**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе № 3**

**по дисциплине «Построение и Анализ Алгоритмов»**

Тема: «**Динамическое программирование**»

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Коршков А.А. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

# Цель работы

Написании программы вычисления расстояния Левенштейна (редакционное расстояние) и предписания алгоритмом Вагнера-Фишера.

# Задания

**№1**

Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

1. replace(ε, a, b) - заменить символ а на символ b.

2. insert(ε, a) - вставить в строку символ а (на любую позицию).

3. delete(ε, b) - удалить из строки символ b.

Каждая операция может иметь некоторую цену выполнения (положительное число).

Даны две строки А и В, а также три числа, отвечающие за цену каждой операции. Определите минимальную стоимость операций, которые необходимы для превращения строки А в строку В.

Входные данные: первая строка - три числа: цена операции replace, цена операции insert, цена операции delete; вторая строка - А; третья строка - В.

Выходные данные: одно число - минимальная стоимость операций.

**Sample Input:**

111

entrance

reenterable

**Sample Output:**

5

**№2**

Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

1. replace(ε, a, b) - заменить символ а на символ b.

2. insert(ε, a) - вставить в строку символ а (на любую позицию).

3. delete(ε, b) - удалить из строки символ b.

Каждая операция может иметь некоторую цену выполнения (положительное число).

Даны две строки А и В, а также три числа, отвечающие за цену каждой операции. Определите последовательность операций (редакционное предписание) с минимальной стоимостью, которые необходимы для превращения строки А в строку В.

Входные данные: первая строка - три числа: цена операции replace, цена операции insert, цена операции delete; вторая строка - А; третья строка - В.

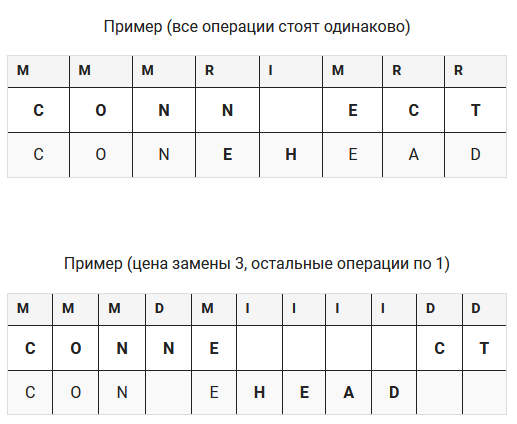


Рисунок 1 - Пример

Выходные данные: первая строка - последовательность операций (M - совпадение, ничего делать не надо; R - заменить символ на другой; I - вставить символ на текущую позицию; D - удалить символ из строки); вторая строка - исходная строка А; третья строка - исходная строка В.

**Sample Input:**

111

entrance

reenterable

**Sample Output:**

IMIMMIMMRRM

entrance

reenterable

**№3**

Расстоянием Левенштейна назовём минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Разработайте программу, осуществляющую поиск расстояния Левенштейна между двумя строками.

Пример:

Для строк pedestal и stien расстояние Левенштейна равно 7:

• Сначала нужно совершить четыре операции удаления символа: pedestal -> stal.

• Затем необходимо заменить два последних символа: stal -> stie.

• Потом нужно добавить символ в конец строки: stie -> stien.

Параметры входных данных:

Первая строка входных данных содержит строку из строчных латинских букв. (S, 1 **≤** |S| ≤ 2550).

Вторая строка входных данных содержит строку из строчных латинских букв. (Т,1 **≤** |T| ≤ 2550).

Параметры выходных данных:

Одно число L, равное расстоянию Левенштейна между строками S и T.

**Sample Input:**

pedestal

stien

**Sample Output:**

7

**Задание варианта:**

**3а.** Добавляется 4-я операция со своей стоимостью: последовательная вставка двух одинаковых символов.

**Примечания для варианта:**

1) Предполагается, что для весов операций действует правило треугольника: если две последовательные операции можно заменить одной, то это не ухудшает общую цену.

2) При выполнении операций запрещается применять операции к символам, которые уже были получены в результате выполнения операций (т.е. строка преобразуется всегда только слева направо).

## Основные теоретические положения

**Описание алгоритмов:**

Алгоритм Вагнера-Фишера — это алгоритм динамического программирования, предназначенный для вычисления редакционного расстояния между двумя строками. Это минимальное количество операций вставки, удаления или замены символов, необходимых для преобразования одной строки в другую с учётом заданных им стоимостей.

