



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 **1.1**

**Тема: ЭМПИРИЧЕСКИЙ АНАЛИЗ СЛОЖНОСТИ  
АЛГОРИТМОВ**

Дисциплина: Структуры и Алгоритмы обработки данных

Выполнил студент  
группа

Багинян А.В.  
ИКБО-50-25

**Москва 2026**

# Содержание

1. Задание 1 .....	2
1.1. Начало .....	2
1.2. Алгоритм №1 .....	3
1.2.1. Тестирование алгоритма .....	4
1.2.2. Выводы по тестированию алгоритма .....	9
1.3. Алгоритм №2 .....	9
1.3.1. Тестирование алгоритма .....	10
1.3.2. Вывод по тестированию алгоритма .....	15
1.4. Сравнение двух алгоритмов .....	16
1.5. Вывод из задания 1 .....	17
1.5.1. Сравнение по времени выполнения .....	17
1.5.2. Сравнение по расходу памяти (Ёмкостная сложность) .....	18
2. Задание 2 .....	19
2.1. Начало .....	19
2.2. Алгоритм простой сортировки (bubble sort) .....	19
2.2.1. Тестирование алгоритма .....	20
2.3. Вывод по тестированию алгоритма .....	23
2.3.1. Ёмкостная сложность алгоритма .....	23
2.3.2. Вывод об эмпирической вычислительной сложности .....	23
3. Задание 3 .....	24
3.1. Тестирование при массиве в убывающем порядке значений (худший случай) .....	26
3.2. Тестирование при массиве в возрастающем порядке значений (лучший случай) .....	27
3.3. Вывод о зависимости алгоритма от исходной упорядоченности .....	27
4. Задание 3 .....	28
4.1. Алгоритм простой сортировки (insertion sort) .....	28
4.2. Тестирование двух алгоритмов .....	29
4.3. Выводы по тестированию алгоритма (insertion sort) .....	36
4.3.1. Ёмкостная сложность алгоритма .....	36
4.4. Сравнение обоих алгоритмов .....	36
4.4.1. Лучший случай .....	38
4.4.2. Худший и средний случаи .....	38
4.4.3. Вывод .....	38
5. Литература .....	39

**Цель работы:** актуализация знаний и приобретение практических умений и навыков по определению вычислительной сложности алгоритмов (эмпирический подход).

## 1. Задание 1

### 1.1. Начало

Определим структуру `ComplexityMetrics` для подсчета числа выполненных сравнений ( $C_n$ ) + перемещений элементов в памяти ( $M_n$ ), а также суммарное число критических операций  $T_n = C_n + M_n$ .

```
1 struct ComplexityMetrics {
2     size_t comparisons; // Cn – число сравнений
3     size_t moves;       // Mn – число перемещений
4     size_t total;       // Tn = Cn + Mn
5     double duration;    // время выполнения функции (мс)
6
7     ComplexityMetrics() : comparisons(0), moves(0), total(0),
8         duration(0) {}
9
10    std::string toString() const {
11        return "Cn=" + std::to_string(comparisons) +
12            ", Mn=" + std::to_string(moves) + ", Tn=" +
13            std::to_string(total) +
14            ", took " + std::to_string(duration) + " ms";
15    }
16};
```

Определим функцию для подсчета времени выполнения алгоритма:

```
1 #include <chrono>
2
3 using TestFunc = std::function<ComplexityMetrics(size_t)>;
4
5 ComplexityMetrics measureTime(TestFunc func, size_t len) {
6     auto start = std::chrono::high_resolution_clock::now();
7     ComplexityMetrics metrics = func(len);
8     auto end = std::chrono::high_resolution_clock::now();
9
10    std::chrono::duration<double, std::milli> duration = end -
11        start;
12    metrics.duration = duration.count();
13    return metrics;
14}
```

Функция принимает функцию, соответствующую типу `ComplexityMetrics (size_t)`, и возвращает метрику.

Определим функцию для генерации случайно заполненного массива:

```
1  #include <random>
2
3  static std::mt19937 rng(std::random_device{}());
4
5  char *generateRandomArray(size_t size, const char key, const char
otherChar) {
6      char *arr = new char[size + 1];
7
8      std::bernoulli_distribution dist(0.5);
9
10     for (size_t i = 0; i < size; i++) {
11         arr[i] = dist(rng) ? key : otherChar;
12     }
13     arr[size] = '\0';
14     return arr;
15 }
```

Функция возвращает указатель на динамически аллоцированный массив символов длиной `size+1` (последний элемент массива под `'\0'`), заполненный символами `key` (символ к удалению) и `otherChar`.

## 1.2. Алгоритм №1

Реализуем первый метод на языке C++:

```
1  void delFirstMethod(char *x, size_t &n, const char &key) {
2      size_t i = 0;
3
4      while (i < n) {
5          if (x[i] == key) {
6              for (size_t j = i; j < n - 1; j++) {
7                  x[j] = x[j + 1];
8              }
9              n--;
10         } else {
11             i++;
12         }
13     }
14 }
```

Изменим код реализованного метода для возвращения метрик:

```

1 ComplexityMetrics delFirstMethod(char *x, size_t &n, const char
&key) {
2     ComplexityMetrics metrics;
3     size_t i = 0;
4
5     while (i < n) {
6         metrics.comparisons++; // Сравнение x[i] = key
7         if (x[i] == key) {
8             for (size_t j = i; j < n - 1; j++) {
9                 metrics.moves++; // Перемещение x[j] = x[j + 1]
10                x[j] = x[j + 1];
11            }
12            n--;
13        } else {
14            i++;
15        }
16    }
17    metrics.total = metrics.comparisons + metrics.moves;
18    return metrics;
19 }

```

Теперь функция возвращает метрику.

### 1.2.1. Тестирование алгоритма

Протестируем функцию в 3х случаях (лучший, худший и средний) на массивах длиной  $n = 100, 200, 500, 1000, 2000, 5000, 10000$

```

1 void testFirstMethod(size_t n, int runs = 100) {
2     std::cout << "\ttestFirstMethod() for " << runs << " runs:" <<
std::endl;
3     std::cout << "\t\ttestFirstMethodWorst() statistics: "
4         << testTimes(
5             [](size_t len) {
6                 char x[len];
7                 std::memset(x, '_', len);
8
9                 return delFirstMethod(x, len, '_');
10            },
11            runs, n)
12         .toString()
13     << std::endl;
14     std::cout << "\t\ttestFirstMethodBest() statistics: "
15         << testTimes(
16             [](size_t len) {
17                 char x[len];

```

```

18         std::memset(x, 'A', len);
19
20         return delFirstMethod(x, len, '_');
21     },
22     runs, n)
23     .toString()
24     << std::endl;
25 }

```

Определим функцию для тестирования алгоритма в среднем (случайном) случае:

```

1 void testBothMethodsMedium(size_t n, int runs = 100) {
2     char *arr = generateRandomArray(n, '_', 'A');
3     char *copy = new char[n + 1];
4     std::memcpy(copy, arr, n);
5
6     auto testFirst = [&](size_t n) { return delFirstMethod(arr, n,
7     '_'); };
8     auto testOther = [&](size_t n) { return delOtherMethod(copy, n,
9     '_'); };
10    std::cout << "\ttestBothMethodsMedium() for " << runs
11    << " runs: " << std::endl;
12    std::cout << "\t\ttestFirstMethodMedium() statistics: "
13    << measureTime(testFirst, n).toString() << std::endl;
14    std::cout << "\t\ttestOtherMethodMedium() statistics: "
15    << measureTime(testOther, n).toString() << std::endl;
16
17    delete[] arr;
18    delete[] copy;
19 }

```

Листинг 1. Общая функция проверки алгоритмов в среднем (случайном) случае.

Функция в данной реализации тестирует оба алгоритма на одном и том же массиве для точности сравнения. Дальше мы просто проигнорируем вывод тестирования для второго алгоритма.

Теперь в файле `main.cpp` вызовем методы для тестирования первого алгоритма:

```

1 #include "include/test.h"
2 #include <iostream>
3
4 int main() {

```

```

5     const auto runs = 1;
6
7     for (auto n : {100, 200, 500, 1000, 2000, 5000, 10000}) {
8         std::cout << "RUNNING FOR N=" << n << std::endl;
9         testFirstMethod(n, runs);
10        std::cout << "-----" << std::endl;
11        testBothMethodsMedium(n, runs);
12        std::cout << "-----" << std::endl;
13    }
14    return 0;
15 }

```

Вывод программы:

```

1  RUNNING FOR N=100
2      testFirstMethod() for 1 runs:
3          testFirstMethodWorst() statistics: Cn=100, Mn=4950, Tn=5050,
took 0.028183 ms
4          testFirstMethodBest() statistics: Cn=100, Mn=0, Tn=100, took
0.000802 ms
5  -----
6      testBothMethodsMedium() for 1 runs:
7          testFirstMethodMedium() statistics: Cn=100, Mn=2266, Tn=2366,
took 0.013666 ms
8  -----
9
10 RUNNING FOR N=200
11     testFirstMethod() for 1 runs:
12         testFirstMethodWorst() statistics: Cn=200, Mn=19900,
Tn=20100, took 0.100721 ms
13         testFirstMethodBest() statistics: Cn=200, Mn=0, Tn=200, took
0.001143 ms
14     -----
15     testBothMethodsMedium() for 1 runs:
16         testFirstMethodMedium() statistics: Cn=200, Mn=8564, Tn=8764,
took 0.047741 ms
17     -----
18
19 RUNNING FOR N=500
20     testFirstMethod() for 1 runs:
21         testFirstMethodWorst() statistics: Cn=500, Mn=124750,
Tn=125250, took 0.606165 ms
22         testFirstMethodBest() statistics: Cn=500, Mn=0, Tn=500, took
0.002314 ms
23     -----

