МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМ. Н. Э. БАУМАНА

УДК ______
№ госрегистрации ______
Инв. №______

УТВЕРЖДАЮ

Преподаватель

____

«______» ____________________ 2019 г.

ДИСЦИПЛИНА АНАЛИЗ АЛГОРИТМОВ
ОТЧЁТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

Расстояние Левенштейна
(промежуточный)

Студент _________________ Ф.М. Набиев

Преподаватели Л.Л. Волкова, Ю.В. Строганов

Москва, 2019

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является изучение динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Данные алгоритмы решают проблему поиска редакционного расстояния между двумя строками. Редакционное расстояние определяется количеством некоторых операций, необходимых для превращения одного слова в другое, а так же стоимостью этих операций.

Для достижения поставленной цели необходимо решить следующие задачи:

— изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;

— применение метода динамического программирования для матричной реализации указанных алгоритмов;

— получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;

— сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

— экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

— описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитический раздел

1.1 Описание алгоритмов

Алгоритм Дамерау-Левенштейна является модификацией алгоритма Левенштейна. Рассмотрим данные методы подробнее.

1.1.1 Алгоритм Левенштейна

Расстояние Левенштейна между двумя строками - это минимальная сумма произведений количества операций вставки, удаления и замены одного символа, необходимых для перврващения одной строки в другую, на их стоимость.

Вышеописанные операции имеют следующие обозначения:

— I (insert) - вставка;

— D (delete) - удаление;

— R (replace) - замена;

При этом cost(x) есть обозначение стоимости некоторой операции x. Будем считать, что символы в строках нумеруются с первого. Пусть S_1 и S_2 - две строки с длинами N и M соответственно. Тогда расстояние Левенштейна D(M, N) вычисляется по формуле (1.1):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i * cost(D), & j = 0, i > 0 \\ j * cost(I), & i = 0, j > 0 \\ D(i-1, j-1), & S_1[i] = S_2[j] \\ min(& \\ D(i,j-1) + cost(I), & \\ D(i-1,j) + cost(D), & j > 0, i > 0, S_1[i] \neq S_2[j] \\ D(i-1,j-1) + cost(R) & \\ ), & \end{cases} \tag{1.1}$$

где min(a, b, c) возвращает наименьшее значение из a, b, c.

1.1.2 Алгоритм Дамерау-Левенштейна

Определение расстояния Дамерау-Левенштейна аналогично определению расстояния Левенштейна с учётом новой операции - перестановки соседних символов (транспозиции). Соответственно, обозначения операций:

- I (insert) - вставка;
- D (delete) - удаление;
- R (replace) - замена;
- T (transpose) - перестановка соседних символов.

При тех же обозначениях имеем формулы (1.2) и (1.3):

$$D(i,j) = \begin{cases} min(A, D(i-2, j-2) + cost(T), & i > 1, j > 1, \\ & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j] \\ A & \text{Иначе} \end{cases} \tag{1.2}$$

где A:

$$A = \begin{cases} 0, & i = 0, j = 0 \\ i * cost(D), & j = 0, i > 0 \\ j * cost(I), & i = 0, j > 0 \\ D(i-1, j-1), & S_1[i] = S_2[j] \\ min(& \\ D(i,j-1) + cost(I), & \\ D(i-1, j) + cost(D), & j > 0, i > 0, S_1[i] \neq S_2[j] \\ D(i-1, j-1) + cost(R) & \\ ), & \end{cases} \tag{1.3}$$

2 Конструкторский раздел

2.1 Модель

IDEFØ модель задачи вычисления редакционного расстояния приведена на рисунке 2.1.

