Содержание

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 ($C_п$) + перемещений элементов в памяти ($M_п$), а также суммарное число критических операций $T_п = C_п + M_п$.

```
struct ComplexityMetrics {
  size_t comparisons; // Cп — число сравнений
  size_t moves;       // Mп — число перемещений
  size_t total;       // Tп = Cп + Mп
  double duration;    // время выполнения функции (мс)

  ComplexityMetrics() : comparisons(0), moves(0), total(0), duration(0) {}

  std::string toString() const {
     return "Cп=" + std::to_string(comparisons) +
            ", Mп=" + std::to_string(moves) + ", Tп=" + std::to_string(total) +
            ", took " + std::to_string(duration) + " ms";
   }
};
```

Определим функцию для подсчета времени выполнения алгоритма:

```
#include <chrono>

using TestFunc = std::function<ComplexityMetrics(size_t)>;

ComplexityMetrics measureTime(TestFunc func, size_t len) {
  auto start = std::chrono::high_resolution_clock::now();
  ComplexityMetrics metrics = func(len);
  auto end = std::chrono::high_resolution_clock::now();

   std::chrono::duration<double, std::milli> duration = end - start;
   metrics.duration = duration.count();
   return metrics;
}
```

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

Определим функцию для генерации рандомно заполненного массива:

```
#include <random>

static std::mt19937 rng(std::random_device{}());

char *generateRandomArray(size_t size, const char key, const char otherChar) {
   char *arr = new char[size + 1];

   std::bernoulli_distribution dist(0.5);

    for (size_t i = 0; i < size; i++) {
      arr[i] = dist(rng) ? key : otherChar;
    }
    arr[size] = '\0';
    return arr;
}
```

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

1.2. Алгоритм №1

Реализуем первый метод на языке C++:

```
void delFirstMethod(char *x, size_t &n, const char &key) {
  size_t i = 0;

  while (i < n) {
    if (x[i] == key) {
      for (size_t j = i; j < n - 1; j++) {
        x[j] = x[j + 1];
      }
      n--;
    } else {
      i++;
    }
  }
}
```

Изменим код реализованного метода для возвращения метрик:

```
ComplexityMetrics delFirstMethod(char *x, size_t &n, const char &key) {
  ComplexityMetrics metrics;
  size_t i = 0;

  while (i < n) {
    metrics.comparisons++; // Сравнение x[i] == key
    if (x[i] == key) {
      for (size_t j = i; j < n - 1; j++) {
        metrics.moves++; // Перемещение x[j] = x[j + 1]
        x[j] = x[j + 1];
      }
      n--;
    } else {
      i++;
    }
  }
  metrics.total = metrics.comparisons + metrics.moves;
  return metrics;
}
```

Теперь функция возвращает метрику.

1.2.1. Тестирование алгоритма

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

```
void testFirstMethod(size_t n, int runs = 100) {
  std::cout << "\ttestFirstMethod() for " << runs << " runs:" << std::endl;
  std::cout << "\t\ttestFirstMethodWorst() statistics: "
            << testTimes(
                   [](size_t len) {
                     char x[len];
                     std::memset(x, '_', len);

                     return delFirstMethod(x, len, '_');
                   },
                   runs, n)
                   .toString()
            << std::endl;
  std::cout << "\t\ttestFirstMethodBest() statistics: "
            << testTimes(
                   [](size_t len) {
                     char x[len];
```

```
                    std::memset(x, 'A', len);

                    return delFirstMethod(x, len, '_');
                },
                runs, n)
                .toString()
            << std::endl;
}
```

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

```
void testBothMethodsMedium(size_t n, int runs = 100) {
  char *arr = generateRandomArray(n, '_', 'A');
  char *copy = new char[n + 1];
  std::memcpy(copy, arr, n);

  auto testFirst = [&](size_t n) { return delFirstMethod(arr, n,
'_'); };
  auto testOther = [&](size_t n) { return delOtherMethod(copy, n,
'_'); };
  std::cout << "\ttestBothMethodsMedium() for " << runs
            << " runs: " << std::endl;
   std::cout << "\t\ttestFirstMethodMedium() statistics: "
             << measureTime(testFirst, n).toString() << std::endl;
  std::cout << "\t\ttestOtherMethodMedium() statistics: "
             << measureTime(testOther, n).toString() << std::endl;

  delete[] arr;
  delete[] copy;
}
```