```

```

24     testBothMethodsMedium() for 1 runs:
25         testFirstMethodMedium() statistics: Cn=500, Mn=61134,
Tn=61634, took 0.302020 ms
26 -----
27
28 RUNNING FOR N=1000
29     testFirstMethod() for 1 runs:
30         testFirstMethodWorst() statistics: Cn=1000, Mn=499500,
Tn=500500, took 2.427775 ms
31         testFirstMethodBest() statistics: Cn=1000, Mn=0, Tn=1000,
took 0.004569 ms
32 -----
33     testBothMethodsMedium() for 1 runs:
34         testFirstMethodMedium() statistics: Cn=1000, Mn=254091,
Tn=255091, took 1.250503 ms
35 -----
36
37 RUNNING FOR N=2000
38     testFirstMethod() for 1 runs:
39         testFirstMethodWorst() statistics: Cn=2000, Mn=1999000,
Tn=2001000, took 9.523869 ms
40         testFirstMethodBest() statistics: Cn=2000, Mn=0, Tn=2000,
took 0.007865 ms
41 -----
42     testBothMethodsMedium() for 1 runs:
43         testFirstMethodMedium() statistics: Cn=2000, Mn=1005082,
Tn=1007082, took 4.767976 ms
44 -----
45
46 RUNNING FOR N=5000
47     testFirstMethod() for 1 runs:
48         testFirstMethodWorst() statistics: Cn=5000, Mn=12497500,
Tn=12502500, took 59.287127 ms
49         testFirstMethodBest() statistics: Cn=5000, Mn=0, Tn=5000,
took 0.022282 ms
50 -----
51     testBothMethodsMedium() for 1 runs:
52         testFirstMethodMedium() statistics: Cn=5000, Mn=6178707,
Tn=6183707, took 29.289154 ms
53 -----
54
55 RUNNING FOR N=10000
56     testFirstMethod() for 1 runs:
57         testFirstMethodWorst() statistics: Cn=10000, Mn=49995000,
Tn=50005000, took 236.746269 ms
58         testFirstMethodBest() statistics: Cn=10000, Mn=0, Tn=10000,

```



```

took 0.044735 ms
59 -----
60     testBothMethodsMedium() for 1 runs:
61         testFirstMethodMedium() statistics: Cп=10000, Mп=25042001,
        Tп=25052001, took 119.494483 ms
62 -----

```

Представим результаты тестирования в таблицы.

п	время, мс	$C_{\pi}$	$M_{\pi}$	$T_{\pi} = C_{\pi} + M_{\pi}$
100	0.0008	100	0	100
200	0.0011	200	0	200
500	0.0023	500	0	500
1000	0.0046	1000	0	1000
2000	0.0079	2000	0	2000
5000	0.0223	5000	0	5000
10000	0.0447	10000	0	10000

Таблица 1. Сводная таблица результатов **лучшего** случая

п	время, мс	$C_{\pi}$	$M_{\pi}$	$T_{\pi} = C_{\pi} + M_{\pi}$
100	0.0137	100	2266	2366
200	0.0477	200	8564	8764
500	0.3020	500	61134	61634
1000	1.2505	1000	254091	255091
2000	4.7680	2000	1005082	1007082
5000	29.2892	5000	6178707	6183707
10000	119.4945	10000	25042001	25052001

Таблица 2. Сводная таблица результатов **среднего** случая

п	время, мс	$C_{\pi}$	$M_{\pi}$	$T_{\pi} = C_{\pi} + M_{\pi}$
100	0.0282	100	4950	5050
200	0.1007	200	19900	20100
500	0.6062	500	124750	125250
1000	2.4278	1000	499500	500500
2000	9.5239	2000	1999000	2001000
5000	59.2871	5000	12497500	12502500
10000	236.7463	10000	49995000	50005000

Таблица 3. Сводная таблица результатов **худшего** случая

Построим график на основе полученных данных.

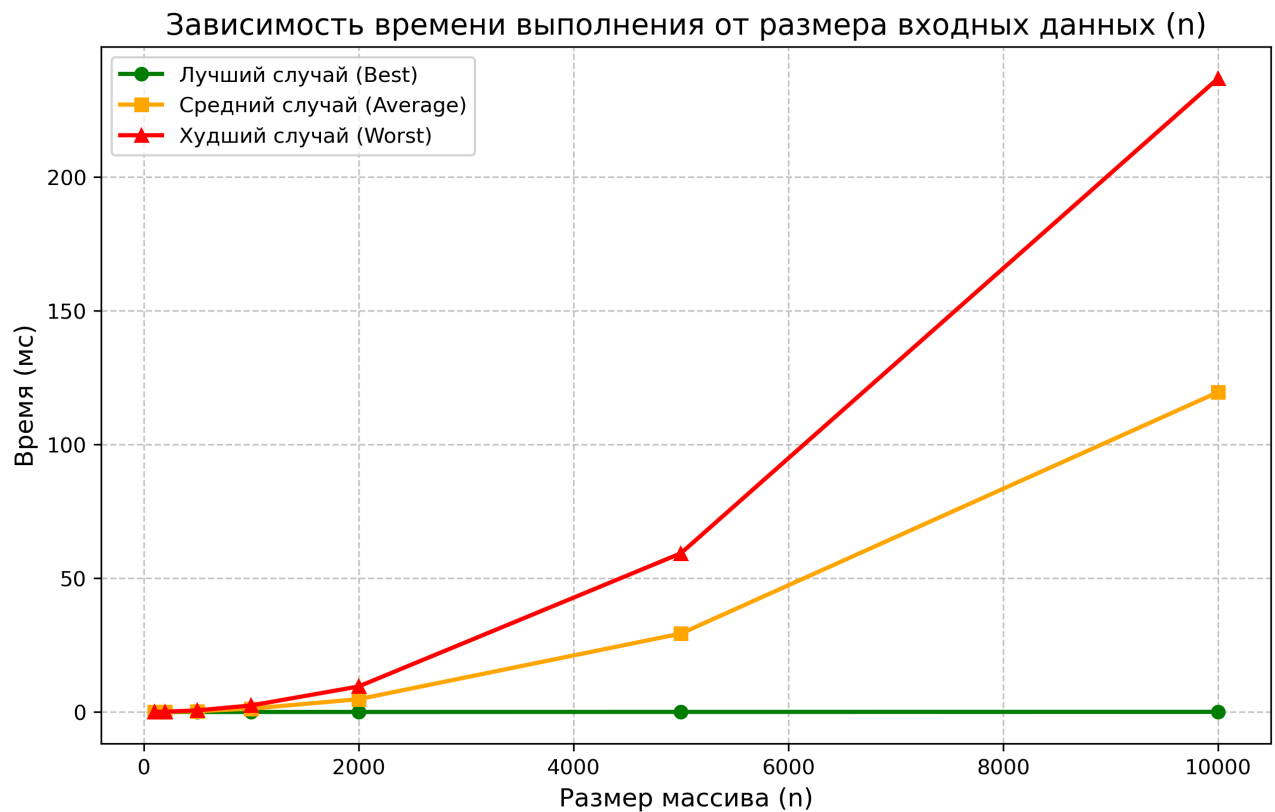


Рис. 1. Зависимость времени работы алгоритма от  $n$ . Видно квадратичное возрастание для среднего и худшего случаев.

### 1.2.2. Выводы по тестированию алгоритма

#### ✓ Вывод

Функция тестирует алгоритм в двух случаях - худшем и лучшем. **Худшим** для данного алгоритма будет массив, полностью заполненный ключами к удалению, **лучшим** - массив без элементов к удалению.

Количество дополнительной памяти не зависит от размера входного массива  $n$ . Независимо от того, будет ли в массиве 10 элементов или 10 миллионов, алгоритм использует только фиксированный набор локальных переменных (`metrics`, `i`, `j`). Следовательно, ёмкостная сложность:  $O(1)$  (Константная сложность). Алгоритму требуется константное число ячеек, то есть  $C_{space} = const$ .

## 1.3. Алгоритм №2

Реализуем второй метод на языке C++:

```
1 void delOtherMethod(char *x, size_t &n, const char &key) {
2     size_t j = 0;
3 }
```

```

4   for (size_t i = 0; i < n; i++) {
5       if (x[i] != key) {
6           if (i != j) {
7               x[j] = x[i];
8           }
9           j++;
10      }
11  }
12  n = j;
13 }

```

### 1.3.1. Тестирование алгоритма

Изменим код реализованного метода для возвращения метрик:

```

1  ComplexityMetrics delOtherMethod(char *x, size_t &n, const char
&key) {
2      ComplexityMetrics metrics;
3      size_t j = 0;
4
5      for (size_t i = 0; i < n; i++) {
6          metrics.comparisons++; // Сравнение x[i] != key
7          if (x[i] != key) {
8              if (i != j) {
9                  metrics.moves++; // Перемещение x[j] = x[i]
10                 x[j] = x[i];
11             }
12             j++;
13         }
14     }
15     n = j;
16
17     metrics.total = metrics.comparisons + metrics.moves;
18     return metrics;
19 }

```

Теперь функция возвращает метрику.

Протестируем функцию в 3х случаях (лучший, худший и средний) на массивах длиной  $n = 100, 200, 500, 1000, 2000, 5000, 10000$

```

1  void testOtherMethod(size_t n, int runs = 100) {
2      std::cout << "\ttestOtherMethod() for " << runs << " runs:" <<
std::endl;
3      std::cout << "\t\ttestOtherMethodWorst() statistics: "

```

```

4         << testTimes(
5             [](size_t len) {
6                 char x[len];
7                 std::memset(x, 'A', len);
8                 return del0therMethod(x, len, '_');
9             },
10            runs, n)
11            .toString()
12        << std::endl;
13    std::cout << "\\t\\ttest0therMethodBest() statistics: "
14        << testTimes(
15            [](size_t len) {
16                char x[len];
17                std::memset(x, '_', len);
18                return del0therMethod(x, len, '_');
19            },
20            runs, n)
21            .toString()
22        << std::endl;
23 }

```

Тестирование алгоритма в среднем случае определено в Листинг 1.