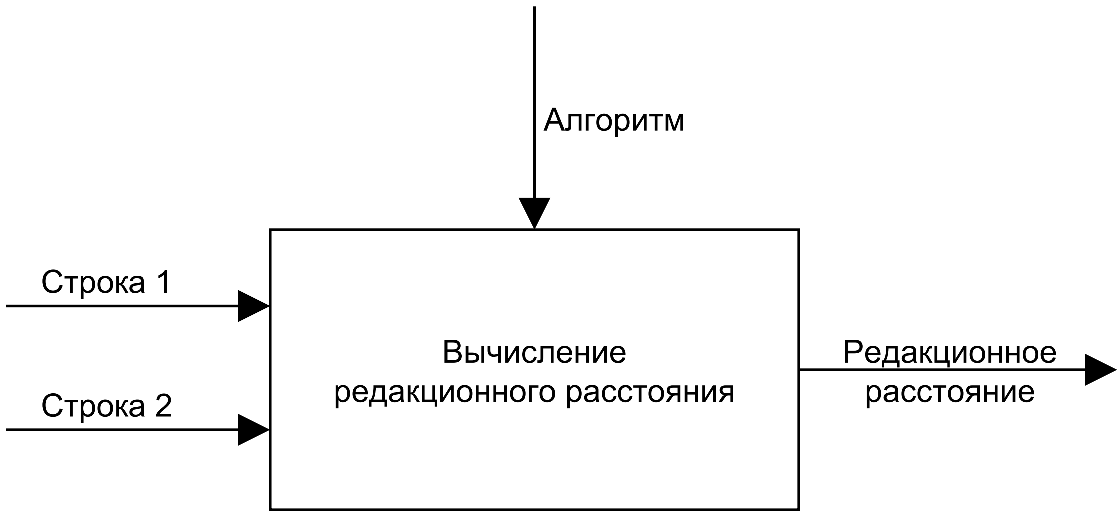

Рисунок 2.1 — IDEFØ модель

2.2 Разработка алгоритмов

Для непосредственной реализации вышеописанных алгоритмов важно иметь их некоторые упрощённые формальные представления, так как чтение таких представлений упрощает написание кода. Подходящим для этого вариантом визуализации являются схемы алгоритмов.

2.2.1 Алгоритм Вагнера-Фишера

Алгоритм Вагнера-Фишера является матричной реализацией поиска расстояния Левенштейна. Схема данного алгоритма приведена на рисунке 2.2.