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:

```
#include "include/test.h"
#include <iostream>

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  ---------
```

```
  testBothMethodsMedium() for 1 runs:
    testFirstMethodMedium() statistics: Cп=500, Mп=61134,
Tп=61634, took 0.302020 ms
---------

RUNNING FOR N=1000
  testFirstMethod() for 1 runs:
    testFirstMethodWorst() statistics: Cп=1000, Mп=499500,
Tп=500500, took 2.427775 ms
    testFirstMethodBest() statistics: Cп=1000, Mп=0, Tп=1000,
took 0.004569 ms
---------
  testBothMethodsMedium() for 1 runs:
    testFirstMethodMedium() statistics: Cп=1000, Mп=254091,
Tп=255091, took 1.250503 ms
---------

RUNNING FOR N=2000
  testFirstMethod() for 1 runs:
    testFirstMethodWorst() statistics: Cп=2000, Mп=1999000,
Tп=2001000, took 9.523869 ms
    testFirstMethodBest() statistics: Cп=2000, Mп=0, Tп=2000,
took 0.007865 ms
---------
  testBothMethodsMedium() for 1 runs:
    testFirstMethodMedium() statistics: Cп=2000, Mп=1005082,
Tп=1007082, took 4.767976 ms
---------

RUNNING FOR N=5000
  testFirstMethod() for 1 runs:
    testFirstMethodWorst() statistics: Cп=5000, Mп=12497500,
Tп=12502500, took 59.287127 ms
    testFirstMethodBest() statistics: Cп=5000, Mп=0, Tп=5000,
took 0.022282 ms
---------
  testBothMethodsMedium() for 1 runs:
    testFirstMethodMedium() statistics: Cп=5000, Mп=6178707,
Tп=6183707, took 29.289154 ms
---------

RUNNING FOR N=10000
  testFirstMethod() for 1 runs:
    testFirstMethodWorst() statistics: Cп=10000, Mп=49995000,
Tп=50005000, took 236.746269 ms
    testFirstMethodBest() statistics: Cп=10000, Mп=0, Tп=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.

n	**время, мс**	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
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

n	**время, мс**	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
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

n	**время, мс**	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
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.

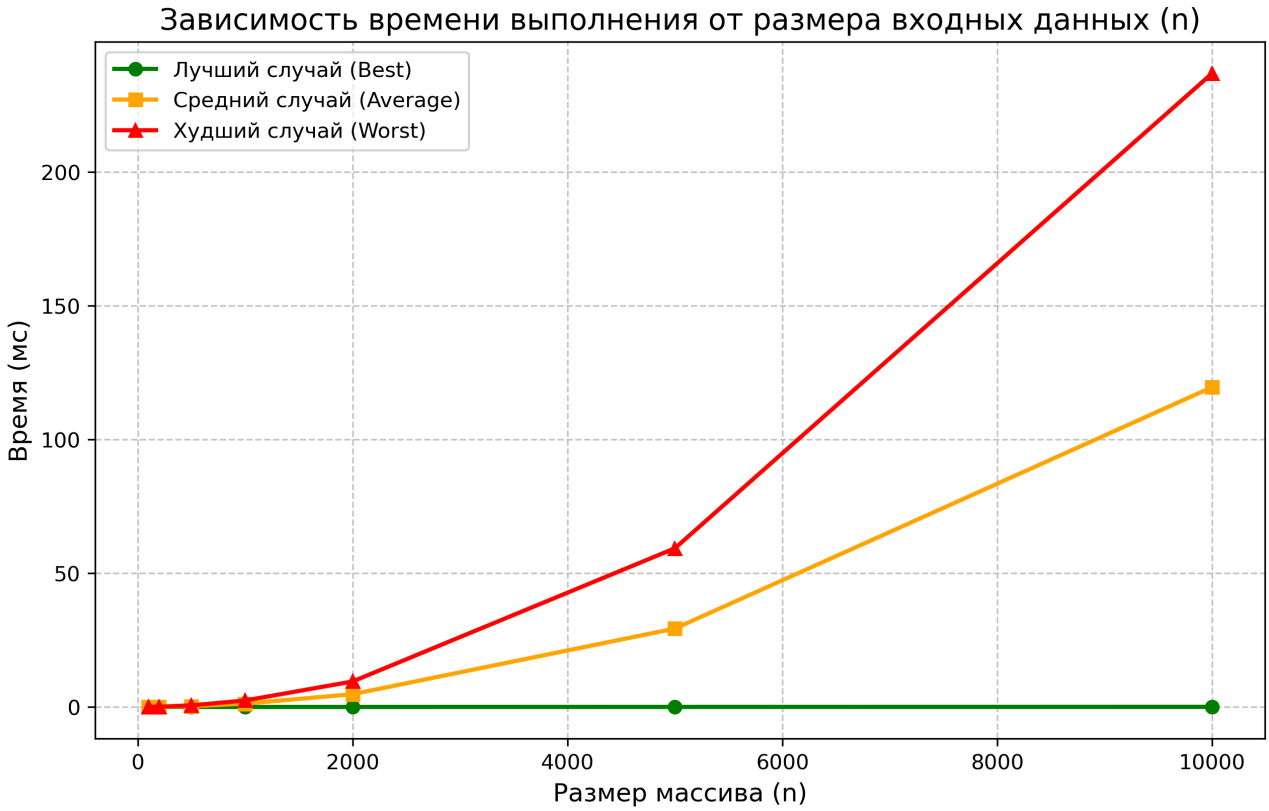

Рис. 1. Зависимость времени работы алгоритма от n. Видно квадратичное возрастание для среднего и худшего случаев.

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 $C_{\text{space}} = const$.

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
```

```
    for (size_t i = 0; i < n; i++) {
      if (x[i] != key) {
        if (i != j) {
          x[j] = x[i];
        }
        j++;
      }
    }
    n = j;
}
```