Теперь в файле `main.cpp` вызовем методы для тестирования второго алгоритма:

```

1  #include "include/test.h"
2  #include <iostream>
3
4  int main() {
5      const auto runs = 1;
6
7      for (auto n : {100, 200, 500, 1000, 2000, 5000, 10000}) {
8          std::cout << "RUNNING FOR N=" << n << std::endl;
9          test0therMethod(n, runs);
10         std::cout << "-----" << std::endl;
11         testBothMethodsMedium(n, runs);
12         std::cout << "-----" << std::endl;
13     }
14     return 0;
15 }

```

Вывод программы:

```

1  RUNNING FOR N=100
2      test0therMethod() for 1 runs:

```

```

3      testOtherMethodWorst() statistics: Cn=100, Mn=0, Tn=100, took
0.000501 ms
4      testOtherMethodBest() statistics: Cn=100, Mn=0, Tn=100, took
0.000340 ms
5  -----
6      testBothMethodsMedium() for 1 runs:
7          testOtherMethodMedium() statistics: Cn=100, Mn=52, Tn=152,
took 0.001884 ms
8  -----
9
10     RUNNING FOR N=200
11     testOtherMethod() for 1 runs:
12         testOtherMethodWorst() statistics: Cn=200, Mn=0, Tn=200, took
0.000611 ms
13         testOtherMethodBest() statistics: Cn=200, Mn=0, Tn=200, took
0.000461 ms
14  -----
15     testBothMethodsMedium() for 1 runs:
16         testOtherMethodMedium() statistics: Cn=200, Mn=109, Tn=309,
took 0.003527 ms
17  -----
18
19     RUNNING FOR N=500
20     testOtherMethod() for 1 runs:
21         testOtherMethodWorst() statistics: Cn=500, Mn=0, Tn=500, took
0.001262 ms
22         testOtherMethodBest() statistics: Cn=500, Mn=0, Tn=500, took
0.000952 ms
23  -----
24     testBothMethodsMedium() for 1 runs:
25         testOtherMethodMedium() statistics: Cn=500, Mn=253, Tn=753,
took 0.007595 ms
26  -----
27
28     RUNNING FOR N=1000
29     testOtherMethod() for 1 runs:
30         testOtherMethodWorst() statistics: Cn=1000, Mn=0, Tn=1000,
took 0.002395 ms
31         testOtherMethodBest() statistics: Cn=1000, Mn=0, Tn=1000,
took 0.001783 ms
32  -----
33     testBothMethodsMedium() for 1 runs:
34         testOtherMethodMedium() statistics: Cn=1000, Mn=509, Tn=1509,
took 0.015289 ms
35  -----
36

```

```

37  RUNNING FOR N=2000
38      testOtherMethod() for 1 runs:
39          testOtherMethodWorst() statistics: Cn=2000, Mn=0, Tn=2000,
took 0.004659 ms
40          testOtherMethodBest() statistics: Cn=2000, Mn=0, Tn=2000,
took 0.003416 ms
41  -----
42      testBothMethodsMedium() for 1 runs:
43          testOtherMethodMedium() statistics: Cn=2000, Mn=971, Tn=2971,
took 0.030668 ms
44  -----
45
46  RUNNING FOR N=5000
47      testOtherMethod() for 1 runs:
48          testOtherMethodWorst() statistics: Cn=5000, Mn=0, Tn=5000,
took 0.012023 ms
49          testOtherMethodBest() statistics: Cn=5000, Mn=0, Tn=5000,
took 0.008796 ms
50  -----
51      testBothMethodsMedium() for 1 runs:
52          testOtherMethodMedium() statistics: Cn=5000, Mn=2532,
Tn=7532, took 0.078919 ms
53  -----
54
55  RUNNING FOR N=10000
56      testOtherMethod() for 1 runs:
57          testOtherMethodWorst() statistics: Cn=10000, Mn=0, Tn=10000,
took 0.026239 ms
58          testOtherMethodBest() statistics: Cn=10000, Mn=0, Tn=10000,
took 0.016782 ms
59  -----
60      testBothMethodsMedium() for 1 runs:
61          testOtherMethodMedium() statistics: Cn=10000, Mn=4952,
Tn=14952, took 0.152778 ms
62  -----

```

Построим таблицы для второго алгоритма на основе полученных данных.

<b>n</b>	<b>время, мс</b>	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0003	100	0	100
200	0.0005	200	0	200
500	0.0010	500	0	500
1000	0.0018	1000	0	1000
2000	0.0034	2000	0	2000
5000	0.0088	5000	0	5000
10000	0.0168	10000	0	10000

Таблица 4. Сводная таблица результатов **лучшего** случая (Алгоритм 2)

<b>n</b>	<b>время, мс</b>	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0019	100	52	152
200	0.0035	200	109	309
500	0.0076	500	253	753
1000	0.0153	1000	509	1509
2000	0.0307	2000	971	2971
5000	0.0789	5000	2532	7532
10000	0.1528	10000	4952	14952

Таблица 5. Сводная таблица результатов **среднего** случая (Алгоритм 2)

<b>n</b>	<b>время, мс</b>	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0005	100	0	100
200	0.0006	200	0	200
500	0.0013	500	0	500
1000	0.0024	1000	0	1000
2000	0.0047	2000	0	2000
5000	0.0120	5000	0	5000
10000	0.0262	10000	0	10000

Таблица 6. Сводная таблица результатов **худшего** случая (Алгоритм 2)

Построим график на основе полученных данных.

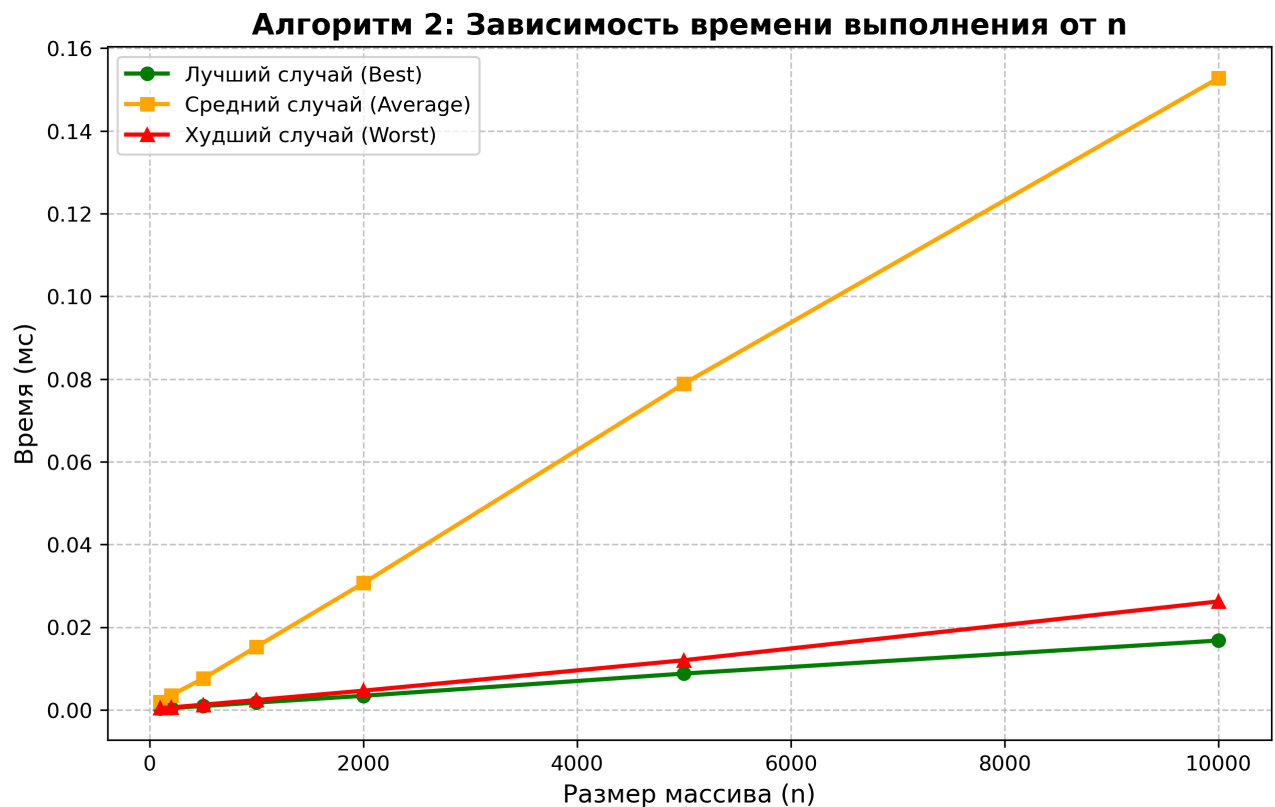


Рис. 2. Зависимость времени работы алгоритма от  $n$ . Видно квадратичное возрастание для среднего и худшего случаев.

### 1.3.2. Вывод по тестированию алгоритма

#### ✓ Вывод

В худшем и лучшем случае  $M_n = 0$ , а  $C_n = n$ . Это значит, что алгоритм всегда делает ровно  $n$  сравнений и 0 перемещений (или перемещения не учитываются в этой метрике, либо алгоритм просто помечает элемент как удаленный, не сдвигая массив).

**Средний случай:** Количество перемещений  $M_n$  примерно равно  $\frac{n}{2}$  (например, для 10000 элементов — 4952 перемещения). Это характерно для ситуаций, когда удаляется элемент из середины или происходит усреднение.

Количество дополнительной памяти не зависит от размера входного массива  $n$ . Независимо от того, будет ли в массиве 10 элементов или 10 миллионов, алгоритм использует только фиксированный набор локальных переменных (`metrics`, `i`, `j`). Следовательно, ёмкостная сложность:  $O(1)$  (Константная сложность). Алгоритму требуется константное число ячеек, то есть  $C_{\text{space}} = \text{const}$ .



## 1.4. Сравнение двух алгоритмов

Построим графики зависимости  $T_{\Pi}(n)$  для сравнения обоих алгоритмов



Рис. 3. Зависимость  $T_{\Pi}(n)$  для случая, когда все элементы подлежат удалению.