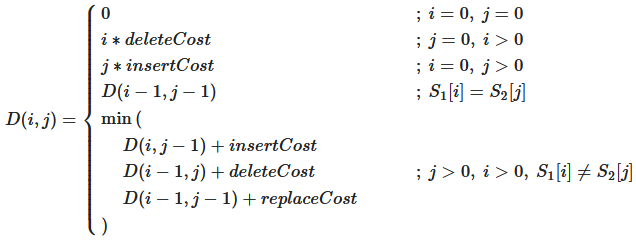


Рисунок 2 – Рекуррентная формула для алгоритма Вагнера-Фишера

Расстояние Левенштейна – частный случай алгоритма Вагнера-Фишера, где стоимостей всех операций (вставки, замены, удаления) равна 1. Для расстояния Левенштейна также задаётся рекуррентная формула.

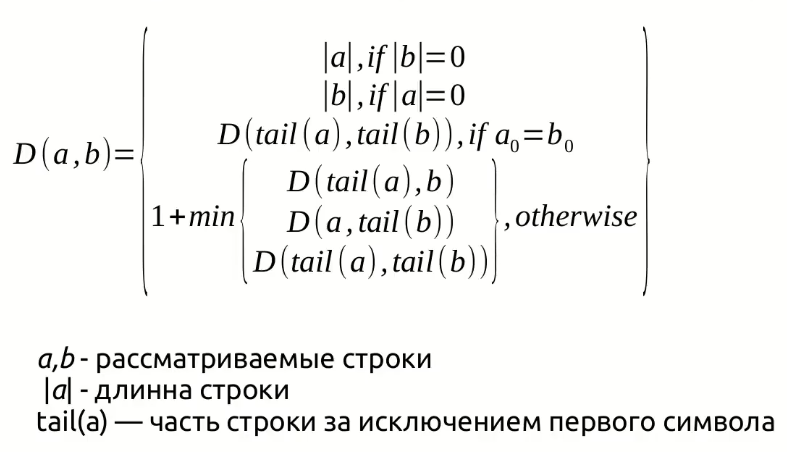


Рисунок 3 – Рекуррентная формула для расстояния Левенштейна

a, b - рассматриваемые строки

|a| - длина строки

tail(a) - часть строки за исключением первого символа

Расстояние Левенштейна можно находить без алгоритма Вагнера-Фишера при помощи рекурсивного метода, однако он очень затратный по ресурсам и очень неэффективный для больших строк из-за огромного кол-ва рекурсивных вызовов. Можно использовать кэширование, чтобы сохранять повторные результаты вызовов функций, однако по сравнению с алгоритмом Вагнера-Фишера это всё равно будет неэффективный метод.

Сложность по времени экспоненциальная для рекурсивного метода: O()

Сложность по памяти: O(n+m) из-за глубины рекурсии.

Суть алгоритма Вагнера-Фишера сводиться к тому, чтобы построить матрицу размером (n+1) \* (m+1) и заполнить её элементы на основе рекуррентной формулы ниже.

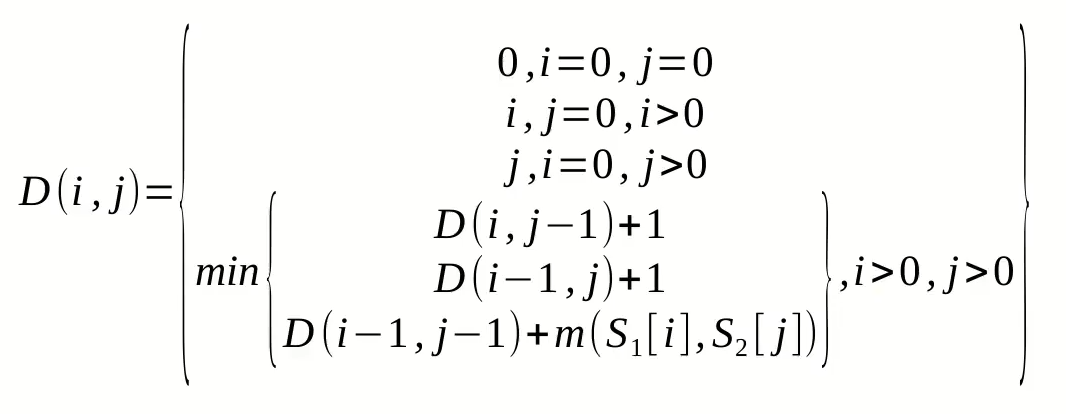


Рисунок 4 – Рекуррентная формула Вагнера-Фишера для нахождения расстояния Левенштейна

- рассматриваемые строки (индексация начинается с единицы). m([i], [i]) = 0 в случае, если [i] = S2[i], иначе m([i], [i]) = 1.