1.3.1. Тестирование алгоритма

Изменим код реализованного метода для возвращения метрик:

```
ComplexityMetrics delOtherMethod(char *x, size_t &n, const char &key) {
    ComplexityMetrics metrics;
    size_t j = 0;

    for (size_t i = 0; i < n; i++) {
      metrics.comparisons++; // Сравнение x[i] != key
      if (x[i] != key) {
        if (i != j) {
          metrics.moves++; // Перемещение x[j] = x[i]
          x[j] = x[i];
        }
        j++;
      }
    }
    n = j;

    metrics.total = metrics.comparisons + metrics.moves;
    return metrics;
}
```

Теперь функция возвращает метрику.

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

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

```
            << testTimes(
                   [](size_t len) {
                     char x[len];
                     std::memset(x, 'A', len);
                     return delOtherMethod(x, len, '_');
                   },
                    runs, n)
                    .toString()
             << std::endl;
   std::cout << "\t\ttestOtherMethodBest() statistics: "
             << testTimes(
                   [](size_t len) {
                     char x[len];
                     std::memset(x, '_', len);
                     return delOtherMethod(x, len, '_');
                   },
                   runs, n)
                   .toString()
             << std::endl;
}
```

Тестирование алгоритма в среднем случае определенно в Листинг 1.

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

```
#include "include/test.h"
#include <iostream>

int main() {
  const auto runs = 1;

  for (auto n : {100, 200, 500, 1000, 2000, 5000, 10000}) {
    std::cout << "RUNNING FOR N=" << n << std::endl;
    testOtherMethod(n, runs);
     std::cout << "---------" << std::endl;
    testBothMethodsMedium(n, runs);
    std::cout << "---------" << std::endl;
  }
  return 0;
}
```