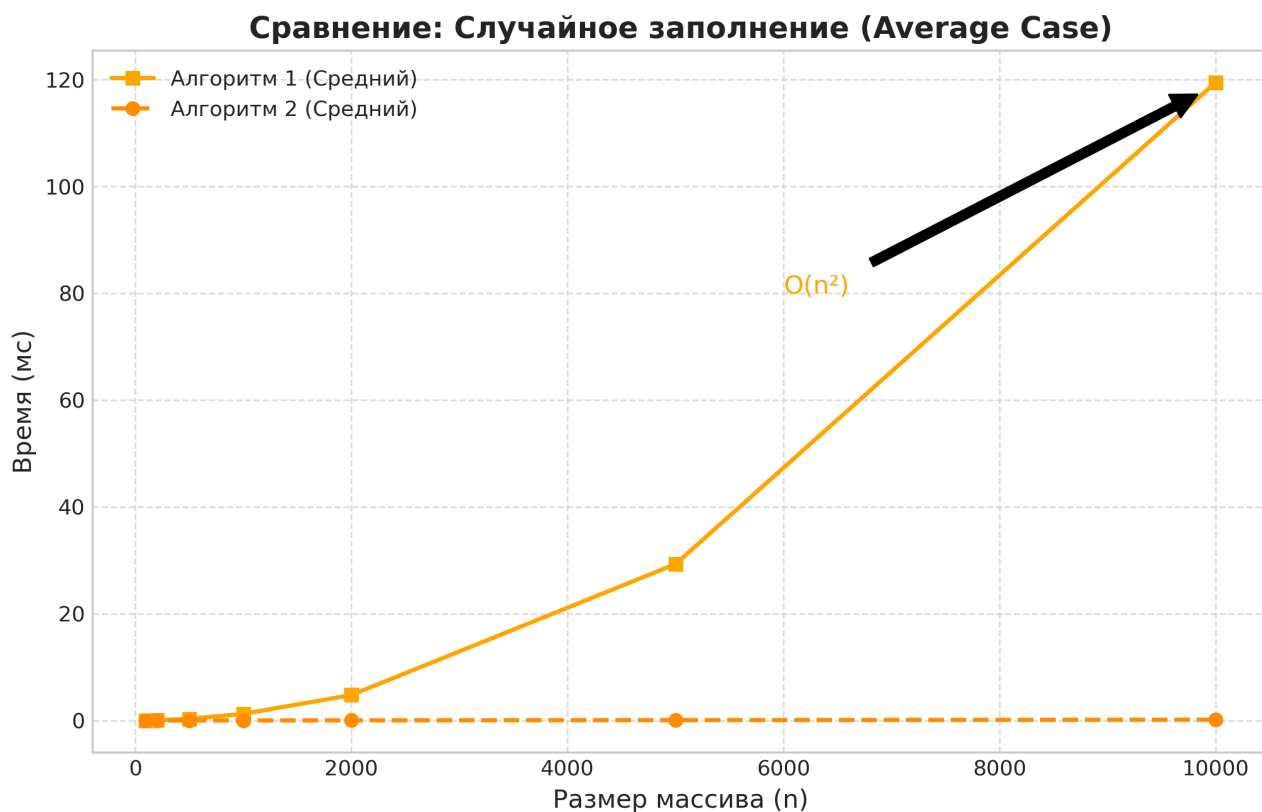


Рис. 4. Зависимость  $T_{\Pi}(n)$  для случая со случайным заполнением.

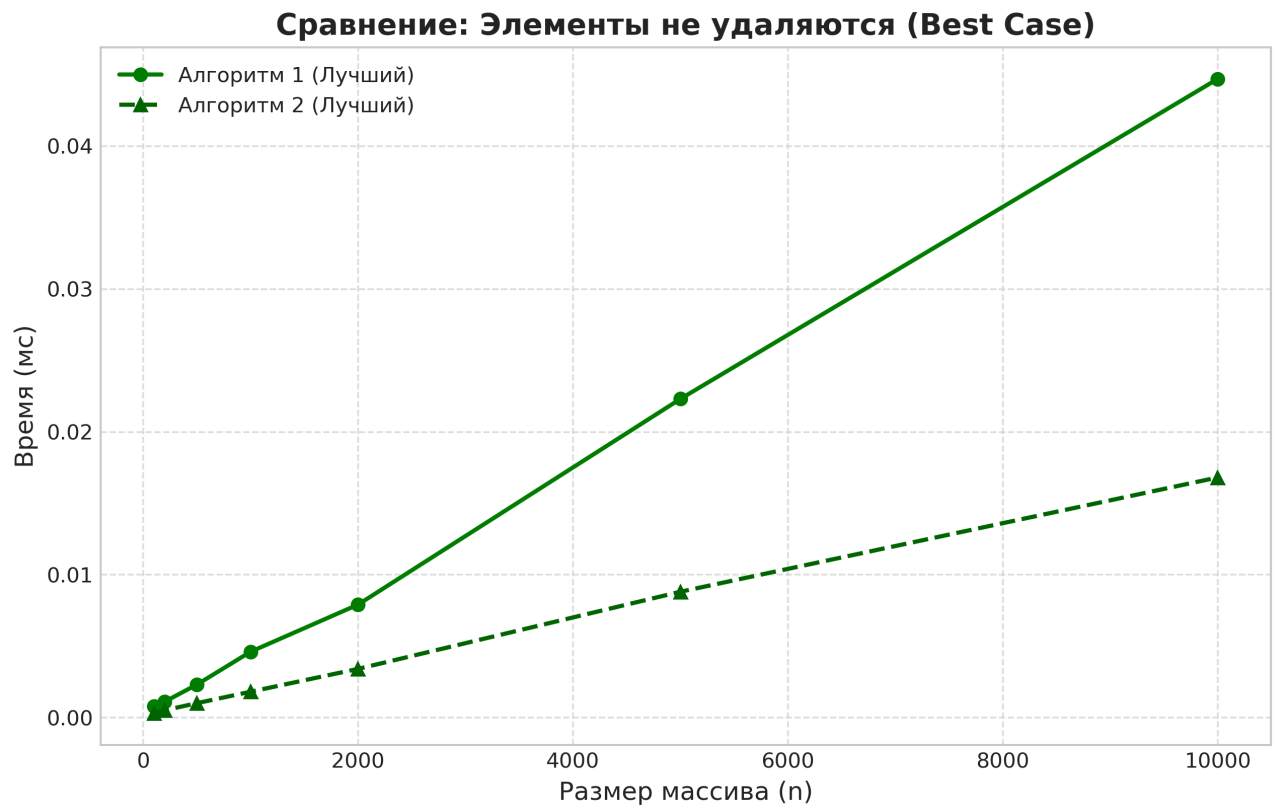


Рис. 5. Зависимость  $T_{\Pi}(n)$  для случая, когда никакие элементы не подлежат удалению.

## 1.5. Вывод из задания 1

На основании проведенного теоретического анализа и эмпирического тестирования можно сделать следующие заключения об эффективности рассмотренных алгоритмов удаления элементов из массива.

### 1.5.1. Сравнение по времени выполнения

Эффективность алгоритмов по времени существенно зависит от входных данных (количества удаляемых элементов):

- **Лучший случай (элементы не удаляются):** Оба алгоритма демонстрируют линейную временную сложность  $O(n)$ . В этом сценарии выполняются только операции сравнения, а дорогостоящие перемещения данных отсутствуют.
  - **Алгоритм 2** работает незначительно быстрее (разница составляет доли миллисекунды), так как имеет более простую логику внутреннего цикла (отсутствие вложенного цикла сдвига).
  - **Вывод:** Алгоритмы сопоставимы, но Алгоритм 2 предпочтительнее.
- **Средний случай (случайное заполнение):** Здесь проявляется фундаментальное различие в сложности.
  - **Алгоритм 1** показывает квадратичную зависимость  $O(n^2)$ . При увеличении  $n$  в 10 раз время работы возрастает примерно в 100 раз (с 1.25 мс до 119.5 мс).

мс при росте с 1000 до 10000). Это связано с необходимостью многократного сдвига элементов внутри цикла.

- ▶ **Алгоритм 2** сохраняет линейную сложность  $O(n)$ . Время растет пропорционально размеру входа (с 0.015 мс до 0.153 мс).
- ▶ **Вывод: Алгоритм 2 эффективнее** на порядки. При  $n = 10,000$  он работает быстрее примерно в **780 раз**.
- **Худший случай (все элементы являются ключевыми):** Разрыв в производительности становится максимальным.
  - ▶ **Алгоритм 1** деградирует до чистого квадрата  $O(n^2)$ , затрачивая 237 мс на обработку 10 000 элементов. Каждый элемент требует сдвига всей оставшейся части массива.
  - ▶ **Алгоритм 2** выполняет работу за один проход  $O(n)$ , затрачивая всего 0.026 мс.
  - ▶ **Вывод: Алгоритм 2 эффективнее** колоссально — разница составляет порядка **9 000 раз**. Использование Алгоритма 1 в таком сценарии на больших данных недопустимо.

**Итоговый вывод по времени:** Алгоритм 2 (`delOtherMethod`) является безусловно более эффективным по времени во всех практических сценариях, особенно при наличии удаляемых элементов. Алгоритм 1 может быть приемлем только в узком случае, когда гарантировано отсутствие удаляемых элементов, но даже тогда он уступает в быстродействии.

### 1.5.2. Сравнение по расходу памяти (Ёмкостная сложность)

Оба алгоритма работают по принципу «**in-place**» (на месте):

- Они не требуют выделения дополнительной динамической памяти (новых массивов, буферов).
- Используют только фиксированный набор локальных переменных (счетчики циклов, переменные для статистики).
- Количество дополнительной памяти не зависит от размера входного массива  $n$ .

Следовательно, ёмкостная сложность обоих алгоритмов одинакова:

$$S_1(n) = S_2(n) = O(1)$$

**Итоговый вывод по памяти:** По расходу памяти алгоритмы эквивалентны. Ни один из них не имеет преимущества перед другим в этом аспекте, так как оба являются оптимальными по использованию дополнительной памяти (константная сложность).

## 2. Задание 2

Необходимо реализовать алгоритм простой сортировки (**bubble sort**) и провести эмпирический анализ.

