МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Отчет по выполнению практического задания №3

Тема: Эмпирический анализ алгоритмов сортировки

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент Антонов А.Д.

группа ИКБО-01-20

Москва 2021

Содержание

1. Задание 1: оценка эффективности алгоритма простых вставок 3
1.1. Алгоритм сортировки методом простой вставки 3
1.2. Функция роста времени выполнения сортировки методом простых вставок 4
1.3. Сводная таблица результатов выполнения сортировки 7
1.4. Код алгоритма и основной программы 8
1.5. Зависимость теоретической и практической сложности 10
1.6. Анализ результатов 12
2. Задание 2: оценка вычислительной сложности алгоритма в наихудшем и наилучшем случаях 13
2.1. Тестирование на упорядоченном по возрастанию массиве 13
2.2. Тестирование на упорядоченном по убыванию массиве 14
2.3. Код программы 14
2.4. График зависимости вычислительных сложностей для 3 случаев 16
2.5. Емкостная сложность алгоритма от n 17
2.6. Анализ результатов 17
3. Задание 3: оценка эффективности алгоритма простого выбора 18
3.1. Алгоритм сортировки методом простого выбора 18
3 2. Функция роста времени выполнения сортировки методом простых вставок 19
3.3. Сводная таблица результатов выполнения сортировки 22
3.4. Код алгоритма и основной программы 22
3.5. График зависимости вычислительных сложностей для 2 случаев 24
3.6. Определение эффективности алгоритма 26
3.7. Анализ результатов 26
4. Ответы на вопросы 7-9 26
ВЫВОДЫ 29
СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ 29

1. Задание 1 (Вариант 1): оценка эффективности алгоритма простых вставок

1.1. Алгоритм сортировки методом простой вставки

Сортировка вставками (Insertion Sort) — это простой алгоритм сортировки. Суть его заключается в том, что на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Стоит отметить что массив из 1-го элемента считается отсортированным.

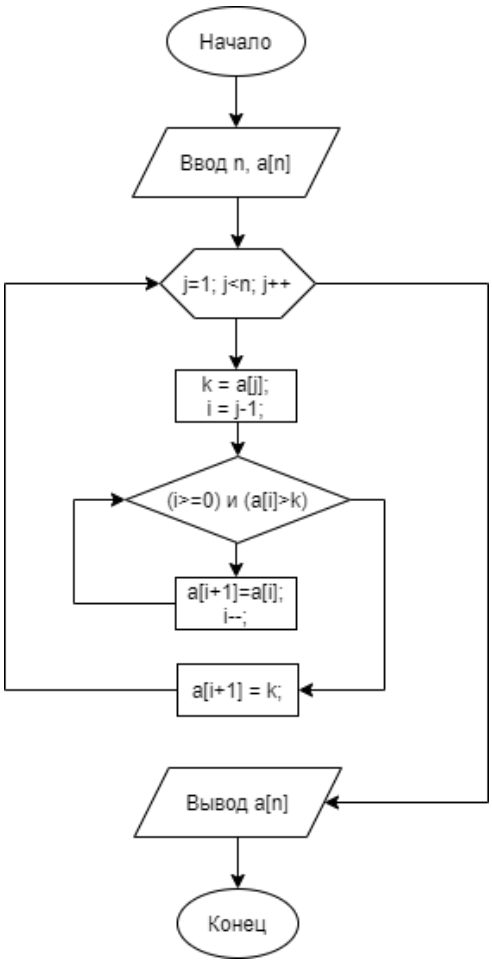

Рис. 1 Схема алгоритма сортировки простыми вставками

Работа алгоритма на случайно заполненном массиве на разном наборе данных приведена на рис. 2 - 4

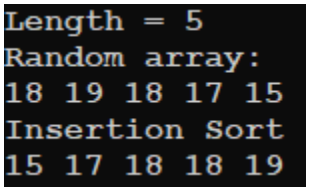

Рис. 2 Работа алгоритма с длиной length = 5

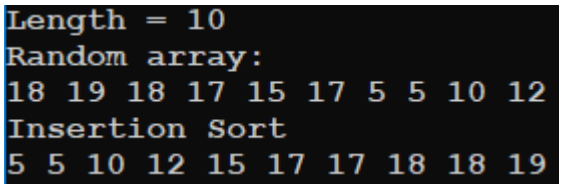

Рис. 3 Работа алгоритма с длиной length = 10

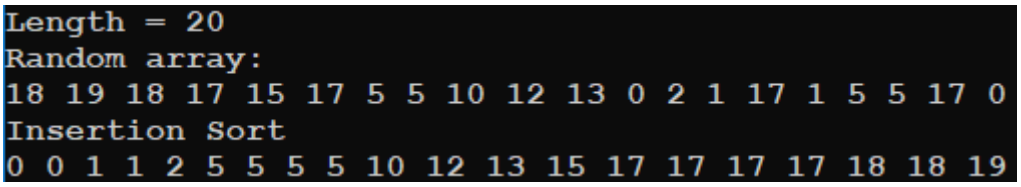

Рис. 4 Работа алгоритма с длиной length = 20

На рис. 2 - 4 видно, что алгоритм сортировки простыми вставками на разном наборе входных данных работает корректно

1.2. Функция роста времени выполнения сортировки методом простых вставок

Для определения функции роста времени выполнения алгоритма при увеличении объема массива рассмотрим операторы вызываемой функции и количество их выполнений исходя из блок-схемы алгоритма. Для наглядности представим их в виде соответствующих операторов языка программирования.