Вывод программы:

```
RUNNING FOR N=100
  testOtherMethod() 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
```

```
RUNNING FOR N=2000
  testOtherMethod() for 1 runs:
    testOtherMethodWorst() statistics: Cп=2000, Mп=0, Tп=2000, took 0.004659 ms
    testOtherMethodBest() statistics: Cп=2000, Mп=0, Tп=2000, took 0.003416 ms
---------
  testBothMethodsMedium() for 1 runs:
    testOtherMethodMedium() statistics: Cп=2000, Mп=971, Tп=2971, took 0.030668 ms
---------

RUNNING FOR N=5000
  testOtherMethod() for 1 runs:
    testOtherMethodWorst() statistics: Cп=5000, Mп=0, Tп=5000, took 0.012023 ms
    testOtherMethodBest() statistics: Cп=5000, Mп=0, Tп=5000, took 0.008796 ms
---------
  testBothMethodsMedium() for 1 runs:
    testOtherMethodMedium() statistics: Cп=5000, Mп=2532, Tп=7532, took 0.078919 ms
---------

RUNNING FOR N=10000
  testOtherMethod() for 1 runs:
    testOtherMethodWorst() statistics: Cп=10000, Mп=0, Tп=10000, took 0.026239 ms
    testOtherMethodBest() statistics: Cп=10000, Mп=0, Tп=10000, took 0.016782 ms
---------
  testBothMethodsMedium() for 1 runs:
    testOtherMethodMedium() statistics: Cп=10000, Mп=4952, Tп=14952, took 0.152778 ms
---------
```

Построим таблицы для второго алгоритма на основе полученных данных.

n	**время, мс**	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
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)

n	**время, мс**	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
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)

n	**время, мс**	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
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.

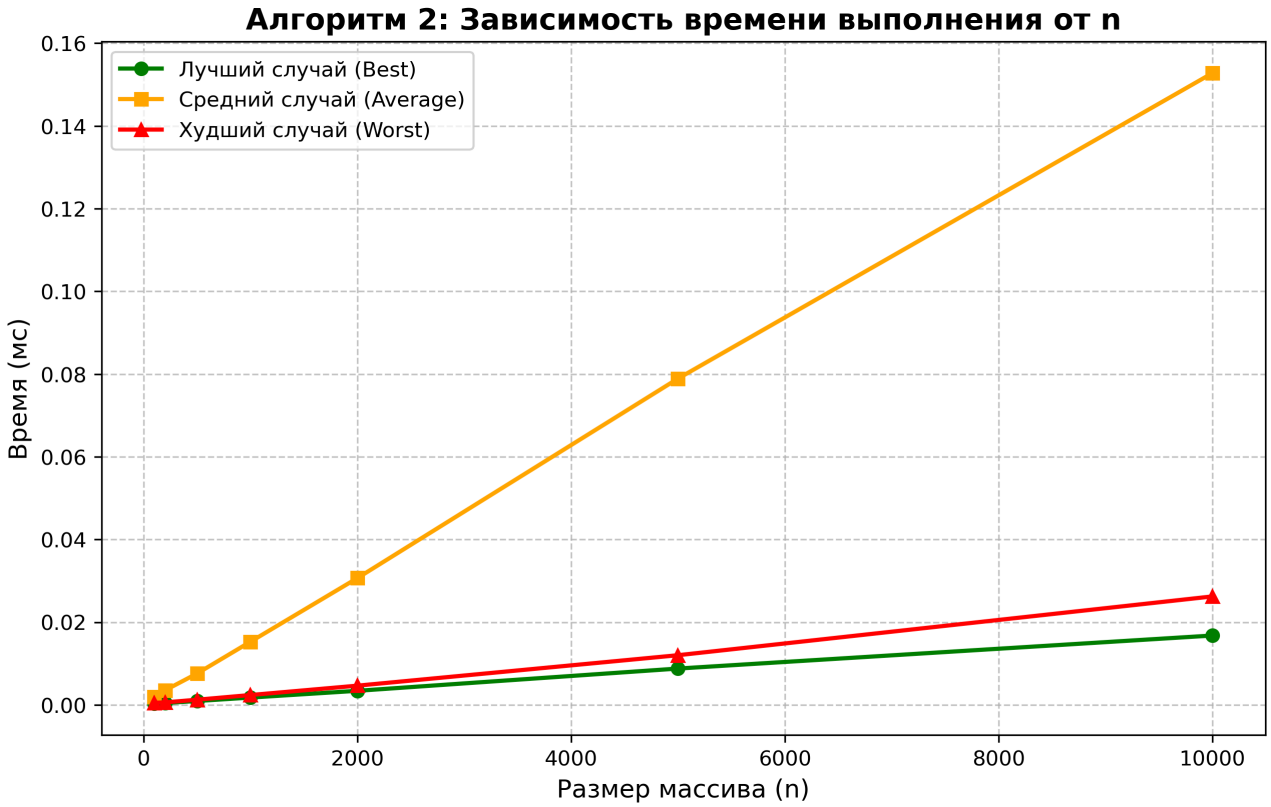

Рис. 2. Зависимость времени работы алгоритма от n. Видно квадратичное возрастание для среднего и худшего случаев.