Принцип работы алгоритма Вагнера-Фишера.

1) Используем любую структуру данных способную представлять матрицу размером (М+1, N+1).

2) Для каждого элемента этой матрицы вычисляем значение расстояния Левенштейна по указанной рекуррентной формуле. Прохождение последовательное по рядам и столбцам.

3) Возвращаем значение по индексу M, N (при условии, что индексация начинается с нуля).

**Оценка сложности по памяти и операциям**

Сложность по времени в наихудшем случае O((N+1) (M+1)), т.к. необходимо пройтись по всей таблице.

Сложность по памяти O((N+1) (M+1)), т.к. необходимо хранить матрицу размером (n+1) \* (m+1).

Примечание: для нахождения последовательностей операций необходимо хранить вторую таблицу с наилучшими выборами операции на каждом шаге. Поэтому в этом случае сложность по памяти .

## Выполнение работы

**Описание работы**

Для решения заданий были написаны три функции.

*def \_wagner\_fisher\_step(i: int, j: int, s1: str, s2: str, matrix: list[list[int]], rep\_cost: int, ins\_cost: int, del\_cost: int, ins2\_cost: int) -> int*

Внутренняя функция для вычисления значения на определённом шаге алгоритма. Принимает позицию i и j, исходную и конечную строки, матрица значений, стоимость операций, включая вставку двух одинаковых символов.

*def calculate\_edit\_distance(s1: str, s2: str, rep\_cost: int = 1, ins\_cost: int = 1, del\_cost: int = 1, ins2\_cost: int = 1) -> int*

Функция, вычисляющая кол-во минимальных операций с заданной для них стоимостью.

*def compute\_edit\_sequence(s1: str, s2: str, rep\_cost: int, ins\_cost: int, del\_cost: int, ins2\_cost: int) -> str*

Функция, принимающая на вход две строки и стоимость операций. Возвращает строку с порядком операций.

# Тестирование

Таблица 1 – Тестирование алгоритмов

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарии |
| 1 | 1 1 1 1  entrance  reenterable | 4 | Алгоритм Вагнера-Фишера. Кол-во операций.  Вычислено корректно (стоимость операций совпадает со случаем расстояния Левенштейна). |
| 2 | 1 3 2 4  ironman  spiderman | 7 | Алгоритм Вагнера-Фишера. Кол-во операций. Вычислено корректно. |
| 3 | 1 1 1 1  entrance  reenterable | PMMMIMMRRM  entrance  reenterable | Алгоритм Вагнера-Фишера. Порядок операций.  Вычислено корректно (стоимость операций совпадает со случаем расстояния Левенштейна). |
| 4 | 1 3 2 4  ironman  spiderman | PMRRRMMM ironman  spiderman | Алгоритм Вагнера-Фишера. Порядок операций.  Вычислено корректно. |
| 5 | pedestal  stien | 7 | Расстояние Левенштейна. Вычислено корректно. |
| 6 | spiderman  ironman | 5 | Расстояние Левенштейна. Вычислено корректно. |

# Выводы

Был реализован алгоритм Вагнера-Фишера для расстояния Левенштейна и редакционного предписания между двумя строками, определяя минимальное количество операций (вставки, удаления, замены, *вставка двух одинаковых символов подряд*) для преобразования одной строки в другую.

ПРИЛОЖЕНИЕ А  
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

"""

Главный файл программы.

Вар. 3а. Добавляется 4-я операция со своей стоимостью: последовательная вставка

двух одинаковых символов.

"""

from modules.vagner\_fisher import calculate\_edit\_distance, compute\_edit\_sequence

def main() -> None:

"""

Главная функция

:return:

"""

print("Задание #1: Алгоритм Вагнера-Фишера")

rep\_cost, ins\_cost, del\_cost, ins2\_cost = map(int, input().split())

s1: str = input()

s2: str = input()

print("Результат:", calculate\_edit\_distance(s1, s2, rep\_cost, ins\_cost, del\_cost, ins2\_cost))

print("Задание #2: Алгоритм Вагнера-Фишера. Порядок операции")

rep\_cost, ins\_cost, del\_cost, ins2\_cost = map(int, input().split())

s1: str = input()

s2: str = input()

print(compute\_edit\_sequence(s1, s2, rep\_cost, ins\_cost, del\_cost, ins2\_cost), s1, s2, sep="\n")

print("Задание #3: Расстояние Левенштейна")

s1: str = input()

s2: str = input()

print("Результат:", calculate\_edit\_distance(s1, s2))

if \_\_name\_\_ == "\_\_main\_\_":

main()

Название файла: vagner\_fisher.py

"""

Модуль для вычисления алгоритма Вагнера-Фишера.

