МИНОБРНАУКИ РОСИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания №1.1

Тема:

Оценка вычислительной сложности алгоритма

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Павлов Н.С.

Группа: ИКБО-30-23

Москва 2024

СОДЕРЖАНИЕ

ЦЕЛЬ РАБОТЫ

Приобретение практических навыков:

— эмпирическому определению вычислительной сложности алгоритмов на теоретическом и практическом уровнях;

— выбору эффективного алгоритма решения вычислительной задачи из нескольких.

1 ЗАДАНИЕ №1

1.1 ФОРМУЛИРОВКА ЗАДАЧИ

Пусть имеется вычислительная задача:

– дан массив **x** из **n** элементов целого типа; удалить из этого массива все значения равные заданному (ключевому) **key**.

Необходимо выбрать эффективный алгоритм вычислительной задачи из двух предложенных, используя теоретическую и практическую оценку вычислительной сложности каждого из алгоритмов, а также его ёмкостную сложность.

1.2 ПЕРВЫЙ АЛГОРИТМ

1.2.1 Математическая модель решения задачи

Описание действий алгоритма:

Переменой целого типа i присваивается значение 0. Пока значение i меньше размера массива x (n) выполняется следующий цикл:

1. Очередной i-й элемент массива x сравнивается с ключевым значением key
2. Если эти значения равны, то выполняется смещение элементов массива x на 1 позицию влево, начиная с (i+1) элемента. Затем размер массива x уменьшается на 1. Иначе переменная i увеличивается на 1.

Подробный порядок действий алгоритма обозначен на блок-схеме (рис.1).

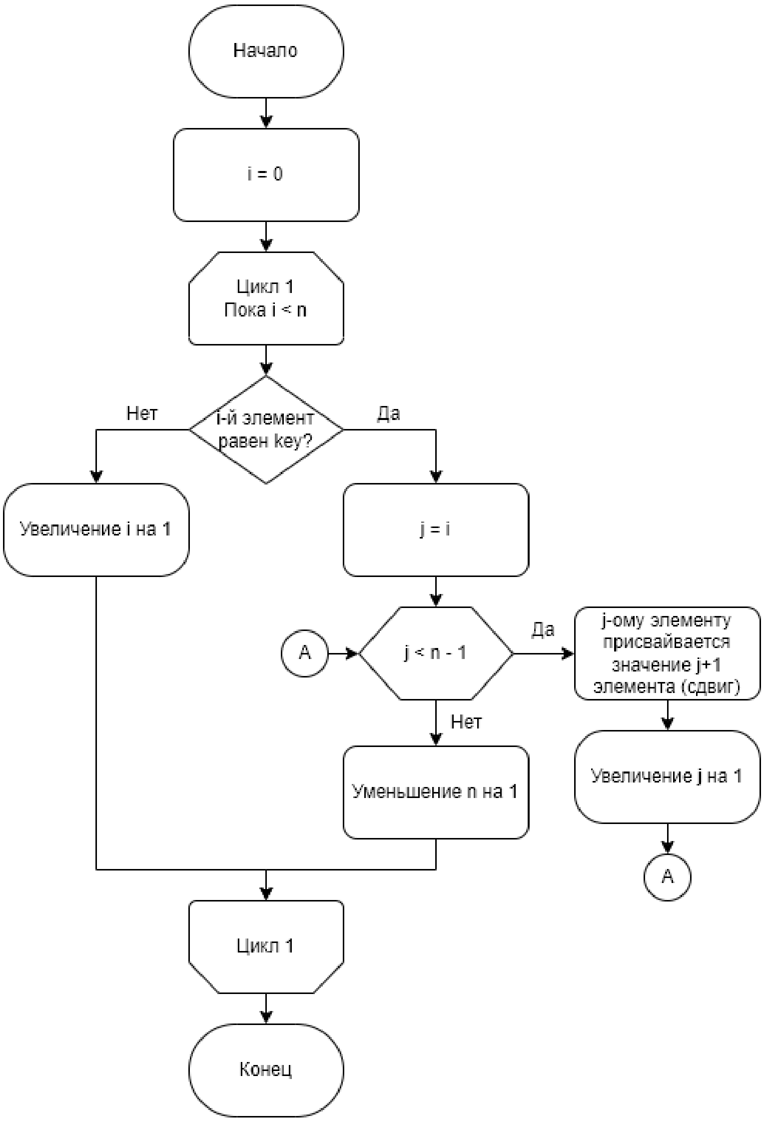

Рисунок 1 – Блок-схема первого алгоритма

Доказательство корректности цикла(внешнего):

Определим инвариант цикла: «В массиве x среди первых i элементов нет значений равных контрольному (key)»

Область изменения параметров этой задачи [x[1]; x[n]] можно разделить на две части:

— исследованную область, для которой нет элементов равных контрольному значению в [x[1]; x[i]];

— область неопределенности [x[i+1]; x[n]].