Таблица 1. Подсчет количества операторов

Номер оператора	Оператор	Время выполнения одного оператора	Кол-во выполнений оператора в строке
Номер столбца	1	2	3
1	int i, j, k;	C1	1 раз
2	for (j = 1; j < n; j++){	C2	n раз
3	k = a[j];	C3	n-1 раз
4	i = j - 1;	C4	n-1 раз
5	while ((i >= 0) && (a[i] > k)) {	C5	$\sum_{j=1}^{n-1} t_j$
6	a[i + 1] = a[i];	C6	$\sum_{j=1}^{n-2} t_j$
7	i--;}	C7	$\sum_{j=1}^{n-2} t_j$
8	a[i + 1] = k;	C8	n раз
9	}		
10	}		

Оператор 1 выполняется один раз.

Оператор 2 выполнится n-1 раз и ещё один раз, чтобы проверить значение i и завершить цикл.

Оператор 3. Это оператор тела цикла, т.е. он выполнится n-1 раз – кол-во входов в тело цикла.

Оператор 4. Как и Оператор 3 выполнится n-1 раз.

Оператор 5. Определяем количество выполнений при худшем случае (массив отсортирован в порядке уменьшения значений).

В этом случае в сумме

$\sum_{j=1}^{n} t_j$ значение t_j равно j.

Тогда значение

$$\sum_{j=1}^{n-1} t_j = \sum_{j=1}^{n-1} t_j = \frac{(n-1)(n-1)}{2} = \frac{n^2-2n+1}{2}$$

Оператор 6. Выполнится на 1 раз меньше, чем Оператор 5, тогда количество операторов:

$$\sum_{j=1}^{n-2} t_j = \frac{(n-2)(n-1)}{2} = \frac{n^2-3n+2}{2}$$

Оператор 7. Выполнится столько же раз, сколько и Оператор 6.

Оператор 8. Выполнится n раз.

Выведем функцию роста для времени выполнения алгоритма:

$$\mathrm{T(n)} = \mathrm{C1} + \mathrm{C2} \cdot \mathrm{n} + \mathrm{C3} \cdot (\mathrm{n} - 1) + \mathrm{C4} \cdot (n - 1) + \mathrm{C5} \cdot \sum_{j=1}^{n-1} t_j + \mathrm{C6} \cdot \sum_{j=1}^{n-2} t_j +$$

$$\mathrm{C7} \cdot \sum_{j=}^{n-2} t_j + \mathrm{C8} \cdot n = (1)$$

Определим порядок роста в худшем случае, когда массив отсортирован по убыванию. Значит операторы 5, 6, 7 выполняются полное число раз. Тогда функция имеет вид:

$$T(n) = C1 + C2 \cdot n + C3 \cdot (n-1) + C4 \cdot (n-1) + C5 \cdot \frac{n^2-2n+1}{2} + C6 \cdot \frac{n^2-3n+2}{2} + C7 \cdot \frac{n^2-3n+2}{2} + C8 \cdot n = (C5 + C6 + C7) \cdot \frac{n^2}{2} + (C2 + C3 + C4 - C5 - C6 \cdot \frac{3}{2} - C7 \cdot \frac{3}{2}) \cdot n + (C1 - C3 - C4 + C5 \cdot \frac{1}{2} + C6 + C7) = A \cdot n + B \cdot n + C$$

Функция n^2 является доминирующей, и она определяет порядок роста времени в худшем случае, т.е. порядок роста $T(n) = O(f(n)) = O(n^2)$

Определим порядок роста в лучшем случае, когда массив уже отсортирован. Значит операторы 5, 6, 7 не выполняются. Тогда функция имеет вид:

$$T(n) = C1 + C2 \cdot n + C3 \cdot (n-1) + C4 \cdot (n-1) + C8 \cdot n = (C2 + C3 + C4 + C8) \cdot n + (C1 - C3 - C4) = B \cdot n + C$$

Функция n является доминирующей, и она определяет порядок роста времени в лучшем случае, т.е. порядок роста $T(n) = O(f(n)) = O(n)$

1.3. Сводная таблица результатов выполнения сортировки

Таблица 2. Сводная таблица результатов

n	T(n)	T_T=f(C+M)	$T_п$=$C_ф$+$M_ф$
100	Length = 100 Limit = 100 Time = 707 mcs	$O(n^2)$	Comps: 2579 Moves: 2678 Total: 5257
1000	Length = 1000 Limit = 1000 Time = 1551 mcs		Comps: 255383 Moves: 256382 Total: 511765

10000	Length = 10000 Limit = 10000 Time = 73 ms		Comps: 24968371 Moves: 24978370 Total: 49946741
100000	Length = 100000 Limit = 100000 Time = 8 s		Comps: 2507095227 Moves: 2507195226 Total: 5014290453
1000000	Length = 1000000 Limit = 1000000 Time = 774 s		Comps: 250365280187 Moves: 250369280168 Total: 500734560355

Для наибольшей точности в вычислении времени работы алгоритма на малых n используются микросекунды. Для n = 10000 применимо измерение с точностью до миллисекунд, а при большей длине массива достаточно секунд.