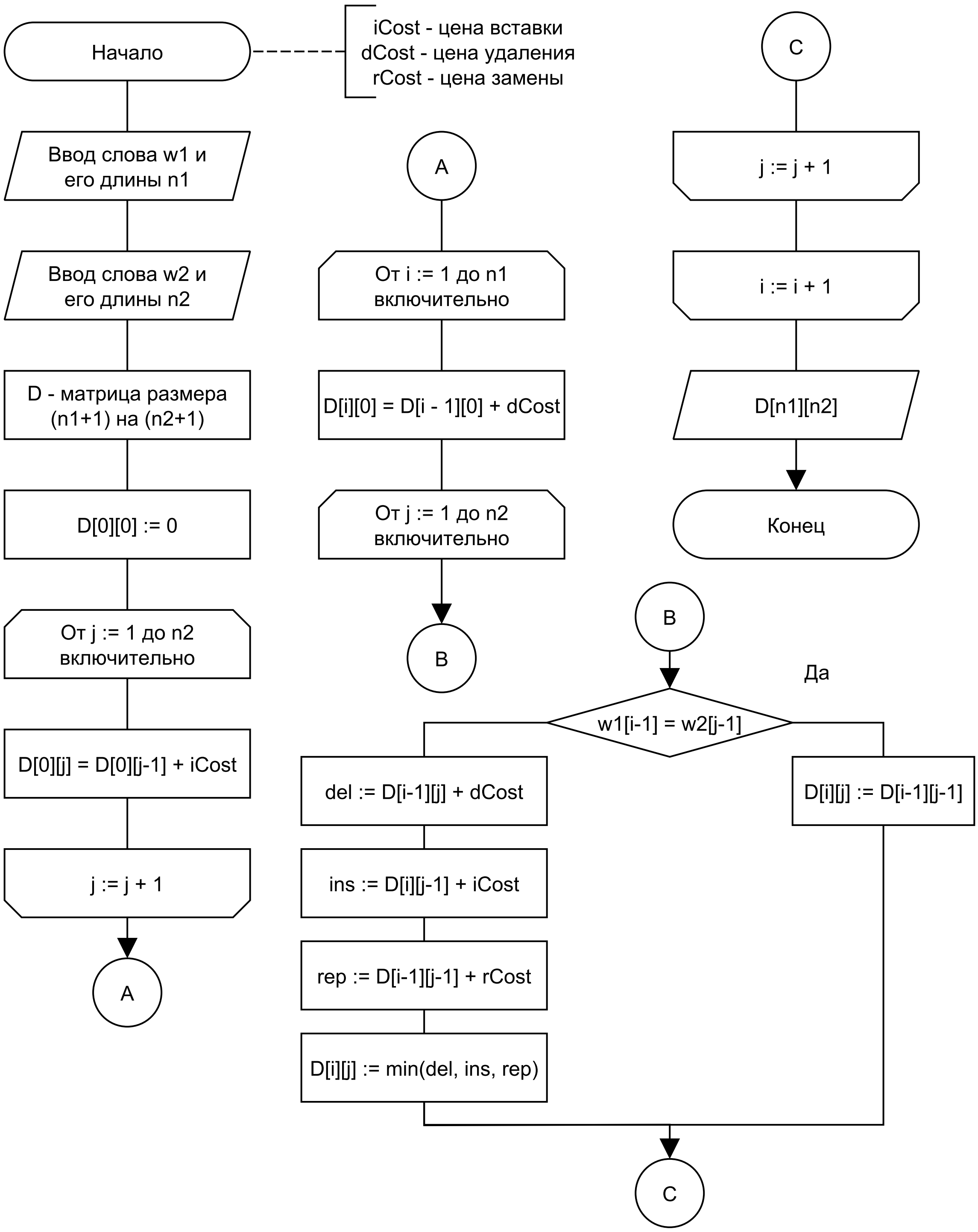

Рисунок 2.2 — Алгоритм Вагнера-Фишера

2.2.2 Матричный алгоритм Дамерау-Левенштейна

Матричный алгоритм Дамерау-Левенштейна представляет из себя модификацию алгоритма Вагнера-Фишера, в котором происходит дополнительная проверка на возможность проведения операции транспозиции. Схема данного алгоритма приведена на рисунках 2.3 и 2.4