Необходимо составлять цикл так, чтобы на каждой итерации область неопределенности сокращалась, это, собственно, и будет доказывать, что условие завершения рано или поздно будет выполнено, т.е. цикл конечный:

— перед началом цикла исследованная область представляет собой пустой массив, а область неопределенности составляет [x[1]; x[n]]

— после первого выполнения цикла может быть 2 варианта. Если первый элемент был равен контрольному значению, то элемент удаляется и размер массива становится на 1 меньше. Поэтому исследованная область остается прежней, а область неопределенности [x[1]; x[n]], но n на 1 меньше исходного значения.. Если же первый элемент не равен контрольному значению, то исследованная область состоит из одного элемента, а область неопределенности [x[2]; x[n]].

— после очередного i-ого выполнения цикла может быть 2 варианта. Если i-й элемент был равен контрольному значению, то элемент удаляется и размер массива становится на 1 меньше. Поэтому исследованная область остается прежней [x[1]; x[i-1]], а область неопределенности [x[i]; x[n]], но n на 1 меньше предыдущего значения.. Если же первый элемент не равен контрольному значению, то исследованная область состоит из i элементов, а область неопределенности [x[i+1]; x[n]]. И так до тех пор, пока область неопределенности не превратится в пустое множество

Таким образом, выражение инварианта сохраняет свою истинность на протяжении всего цикла, а это значит, что цикл является корректным.

Определение вычислительной сложности алгоритма:

Определим вычислительную сложность алгоритма используя теоретический подход, т.е. выведем функцию роста T(n) отдельно для худшего и лучшего случаев.

Таблица 1 – Таблица подсчета кол-ва выполнений оператора в строке

Номер оператора	Оператор	Кол-во выполнений оператора в строке	
		В лучшем случае	В худшем случае
1	int i = 0;	1	1
2	while (i < n) {	n+1	n+1
3	if (x[i] == key) {	n	n
4	for (int j = i; j < n - 1; j++) {	0	(n-1)/2*n+1
5	x[j] = x[j + 1];	0	(n-1)/2*n
6	}		
7	n--;	0	n
8	}		
9	else {		
10	i++;	n	0
11	}		
12	}		

В лучшем случае: в массиве не содержится элементов, значение которых равняется контрольному. Условие строки 3 будет ложным, поэтому выполняется оператор 10 строки n раз.

$$T(n) = 1 + (n + 1) + n + n = 3n + 2 \quad (1)$$

В худшем случае: каждый элемент массива равен контрольному значению. Выполняются операторы 4 и 5 строки, а оператор 10 строки не выполнится ни разу.

$$T(n) = 1 + (n+1) + \left(\frac{n-1}{2} * n + 1\right) + \left(\frac{n-1}{2} * n\right) + n = \\ = n^2 + n + 3 \quad (2)$$

В среднем случае: ровно половина элементов массива равняется контрольному значению. Тогда условие строки 3 выполнится n/2 раз и столько же не выполнится.

$$T(n) = 1 + (n+1) + \left(\frac{n-1}{2} * \frac{n}{2} + 1\right) + \left(\frac{n-1}{2} * \frac{n}{2}\right) + \frac{n}{2} + \frac{n}{2} = \\ = \frac{n^2}{2} + \frac{3n}{2} + 3 \quad (3)$$

В лучшем случае скорость роста получается линейная, когда как в худшем и в среднем скорость роста квадратичная. Значит, в целом, скорость роста – квадратичная.

1.2.2 Реализация алгоритма

Реализуем алгоритм в виде функции на языке программирования C++, включив в него подсчет суммарного количества выполненных сравнений (sravn) и перемещений (del) элементов при решении задачи удаления ключевых элементов (рис. 2).

```
void delFirstMetod(int* x, int& n, int key, int& sravn, int& del)
{
    int i = 0;
    while (i < n) {
        sravn++;
        if (x[i] == key) {
            for (int j = i; j < n - 1; j++) {
                sravn++;
                x[j] = x[j + 1];
                del++;
            }
            sravn++;
            n--;
        }
        else {
            i++;
        }
        sravn++;
    }
    sravn++;
}
```