1.4. Код алгоритма и основной программы

Алгоритм сортировки простыми вставками представлен функцией

```
5  void insertionSort(int* a, int n) {
6      int i, j, k;
7      for (j = 1; j < n; j++) {
8          k = a[j];
9          i = j - 1;
10         while ((i >= 0) && (a[i] > k)) {
11             a[i + 1] = a[i];
12             i--;
13         }
14         a[i + 1] = k;
15     }
16 }
```

Также реализована функция заполнения массива случайными числами.

```
18 void randArray(int* a, int length, int limit) {
19     for (int i = 0; i < length; i++) {
20         a[i] = rand() % limit;
21     }
22 }
```

В процессе выполнения основной программы вызываются функциизаполнения массива и алгоритма сортировки и подсчитывается время выполнения алгоритма.

```
24 int main() {
25     srand(NULL);
26
27     int length, limit;
28     std::cout << "Length = ";
29     std::cin >> length;
30
31     std::cout << std::endl;
32     std::cout << "Limit = ";
33     std::cin >> limit;
34
35     int* a = new int[length];
36     randArray(a, length, limit);
37
38     auto begin = std::chrono::steady_clock::now();
39     insertionSort(a, length);
40     auto end = std::chrono::steady_clock::now();
41     auto elapsed_ms = std::chrono::duration_cast<std::chrono::seconds>(end - begin);
42     std::cout << "Time = " << elapsed_ms.count() <<" s";
43
44     delete[] a;
45     return 0;
46 }
```

Для возможности подсчет времени в микросекундах программа используется библиотеку времени chrono.

Для определения практической вычислительной сложности алгоритма в сортировку добавлены счетчики сравнений и перестановок.

```cpp
void insertionSort(int* a, int n) {
    std::cout << "Insertion Sort" << std::endl;
    long long comps = 0, moves = 0;
    int i, j, k;
    for (j = 1; j < n; j++) {
        k = a[j];
        i = j - 1;
        moves++;
        while ((i >= 0) && (a[i] > k)) {
            a[i + 1] = a[i];
            i--;
            comps++;
            moves++;
        }
        a[i + 1] = k;
    }
    std::cout << "Comps = " << comps <<
    std::endl << "Moves = " << moves;
}
```

1.5. Зависимость теоретической и практической сложности

Для вычисления зависимости практической вычислительной сложности алгоритма от размера n массива можно воспользоваться калькулятором методов регрессии для аппроксимации функции одной переменной. Наименьшая ошибка аппроксимации приходится на степенную модель регрессии с показателем, приблизительно равным 2

Степенная регрессия

$$y = 0.5906x^{1.9833}$$

Коэффициент корреляции	Коэффициент детерминации	Средняя ошибка аппроксимации, %
0.9994	0.9987	3.8539 %