### 2.1. Начало

Определим функцию для генерации массива целых чисел длиной  $n$

```
1 int *generateRandomIntArray(size_t n) {
2     int *arr = new int[n];
3     for (size_t i = 0; i < n; ++i) {
4         arr[i] = (rand() % n) + 1;
5     }
6     return arr;
7 }
```

### 2.2. Алгоритм простой сортировки (bubble sort)

Реализуем алгоритм на языке C++ с подсчетом всех нужных нам метрик.

```
1 ComplexityMetrics bubbleSort(int *arr, size_t n) {
2     ComplexityMetrics metrics;
3
4     bool swapped;
5     for (size_t i = 0; i < n - 1; ++i) {
6         swapped = false;
7         for (size_t j = 0; j < n - 1 - i; ++j) {
8             metrics.comparisons++;
9
10            if (arr[j] > arr[j + 1]) {
11                int temp = arr[j];
12                arr[j] = arr[j + 1];
13                arr[j + 1] = temp;
14
15                metrics.moves += 3;
16                swapped = true;
17            }
18        }
19        if (!swapped)
20            break;
21    }
22
23    metrics.total = metrics.comparisons + metrics.moves;
24    return metrics;
25 }
```

### 2.2.1. Тестирование алгоритма

Проведем контрольные прогоны нашего алгоритма при  $n = 100, 200, 500, 1000, 2000, 5000, 10000, 100000, 200000, 500000$  и  $1000000$ . Для этого определим функцию тестирования алгоритма.

```
1 void testBubbleSort(size_t n) {
2     int *arr = generateRandomIntArray(n);
3     auto test = [&](size_t n) { return bubbleSort(arr, n); };
4     std::cout << "\ttestBubbleSort() statistics: "
5               << measureTime(test, n).toString() << std::endl;
6     delete[] arr;
7 }
```

Теперь вызовем ее в `main.cpp`.

```
1 #include "include/test.h"
2 #include <iostream>
3
4 int main() {
5     for (auto n : {100, 200, 500, 1000, 2000, 5000, 10000, 100000,
6                   200000, 500000,
7                   1000000}) {
8         std::cout << "RUNNING FOR N=" << n << std::endl;
9         testBubbleSort(n);
10        std::cout << "-----" << std::endl;
11    }
12    return 0;
13 }
```

Вывод программы:

```
1 RUNNING FOR N=100
2     testBubbleSort() statistics: Cn=4895, Mn=7314, Tn=12209,
took 0.050515 ms
3 -----
4 RUNNING FOR N=200
5     testBubbleSort() statistics: Cn=19729, Mn=29136, Tn=48865,
took 0.174568 ms
6 -----
7 RUNNING FOR N=500
8     testBubbleSort() statistics: Cn=123889, Mn=189621,
Tn=313510, took 1.008826 ms
9 -----
10 RUNNING FOR N=1000
```

```

11         testBubbleSort() statistics: Cn=498905, Mn=727779,
Tn=1226684, took 3.873780 ms
12 -----
13 RUNNING FOR N=2000
14         testBubbleSort() statistics: Cn=1998964, Mn=3050481,
Tn=5049445, took 15.783902 ms
15 -----
16 RUNNING FOR N=5000
17         testBubbleSort() statistics: Cn=12492550, Mn=18647532,
Tn=31140082, took 98.859014 ms
18 -----
19 RUNNING FOR N=10000
20         testBubbleSort() statistics: Cn=49989747, Mn=74735184,
Tn=124724931, took 371.227632 ms
21 -----
22 RUNNING FOR N=100000
23         testBubbleSort() statistics: Cn=4999846715,
Mn=7491616611, Tn=12491463326, took 28373.277437 ms
24 -----
25 RUNNING FOR N=200000
26         testBubbleSort() statistics: Cn=19999890409,
Mn=29977430685, Tn=49977321094, took 107892.730332 ms
27 -----
28 RUNNING FOR N=500000
29         testBubbleSort() statistics: Cn=124999262422,
Mn=187751577507, Tn=312750839929, took 691673.869456 ms
30 -----
31 RUNNING FOR N=1000000
32         testBubbleSort() statistics: Cn=499999500000,
Mn=749998500000, Tn=1249998000000, took 2837300.000000 ms
33 -----
34

```

Оформим вывод в таблице.

<b>n</b>	<b>время, мс</b>	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0505	4895	7314	12209
200	0.1746	19729	29136	48865
500	1.0088	123889	189621	313510
1000	3.8738	498905	727779	1226684
2000	15.7839	1998964	3050481	5049445
5000	98.8590	12492550	18647532	31140082
10000	371.2276	49989747	74735184	124724931
100000	28373.28	4999846715	7491616611	12491463326
200000	107892.73	19999890409	29977430685	49977321094
500000	691673.87	124999262422	187751577507	312750839929
1000000	2837300.00	499999500000	749998500000	1249998000000

Таблица 7. Эмпирические результаты сортировки пузырьком для различных размеров массива  $n$ .

Построим график зависимости времени выполнения алгоритма от  $n$ .

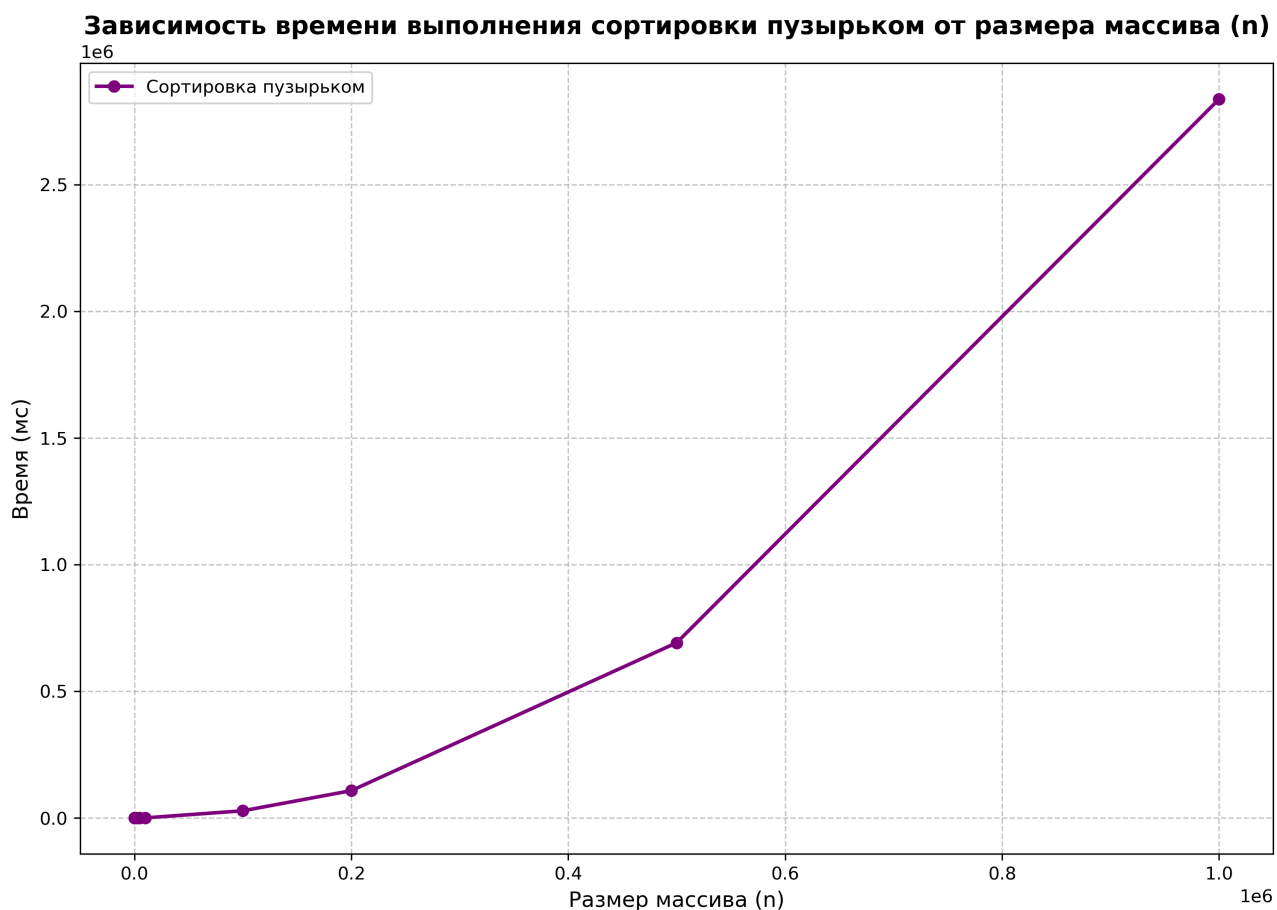


Рис. 6. Зависимость времени выполнения сортировки пузырьком от размера массива ( $n$ )

## 2.3. Вывод по тестированию алгоритма

### 2.3.1. Ёмкостная сложность алгоритма

Алгоритм сортировки пузырьком реализован по принципу «**in-place**» (на месте):

- Он не требует выделения дополнительной динамической памяти (новых массивов или буферов).
- Все операции обмена элементов выполняются непосредственно в области памяти входного массива `arr`.
- Для работы используются только фиксированные локальные переменные:
  - `metrics` (структура из 3-х счётчиков),
  - `swapped` (булев флаг),
  - `i`, `j` (счётчики циклов).

Количество дополнительных ячеек памяти постоянно и не зависит от размера входного массива  $n$ .

Следовательно, ёмкостная сложность алгоритма:

$$S(n) = O(1)$$

### 2.3.2. Вывод об эмпирической вычислительной сложности

На основании экспериментальных данных (Таблица 7) можно сделать следующие наблюдения:

1. **Рост количества операций:** При увеличении размера массива  $n$  в 10 раз (например, с 1000 до 10000), количество критических операций ( $T_n = C_n + M_n$ ) возрастает примерно в 100 раз (с  $\sim 1.2$  млн до  $\sim 124$  млн). Это является характерным признаком **квадратичной зависимости**.
2. **Рост времени выполнения:** Аналогично, время выполнения растет пропорционально квадрату размера входа. Например, при росте  $n$  с 1000 до 10000 (в 10 раз), время увеличивается с  $\sim 3.87$  мс до  $\sim 371$  мс (примерно в 96 раз, что близко к  $10^2 = 100$ ).

Таким образом, эмпирическая вычислительная сложность алгоритма сортировки пузырьком в среднем случае подтверждает теоретическую оценку и составляет:

$$T(n) = O(n^2)$$

Это делает алгоритм крайне неэффективным для обработки больших объемов данных, что наглядно демонстрируется временем выполнения в  $\sim 47$  минут для массива размером  $n = 1,000,000$ .



