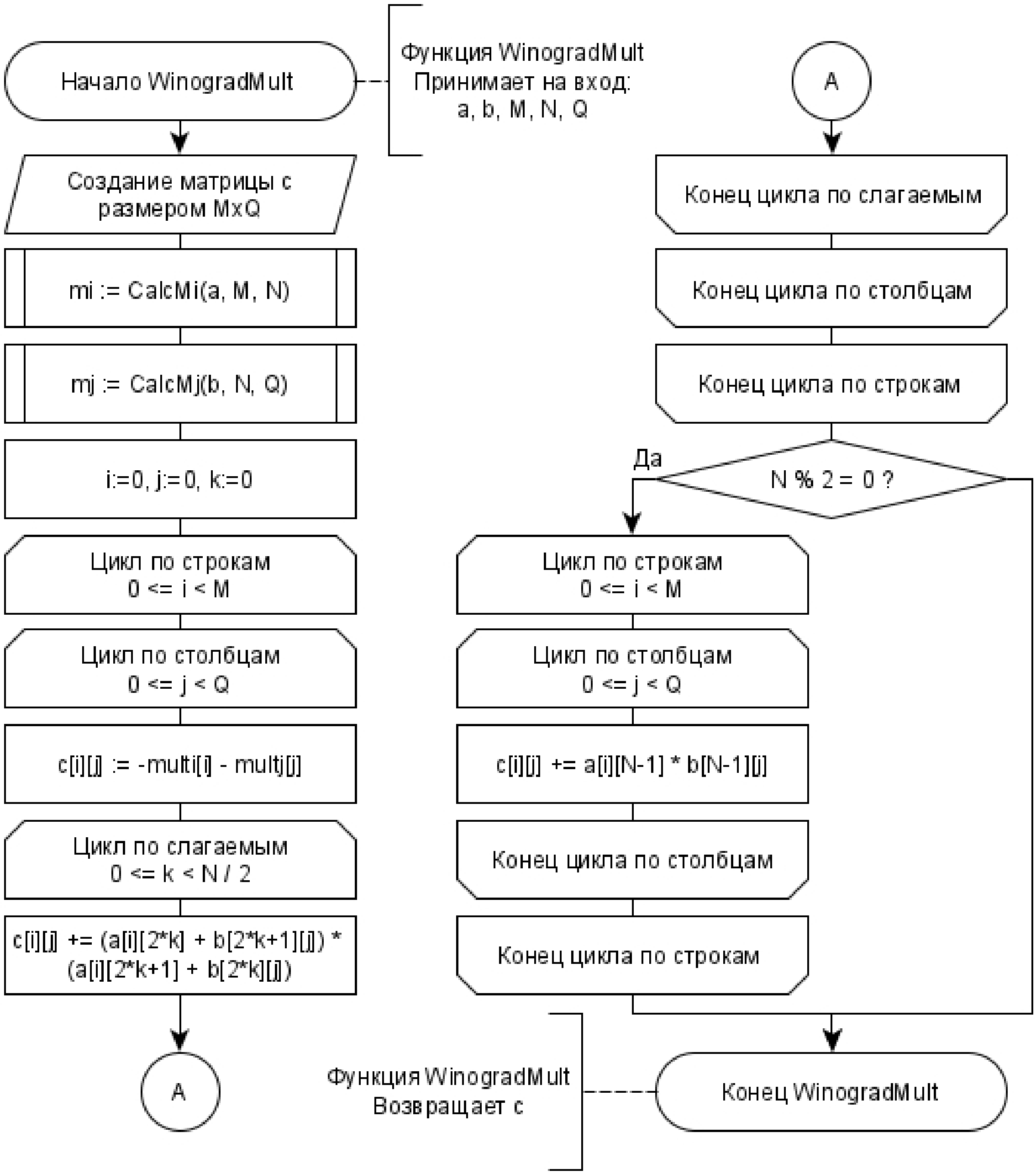

Рис. 2.2: Алгоритм Винограда

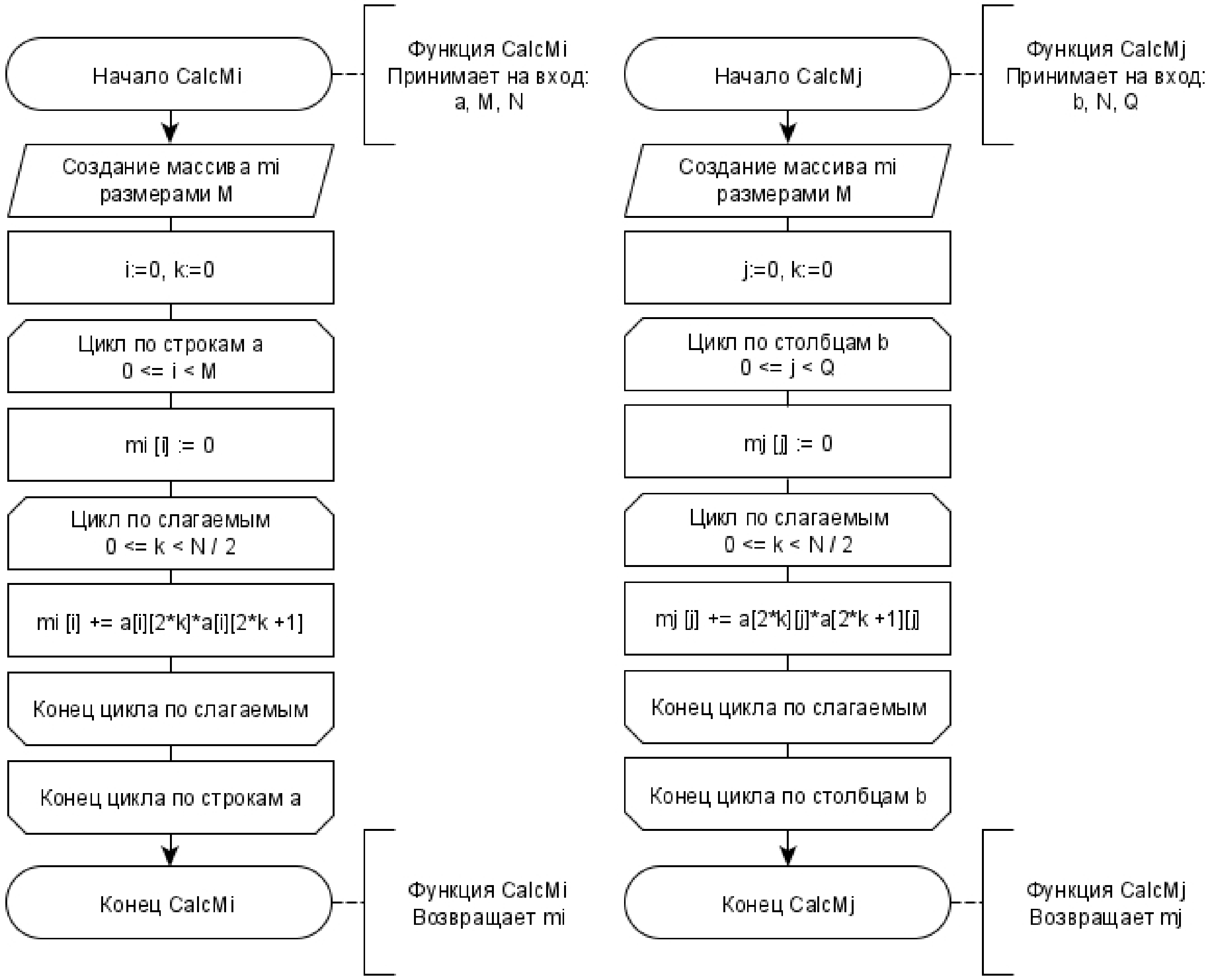

Рис. 2.3: Алгоритм Винограда, функции вычисления вспомогательных массивов

3. Технологическая часть

3.1. Выбор языка программирования

В качестве языка программирования был выбран Python 3, так как имеется опыт работы с ним, и с библиотеками, позволяющими провести исследование и тестирование программы.