1.3.2. Вывод по тестированию алгоритма

> ✓ **Вывод**
>
> В **худшем и лучшем случае** $M_{\text{п}} = 0$, а $C_{\text{п}} = n$. 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 $M_{\text{п}}$ 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 $C_{\text{space}} = const$.

1.4. Сравнение двух алгоритмов

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

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

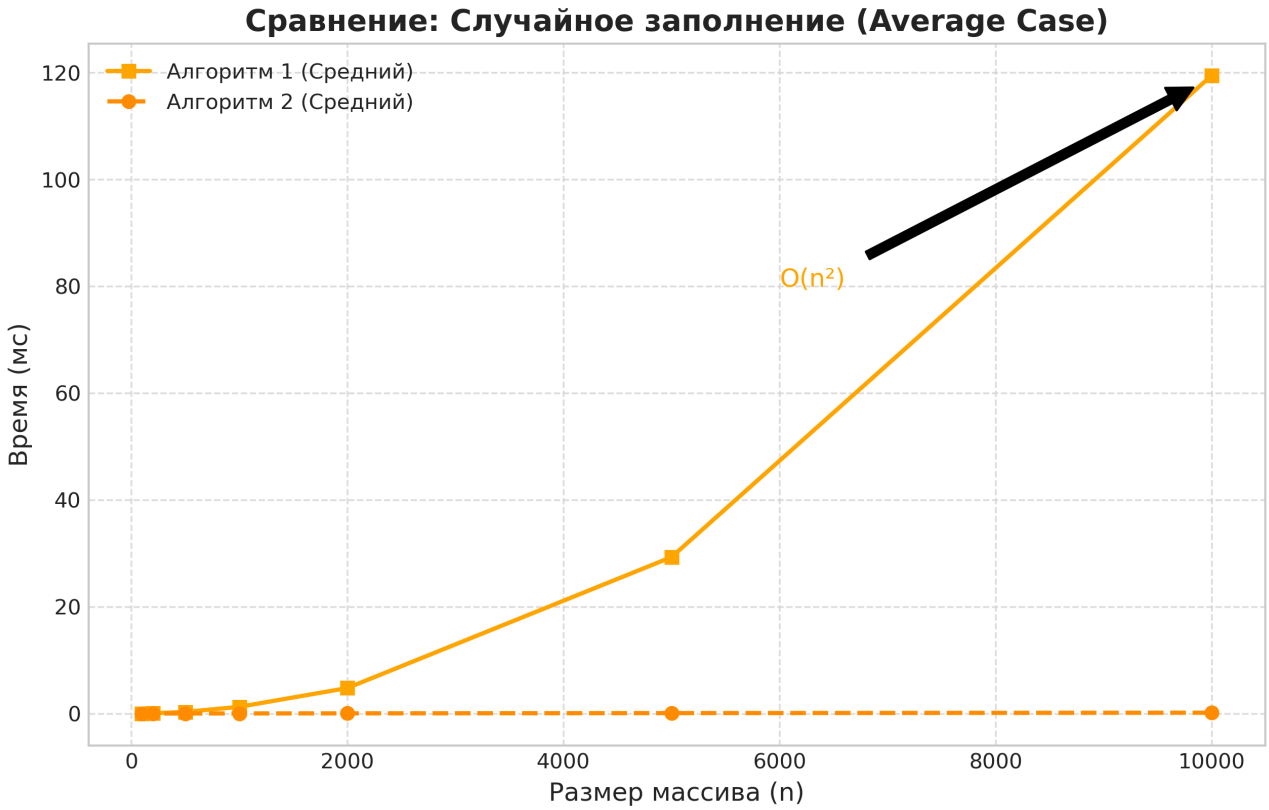

Рис. 4. Зависимость $T_{\text{п}}(n)$ для случая со случайным заполнением.