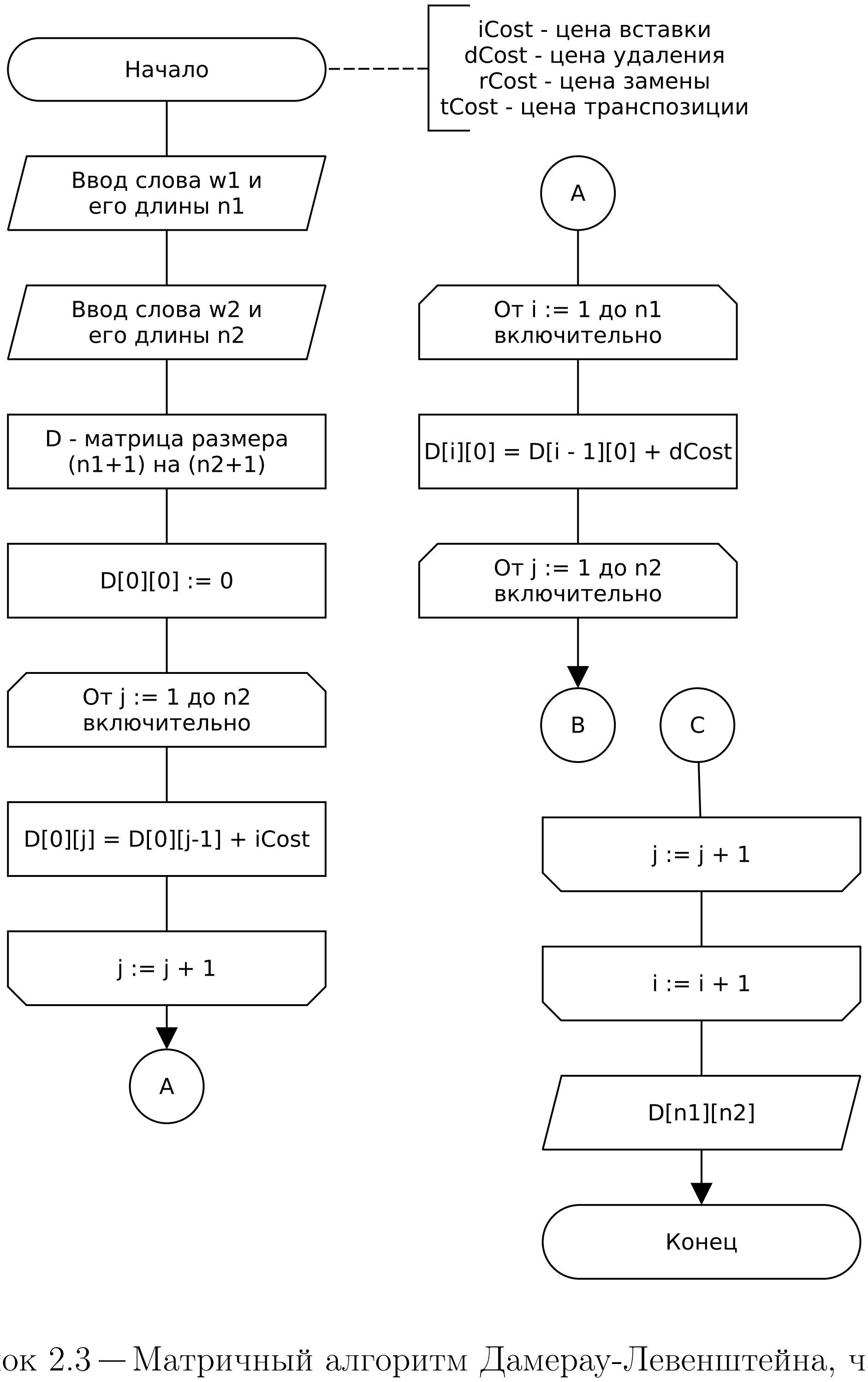

Рисунок 2.3 — Матричный алгоритм Дамерау-Левенштейна, часть 1

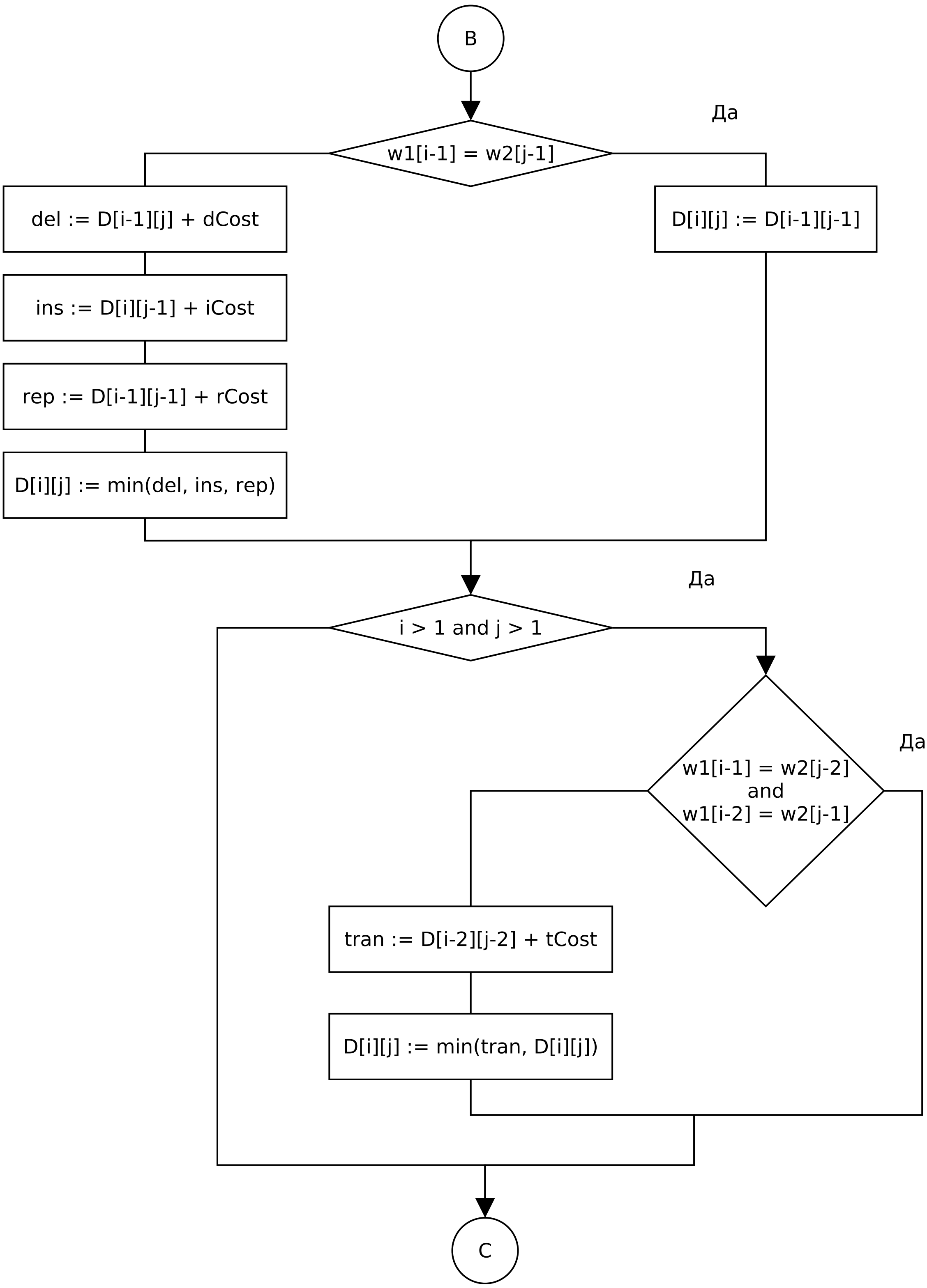

Рисунок 2.4 — Матричный алгоритм Дамерау-Левенштейна, часть 2

2.2.3 Рекурсивный алгоритм Дамерау-Левенштейна

Суть рекурсивного алгоритма Левенштейна состоит в сведении поиска редакционного расстояния до тривиального случая, когда длина хотя бы одного из слов равна 0. Отличие алгоритма Левенштейна от алгоритма Дамерау-Левенштейна было описано ранее. Схема данного алгоритма приведена на рисунках 2.5, 2.6 и 2.7.