3.2. Листинг кода

Реализация алгоритмов поиска расстояний представлена на листингах 3.1-3.4.

Листинг 3.1: Функция нахождения расстояния Левенштейна матричным методом.

```
def lev_matrix(s1, s2, is_print=False):
  matr = [[0] * (len(s1)+1) for i in range(len(s2)+1)]

  for j in range(len(s1)+1):
    matr[0][j] = j
  for i in range(len(s2)+1):
    matr[i][0] = i

  for i in range(1, len(s2)+1):
    for j in range(1, len(s1)+1):
      add = 0 if s1[j-1] == s2[i-1] else 1
      matr[i][j] = min(matr[i-1][j]+1, matr[i][j-1]+1, matr[i-1][j-1]+add)

  if is_print:
    print("Расстояние:", matr[i][j])
    print_matrix(matr)
  return matr[i][j]
```

Листинг 3.2: Функции нахождения расстояния Левенштейна рекурсивным методом.

```
def _lev_rec(s1, s2, len1, len2):
  if len1 == 0: return len2
  elif len2 == 0: return len1
  else:
    return min(_lev_rec(s1, s2, len1, len2-1)+1,
        _lev_rec(s1, s2, len1-1, len2)+1,
        _lev_rec(s1, s2, len1-1, len2-1) +
        (0 if s1[len1-1] == s2[len2-1]
         else 1))
def lev_recursion(s1, s2, is_print=False):
  res = _lev_rec(s1, s2, len(s1), len(s2))
  if is_print:
    print("Расстояние:", res)
  return res
```

Листинг 3.3: Функции нахождения расстояния Левенштейна рекурсивным методом с заполнением матрицы.

```
def _lev_mr(matr, i, j, s1, s2):
  if i+1 < len(matr) and j+1 < len(matr[0]):
    add = 0 if s1[j] == s2[i] else 1
    if matr[i+1][j+1] > matr[i][j] + add:
      matr[i+1][j+1] = matr[i][j] + add
      _lev_mr(matr, i+1, j+1, s1, s2)
  if j+1 < len(matr[0]) and (matr[i][j+1] > matr[i][j] + 1):
    matr[i][j+1] = matr[i][j] + 1
    _lev_mr(matr, i, j+1, s1, s2)
  if i+1 < len(matr) and (matr[i+1][j] > matr[i][j] + 1):
    matr[i+1][j] = matr[i][j] + 1
    _lev_mr(matr, i+1, j, s1, s2)

def lev_matrix_recursion(s1, s2, is_print=False):
```

```
    max_len = max(len(s1), len(s2)) + 1
    matr = [[max_len] * (len(s1)+1) for i in range(len(s2)+1)]
    matr[0][0] = 0
    _lev_mr(matr, 0, 0, s1, s2)

    if is_print:
        print("Расстояние:", matr[-1][-1])
        print_matrix(matr)
    return matr[-1][-1]
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матричным методом.

```
def dem_lev_matrix(s1, s2, is_print=False):
    if len(s1) == 0: return len(s2)
    elif len(s2) == 0: return len(s1)
    matr = [[0] * (len(s1) + 1) for i in range(len(s2) + 1)]
    for j in range(len(s1)+1):
        matr[0][j] = j
    for i in range(len(s2)+1):
        matr[i][0] = i

    for i in range(1, len(s2) + 1):
        addM = 0 if s1[0] == s2[i-1] else 1
        matr[i][1] = min(matr[i-1][1] + 1, matr[i][0] + 1,
        matr[i-1][0] + addM)
    for j in range(2, len(s1) + 1):
        addM = 0 if s1[j-1] == s2[0] else 1
        matr[1][j] = min(matr[0][j] + 1, matr[1][j-1] + 1,
        matr[0][j-1] + addM)

    for i in range(2, len(s2)+1):
        for j in range(2, len(s1)+1):
            addM = 0 if s1[j-1] == s2[i-1] else 1
```