Рисунок 2 – Реализация алгоритма на языке C++

Добавим еще несколько функций (рис. 3): заполнение массива случайными числами и вывод массива на экран монитора

```
void mass_input(int* x, int n) //заполнение случайными числами
{
    srand(time(0));
    for (int i = 0; i < n; i++) {
        x[i] = rand() % 5;
    }
}

void output(int* x, int n) //вывод массива на экран
{
    for (int i = 0; i < n; i++) {
        cout << x[i] << " ";
    }
    cout << endl;
}
```

Рисунок 3 – Функции заполнения массива и вывода его на экран

1.2.3 Тестирование алгоритма

Проведем тестирование алгоритма для следующих ситуаций:

— случайной заполнение массива (рис. 4)

— ни один элемент массива не удаляется (рис. 5)

— все элементы массива удаляются (рис. 6)

```
Введите n: 10
Введите key: 2
Исходный массив: 3 4 2 2 1 0 2 0 0 2
Результат удаления: 3 4 1 0 0 0
Колличество сравнений: 41
Колличество перемещений: 16
```

Рисунок 4 – Тестирование на случайных значениях

```
Введите n: 10
Введите key: 15
Исходный массив: 0 0 2 3 2 0 0 1 0 1
Результат удаления: 0 0 2 3 2 0 0 1 0 1
Колличество сравнений: 21
Колличество перемещений: 0
```

Рисунок 5 – Тестирование с сохранением всего массива

```
Введите n: 10
Введите key: 2
Исходный массив: 2 2 2 2 2 2 2 2 2 2
Результат удаления:
Колличество сравнений: 76
Колличество перемещений: 45
```

Рисунок 6 – Тестирование с удалением всего массива

В результате тестирования мы получили, что на практике вычислительная сложность алгоритма равняется: в лучшем случае 21, в худшем случае 121, в усредненном случае 57.

Высчитаем теоретические значения из формул 1-3 при n=10 и сравним из с фактическими значениями.

Лучший случай: $3 * 10 + 2 = 32$; на практике 21

Худший случай: $10^2 + 10 + 3 = 123$; на практике 121

Усредненный случай: $\frac{10^2}{2} + \frac{3*10}{2} + 3 = 68$; на практике 57

1.2.4 Оценка емкостной сложности алгоритма

В реализации алгоритма участвуют переменные двух типов: переменная целого типа (int) и указатель на переменную целого типа (int*). Определим количество выделяемой памяти на каждый тип переменных с помощью функции sizeof() для конкретной системы (рис. 7).

Рисунок 7 – количество памяти, занимаемой переменными в конкретной системе

Перечислим все переменные, одновременно существующие в алгоритме, и размер в памяти, занимаемый ими (табл. 2).

Таблица 2 – Таблица определения размера переменных

Имя переменной	Тип переменной	Занимаемая память в байтах
i	int	4
j	int	4
n	int	4
key	int	4
x	int*	8
x[1]…x[n]	Массив int	4*n

Сложив значения третьего столбца, получаем, что емкостная сложность алгоритма: 4+4+4+4+8+4*n = 24+4n байт

1.3 ВТОРОЙ АЛГОРИТМ

1.3.1 Математическая модель решения задачи

Описание действий алгоритма:

Переменным целого типа j и i присваивается значение 0. Пока значение i меньше размера массива x (n) выполняется следующий цикл:

1. j-ому элементу массива присваивается значение i-ого элемента.
2. Если значение i-ого элемента массива не равняется контрольному значению key, то j увеличивается на 1.

Размер массива уменьшается до значения j.

Подробный порядок действий алгоритма обозначен на блок-схеме (рис.8).