"""

def \_wagner\_fisher\_step(i: int, j: int, s1: str, s2: str, matrix: list[list[int]],

rep\_cost: int, ins\_cost: int, del\_cost: int, ins2\_cost: int) -> int:

"""

Вычисляет шаг алгоритма Вагнера-Фишера для двух строк s1 и s2.

:param i:

:param j:

:param s1:

:param s2:

:param matrix:

:param rep\_cost:

:param ins\_cost:

:param del\_cost:

:param ins2\_cost:

:return:

"""

print(f"--- Вычисление ячейки ({i}, {j}) ---")

if i == 0 and j == 0:

print("Начальная ячейка (0, 0), значение 0.")

return 0

if j == 0:

val: int = i \* del\_cost

print(f"j=0: удаление {i} символов. Значение = {val}.")

return val

if i == 0:

if j == 1:

val: int = matrix[0][0] + ins\_cost

print(f"i=0, j=1: единичная вставка. Значение = {val} (0 + {ins\_cost}).")

return val

val1: int = matrix[0][j - 1] + ins\_cost

val2: int = matrix[0][j - 2] + ins2\_cost

val: int = min(val1, val2)

print(f"i=0, j={j}: варианты {val1} (одиночная вставка) "

f"и {val2} (двойная вставка). "

f"Минимум: {val}.")

return val

# Замена или совпадение

rep: int = matrix[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1] else rep\_cost)

ins: int = matrix[i][j - 1] + ins\_cost

dele: int = matrix[i - 1][j] + del\_cost

candidates: list[int] = [rep, ins, dele]

if j >= 2:

ins2\_val = matrix[i][j - 2] + ins2\_cost

print(f"Двойная вставка: {ins2\_val} (база {matrix[i][j - 2]} + {ins2\_cost})")

candidates.append(ins2\_val)

else:

print("Двойная вставка недоступна (j < 2)")

print(f"Кандидаты для ячейки ({i}, {j}): {candidates}. "

f"Минимальное значение: {min(candidates)}.")

return min(candidates)

def calculate\_edit\_distance(s1: str, s2: str,

rep\_cost: int = 1, ins\_cost: int = 1,

del\_cost: int = 1, ins2\_cost: int = 1) -> int:

"""

Вычисляет расстояние редактирования между строками s1 и s2

с учётом операций: замены, вставки, удаления и

последовательной вставки двух одинаковых символов.

:param s1:

:param s2:

:param rep\_cost:

:param ins\_cost:

:param del\_cost:

:param ins2\_cost:

:return:

"""

n, m = len(s1), len(s2)

matrix: list[list[int]] = [[0] \* (m + 1) for \_ in range(n + 1)]

for i in range(n + 1):

for j in range(m + 1):

matrix[i][j] = \_wagner\_fisher\_step(i, j, s1, s2, matrix,

rep\_cost, ins\_cost,

del\_cost, ins2\_cost)

print(f"Текущее значение матрицы[{i}][{j}] = {matrix[i][j]}")

print(f"\nСостояние матрицы после строки i={i}:")

for row in matrix[:i + 1]:

print(' '.join(map(str, row)))

print("-" \* 50 + "\n")

return matrix[n][m]

def compute\_edit\_sequence(s1: str, s2: str,

rep\_cost: int = 1, ins\_cost: int = 1,

del\_cost: int = 1, ins2\_cost: int = 1) -> str:

"""

Вычисляет последовательность операций для преобразования строки s1 в s2

с учётом дополнительных затрат при последовательной вставке двух символов.