Рис. 5 Аппроксимирующая функция для практической сложности

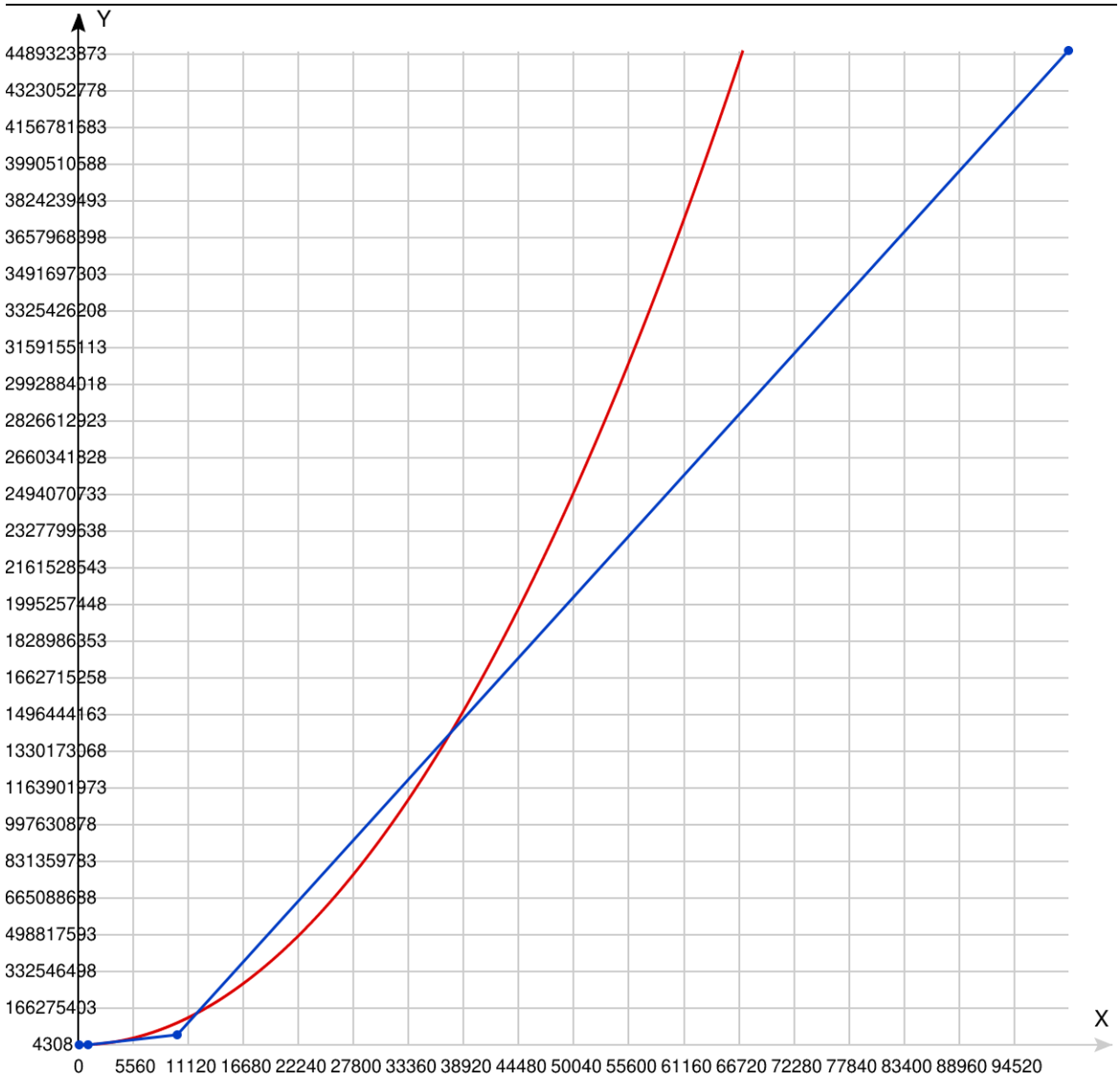

Рис. 6 Графики зависимостей вычислительных сложностей алгоритма от n

На рис. 6 приведены графики зависимостей теоретической $T(n) = O(f(n))$ (синий) и практической $T_п=C_ф+M_ф$ (красный) вычислительных сложностей алгоритма от размера n массива. График практической сложности $T_п$ построен по формуле, найденной ранее. Очевидно, что оба графика представляют собой квадратичную зависимость от n.

1.6. Анализ результатов

Теоретическая сложность алгоритма сортировки простыми вставками, вычисленная по формуле, совпадает с фактически вычисленными временными и эмпирическими характеристиками алгоритма. По таблице 3 можно убедиться, что с возрастание размера массива в 10 раз время работы увеличивается примерно в 100 раз, что означает квадратичную зависимость.

Для оценки практической сложности обратимся к таблице 1. Количество выполнений операции сравнения в лучшем случае $C(n) = 0$, в худшем $C(n) = \frac{n^2-2n+}{2}$, а операции перемещения в лучшем случае $M(n) = n$, в худшем $M(n) = \frac{n^2-2n+}{2}$. Значит сложность алгоритма порядка $O(n^2)$ операций сравнения и $O(n^2)$ операций перемещения, что совпадает с данными, установленными практическим путем.

Таким образом, установленные для алгоритма сортировки простыми вставками вычислительные сложности и реальное время выполнения зависят квадратично от размера массива.

2. Задание 2: оценка вычислительной сложности алгоритма в наихудшем и наилучшем случаях.

2.1. Тестирование на упорядоченном по возрастанию массиве

Таблица 3. Сводная таблица результатов для наилучшего случая

n	T(n)	T_T=f(C+M)	$T_п$=$C_ф$+$M_ф$
100	Length = 100 Time = 519 mcs	$O(n)$	Comps: 0 Moves: 99 Total: 99
1000	Length = 1000 Time = 665 mcs		Comps: 0 Moves: 999 Total: 999
10000	Length = 10000 Time = 420 mcs		Comps: 0 Moves: 9999 Total: 9999
100000	Length = 100000 Time = 1008 mcs		Comps: 0 Moves: 99999 Total: 99999
1000000	Length = 1000000 Time = 3011 mcs		Comps: 0 Moves: 999999 Total: 999999

2.2. Тестирование на упорядоченном по убыванию массиве

Таблица 4. Сводная таблица результатов для худшего случая

n	T(n)	T_T=f(C+M)	$T_п$=$C_ф$+$M_ф$
100	Length = 100 Time = 1670 mcs	$O(n^2)$	Comps: 4950 Moves: 5049 Total: 9999
1000	Length = 1000 Time = 2278 mcs		Comps: 499500 Moves: 500499 Total: 999999
10000	Length = 10000 Time = 173 ms		Comps: 49995000 Moves: 50004999 Total: 99999999
100000	Length = 100000 Time = 16 s		Comps: 4999950000 Moves: 5000049999 Total: 9999999999
1000000	Length = 1000000 Time = 1574 s		Comps: 500000499999 Moves: 499999500000 Total: 999999999999

2.3. Код программы

В целях упорядочения массива добавлена функция, заполняющая массив по возрастания или убыванию целыми значениями от 0 до n-1:

```
void fillSorted(int* a, int n, char sym) {
    switch (sym) {
        case '+':
            for (int i = 0; i < n; i++) a[i] = i;
            return;
        case '-':
            for (int i = 0; i < n; i++) a[i] = n-i-1;
            return;
    }
}
```

Код программы, вычисляющий время выполнения алгоритма сортировки на примере вызова функции заполнения массива по возрастанию:

```
int main() {
    srand(NULL);
    int length, limit;
    std::cout << "Length = ";
    std::cin >> length;
    //std::cout << "Limit = ";
    //std::cin >> limit;

    int* a = new int[length];
    fillSorted(a, length, '+');
    //randArray(a, length, limit);
    //std::cout << "Random array:" << std::endl;
    //print(a, length);

    auto begin = std::chrono::steady_clock::now();
    insertionSort(a, length);
    auto end = std::chrono::steady_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(end - begin);

    std::cout << "Length = " << length << std::endl;
    //std::cout << "Limit = " << limit << std::endl;
    std::cout << "Time = " << elapsed.count() << " s";

    delete[] a;
    return 0;
}
```