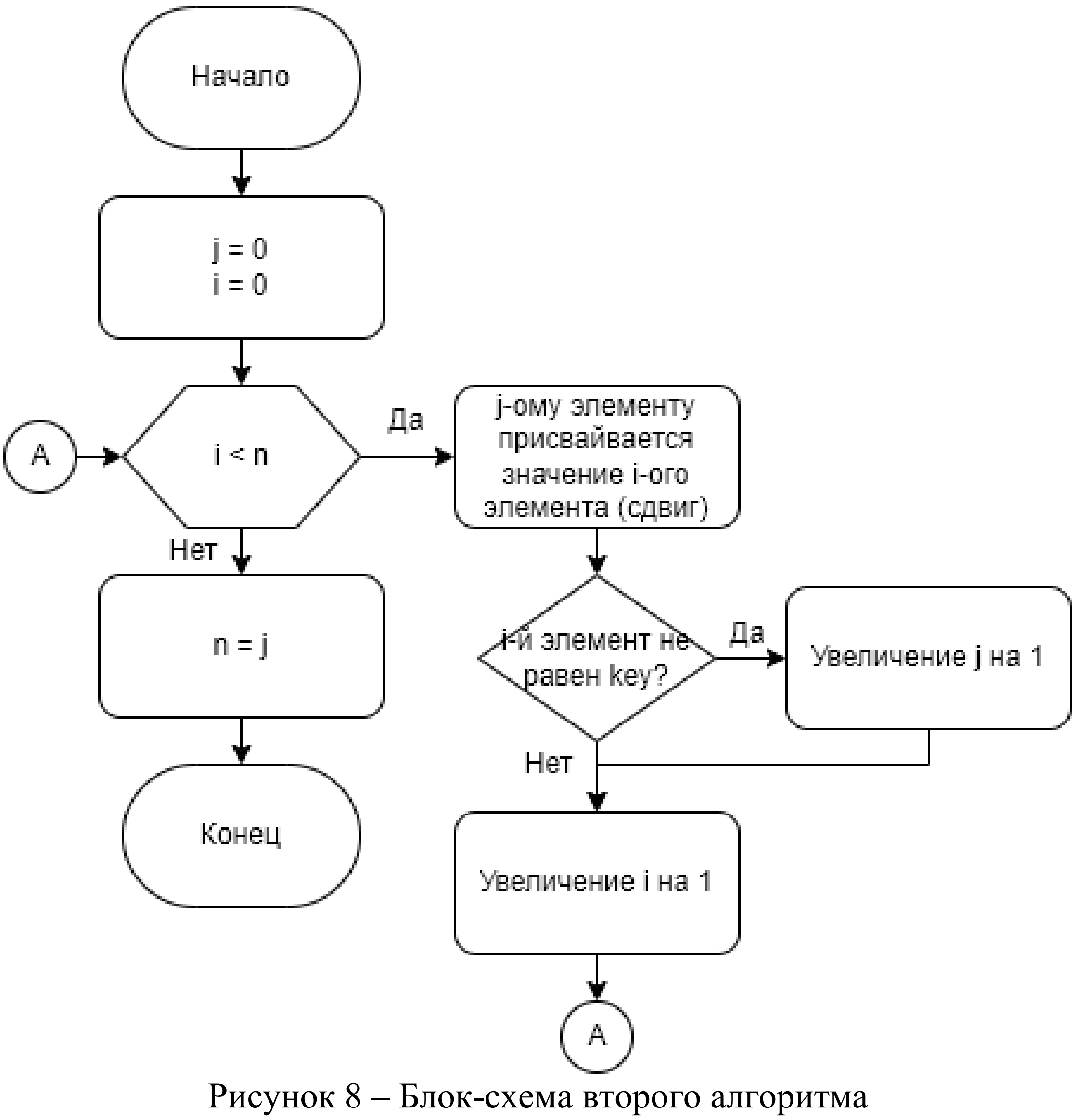

Рисунок 8 – Блок-схема второго алгоритма

Доказательство корректности цикла(внешнего):

Определим инвариант цикла: «В массиве x среди первых j элементов нет значений равных контрольному (key)»

Область изменения параметров этой задачи [x[1]; x[n]] можно разделить на две части:

— исследованную область, для которой нет элементов равных контрольному значению в [x[1]; x[j]];

— область «мусора» (значений, не имеющих роли) [x[j+1]; x[i]]

— область неопределенности [x[i+1]; x[n]].

Необходимо составлять цикл так, чтобы на каждой итерации область неопределенности сокращалась, это, собственно, и будет доказывать, что условие завершения рано или поздно будет выполнено, т.е. цикл конечный:

— перед началом цикла исследованная область и область «мусора» представляют собой пустой массив, а область неопределенности составляет [x[1]; x[n]]

— после первого выполнения цикла может быть 2 варианта. Если первый элемент был равен контрольному значению, то исследованная область остается пустой, область «мусора» состоит из одного элемента, а область неопределенности [x[2]; x[n]]. Если же первый элемент не равен контрольному значению, то исследованная область состоит из одного элемента, область «мусора» пустая, а область неопределенности [x[2]; x[n]].

— после очередного i-ого выполнения цикла может быть 2 варианта. Если i-й элемент был равен контрольному значению, то исследованная область остается прежней, область «мусора» [x[j]; x[i]], а область неопределенности [x[i+1]; x[n]]. Если же первый элемент не равен контрольному значению, то исследованная область состоит из j элементов, область «мусора» [x[j+1]; x[i]], а область неопределенности [x[i+1]; x[n]]. И так до тех пор, пока область неопределенности не превратится в пустое множество

Таким образом, выражение инварианта сохраняет свою истинность на протяжении всего цикла, а это значит, что цикл является корректным.

Определение вычислительной сложности алгоритма:

Определим вычислительную сложность алгоритма используя теоретический подход, т.е. выведем функцию роста T(n) отдельно для худшего и лучшего случаев.