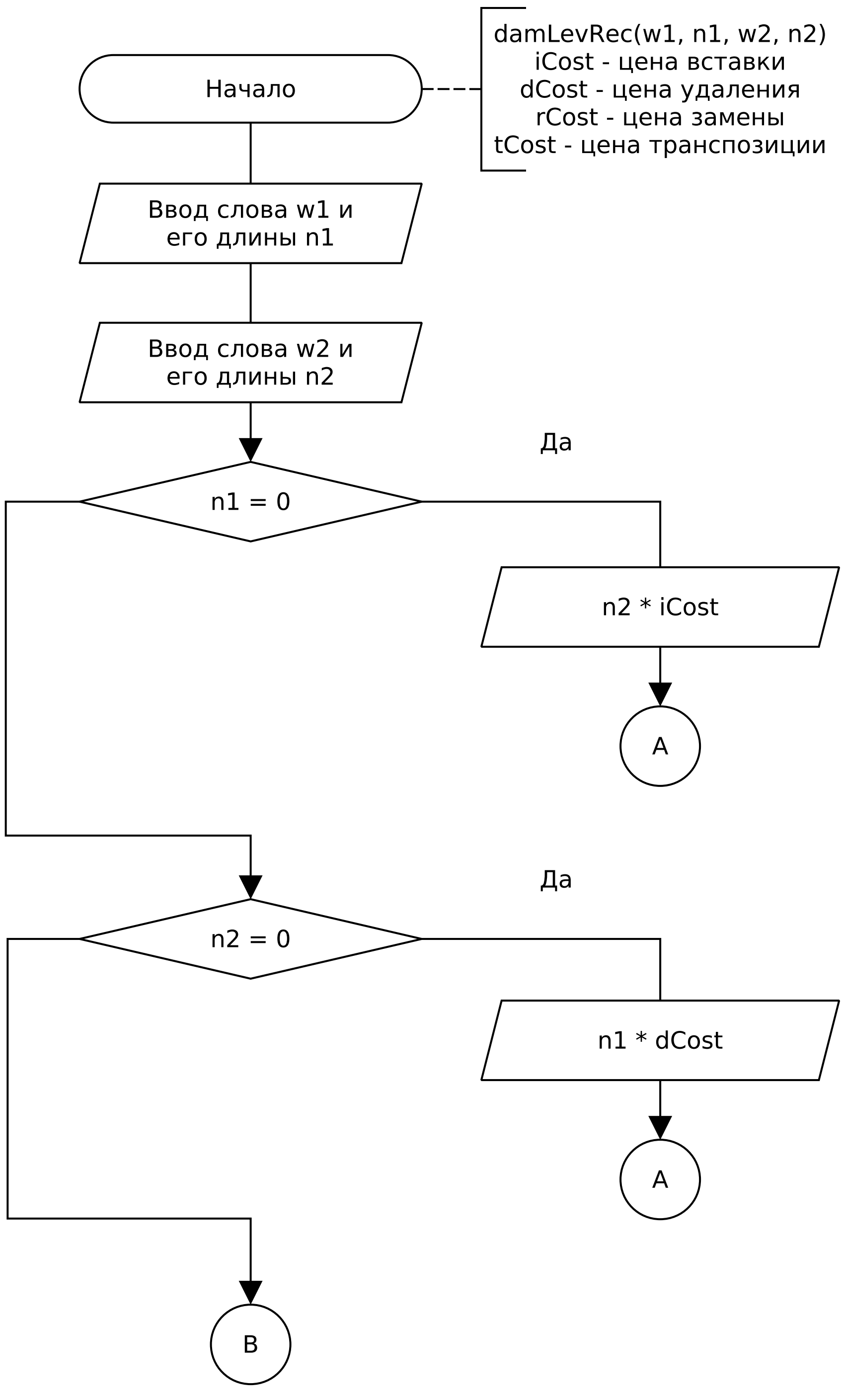

Рисунок 2.5 — Рекурсивный алгоритм Дамерау-Левенштейна, часть 1

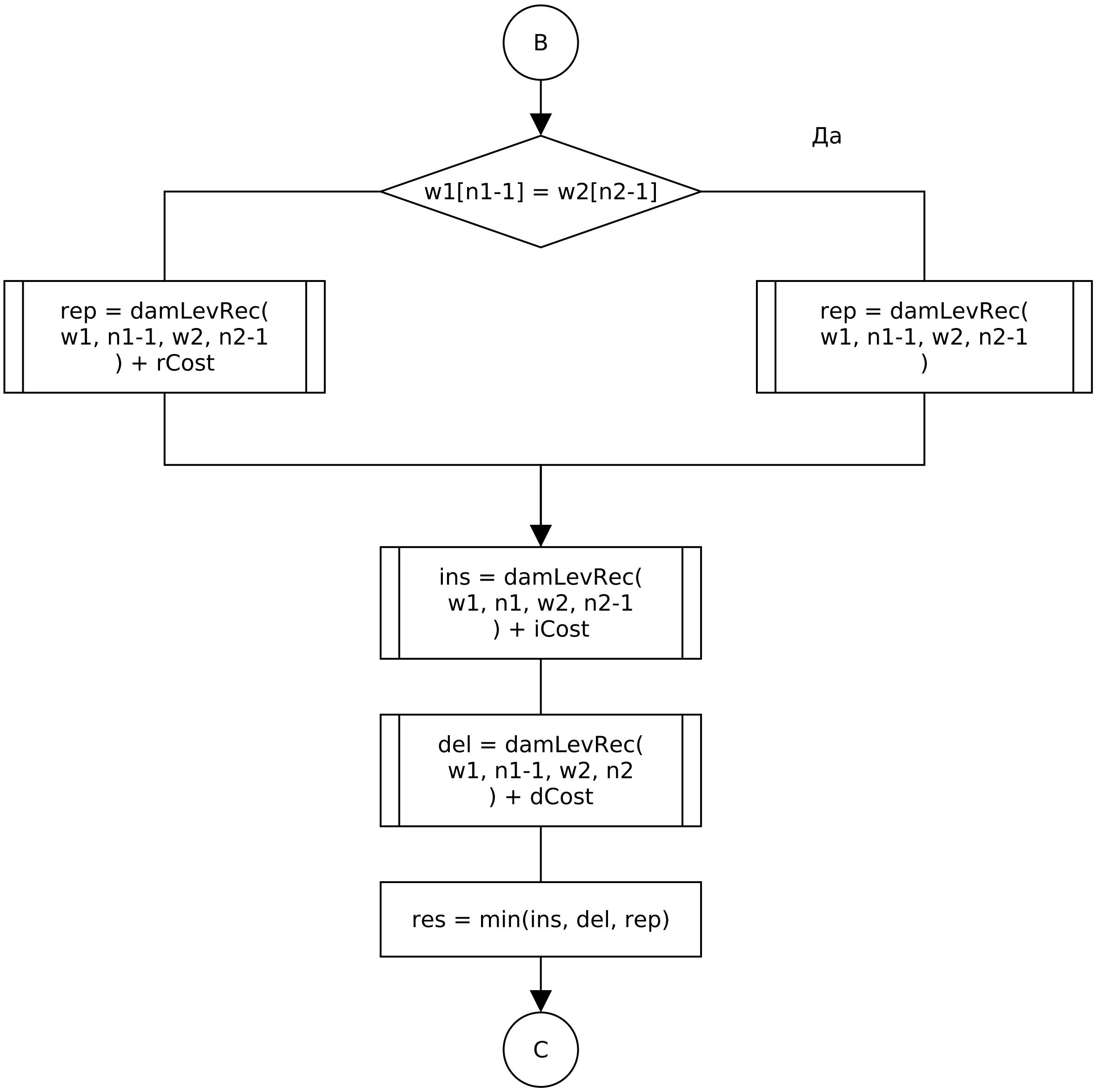

Рисунок 2.6 — Рекурсивный алгоритм Дамерау-Левенштейна, часть 2

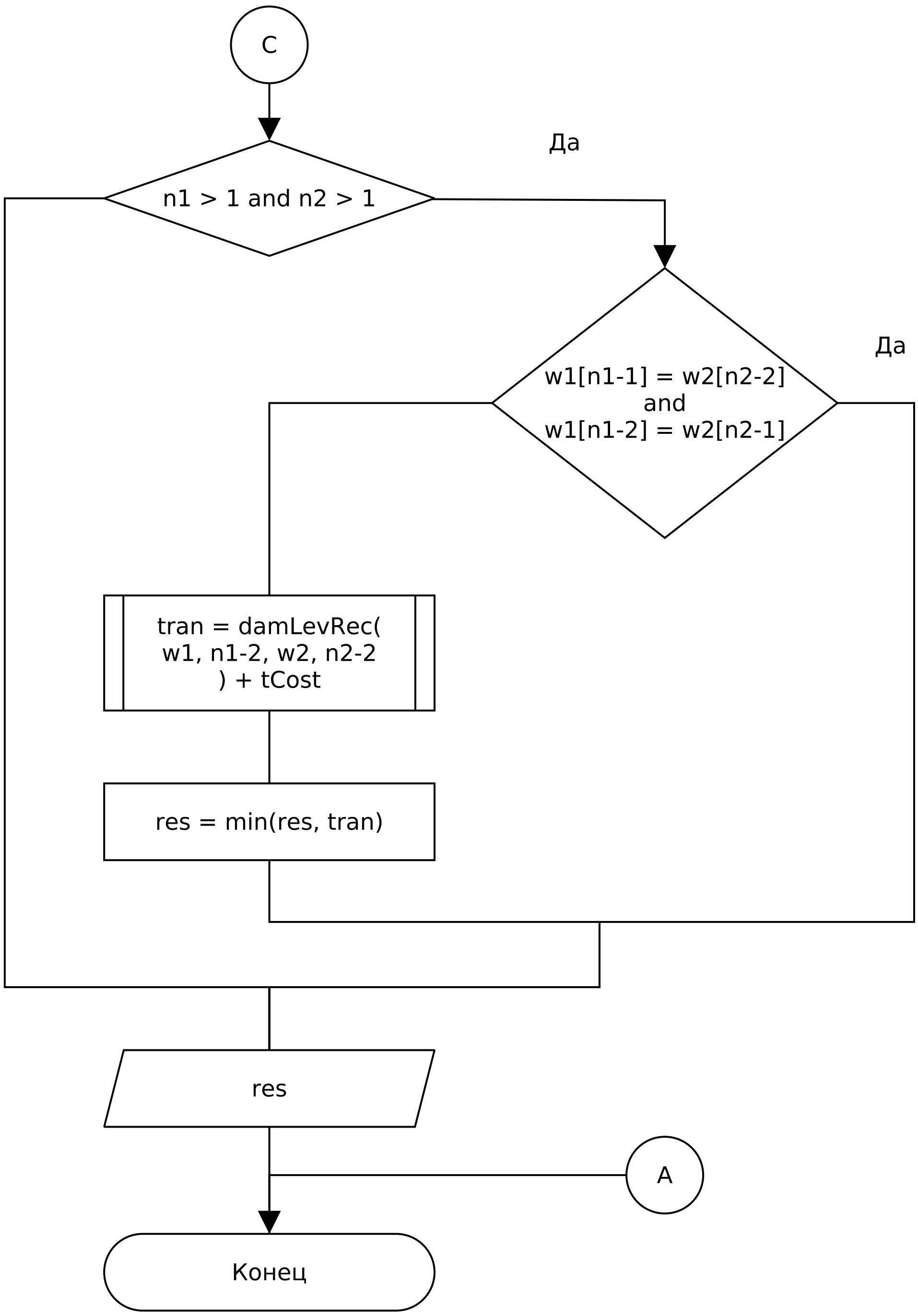

Рисунок 2.7 — Рекурсивный алгоритм Дамерау-Левенштейна, часть 3

2.3 Сравнительный анализ реализаций

Алгоритм Дамерау-Левенштейна был рассмотрен в двух вариациях: рекурсивной и матричной. Сравним эти реализации.

2.3.1 Оценка сложности

Произведем оценку общей сложности рекурсивного алгоритма. В рассмотренной реализации присутствуют 4 точки входа в рекурсию. Условием выхода из рекурсии является равенство длины хотя бы одной из строк нулю. Из этого можно сделать вывод, что рекурсивная вариация алгоритма Дамерау-Левенштейна имеет сложность $O(4^{max(n_1,n_2)})$, где n_1 и n_2 - длины обрабатываемых строк.

Что касается матричной реализации, её задача сводится к полному обходу матрицы размера $(n_1+1)\cdot(n_2+1)$, где n_1 и n_2 - длины обрабатываемых строк. Следовательно, данный алгоритм имеет сложность $O(n_1 \cdot n_2)$.

2.3.2 Оценка памяти

Память, затрачиваемая на выполнение рассматриваемых алгоримтов зависит от используемых типов данных и соглашении о вызовах. В качестве примера, будем считать, что обрабатываемые строки передаются в функции по указателю, размер одного символа составляет 1 байт, целочисленного типа - 4 байта, указателя - 8 байт, используется cdecl (параметры функции передаются через стек, в который так же помещаются значения адреса возврата и указателя на верхушку текущего стекового кадра).