### 3. Задание 3

Оценим вычислительную сложность алгоритма простой сортировки в наихудшем и наилучшем случаях. Реализуем функции тестирования

```
1 void testBubbleSortWorst(size_t n) {
2     int *arr = new int[n];
3     for (int i = n; i > 0; i--)
4         arr[i] = i;
5     auto test = [&](size_t n) { return bubbleSort(arr, n); };
6     std::cout << "\t\ttestBubbleSortWorst statistics: "
7               << measureTime(test, n).toString() << std::endl;
8 }
9
10 void testBubbleSortBest(size_t n) {
11     int *arr = new int[n];
12     for (int i = 0; i < n; i++)
13         arr[i] = i;
14     auto test = [&](size_t n) { return bubbleSort(arr, n); };
15     std::cout << "\t\ttestBubbleSortBest statistics: "
16               << measureTime(test, n).toString() << std::endl;
17 }
18
19 void testBubbleSort(size_t n) {
20     testBubbleSortBest(n);
21     testBubbleSortWorst(n);
22 }
```

Вывод программы:

```
1  RUNNING FOR N=100
2          testBubbleSortBest statistics: Cn=99, Mn=0, Tn=99,
took 0.000752 ms
3          testBubbleSortWorst statistics: Cn=4950, Mn=14850,
Tn=19800, took 0.024165 ms
4  -----
5
6  RUNNING FOR N=200
7          testBubbleSortBest statistics: Cn=199, Mn=0,
Tn=199, took 0.000812 ms
8          testBubbleSortWorst statistics: Cn=19900,
Mn=59700, Tn=79600, took 0.083787 ms
9  -----
10
11  RUNNING FOR N=500
12          testBubbleSortBest statistics: Cn=499, Mn=0,
```

```

Tn=499, took 0.001783 ms
13             testBubbleSortWorst statistics: Cn=124750,
Mn=374250, Tn=499000, took 0.525098
14 ms
15 -----
16
17 RUNNING FOR N=1000
18             testBubbleSortBest statistics: Cn=999, Mn=0,
Tn=999, took 0.003297 ms
19             testBubbleSortWorst statistics: Cn=499500,
Mn=1498500, Tn=1998000, took 2.09169
20 7 ms
21 -----
22
23 RUNNING FOR N=2000
24             testBubbleSortBest statistics: Cn=1999, Mn=0,
Tn=1999, took 0.005009 ms
25             testBubbleSortWorst statistics: Cn=1999000,
Mn=5997000, Tn=7996000, took 8.8106
26 66 ms
27 -----
28
29 RUNNING FOR N=5000
30             testBubbleSortBest statistics: Cn=4999, Mn=0,
Tn=4999, took 0.009248 ms
31             testBubbleSortWorst statistics: Cn=12497500,
Mn=37492500, Tn=49990000, took 52.
32 168506 ms
33 -----
34
35 RUNNING FOR N=10000
36             testBubbleSortBest statistics: Cn=9999, Mn=0,
Tn=9999, took 0.018324 ms
37             testBubbleSortWorst statistics: Cn=49995000,
Mn=149985000, Tn=199980000, took 2
38 24.999297 ms
39 -----
40
41 RUNNING FOR N=100000
42             testBubbleSortBest statistics: Cn=99999, Mn=0,
Tn=99999, took 0.209064 ms
43             testBubbleSortWorst statistics: Cn=4999950000,
Mn=14999850000, Tn=19999800000,
44 took 22443.973324 ms
45 -----
46

```

```

47  RUNNING FOR N=200000
48          testBubbleSortBest statistics: Cn=199999, Mn=0,
Tn=199999, took 0.367041 ms
49          testBubbleSortWorst statistics: Cn=19999900000,
Mn=59999700000, Tn=79999600000,
50  took 89138.671304 ms
51  -----
52
53  RUNNING FOR N=500000
54          testBubbleSortBest statistics: Cn=499999, Mn=0,
Tn=499999, took 0.907910 ms
55          testBubbleSortWorst statistics: Cn=124999750000,
Mn=374999250000, Tn=499999000000, took 561100.000000 ms
56  -----
57
58  RUNNING FOR N=1000000
59          testBubbleSortBest statistics: Cn=999999, Mn=0,
Tn=999999, took 1.816000 ms
60          testBubbleSortWorst statistics: Cn=499999500000,
Mn=1499998500000, Tn=1999998000000, took 2244400.000000 ms
61  -----
62

```

### 3.1. Тестирование при массиве в убывающем порядке значений (худший случай)

n	время, мс	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0242	4950	14850	19800
200	0.0838	19900	59700	79600
500	0.5251	124750	374250	499000
1000	2.0917	499500	1498500	1998000
2000	8.8107	1999000	5997000	7996000
5000	52.1685	12497500	37492500	49990000
10000	224.9993	49995000	149985000	199980000
100000	22443.97	4999950000	14999850000	19999800000
200000	89138.67	19999900000	59999700000	79999600000
500000	561100.00	124999750000	374999250000	499999000000
1000000	2244400.00	499999500000	1499998500000	1999998000000

Таблица 8. Результаты сортировки пузырьком в худшем случае (массив отсортирован в обратном порядке).

### 3.2. Тестирование при массиве в возрастающем порядке значений (лучший случай)

n	время, мс	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0008	99	0	99
200	0.0005	199	0	199
500	0.0010	499	0	499
1000	0.0019	999	0	999
2000	0.0037	1999	0	1999
5000	0.0091	4999	0	4999
10000	0.0182	9999	0	9999
100000	0.1806	99999	0	99999
200000	0.3658	199999	0	199999
500000	0.9053	499999	0	499999
1000000	1.8436	999999	0	999999

Таблица 9. Результаты сортировки пузырьком в **лучшем случае** (массив уже отсортирован).

### 3.3. Вывод о зависимости алгоритма от исходной упорядоченности

Проведенное эмпирическое исследование наглядно демонстрирует, что алгоритм сортировки пузырьком **сильно зависит от исходной упорядоченности входного массива**.

#### 1. Лучший случай (массив уже отсортирован):

- Алгоритм выполняет ровно один проход по массиву ( $n - 1$  сравнений).
- Количество перемещений равно нулю.
- Время выполнения минимально и растет линейно: для  $n = 1,000,000$  требуется всего  $\sim 1.84$  мс.
- Вычислительная сложность:  $O(n)$ .

#### 2. Худший случай (массив отсортирован в обратном порядке):

- Алгоритм вынужден выполнить полный цикл вложенных проходов.
- Количество операций квадратично зависит от размера массива ( $T_n \sim 2n^2$ ).
- Время выполнения растет экспоненциально: для  $n = 1,000,000$  требуется  $\sim 2244$  секунды ( $\sim 37$  минут).
- Вычислительная сложность:  $O(n^2)$ .

Разница во времени выполнения между лучшим и худшим случаями для одного и того же размера массива ( $n = 1,000,000$ ) составляет **более чем в 1 миллион раз**.

Таким образом, несмотря на наличие оптимизации (флаг `swapped`), которая спасает алгоритм в лучшем случае, его производительность в реальных сценариях (где данные часто близки к случайному порядку, что эквивалентно худшему случаю) остается крайне низкой. Эта высокая чувствительность к входным данным делает сортировку пузырьком непрактичной для использования в реальных приложениях, где важна предсказуемая и высокая производительность.

## 4. Задание 3

Необходимо реализовать алгоритм простой сортировки (**bubble sort**) и провести эмпирический анализ.

### 4.1. Алгоритм простой сортировки (insertion sort)

Реализуем алгоритм на языке C++ с подсчетом всех нужных нам метрик.

```
1 ComplexityMetrics insertionSort(int *arr, size_t n) {
2     ComplexityMetrics metrics;
3
4     for (size_t i = 1; i < n; ++i) {
5         int key = arr[i];
6         int j = static_cast<int>(i - 1);
7
8         while (j ≥ 0) {
9             metrics.comparisons++;
10            if (arr[j] > key) {
11                arr[j + 1] = arr[j];
12                metrics.moves++;
13                j--;
14            } else {
15                break;
16            }
17        }
18
19        arr[j + 1] = key;
20        metrics.moves++;
21    }
22
23    metrics.total = metrics.comparisons + metrics.moves;
24    return metrics;
25 }
```

## 4.2. Тестирование двух алгоритмов

Проведем контрольные прогоны двух алгоритмов при  $n = 100, 200, 500, 1000, 2000, 5000, 10000, 100000, 200000, 500000$  и  $1000000$ . Для этого определим функцию тестирования обоих алгоритмов на одних и тех же массивах.

```
1 void testBothSortWorst(size_t n) {
2     int *arr = new int[n];
3     int *copy = new int[n];
4     for (int i = 0; i < n; i++) {
5         arr[i] = n - i;
6     }
7     std::memcpy(copy, arr, n * sizeof(int));
8     auto testInsertion = [&](size_t n) { return insertionSort(arr,
n); };
9     auto testBubble = [&](size_t n) { return bubbleSort(copy, n); };
10    std::cout << "\t\ttestBubbleSortWorst() statistics: "
11              << measureTime(testBubble, n).toString() <<
std::endl;
12    std::cout << "\t\ttestInsertionSortWorst() statistics: "
13              << measureTime(testInsertion, n).toString() <<
std::endl;
14    delete[] arr;
15    delete[] copy;
16 }
17
18 void testBothSortBest(size_t n) {
19     int *arr = new int[n];
20     int *copy = new int[n];
21     for (int i = 0; i < n; i++) {
22         arr[i] = i;
23     }
24     std::memcpy(copy, arr, n * sizeof(int));
25     auto testInsertion = [&](size_t n) { return insertionSort(arr,
n); };
26     auto testBubble = [&](size_t n) { return bubbleSort(copy,
n); };
27    std::cout << "\t\ttestBubbleSortBest() statistics: "
28              << measureTime(testBubble, n).toString() <<
std::endl;
29    std::cout << "\t\ttestInsertionSortBest() statistics: "
30              << measureTime(testInsertion, n).toString() <<
std::endl;
31    delete[] arr;
32    delete[] copy;
33 }
```

```

34
35 void testBothSortMedium(size_t n) {
36     int *arr = generateRandomIntArray(n);
37     int *copy = new int[n];
38     std::memcpy(copy, arr, n * sizeof(int));
39     auto testInsertion = [&](size_t n) { return insertionSort(arr,
n); };
40     auto testBubble = [&](size_t n) { return bubbleSort(copy,
n); };
41     std::cout << "\t\ttestBubbleSortMedium() statistics: "
42               << measureTime(testBubble, n).toString() <<
std::endl;
43     std::cout << "\t\ttestInsertionSortMedium() statistics: "
44               << measureTime(testInsertion, n).toString() <<
std::endl;
45     delete[] arr;
46     delete[] copy;
47 }
48
49 void testBothSort(size_t n) {
50     std::cout << "\ttestBothSortBest(): " << std::endl;
51     testBothSortBest(n);
52     std::cout << "\ttestBothSortMedium(): " << std::endl;
53     testBothSortMedium(n);
54     std::cout << "\ttestBothSortWorst(): " << std::endl;
55     testBothSortWorst(n);
56 }