Таблица 3 – Таблица подсчета кол-ва выполнений оператора в строке

Номер оператора	Оператор	Кол-во выполнений оператора в строке	
		В лучшем случае	В худшем случае
1	int j = 0;	1	1
2	for (int i = 0; i < n; i++) {	n+1	n+1
3	x[j] = x[i];	n	n
4	if (x[i] != key) {	n	n
5	j++;	0	n
6	}		
7	}		
8	n = j;	1	1

В лучшем случае: все значения в массиве равняются контрольному значению. Тогда строка 5 будет игнорироваться.

$$T(n) = 1 + (n + 1) + n + n + 1 = 3n + 3 \qquad (4)$$

В худшем случае: ни один из элементов массива не равен контрольному значению. Тогда строка 5 будет выполняться каждый раз, т.е. n раз.

$$T(n) = 1 + (n + 1) + n + n + n + 1 = 4n + 3 \qquad (5)$$

В среднем случае: ровно половина элементов массива равняется контрольному значению. Тогда условие строка 5 выполнится n/2 раз и столько же не выполнится.

$$T(n) = 1 + (n + 1) + n + n + \frac{n}{2} + 1 = \frac{7n}{2} + 3 \qquad (6)$$

Мы видим, что во всех трех случаях скорость роста – линейная. Значит, в целом, скорость роста – линейная.

1.3.2 Реализация алгоритма

Реализуем алгоритм в виде функции на языке программирования C++, включив в него подсчет суммарного количества выполненных сравнений (sravn) и перемещений (del) элементов при решении задачи удаления ключевых элементов (рис. 9).

```cpp
void delOtherMetod(int* x, int& n, int key, int& sravn, int& del)
{
    int j = 0;
    for (int i = 0; i < n; i++) {
        sravn++;
        x[j] = x[i];
        del++;
        if (x[i] != key) {
            j++;
        }
        sravn++;
    }
    sravn++;
    n = j;
}
```

Рисунок 9 – Реализация алгоритма на языке C++

Добавим еще несколько функций (рис. 10): заполнение массива случайными числами и вывод массива на экран монитора

```cpp
void mass_input(int* x, int n) //заполнение случайными числами
{
    srand(time(0));
    for (int i = 0; i < n; i++) {
        x[i] = rand() % 5;
    }
}

void output(int* x, int n) //вывод массива на экран
{
    for (int i = 0; i < n; i++) {
        cout << x[i] << " ";
    }
    cout << endl;
}
```

Рисунок 10 – Функции заполнения массива и вывода его на экран

1.3.3 Тестирование алгоритма

Проведем тестирование алгоритма для следующих ситуаций:

— случайной заполнение массива (рис. 11)

— ни один элемент массива не удаляется (рис. 12)

— все элементы массива удаляются (рис. 13)

```
Введите n: 10
Введите key: 2
Исходный массив: 0 2 3 2 4 2 2 2 2 4
Результат удаления: 0 3 4 4
Колличество сравнений: 21
Колличество перемещений: 10
```

Рисунок 11 – Тестирование на случайных значениях

```
Введите n: 10
Введите key: 15
Исходный массив: 3 2 4 0 2 4 1 2 1 1
Результат удаления: 3 2 4 0 2 4 1 2 1 1
Колличество сравнений: 21
Колличество перемещений: 10
```

Рисунок 12 – Тестирование с сохранением всего массива

```
Введите n: 10
Введите key: 2
Исходный массив: 2 2 2 2 2 2 2 2 2 2
Результат удаления:
Колличество сравнений: 21
Колличество перемещений: 10
```

Рисунок 13 – Тестирование с удалением всего массива

В результате тестирования мы получили, что на практике вычислительная сложность алгоритма равняется: в лучшем случае 31, в худшем случае 31, в усредненном случае 31. Фактически получается, что данный алгоритм имеет одинаковую вычислительную сложность вне зависимости от исходной ситуации.

Высчитаем теоретические значения из формул 4-6 при n=10 и сравним из с фактическими значениями.

Лучший случай: $3 * 10 + 3 = 33$; на практике 31

Худший случай: $4 * 10 + 3 = 43$; на практике 31

Усредненный случай: $\frac{7*10}{2} + 3 = 38$; на практике 31

1.3.4 Оценка емкостной сложности алгоритма

В реализации алгоритма участвуют переменные двух типов: переменная целого типа (int) и указатель на переменную целого типа (int*). Определим количество выделяемой памяти на каждый тип переменных с помощью функции sizeof() для конкретной системы (рис. 14).