Как было показано ранее, в рекурсивной реализации происходит $4^{max(n_1,n_2)}$ вызовов функции. Допустим, что в аргументах функции передаются два указателя на обрабатываемые строки и 2 целочисленные переменные, означающие длины этих строк. Без учета возможного использования локальных переменных, имеем следуюшие затраты памяти:

$$4^{max(n_1,n_2)} \cdot (8+8+8+4+8+4) = 10 \cdot 4^{max(n_1,n_2)+1} \tag{2.1}$$

В случае матричной вариации алгоритма Дамерау-Левенштейна будем считать, что функция использует матрицу размера $(n_1+1)\cdot(n_2+1)$, указатель на начало этой матрицы и два целочисленных счётчика для циклов. Тогда имеем следующие затраты памяти:

$$\begin{aligned}
&40 + (n_1 + 1) \cdot (n_2 + 1) \cdot 4 + 8 + 4 + 4 = \\
&56 + 4 \cdot (n_1 + 1) \cdot (n_2 + 1)
\end{aligned} \tag{2.2}$$

2.3.3 Итог

Таким образом, можно сделать вывод о том, что матричная реализация алгоритма Дамерау-Левенштейна работает быстрее и потребляет меньше памяти, чем рекурсивная.

3 Технологический раздел

3.1 Требования к программному обеспечению

3.2 Средства реализации

3.3 Листинги кода

Листинг 3.1 — Расстояние Левенштейна (матричная реализация)

```
template<typename _Word_t>
int WagnerFischer<_Word_t>::distance(_Word_t w1, int n1,
                                     _Word_t w2, int n2) {
    int **D = Util::createMatrix<int>(n1 + 1, n2 + 1);
    D[0][0] = 0;

    for (int j = 1; j <= n2; j++) {
        D[0][j] = D[0][j - 1] + insertCost;
    }

    for (int i = 1; i <= n1; i++) {
        D[i][0] = D[i - 1][0] + deleteCost;

        for (int j = 1; j <= n2; j++) {
            if (w1[i - 1] == w2[j - 1]) {
                D[i][j] = D[i - 1][j - 1];
            }
            else {
                D[i][j] = std::min(std::min(D[i - 1][j] + deleteCost,
                                            D[i][j - 1] + insertCost),
                                   D[i - 1][j - 1] + replaceCost);
            }
        }
    }

    int res = D[n1][n2];
    delete[] D;

    return res;
}
```

Листинг 3.2 — Расстояние Дамерау-Левенштейна (матричная реализация)

```
template<typename _Word_t>
int DamerauLevenshtein<_Word_t>::distance(_Word_t w1, int n1,
                                          _Word_t w2, int n2) {
    int **D = Util::createMatrix<int>(n1 + 1, n2 + 1);
    D[0][0] = 0;

    for (int j = 1; j <= n2; j++) {
        D[0][j] = D[0][j - 1] + insertCost;
    }

    for (int i = 1; i <= n1; i++) {
        D[i][0] = D[i - 1][0] + deleteCost;

        for (int j = 1; j <= n2; j++) {
            if (w1[i - 1] == w2[j - 1]) {
                D[i][j] = D[i - 1][j - 1];
            }
            else {
                D[i][j] = std::min(std::min(D[i - 1][j] + deleteCost,
                                            D[i][j - 1] + insertCost),
                                   D[i - 1][j - 1] + replaceCost);

                if (i > 1 && j > 1) {
                    if (w1[i - 1] == w2[j - 2] && w1[i - 2] == w2[j - 1]) {
                        D[i][j] = std::min(D[i - 2][j - 2] + transposeCost,
                            D[i][j]);
                    }
                }
            }
        }
    }

    int res = D[n1][n2];
    delete[] D;

    return res;
}
```

Листинг 3.3 — Расстояние Дамерау-Левенштейна (рекурсивная реализация)

```
template<typename _Word_t>
int DamerauLevenshteinRecursive<_Word_t>::distance(_Word_t w1, int n1,
                                                   _Word_t w2, int n2) {
    if (n1 == 0) {
        return n2 * insertCost;
    }
    if (n2 == 0) {
        return n1 * deleteCost;
    }

    bool isSame = w1[n1 - 1] == w2[n2 - 1];

    int res = std::min(std::min(distance(w1, n1 - 1, w2, n2) + deleteCost,
                                distance(w1, n1, w2, n2 - 1) + insertCost),
                       distance(w1, n1 - 1, w2, n2 - 1) + replaceCost *
                           !isSame);

    if (n1 > 1 && n2 > 1) {
        if (w1[n1 - 1] == w2[n2 - 2] && w1[n1 - 2] == w2[n2 - 1]) {
            res = std::min(distance(w1, n1 - 2, w2, n2 - 2) +
                transposeCost, res);
        }
    }

    return res;
}
```

3.4 Описание тестирования

4 Исследовательский раздел

ЗАКЛЮЧЕНИЕ