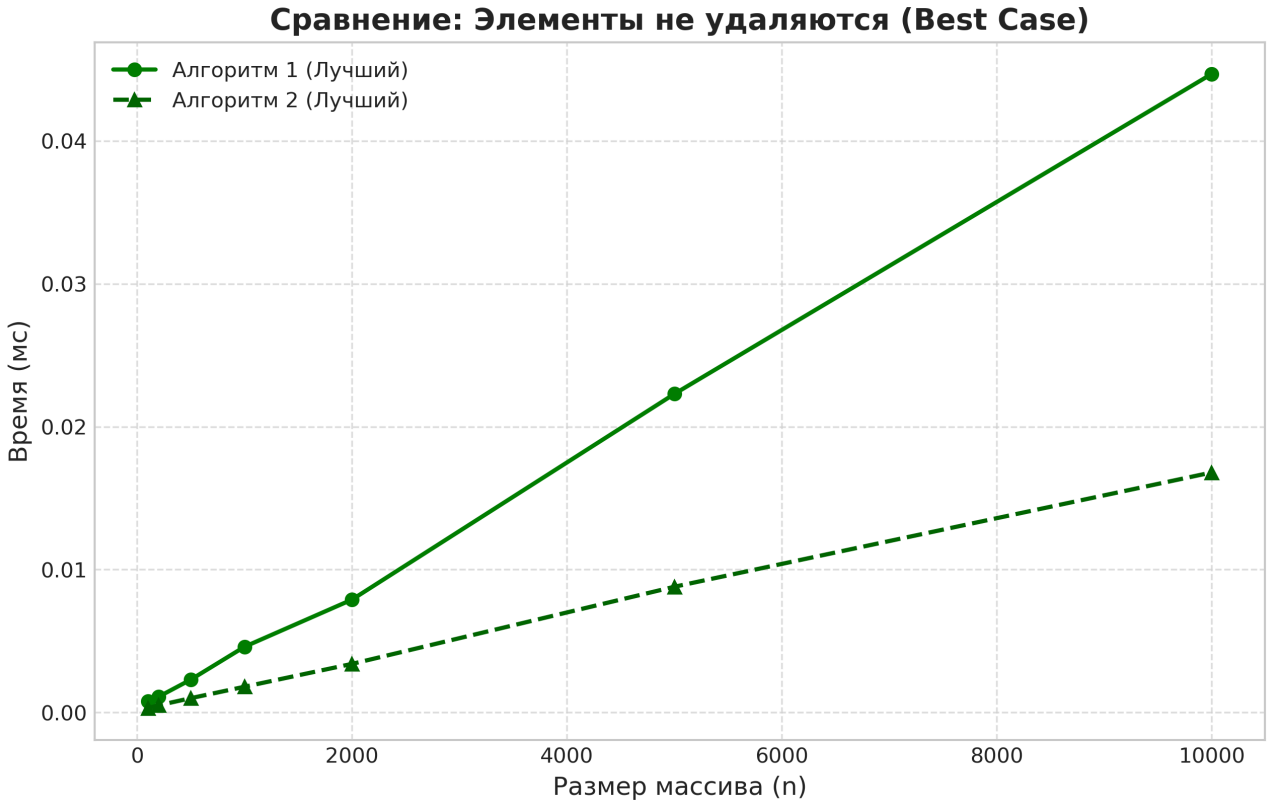

Рис. 5. Зависимость $T_{\text{п}}(n)$ для случая, когда никакие элементы не подлежат удалению.

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 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

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

2.2. Алгоритм простой сортировки (bubble sort)

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

```
ComplexityMetrics bubbleSort(int *arr, size_t n) {
  ComplexityMetrics metrics;

  bool swapped;
  for (size_t i = 0; i < n - 1; ++i) {
    swapped = false;
    for (size_t j = 0; j < n - 1 - i; ++j) {
      metrics.comparisons++;

       if (arr[j] > arr[j + 1]) {
         int temp = arr[j];
         arr[j] = arr[j + 1];
         arr[j + 1] = temp;

         metrics.moves += 3;
         swapped = true;
       }
     }
     if (!swapped)
       break;
   }

   metrics.total = metrics.comparisons + metrics.moves;
   return metrics;
}
```

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.