Рисунок 14 – количество памяти, занимаемой переменными в конкретной системе

Перечислим все переменные, одновременно существующие в алгоритме, и размер в памяти, занимаемый ими (табл. 4).

Таблица 4 – Таблица определения размера переменных

Имя переменной	Тип переменной	Занимаемая память в байтах
i	int	4
j	int	4
n	int	4
key	int	4
x	int*	8
x[1]…x[n]	Массив int	4*n

Сложив значения третьего столбца, получаем, что емкостная сложность алгоритма: 4+4+4+4+8+4*n = 24+4n байт

1.4 ВЫВОДЫ

Основываясь на проделанном детальном анализе двух алгоритмов решения одной и той же задачи, можно заметить, что оба алгоритма имеют одинаковую емкостную сложность. При этом второй алгоритм имеет гораздо меньшую вычислительную сложность, нежели чем первый (у первого – квадратичная, у второго – линейная).

Из этого можно сделать вполне обоснованный вывод о том, что второй алгоритм решения задачи является более эффективным.

2 ЗАДАНИЕ №2

2.1 ПОСТАНОВКА ЗАДАЧИ

Выполнить разработку алгоритма в соответствии с задачей варианта, включив в отчёт описание следующих этапов:

1) Формулировка задачи.
2) Математическая модель решения
3) Разработка эффективного алгоритма
4) Реализовать алгоритм решения задачи варианта в виде одной функции (без декомпозиции на другие функции).
5) Провести тестирование алгоритма. Для этого разработать таблицу тестов в соответствии с ограничениями постановки задачи. Выполнить тестовые прогоны и убедиться, что все требования выполняются.
6) Выполнить практическую оценку сложности алгоритма для больших n (по счётчикам количества сравнений и перемещений данных в памяти). Показать результаты прогонов для заданного n в лучшем, худшем и среднем случаях.
7) Оценить ёмкостную сложность алгоритма.

Персональный вариант: Умножение квадратных матриц

2.2 РАБОТА С АЛГОРИТМОМ

2.2.1 Математическая модель решения алгоритма

Запустим 3 цикла, вложенных друг в друга: внешний (переменная i меняет значение от 1 до n) для выбора строки матрицы результата, промежуточный (переменная j меняет значение от 1 до n) для выбора элемента в строке матрицы результата и внутренний (переменная k меняет значение от 1 до n) для перебора элементов исходных матриц.

Во внутреннем цикле j-ому элементу i-ой строки массива результата присваивается сумма k-ого элемента i-ой строки первого массива и j-ого элемента k-ой строки второго массива.

Подробный порядок действий алгоритма обозначен на блок-схеме (рис.15).