Обозначения:

M – совпадение

R – замена

I – вставка одного символа

D – удаление символа

P – последовательная вставка двух одинаковых символов

:param s1:

:param s2:

:param rep\_cost:

:param ins\_cost:

:param del\_cost:

:param ins2\_cost:

:return:

"""

n, m = len(s1), len(s2)

cost: list[list[int]] = [[0] \* (m + 1) for \_ in range(n + 1)]

back: list[list[str]] = [[''] \* (m + 1) for \_ in range(n + 1)]

print("\n" + "=" \* 50)

print("Инициализация первого столбца (операции удаления):")

for i in range(1, n + 1):

cost[i][0]: int = cost[i - 1][0] + del\_cost

back[i][0]: str = 'D'

print(f"\ti={i}, j=0 → УДАЛЕНИЕ (D). cost[{i}][0] = {cost[i][0]} "

f"(предыдущее {cost[i-1][0]} + {del\_cost})")

print("\n" + "=" \* 50)

print("Инициализация первой строки (операции вставки):")

if m >= 1:

cost[0][1]: int = cost[0][0] + ins\_cost

back[0][1]: str = 'I'

print(f"\ti=0, j=1 → ВСТАВКА (I). cost[0][1] = {cost[0][1]} (0 + {ins\_cost})")

for j in range(2, m + 1):

candidate\_single: int = cost[0][j - 1] + ins\_cost

candidate\_double: int = cost[0][j - 2] + ins2\_cost

print(f"\n i=0, j={j}:")

print(f"\tВариант 1: одиночная вставка → {candidate\_single} "

f"(cost[0][{j - 1}]={cost[0][j - 1]} + {ins\_cost})")

print(f"\tВариант 2: двойная вставка → {candidate\_double} "

f"(cost[0][{j - 2}]={cost[0][j - 2]} + {ins2\_cost})")

if candidate\_double < candidate\_single:

cost[0][j]: int = candidate\_double

back[0][j]: str = 'P'

print("\tВыбрана ДВОЙНАЯ ВСТАВКА (P)")

else:

cost[0][j]: int = candidate\_single

back[0][j]: str = 'I'

print("\tВыбрана ОДИНОЧНАЯ ВСТАВКА (I)")

print(f"\tcost[0][{j}] = {cost[0][j]}, back[0][{j}] = '{back[0][j]}'")

print("\n" + "=" \* 50)

print("Заполнение основной матрицы:")

for i in range(1, n + 1):

print(f"\nОбработка строки i={i}:")

for j in range(1, m + 1):

print(f"\n--- Ячейка ({i}, {j}) ---")

print(f"\tСимволы: s1[{i - 1}] = '{s1[i - 1]}', s2[{j - 1}] = '{s2[j - 1]}'")

if s1[i - 1] == s2[j - 1]:

rep\_val: int = cost[i - 1][j - 1]

op\_rep: str = 'M'

print(f"\tСОВПАДЕНИЕ (M): cost = {rep\_val}")

else:

rep\_val: int = cost[i - 1][j - 1] + rep\_cost

op\_rep: str = 'R'

print(f"\tЗАМЕНА (R): cost = {cost[i - 1][j - 1]} + {rep\_cost} = {rep\_val}")

ins\_val: int = cost[i][j - 1] + ins\_cost

op\_ins: str = 'I'

print(f"\tВСТАВКА (I): cost = {cost[i][j - 1]} + {ins\_cost} = {ins\_val}")

del\_val: int = cost[i - 1][j] + del\_cost

op\_del: str = 'D'

print(f"\tУДАЛЕНИЕ (D): cost = {cost[i - 1][j]} + {del\_cost} = {del\_val}")

best: int = rep\_val

best\_op: str = op\_rep

if ins\_val < best:

best: int = ins\_val

best\_op: str = op\_ins

if del\_val < best:

best: int = del\_val

best\_op: str = op\_del

if j >= 2:

double\_ins\_val: int = cost[i][j - 2] + ins2\_cost

print(f"\tДВОЙНАЯ ВСТАВКА (P): cost = {cost[i][j - 2]} + {ins2\_cost} = {double\_ins\_val}")

if double\_ins\_val < best:

best: int = double\_ins\_val

best\_op: str = 'P'

cost[i][j]: int = best

back[i][j]: str = best\_op

print(f"\tВыбранная операция: '{best\_op}' → cost[{i}][{j}] = {best}")

print("\n" + "=" \* 50)

print("Матрица стоимостей операций")

for row in cost:

print('\t'.join(map(str, row)))

print("\n" + "=" \* 50)

print("Матрица оптимальных операций")

for row in back:

print('\t'.join(row))

print("\n" + "=" \* 50)

print("Восстановление последовательности операций:")

i, j = n, m

operations: list = []

while i > 0 or j > 0:

op: str = back[i][j]

operations.append(op)

print(f"\tПозиция ({i}, {j}): операция '{op}'")

if op in ('M', 'R'):

i -= 1

j -= 1

elif op == 'I':

j -= 1

elif op == 'D':

i -= 1

elif op == 'P':

j -= 2

operations.reverse()

return ''.join(operations)