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

2.2.1. Тестирование алгоритма

Теперь вызовем ее в `main.cpp`.

```
#include "include/test.h"
#include <iostream>

int main() {
  for (auto n : {100, 200, 500, 1000, 2000, 5000, 10000, 100000,
200000, 500000,
                 1000000}) {
    std::cout << "RUNNING FOR N=" << n << std::endl;
    testBubbleSort(n);
    std::cout << "---------" << std::endl;
   }
   return 0;
 }
```

Вывод программы:

```
RUNNING FOR N=100
        testBubbleSort() statistics: Cп=4895, Mп=7314, Tп=12209,
took 0.050515 ms
---------
RUNNING FOR N=200
        testBubbleSort() statistics: Cп=19729, Mп=29136, Tп=48865,
took 0.174568 ms
---------
RUNNING FOR N=500
        testBubbleSort() statistics: Cп=123889, Mп=189621,
Tп=313510, took 1.008826 ms
---------
 RUNNING FOR N=1000
```

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

Оформим вывод в таблице.

n	время, мс	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
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.

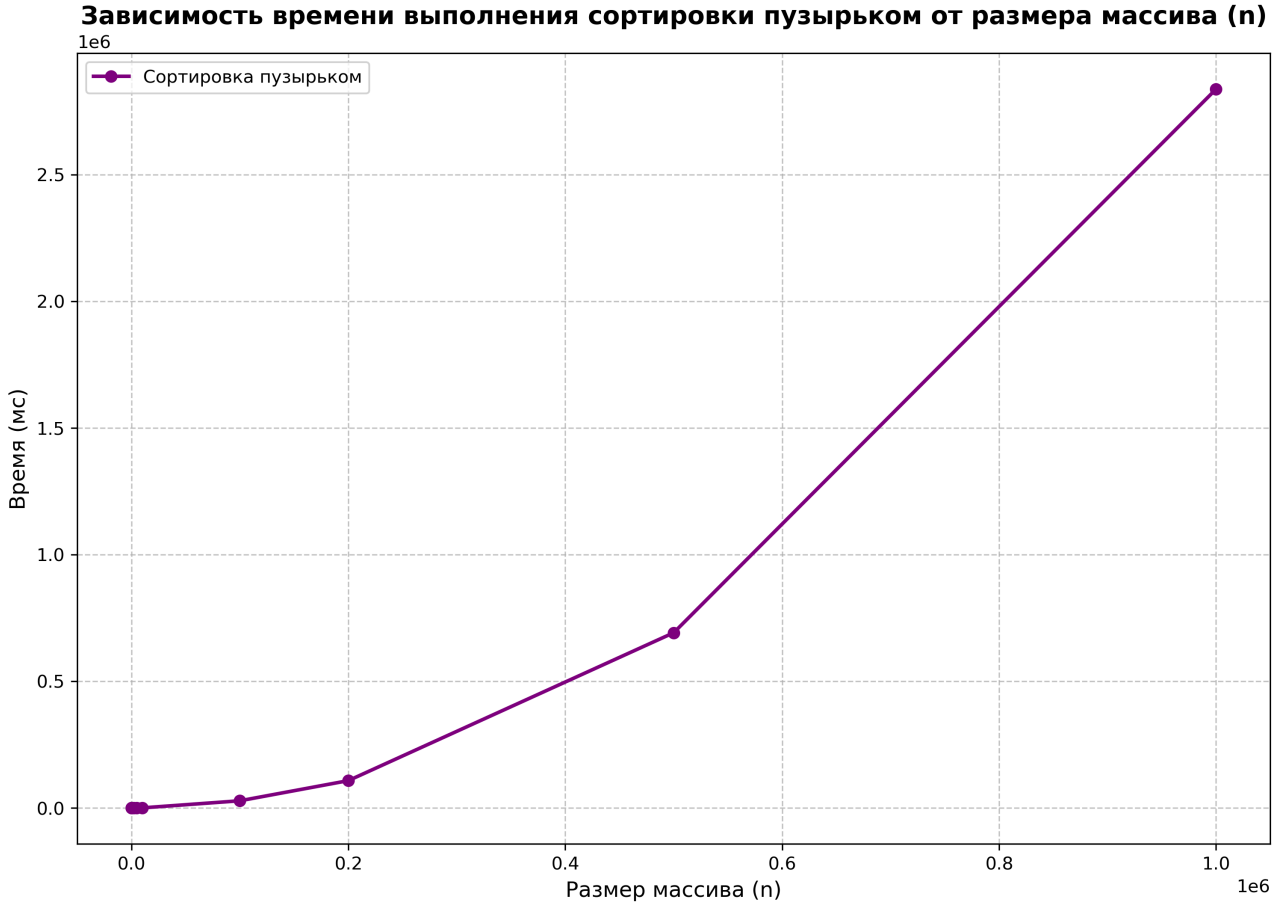

Рис. 6. Зависимость времени выполнения сортировки пузырьком от размера массива (n)

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 ($T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$) возрастает примерно в 100 раз (с ~1.2 млн до ~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 ~3.87 мс до ~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 ~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

```
void testBubbleSortWorst(size_t n) {
  int *arr = new int[n];
  for (int i = n; i > 0; i--)
    arr[i] = i;
  auto test = [&](size_t n) { return bubbleSort(arr, n); };
  std::cout << "\t\ttestBubbleSortWorst statistics: "
            << measureTime(test, n).toString() << std::endl;
}

 void testBubbleSortBest(size_t n) {
   int *arr = new int[n];
   for (int i = 0; i < n; i++)
     arr[i] = i;
   auto test = [&](size_t n) { return bubbleSort(arr, n); };
   std::cout << "\t\ttestBubbleSortBest statistics: "
             << measureTime(test, n).toString() << std::endl;
 }

 void testBubbleSort(size_t n) {
   testBubbleSortBest(n);
   testBubbleSortWorst(n);
 }
```

Вывод программы:

```
 RUNNING FOR N=100
         testBubbleSort() statistics:
                 testBubbleSortBest statistics: Cn=99, Mn=0, Tn=99,