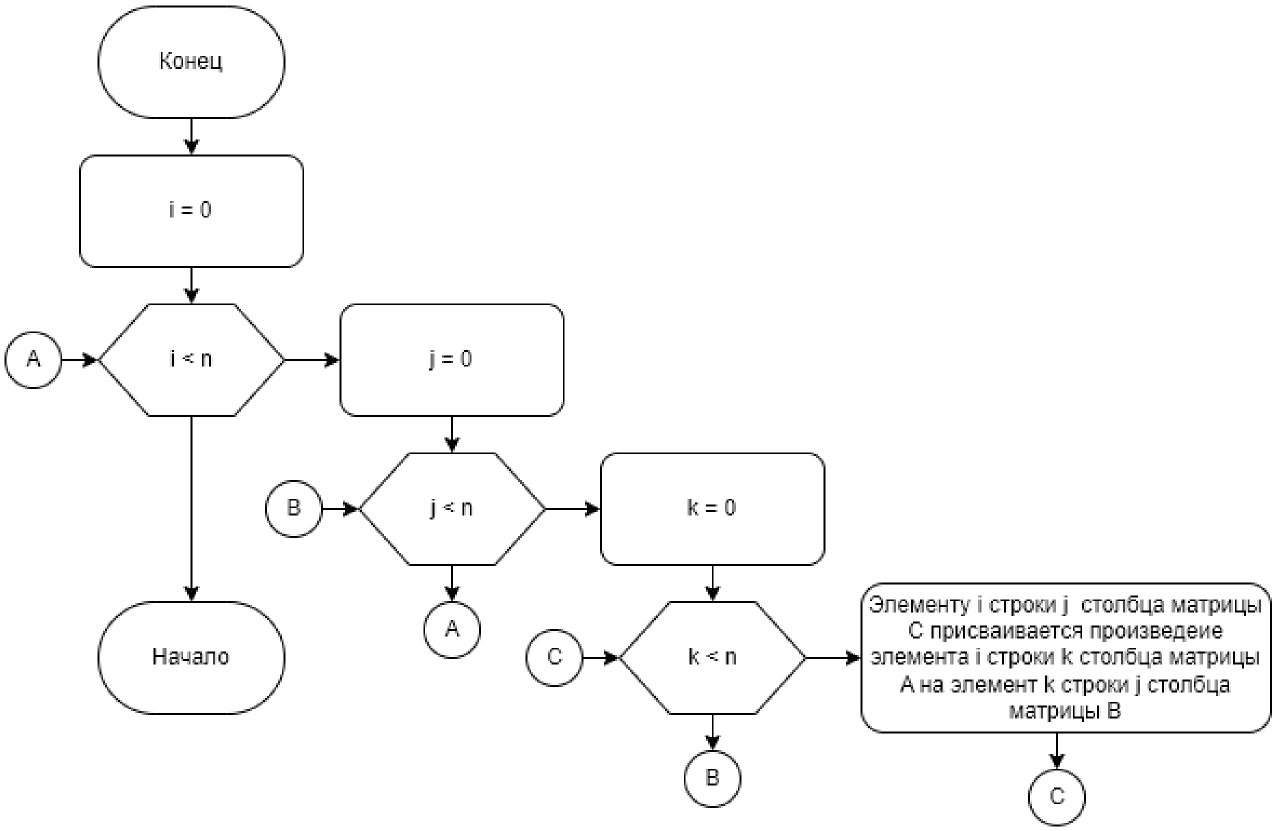

Рисунок 15 – Блок-схема алгоритма

Доказательство корректности цикла(внешнего):

Определим инвариант цикла: «В матрице x первые k элементов получены из произведения i-ой строки исходной матрицы 1 на j-й столбец исходной матрицы 2»

Область изменения параметров этой задачи [x[1]; $x[n^2]$] можно разделить на две части:

— исследованную область, для которой посчитаны элементы из [x[1]; x[k]];

— область неопределенности [x[k+1]; $x[n^2]$].

Необходимо составлять цикл так, чтобы на каждой итерации область неопределенности сокращалась, это, собственно, и будет доказывать, что условие завершения рано или поздно будет выполнено, т.е. цикл конечный:

— перед началом цикла исследованная область представляют собой пустой массив, а область неопределенности составляет [x[1]; $x[n^2]$]

— после первого выполнения цикла исследованная область состоит из одного элемента, а область неопределенности [x[2]; $x[n^2]$].

— после очередного k-ого выполнения цикла исследованная область – [x[1]; x[k]], а область неопределенности [x[k+1]; $x[n^2]$]. И так до тех пор, пока область неопределенности не превратится в пустое множество

Таким образом, выражение инварианта сохраняет свою истинность на протяжении всего цикла, а это значит, что цикл является корректным.

Определение вычислительной сложности алгоритма:

Определим вычислительную сложность алгоритма используя теоретический подход, т.е. выведем функцию роста T(n) отдельно для худшего и лучшего случаев.

Таблица 5 – Таблица подсчета кол-ва выполнений оператора в строке

Номер оператора	Оператор	Кол-во выполнений оператора в строке	
		В лучшем случае	В худшем случае
1	for (int i = 0; i < n; i++) {	n+1	n+1
2	for (int j = 0; j < n; j++) {	(n+1)n	(n+1)n
3	for (int k = 0; k < n; k++) {	$(n+1)n^2$	$(n+1)n^2$
4	matrix_C[i][j] += matrix_A[i][k] * matrix_B[k][j];	n^3	n^3
5	}		
6	}		
7	}		

В алгоритме решения задачи не существует худшего или лучшего решения. В алгоритме выполняется одинаковое количество действий для матриц одного размера, т.к. сложность такого алгоритма зависит только от количества исходных элементов, а не от их значения:

$$T(n) = (n+1) + (n+1)n + (n+1)n^2 + n^3 = 2n^3 + 2n^2 + 2n + 1 \quad (7)$$

Исходя из функции роста, очевидно, что сложность алгоритма – кубическая.

2.2.2 Разработка эффективного алгоритма

Реализуем алгоритм в виде функции на языке программирования C++, включив в него подсчет суммарного количества выполненных сравнений и перемещений (count) элементов при решении задачи умножения квадратных матриц (рис. 16).

```cpp
void multiply(int** matrix_A, int** matrix_B, int** matrix_C, int n, int& count)
{
    for (int i = 0; i < n; i++) {
        count++;
        for (int j = 0; j < n; j++) {
            count++;
            for (int k = 0; k < n; k++) {
                count++;
                matrix_C[i][j] += matrix_A[i][k] * matrix_B[k][j];
                count++;
            }
            count++;
        }
        count++;
    }
    count++;
}
```

Рисунок 16 – Реализация алгоритма на языке С++

Добавим еще несколько функций (рис. 17): заполнение массива случайными числами и вывод массива на экран монитора

```cpp
void input(int** matrix, int n)
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = rand() % 10;
        }
    }
}

void output(int** matrix, int n)
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << setw(3) << matrix[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}
```

Рисунок 17 – Функции заполнения массива и вывода его на экран

2.2.3 Тестирование алгоритма

Разработаем таблицу (табл. 6) тестов с ограничениями постановки задачи и убедимся, в правильности работы алгоритма.