```
        addT = 1 if (s1[j-2] == s2[i-1] and s1[j-1] == s2[i-2])
      else 2
        matr[i][j] = min(matr[i-1][j]+1, matr[i][j-1]+1,
        matr[i-1][j-1]+addM, matr[i-2][j-2]+addT)

    if is_print:
      print("Расстояние:", matr[i][j])
      print_matrix(matr)
    return matr[i][j]

```

3.3. Результаты тестирования

Для тестирования написанных функций была использована библиотека unittest[1]. Тестирование функций проводилось за счёт сравнения результата, возвращённого функцией и ожидаемого расстояния для разных наборов строк.

Состав тестов приведён в листинге 3.5.

Листинг 3.5: Модульные тесты

```
import unittest
import main

# Общий набор тестов для всех алгоритмов
class GeneralTest(unittest.TestCase):
    # Данный класс являтся абстрактным, поэтому для него тесты
      пропускаются
    @unittest.skip("Skip GeneralTest")
    def setUp(self):
        self.function = None

    # Проверка пустыми строками
    def test_empty(self):
```

```
        self.assertEqual(self.function("", ""), 0)
        self.assertEqual(self.function("a", ""), 1)
        self.assertEqual(self.function("", "b"), 1)

    # Проверка нахождения совпадений
    def test_match(self):
        self.assertEqual(self.function("abc", "abc"), 0)
        self.assertEqual(self.function("a", "a"), 0)
        self.assertEqual(self.function("A", "a"), 1)

    # Прочие общие тесты
    def test_other(self):
        self.assertEqual(self.function("q", "w"), 1)
        self.assertEqual(self.function("aq", "aw"), 1)
        self.assertEqual(self.function("a", "aw"), 1)
        self.assertEqual(self.function("aw", "a"), 1)


# Набор тестов для алгоритмов поиска расстояния Левенштейна
class LevTest(GeneralTest):
    def test_lev(self):
        self.assertEqual(self.function("stolb", "telo"), 3)
        self.assertEqual(self.function("kult_tela", "tela_kult"), 6)
        self.assertEqual(self.function("развлечение", "увлечение"), 3)


# Набор тестов для алгоритма поиска расстояния ДамерауЛевенштейна—
class DemLevMatrixTest(GeneralTest):
    def setUp(self):
        self.function = main.dem_lev_matrix

    def dem_lev_test(self):
```

```
        self.assertEqual(self.function("aba", "aab"), 1)
        self.assertEqual(self.function("ab", "ba"), 1)
        self.assertEqual(self.function("abb", "bab"), 1)


# Алгоритмы поиска расстояния Левенштейна проходят одинковые тесты из
    класса LevTest
# Алгоритм поиска расстояния Левенштейна, матричный метод
class LevMatrixTest(LevTest):
    def setUp(self):
        self.function = main.lev_matrix
# Алгоритм поиска расстояния Левенштейна, рекурсивный метод
class LevRecursionTest(LevTest):
    def setUp(self):
        self.function = main.lev_recursion
# Алгоритм поиска расстояния Левенштейна, рекурсивный метод с
    заполнением матрицы
class LevMatRecTest(LevTest):
    def setUp(self):
    self.function = main.lev_matrix_recursion

# Точка входа, запуск тестов
if __name__ == "__main__":
    unittest.main()
```

3.4. Оценка памяти

Произведём оценку наибольшей затрачиваемой алгоритмом памяти M_{max} при поиске расстояний для строк s1 и s2. Для удобства оценки примем длину обеих строк за n.

Расстояние Левенштейна, матричный метод. Память затрачивается на матрицу и две строки.

$M_{max} = (n+1)*(n+1)*sizeof(int)+(n+n)*sizeof(char) = (n+1)*(n+1)*16+(n+n) = 16*n^2+2*17n+16$ байт

Расстояние Дамерау-Левенштейна, матричный метод. Аналогично.

$M_{max} = 16*n^2+2*17n+16$ байт

Расстояние Левенштейна, рекурсивный метод. Память используется при каждом вызове функции. Одна функция принимает в качестве аргумента 2 строки по значению, 2 размера строк. Максимальная глубина рекурсии = n+n.

$M_{max} = (n+n)*(2n*sizeof(char)+2*sizeof(int)) = 2n*(2n+32) = 4n^2+64n$ байт

Расстояние Левенштейна, рекурсивный метод с матрицей. Память используется для матрицы и при каждом вызове функции. Максимальная глубина рекурсии = n+n.

$M_{max} = (n+1)*(n+1)*sizeof(int)+(n+n)*(2n*sizeof(char)+2*sizeof(int)) = (n^2+2n+1)*16+2n*(2n+32) = 20n^2+96n+16$ байт

3.5. Оценка времени

Для замера процессорного времени исполнения функции используется библиотека time. Проведение измерений производится в функции, приведённой в листинге 3.6. Также в листинге приведена функция *random_str* для создания строки заданной длины из случайной последовательности символов, с использованием библиотеки random.

Листинг 3.6: Функция замера процессорного времени работы функции