2.4. График зависимости вычислительных сложностей для 3 случаев

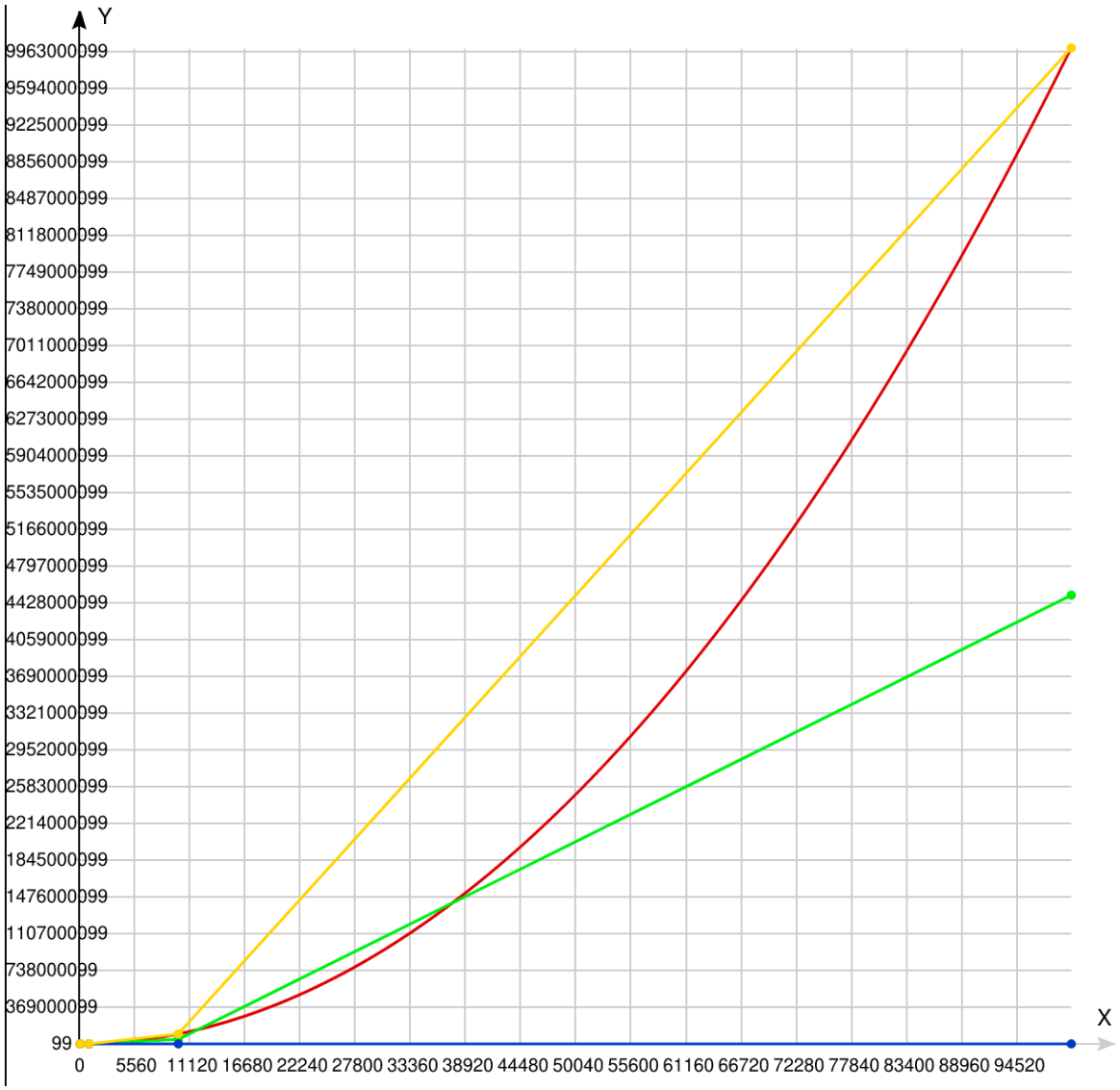

Рис. 7 Графики зависимостей вычислительных сложностей алгоритма от n

На рис. 7 представлены графики зависимостей практической (красный для табл. 1, зеленый для табл. 2, синий для табл. 3) вычислительных сложностей алгоритма. При этом квадратичная зависимость сохраняется только в худшем

случае. Графики практических сложностей для упорядоченного массива являются квадратичными зависимостями с формулами $T_{\text{п}}(n) = n$ для лучшего и $T_{\text{п}}(n) = n^2$.

2.5. Емкостная сложность алгоритма от n

Алгоритм сортировки методом простого обмена имеет линейно зависящую емкостную сложность O(n), т.к. в процессе сортировки используется только один исходный массив размера n.

2.6. Анализ результатов

Установленная эмпирическим путем зависимость вычислительной сложности алгоритма подтверждает найденную ранее теоретическую сложность. Для лучшего случая в алгоритме количество операций сравнения при увеличении размера массива в 10 раз увеличивается в 10, а операции перестановки не выполняются совсем. Для худшего же случая сохраняется квадратичная зависимость критических операций от размера массива.

3. Задание 3 (Вариант 1): оценка эффективности алгоритма простого выбора

3.1. Алгоритм сортировки методом простого выбора

Находим номер минимального значения в текущем списке. Производим обмен этого значения со значением первой неотсортированной позиции (обмен не нужен, если минимальный элемент уже находится на данной позиции). Теперь сортируем хвост списка, исключив из рассмотрения уже отсортированные элементы.

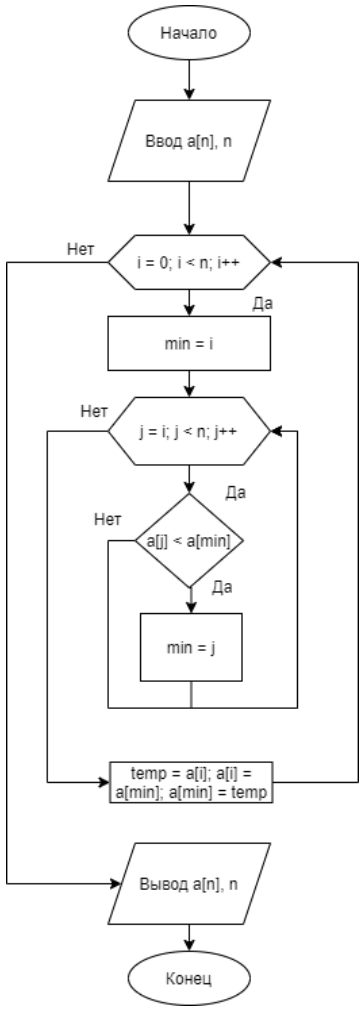

Рис 8. Блок-схема алгоритма простого выбора

Работа алгоритма на случайно заполненном массиве на разном наборе данных приведена на рис. 9 – 11.