Таблица 6 – Таблица тестов алгоритма

Входные данные	Ожидаемый результат	Результат работы программы
$\begin{bmatrix} 6 & 2 \\ 3 & 0 \end{bmatrix}; \begin{bmatrix} 0 & 2 \\ 3 & 7 \end{bmatrix}$	$\begin{bmatrix} 6 & 26 \\ 0 & 6 \end{bmatrix}$	Исходная матрица A: 6 2 3 0 Исходная матрица B: 0 2 3 7 Произведение матриц: 6 26 0 6 Сложность алгоритма: 29
$\begin{bmatrix} 3 & 7 & 3 \\ 3 & 7 & 3 \\ 7 & 9 & 5 \end{bmatrix}; \begin{bmatrix} 1 & 0 & 9 \\ 7 & 0 & 4 \\ 1 & 8 & 0 \end{bmatrix}$	$\begin{bmatrix} 55 & 24 & 55 \\ 55 & 24 & 55 \\ 75 & 40 & 99 \end{bmatrix}$	Исходная матрица A: 3 7 3 3 7 3 7 9 5 Исходная матрица B: 1 0 9 7 0 4 1 8 0 Произведение матриц: 55 24 55 55 24 55 75 40 99 Сложность алгоритма: 79
$[4]; [4]$	$[16]$	Исходная матрица A: 4 Исходная матрица B: 4 Произведение матриц: 16 Сложность алгоритма: 7

При тестировании мы получили, что при умножении матриц 3*3, практическая сложность алгоритма равняется 79. Вычислим теоретическое значение, подставив в формулу 7 значение n=3:

$$T(3) = 2 * 3^3 + 2 * 3^2 + 2 * 3 + 1 = 79$$

Оказалось, что теоретическое значение равняется практическому.

2.2.4 Оценка емкостной сложности алгоритма

В реализации алгоритма участвуют переменные двух типов: переменная целого типа (int) и указатели. Определим количество выделяемой памяти на каждый тип переменных с помощью функции sizeof() для конкретной системы (рис. 18).

```
int     4bytes
int*    8bytes
```

Рисунок 18 – количество памяти, занимаемой переменными в конкретной системе

Перечислим все переменные, одновременно существующие в алгоритме, и размер в памяти, занимаемый ими (табл. 7).

Таблица 7 – Таблица определения размера переменных

Имя переменной	Тип переменной	Занимаемая память в байтах
matrix_A	int*	8
Matrix_B	int*	8
Matrix_C	int*	8
n	int	4
i	int	4
j	int	4
k	int	4
matrix_A[i]	Массив int*	8*n
matrix_B[i]	Массив int*	8*n
matrix_C[i]	Массив int*	8*n
matrix_A[i][j]	Массив int	4*n
matrix_B[i][j]	Массив int	4*n
matrix_C[i][j]	Массив int	4*n

Сложив значения третьего столбца, получаем, что емкостная сложность алгоритма: 8*3+4*4+3*8*n+3*4*n = 40+36n байт

2.3 ВЫВОДЫ

Мною была проведена работа по разработке и реализации алгоритма решения задачи умножения квадратных матриц размерностью n. В ходе работы над алгоритмом была доказана его корректность, определена функция для расчета вычислительной сложности алгоритма и определена емкостная сложность алгоритма. Алгоритм протестирован и работает корректно.

СПИСОК ИНФОРМАЦТОННЫХ ИСТОЧНИКОВ

1. Рысин, М. Л. Введение в структуры и алгоритмы обработки данных. Часть 1. Сложность алгоритмов. Сортировки. Линейные структуры данных. Поиск в таблице. : учеб. пособие / М. Л. Рысин, М. В. Сартаков, М. Б. Туманова ; МИРЭА – Российский технологический университет, 2022. – 109 с. – ISBN 978-5-7339-1612-5.

2. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения : межгосударственный стандарт : дата введения 1992-01-01 / Федеральное агентство по техническому регулированию. – Изд. официальное. – Москва : Стандартинформ, 2010. – 23 с.