```python
def random_str(length):
    a = []
    for i in range(length):
        a.append(random.choice("qwerty"))
    return "".join(a)

def test_time(func):
    length = int(input("Введите длину строки: "))
    s1 = random_str(length)
    s2 = random_str(length)
    print("Строка 1:", s1)
    print("Строка 2:", s2)

    begin_t = time.process_time()
    count = 0
    while time.process_time() - begin_t < 1.0:
        func(s1, s2)
        count += 1

    t = time.process_time() - begin_t
    print("Выполнено {:} операций за {:} секунд".format(count, t))
    print("Время: {:7.4} секунд".format(t / count))

```

Исследовательская часть

План экспериментов

Измерения процессорного времени проводятся при равных длинах строк s1 и s2. Содержание строк сгенерировано случайным образом. Изучается время работы при длинах: 1, 3, 10, 20, 100, 1000. Для повышения точности, каждый замер производится пять раз, за результат берётся среднее арифметическое.

Результат экспериментов

По результатам измерений процессорного времени можно составить таблицу 4.1

Таблица 4.1: Результат измерений процессорного времени (в секундах)

	1	3	10	20	100	1000
Лев., матрица	$7 * 10^{-6}$	$1.9 * 10^{-5}$	$1.3 * 10^{-4}$	$4.7 * 10^{-4}$	0.013	1.405
Лев., рекурсия	$3 * 10^{-6}$	$4.7 * 10^{-5}$	6.984	–	–	–
Лев., рекурсия с матрицей	$1 * 10^{-5}$	$4.1 * 10^{-5}$	$4.1 * 10^{-4}$	$2.5 * 10^{-3}$	0.38	–
Д-Л, матрица	$8 * 10^{-6}$	$2.8 * 10^{-5}$	$1.7 * 10^{-4}$	$6.1 * 10^{-4}$	0.016	2.031

В алгоритме нахождения расстояния Левенштейна с помощью рекурсии замеры на длине строк более 10 не проводились, так как время выполнения было слишком велико (более 10 минут). В алгоритме рекурсии с заполнением матрицы не удалось провести измерения при длине 1000, так как была превышена максимальная глубина рекурсии.

Сравнительный анализ

По результатам эксперимента можно заключить следующее.

- Наиболее быстродейственным алгоритмом поиска расстояния Левенштейна является алгоритм, использующий матрицу.
- Рекурсивный алгоритм с использованием матрицы показывает значительно более низкую скорость роста времени по сравнению с рекурсивным алгоритмом.
- Алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна с помощью матрицы показывают схожую скорость роста времени, однако первый алгоритм несколько быстрее.

Заключение

В ходе лабораторной работы достигнута поставленная цель: реализация и сравнение алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. Решены все задачи работы.

Были изучены и описаны понятия расстояний Левенштейна и Дамерау-Левенштейна. Также были описаны и реализованы алгоритмы поиска расстояний. Проведены замеры процессорного времени работы каждого алгоритмах при различных строках, оценена наибольшая занимаемая память. На основании оценок и экспериментов проведён сравнительный анализ.

Список литературы

1. Документация unittest [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/unittest.html, свободный (дата обращения: 27.09.2020).

2. Трудоёмкость программ [Электронный ресурс] Режим доступа: http://ermak.cs.nstu.ru/cprog/html/041.htm , свободный (дата обращения: 27.09.2020).