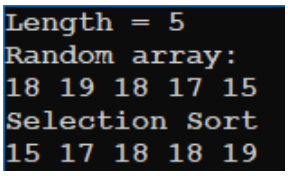

Рис 9. Работа алгоритма с длиной n = 5

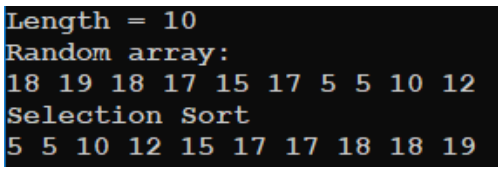

Рис 10. Работа алгоритма с длиной n = 10

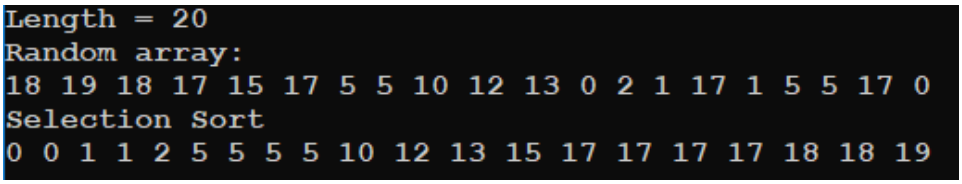

Рис 11. Работа алгоритма с длиной n = 20

На рис. 9 – 11 видно, что алгоритм сортировки простого выбора на случайном наборе входных данных работает корректно.

3.2. Функция роста времени выполнения сортировки методом простых вставок

Для определения функции роста времени выполнения алгоритма при увеличении объема массива рассмотрим операторы вызываемой функции и количество их выполнений исходя из блок-схемы алгоритма. Для наглядности представим их в виде соответствующих операторов языка программирования.

Таблица 5. Подсчет количества операторов

Номер оператора	Оператор	Время выполнения одного оператора	Кол-во выполнений оператора в строке
Номер столбца	1	2	3
1	int min, temp;	C1	1 раз
2	for (int i = 0; i < n; i++) {	C2	n+1 раз
	min = i;	C3	n раз
	for (int j = i + 1; j < n; j++) {	C4	n^2 раз
3	if (a[j] < a[min]) {	C5	$n^2 - 1$ раз
4	min = j; }}	C6	$n^2 - 1$ раз
5	temp = a[i];	C7	n раз
6	a[i] = a[min];	C8	n раз
7	a[min] = temp;	C9	n раз
8	}		

Оператор 1 выполняется один раз.

Оператор 2 выполнится n раз и ещё один раз, чтобы проверить значение i и завершить цикл.

Оператор 3. Это оператор тела цикла, т.е. он выполнится n раз – кол-во входов в тело цикла.

Оператор 4. Как и Оператор 3 выполнится n раз и поскольку это цикл, он выполнится n-1 раз и ещё один раз, чтобы проверить условие, т.е. n^2 раз.

Оператор 5. Это оператор тела цикла, т.е. он выполнится $n^2 - 1$ раз – кол-во входов в тело цикла.

Оператор 6. Так как мы рассматриваем худший случай (массив отсортирован в порядке убывания), то Оператор 6 выполнится столько же раз, сколько и Оператор 5.

Оператор 7 Выполнится n раз.

Оператор 8. Выполнится n раз.

Оператор 9. Выполнится n раз.

Выведем функцию роста для времени выполнения алгоритма:

$$T(n) = C1 + C2 \cdot (n + 1) + C3 \cdot n + C4 \cdot n^2 + C5 \cdot (n^2 - 1) + C6 \cdot (n^2 - 1) + C7 \cdot n + C8 \cdot n + C9 \cdot n \quad (1)$$

Определим порядок роста в худшем случае, когда массив отсортирован по убыванию. Значит все операторы выполняются полное число раз. Тогда функция имеет вид:

$$T(n) = C1 + C2 \cdot (n + 1) + C3 \cdot n + C4 \cdot n^2 + C5 \cdot (n^2 - 1) + C6 \cdot (n^2 - 1) + C7 \cdot n + C8 \cdot n + C9 \cdot n = (C4 + C5 + C6) \cdot n^2 + (C2 + C3 + C7 + C8 + C9) \cdot n + (C1 + C2 - C5 - C6) = A \cdot n^2 + B \cdot n + C$$

Функция n^2 является доминирующей и она определяет порядок роста времени в худшем случае, т.е. порядок роста $T(n) = O(f(n)) = O(n^2)$

Определим порядок роста в лучшем случае, когда массив уже отсортирован. Значит операторы 5, 6 не выполняются. Тогда функция имеет вид:

$$T(n) = C1 + C2 \cdot (n+1) + C3 \cdot n + C4 \cdot n^2 + C7 \cdot n + C8 \cdot n + C9 \cdot n = C4 \cdot n^2 + (C2 + C3 + C7 + C8 + C9) \cdot n + (C1 + C2) = A \cdot n^2 + B \cdot n + C$$

Функция n^2 является доминирующей и она определяет порядок роста времени в лучшем случае, т.е. порядок роста $T(n) = O(f(n)) = O(n^2)$.