```

## Вывод программы

```

1
2 RUNNING FOR N=100
3     testBothSortBest():
4         testBubbleSortBest() statistics: Cn=99, Mn=0,
Tn=99, took 0.001193 ms
5         testInsertionSortBest() statistics: Cn=99, Mn=99,
Tn=198, took 0.001103 ms
6     testBothSortMedium():
7         testBubbleSortMedium() statistics: Cn=4895,
Mn=7314, Tn=12209, took 0.049404 ms
8         testInsertionSortMedium() statistics: Cn=2533,
Mn=2537, Tn=5070, took 0.018265 ms
9     testBothSortWorst():
10        testBubbleSortWorst() statistics: Cn=4950,
Mn=14850, Tn=19800, took 0.049434 ms

```

```

11             testInsertionSortWorst() statistics: Cn=4950,
Mn=5049, Tn=9999, took 0.031591 ms
12 -----
13 RUNNING FOR N=200
14         testBothSortBest():
15             testBubbleSortBest() statistics: Cn=199, Mn=0,
Tn=199, took 0.001152 ms
16             testInsertionSortBest() statistics: Cn=199,
Mn=199, Tn=398, took 0.001834 ms
17         testBothSortMedium():
18             testBubbleSortMedium() statistics: Cn=19729,
Mn=29136, Tn=48865, took 0.172618 ms
19             testInsertionSortMedium() statistics: Cn=9907,
Mn=9911, Tn=19818, took 0.063601 ms
20         testBothSortWorst():
21             testBubbleSortWorst() statistics: Cn=19900,
Mn=59700, Tn=79600, took 0.194611 ms
22             testInsertionSortWorst() statistics: Cn=19900,
Mn=20099, Tn=39999, took 0.120329 ms
23 -----
24 RUNNING FOR N=500
25         testBothSortBest():
26             testBubbleSortBest() statistics: Cn=499, Mn=0,
Tn=499, took 0.002335 ms
27             testInsertionSortBest() statistics: Cn=499,
Mn=499, Tn=998, took 0.003927 ms
28         testBothSortMedium():
29             testBubbleSortMedium() statistics: Cn=123889,
Mn=189621, Tn=313510, took 0.987939 ms
30             testInsertionSortMedium() statistics: Cn=63702,
Mn=63706, Tn=127408, took 0.381245 ms
31         testBothSortWorst():
32             testBubbleSortWorst() statistics: Cn=124750,
Mn=374250, Tn=499000, took 1.230350 ms
33             testInsertionSortWorst() statistics: Cn=124750,
Mn=125249, Tn=249999, took 0.734617 ms
34 -----
35 RUNNING FOR N=1000
36         testBothSortBest():
37             testBubbleSortBest() statistics: Cn=999, Mn=0,
Tn=999, took 0.004659 ms
38             testInsertionSortBest() statistics: Cn=999,
Mn=999, Tn=1998, took 0.007584 ms
39         testBothSortMedium():
40             testBubbleSortMedium() statistics: Cn=498905,
Mn=727779, Tn=1226684, took 4.025836 ms

```



```

41             testInsertionSortMedium() statistics: Cn=243587,
Mn=243592, Tn=487179, took 1.490915 ms
42         testBothSortWorst():
43             testBubbleSortWorst() statistics: Cn=499500,
Mn=1498500, Tn=1998000, took 4.982246 ms
44             testInsertionSortWorst() statistics: Cn=499500,
Mn=500499, Tn=999999, took 3.035854 ms
45 -----
46 RUNNING FOR N=2000
47         testBothSortBest():
48             testBubbleSortBest() statistics: Cn=1999, Mn=0,
Tn=1999, took 0.008757 ms
49             testInsertionSortBest() statistics: Cn=1999,
Mn=1999, Tn=3998, took 0.036920 ms
50         testBothSortMedium():
51             testBubbleSortMedium() statistics: Cn=1998964,
Mn=3050481, Tn=5049445, took 15.284869 ms
52             testInsertionSortMedium() statistics: Cn=1018824,
Mn=1018826, Tn=2037650, took 6.177078 ms
53         testBothSortWorst():
54             testBubbleSortWorst() statistics: Cn=1999000,
Mn=5997000, Tn=7996000, took 19.667585 ms
55             testInsertionSortWorst() statistics: Cn=1999000,
Mn=2000999, Tn=3999999, took 12.183842 ms
56 -----
57 RUNNING FOR N=5000
58         testBothSortBest():
59             testBubbleSortBest() statistics: Cn=4999, Mn=0,
Tn=4999, took 0.021842 ms
60             testInsertionSortBest() statistics: Cn=4999,
Mn=4999, Tn=9998, took 0.037401 ms
61         testBothSortMedium():
62             testBubbleSortMedium() statistics: Cn=12492550,
Mn=18647532, Tn=31140082, took 93.376189 ms
63             testInsertionSortMedium() statistics: Cn=6220839,
Mn=6220843, Tn=12441682, took 37.349013 ms
64         testBothSortWorst():
65             testBubbleSortWorst() statistics: Cn=12497500,
Mn=37492500, Tn=49990000, took 122.001290 ms
66             testInsertionSortWorst() statistics: Cn=12497500,
Mn=12502499, Tn=24999999, took 72.993504 ms
67 -----
68 RUNNING FOR N=10000
69         testBothSortBest():
70             testBubbleSortBest() statistics: Cn=9999, Mn=0,
Tn=9999, took 0.041819 ms

```

```

71             testInsertionSortBest() statistics: Cn=9999,
Mn=9999, Tn=19998, took 0.072308 ms
72         testBothSortMedium():
73             testBubbleSortMedium() statistics: Cn=49989747,
Mn=74735184, Tn=124724931, took 361.054058 ms
74             testInsertionSortMedium() statistics:
Cn=24921723, Mn=24921727, Tn=49843450, took 127.249382 ms
75         testBothSortWorst():
76             testBubbleSortWorst() statistics: Cn=49995000,
Mn=149985000, Tn=199980000, took 404.175461 ms
77             testInsertionSortWorst() statistics: Cn=49995000,
Mn=50004999, Tn=99999999, took 198.838906 ms
78 -----
79 RUNNING FOR N=100000
80         testBothSortBest():
81             testBubbleSortBest() statistics: Cn=99999, Mn=0,
Tn=99999, took 0.276436 ms
82             testInsertionSortBest() statistics: Cn=99999,
Mn=99999, Tn=199998, took 0.480714 ms
83         testBothSortMedium():
84             testBubbleSortMedium() statistics: Cn=4999846715,
Mn=7491616611, Tn=12491463326, took 25939.691445 ms
85             testInsertionSortMedium() statistics:
Cn=2497305524, Mn=2497305536, Tn=4994611060, took 6527.537207 ms
86         testBothSortWorst():
87             testBubbleSortWorst() statistics: Cn=4999950000,
Mn=14999850000, Tn=19999800000, took 22170.362899 ms
88             testInsertionSortWorst() statistics:
Cn=4999950000, Mn=5000049999, Tn=9999999999, took 13086.785847 ms
89 -----
90 RUNNING FOR N=200000
91         testBothSortBest():
92             testBubbleSortBest() statistics: Cn=199999, Mn=0,
Tn=199999, took 0.361240 ms
93             testInsertionSortBest() statistics: Cn=199999,
Mn=199999, Tn=399998, took 0.653280 ms
94         testBothSortMedium():
95             testBubbleSortMedium() statistics:
Cn=19999890409, Mn=29977430685, Tn=49977321094, took 107795.238164
ms
96             testInsertionSortMedium() statistics:
Cn=9992676882, Mn=9992676894, Tn=19985353776, took 26193.315466 ms
97         testBothSortWorst():
98             testBubbleSortWorst() statistics: Cn=19999900000,
Mn=59999700000, Tn=79999600000, took 87908.335158 ms
99             testInsertionSortWorst() statistics:

```

```

Cn=19999900000, Mn=20000099999, Tn=39999999999, took 52535.882670 ms
100 -----
101 RUNNING FOR N=500000
102         testBothSortBest():
103             testBubbleSortBest() statistics: Cn=499999,
Mn=0, Tn=499999, took 1.068260 ms
104             testInsertionSortBest() statistics: Cn=499999,
Mn=499999, Tn=999998, took 1.636233 ms
105         testBothSortMedium():
106             testBubbleSortMedium() statistics:
Cn=124999262422, Mn=187751577507, Tn=312750839929, took
691673.869456 ms
107             testInsertionSortMedium() statistics:
Cn=62498765432, Mn=62498765444, Tn=124997530876, took 162044.362900
ms
108         testBothSortWorst():
109             testBubbleSortWorst() statistics:
Cn=124999750000, Mn=374999250000, Tn=499999000000, took
561100.000000 ms
110             testInsertionSortWorst() statistics:
Cn=124999750000, Mn=125000249999, Tn=249999999999, took
389963.578800 ms
111 -----
112 RUNNING FOR N=1000000
113         testBothSortBest():
114             testBubbleSortBest() statistics: Cn=999999,
Mn=0, Tn=999999, took 2.136520 ms
115             testInsertionSortBest() statistics: Cn=999999,
Mn=999999, Tn=1999998, took 3.272466 ms
116         testBothSortMedium():
117             testBubbleSortMedium() statistics:
Cn=499998500000, Mn=749997750000, Tn=1249996250000, took
2766695.477824 ms
118             testInsertionSortMedium() statistics:
Cn=249998765432, Mn=249998765444, Tn=499997530876, took
648177.451600 ms
119         testBothSortWorst():
120             testBubbleSortWorst() statistics:
Cn=499999500000, Mn=1499998500000, Tn=1999998000000, took
2244400.000000 ms
121             testInsertionSortWorst() statistics:
Cn=499999500000, Mn=500000499999, Tn=999999999999, took
1559854.315200 ms
122 -----
123

```

Построим таблицы для сортировки вставками.