3.3. Сводная таблица результатов выполнения сортировки

Таблица 6. Сводная таблица результатов

n	T(n)	T_T=f(C+M)	$T_п$=$C_ф$+$M_ф$
100	Length = 100 Limit = 100 Time = 22 mcs	$O(n^2)$	Comps: 4950 Moves: 407 Total: 5357
1000	Length = 1000 Limit = 1000 Time = 1547 mcs		Comps: 499500 Moves: 6318 Total: 505818
10000	Length = 10000 Limit = 10000 Time = 136 ms		Comps: 49995000 Moves: 84431 Total: 50079431
100000	Length = 100000 Limit = 100000 Time = 14 s		Comps: 4999950000 Moves: 1039198 Total: 5000989198
1000000	Length = 1000000 Limit = 1000000 Time = 1391 s		Comps: 499999500000 Moves: 2954824 Total: 31252564265

3.4. Код алгоритма и основной программы

Алгоритм сортировки простым выбором представлен функцией

```
void selection_sort(int* a, int n) {
    int min, temp;
    for (int i = 0; i < n; i++) {
        min = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

В процессе выполнения основной программы вызываются функциизаполнения массива и алгоритма сортировки и подсчитывается время выполнения алгоритма.

```
int main() {
    long long comps = 0, moves = 0;
    int length, limit;
    std::cout << "Length = ";
    std::cin >> length;
    std::cout << "Limit = ";
    std::cin >> limit;

    int* a = new int[length];
    randArray(a, length);

    auto begin = std::chrono::steady_clock::now();
    selection_sort(a, length, comps, moves);
    auto end = std::chrono::steady_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - begin);

    std::cout << "Time = " << elapsed.count() << " mcs" << std::endl;
    std::cout << "Comps: " << comps << std::endl
              << "Moves: " << moves << std::endl
              << "Total: " << comps + moves << std::endl;
    std::cout << comps + moves;

    delete[] a;
    return 0;
}
```

Для определения практической вычислительной сложности алгоритма в сортировку добавлены счетчики сравнений и перестановок.

```
void selection_sort(int* a, int n, long long &comps, long long &moves) {
    int min, temp;
    for (int i = 0; i < n; i++) {
        min = i;
        for (int j = i + 1; j < n; j++) {
            comps++;
            if (a[j] < a[min]) {
                min = j;
                moves++;
            }
        }
        moves++;
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

3.5. График зависимости вычислительных сложностей для 2 случаев

Зависимость практической вычислительной сложности алгоритма простой вставки определена методом, указанным в п. 1.5. Наиболее близкая функция имеет показатель степени приблизительно равный 2:

Степенная регрессия

$y = 0.5301x^{1.9942}$

Коэффициент корреляции	Коэффициент детерминации	Средняя ошибка аппроксимации, %
1.0000	0.9999	0.9416 %

Рис. 12 Аппроксимирующая функция для практической сложности

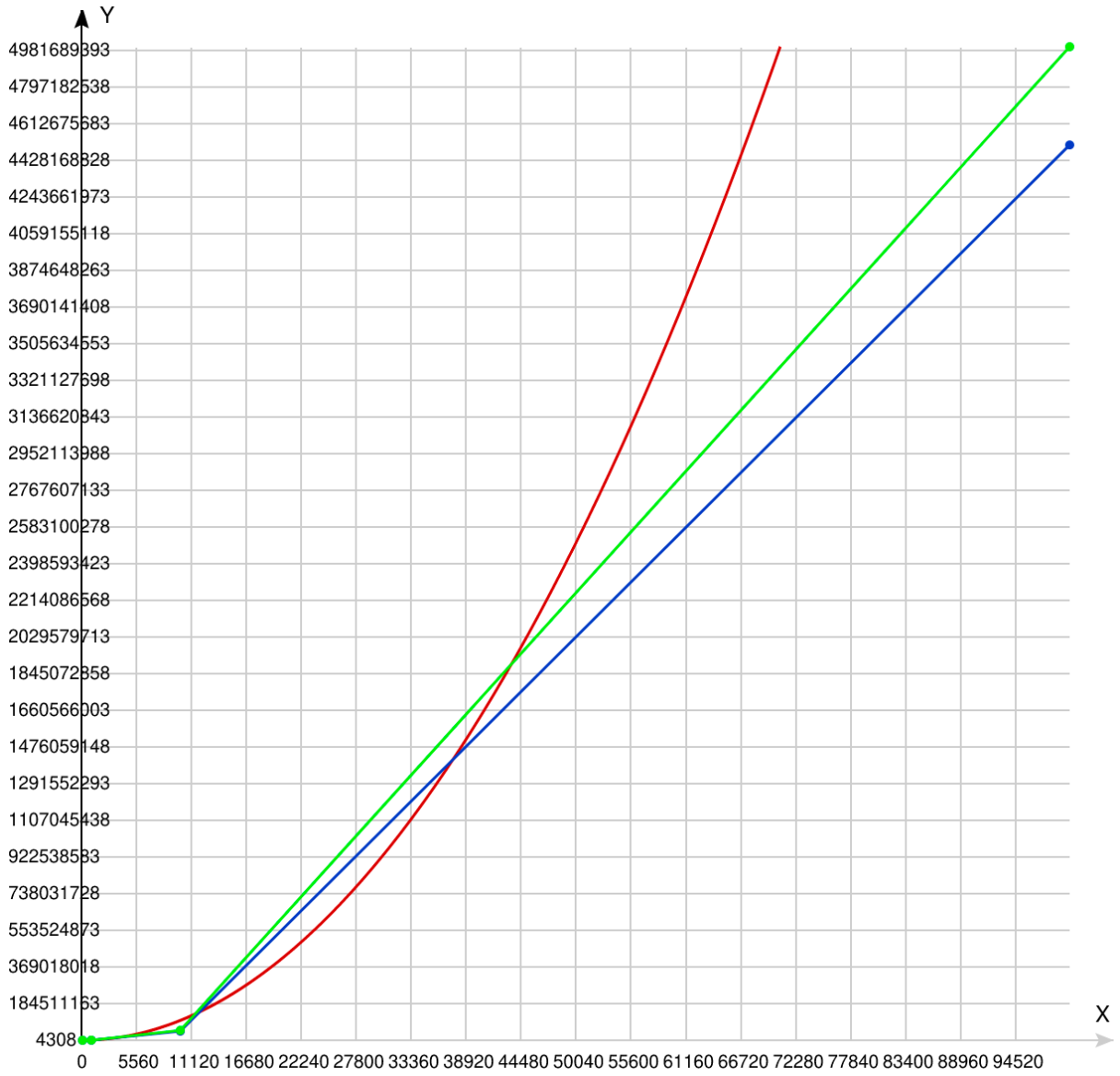

Рис. 13 Графики зависимостей вычислительных сложностей алгоритма от n

На рис. 13 отображена зависимость теоретической (синий) и практической (красный для алгоритма простой вставки, черный для алгоритма простого выбора) вычислительных сложностей. Теоретическая сложность в наихудшем случае у обоих алгоритмов квадратично зависит от размера массива.

3.6. Определение эффективности алгоритма

Эффективность алгоритма — это свойство алгоритма, которое связано с вычислительными ресурсами, используемыми алгоритмом. Алгоритм считается эффективным, если потребляемый им ресурс (или стоимость ресурса) являются наиболее оптимальными. Оценка эффективности алгоритма состоит в определении времени его выполнения и объема потребляемой им памяти. Алгоритм считается эффективнее другого, если время его работы в наихудшем случае имеет более низкий порядок роста.

3.7. Анализ результатов

На рис. 13 по графикам зависимости видно, что практическая сложность для сортировки простой вставкой представлена функцией, растущей со скоростью, аналогичной скорости функции простого выбора. Однако, алгоритм сортировки методом простой вставки имеет линейную зависимость от размера в наилучшем случае, хотя и показывает примерно одинаковое время в среднем случае.

Таким образом, на основании эмпирического анализа алгоритм сортировки методом простой вставки эффективнее метода простого обмена.

4. Ответы на вопросы 7-9

Вопрос 7:

Для применения условия Айверсона к сортировке простого обмена внесены изменения в функцию алгоритма:

```
void exchange_sort(int* arr, int n) {
    bool isSorted;
    for (int i = 0; i < n - 1; ++i) {
        isSorted = true;
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                isSorted = false;
            }
        }
        if (isSorted)
            break;
    }
}
```

Таблица 8 Сводная таблица результатов с условием Айверсона

n	T(n)	T_T=f(C+M)	$T_П$=Сф+Мф
100	Length = 100 Max = 100 114 mcs	$O(n^2)$	Comps: 4797 Moves: 2486 7283
1000	Length = 1000 Max = 1000 8371 mcs		Comps: 498870 Moves: 247314 746184
10000	Length = 10000 Max = 10000 809 ms		Comps: 49981470 Moves: 24980642 74962112
100000	Length = 100000 Max = 100000 83 s		Comps: 4999770899 Moves: 2492732912 7492503811
1000000	Length = 1000000 Max = 1000000 8728 s		Comps: 499999419800 Moves: 249947507992 749946927792

Сравнивая табл. 8 и табл. 3, можно убедиться в том, что применение условия Айверсона не играет весомой роли в изменении эффективности алгоритма простого обмена. Реализация условия Айверсона требует дополнительные операции сравнения на каждой итерации внешнего цикла, а также накладывает дополнительные операции присваивания, выполняемые вместе с перестановками.

Вопрос 8:

Сортировка массива [5 6 1 2 3] с данными шагами является сортировкой простой вставки, суть которой заключается в переборе массива, при котором на каждом шаге элемент перемещается на свое место в левую часть массива с сохранением порядка следования остальных:

1-я итерация: элемент «1» встает на 1-ое место: [1 5 6 2 3];

2-я итерация: элемент «2» встает на 2-ое место: [1 2 5 6 3];

3-я итерация: элемент «3» встает на 3-е место: [1 2 3 5 6].

Вопрос 9:

Теоретическая вычислительная сложность алгоритма сортировки простой вставкой была рассмотрена в п. 3.2. В лучшем случае алгоритм имеет порядок роста времени O(n) (порядка O(n) операций сравнения перестановки), в худшем – $O(n^2)$ (порядка $O(n^2)$ операций сравнения и перестановки). Емкостная сложность составляет O(n), так как используется только один массив.

ВЫВОДЫ

В ходе выполнения работы были освоены следующие алгоритмы сортировки: сортировка обменом (пузырьковая сортировка), сортировка выбором. Данные алгоритмы были описаны на языке блок-схем, реализованы на языке C++, была определена теоретическая сложность в наилучшем и наихудшем случаях, выполнен прогон алгоритмов на тестовых данных для среднего, наилучшего и наихудшего случаев, определена практическая сложность алгоритмов. Следуя из результатов тестирования, теоретические предположения совпали с полученными эмпирически данными. Основываясь на эти данные, алгоритм пузырьковой сортировки во многих случаях эффективнее алгоритма сортировки выбором.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Лекционный материал по структуре и алгоритмам обработки данных Гданского Н.И.

2. Вычислительная сложность // Википедия [Электронный ресурс]. - https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C. – (дата обращения: 25.04.2021)

3. Прата, Стивен Язык программирования C++. Лекции и упражнения/ Стивен Прата. - М.: Вильямс, 2015. - 445 с.