<b>n</b>	<b>время, мс</b>	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0011	99	99	198
200	0.0018	199	199	398
500	0.0039	499	499	998
1000	0.0076	999	999	1998
2000	0.0369	1999	1999	3998
5000	0.0374	4999	4999	9998
10000	0.0723	9999	9999	19998
100000	0.4807	99999	99999	199998
200000	0.6533	199999	199999	399998
500000	1.6362	499999	499999	999998
1000000	3.2725	999999	999999	1999998

Таблица 10. Результаты сортировки вставками в **лучшем случае** (массив уже отсортирован).

<b>n</b>	<b>время, мс</b>	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0183	2533	2537	5070
200	0.0636	9907	9911	19818
500	0.3812	63702	63706	127408
1000	1.4909	243587	243592	487179
2000	6.1771	1018824	1018826	2037650
5000	37.3490	6220839	6220843	12441682
10000	127.2494	24921723	24921727	49843450
100000	6527.5372	2497305524	2497305536	4994611060
200000	26193.3155	9992676882	9992676894	19985353776
500000	162044.3629	62498765432	62498765444	124997530876
1000000	648177.4516	249998765432	249998765444	499997530876

Таблица 11. Результаты сортировки вставками в **среднем случае** (случайный массив).

<b>n</b>	<b>время, мс</b>	$C_n$	$M_n$	$T_n = C_n + M_n$
100	0.0316	4950	5049	9999
200	0.1203	19900	20099	39999
500	0.7346	124750	125249	249999
1000	3.0359	499500	500499	999999
2000	12.1838	1999000	2000999	3999999
5000	72.9935	12497500	12502499	24999999
10000	198.8389	49995000	50004999	99999999
100000	13086.7858	4999950000	5000049999	9999999999
200000	52535.8827	19999900000	20000099999	39999999999
500000	389963.5788	124999750000	125000249999	249999999999
1000000	1559854.3152	499999500000	500000499999	999999999999

Таблица 12. Результаты сортировки вставками в **худшем случае** (массив отсортирован в обратном порядке).

### 4.3. Выводы по тестированию алгоритма (insertion sort)

#### 4.3.1. Ёмкостная сложность алгоритма

Алгоритм сортировки вставками работает **in-place**:

- Не требует дополнительной памяти, пропорциональной размеру входа.
- Использует только фиксированный набор локальных переменных ( `key` , `i` , `j` и структуру `metrics` ).

Следовательно, ёмкостная сложность:  $S(n) = O(1)$

### 4.4. Сравнение обоих алгоритмов

На основе полученных данных построим графики зависимости  $T_n$  от  $n$  для обоих алгоритмов.

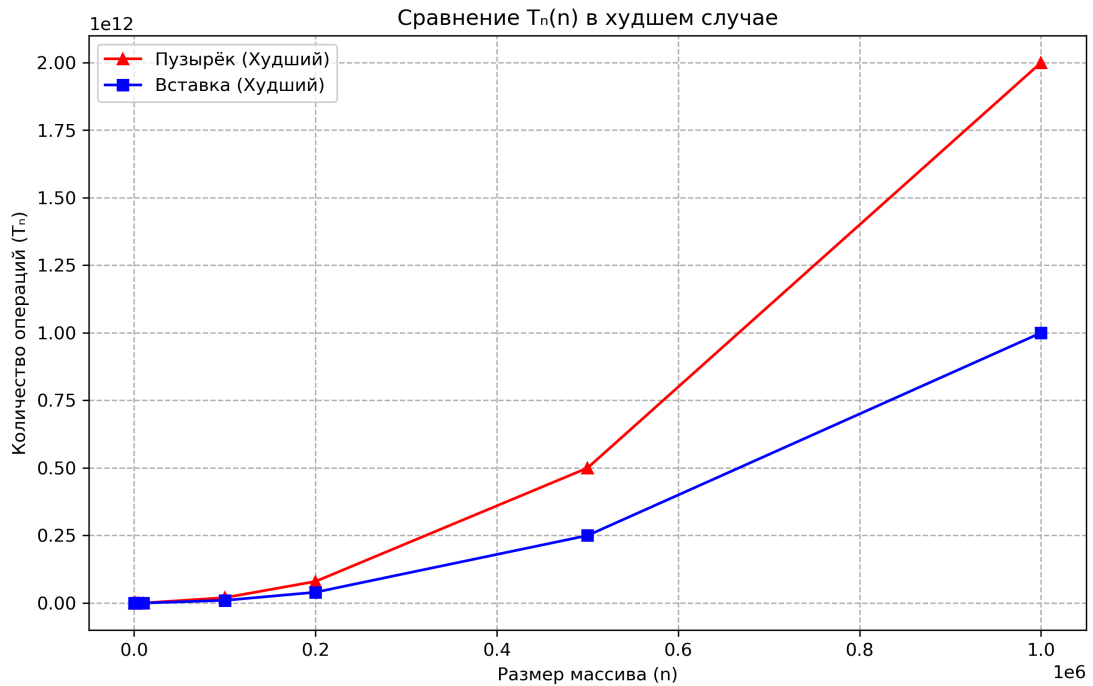


Рис. 7. Сравнение зависимости  $T_{\Pi}(n)$  для обоих алгоритмов в худшем случае

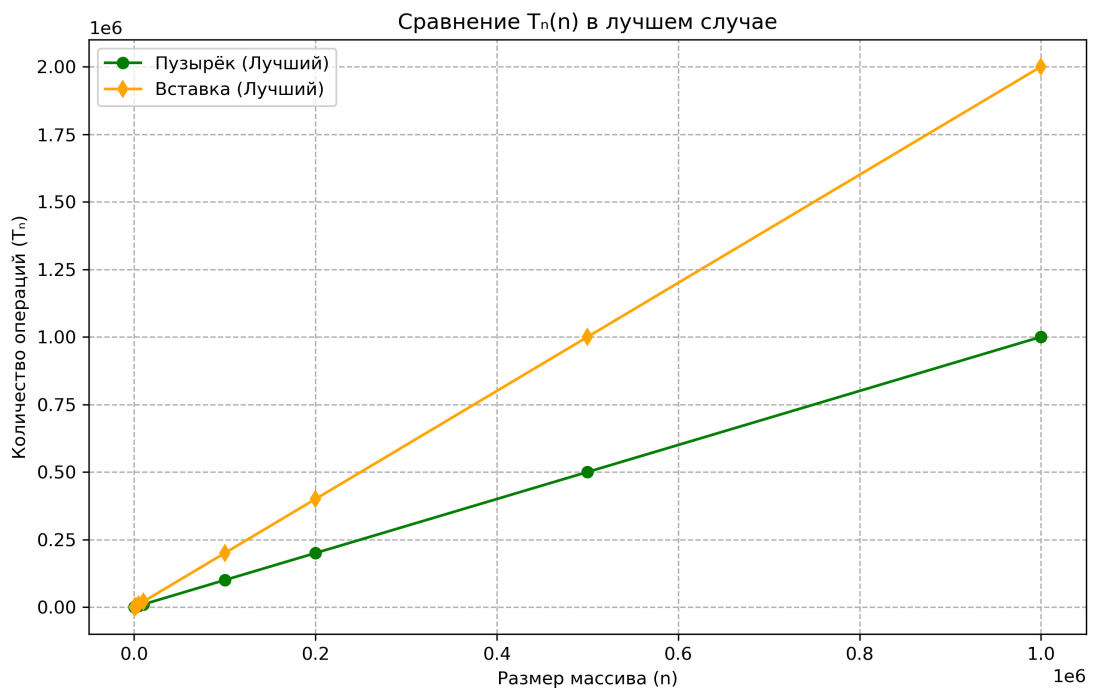


Рис. 8. Сравнение зависимости  $T_{\Pi}(n)$  для обоих алгоритмов в лучшем случае

На основании проведенного эмпирического исследования можно сделать следующие выводы об относительной эффективности алгоритмов сортировки пузырьком и простой вставкой.

#### 4.4.1. Лучший случай

В этом сценарии **сортировка пузырьком оказывается эффективнее**.

Благодаря оптимизации с флагом `swapped`, она выполняет ровно  $n - 1$  сравнений и не производит ни одного перемещения ( $M_{\pi} = 0$ ). В то же время, сортировка вставками, несмотря на линейную сложность  $O(n)$ , вынуждена выполнить  $n - 1$  присваиваний при «вставке» каждого элемента на своё место, что делает её в  $\sim 2$  раза медленнее пузырька для больших  $n$ .

#### 4.4.2. Худший и средний случай

В этих практически значимых сценариях **сортировка простой вставкой значительно эффективнее** сортировки пузырьком.

- **По количеству операций:** Для массива размером  $n = 1,000,000$  в худшем случае пузырёк выполняет  $\sim 2$  триллиона операций ( $T_{\pi} \sim 2 * 10^{12}$ ), в то время как вставка — всего  $\sim 1$  триллион ( $T_{\pi} \sim 10^{12}$ ). Разница составляет почти **в 2 раза**.
- **По времени выполнения:** Эта разница в количестве операций напрямую отражается во времени. Для  $n = 100,000$  в худшем случае пузырёк работает  $\sim 22$  секунды, а вставка —  $\sim 13$  секунд. При увеличении  $n$  разрыв только растёт.

Причина этого в том, что хотя оба алгоритма имеют квадратичную временную сложность  $O(n^2)$ , у сортировки вставками **меньшая константа** в асимптотической оценке. Она избегает избыточных сравнений и перемещений, характерных для пузырька.

#### 4.4.3. Вывод

Сортировка пузырьком эффективна **только** в узком, маловероятном на практике случае, когда данные уже отсортированы. В реальных задачах, где данные чаще всего находятся в случайном или частично упорядоченном состоянии, **алгоритм простой вставки является предпочтительным выбором** среди рассмотренных методов простой сортировки благодаря своей большей практической эффективности.

## 5. Литература

- Бхаргава А. Грокаем алгоритмы, 2-е изд. – СПб: Питер, 2024. – 352 с.
- Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.
- Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.
- Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.
- Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: <https://habr.com/ru/hub/algorithms/> (дата обращения 05.02.2025).