took 0.000771 ms
                 testBubbleSortWorst statistics: Cn=99, Mn=0,
Tn=99, took 0.000321 ms
 ---------

 RUNNING FOR N=200
         testBubbleSort() statistics:
                 testBubbleSortBest statistics: Cn=199, Mn=0,
Tn=199, took 0.000481 ms
                  testBubbleSortWorst statistics: Cn=199, Mn=0,
Tn=199, took 0.000441 ms
 ---------

```

```
RUNNING FOR N=500
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=499, Mп=0, Tп=499, took 0.000992 ms
                testBubbleSortWorst statistics: Cп=499, Mп=0, Tп=499, took 0.000972 ms
---------

RUNNING FOR N=1000
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=999, Mп=0, Tп=999, took 0.001883 ms
                testBubbleSortWorst statistics: Cп=999, Mп=0, Tп=999, took 0.001874 ms
---------

RUNNING FOR N=2000
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=1999, Mп=0, Tп=1999, took 0.003697 ms
                testBubbleSortWorst statistics: Cп=1999, Mп=0, Tп=1999, took 0.003697 ms
---------

RUNNING FOR N=5000
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=4999, Mп=0, Tп=4999, took 0.009117 ms
                testBubbleSortWorst statistics: Cп=4999, Mп=0, Tп=4999, took 0.009107 ms
---------

RUNNING FOR N=10000
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=9999, Mп=0, Tп=9999, took 0.018184 ms
                testBubbleSortWorst statistics: Cп=9999, Mп=0, Tп=9999, took 0.018164 ms
---------

RUNNING FOR N=100000
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=99999, Mп=0, Tп=99999, took 0.180561 ms
                testBubbleSortWorst statistics: Cп=99999, Mп=0, Tп=99999, took 0.180641 ms
```

```
---------

RUNNING FOR N=200000
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=199999, Mп=0, Tп=199999, took 0.365820 ms
                testBubbleSortWorst statistics: Cп=199999, Mп=0, Tп=199999, took 0.375238 ms
---------

RUNNING FOR N=500000
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=499999, Mп=0, Tп=499999, took 0.905348 ms
                testBubbleSortWorst statistics: Cп=499999, Mп=0, Tп=499999, took 0.921188 ms
---------

RUNNING FOR N=1000000
        testBubbleSort() statistics:
                testBubbleSortBest statistics: Cп=999999, Mп=0, Tп=999999, took 1.843608 ms
                testBubbleSortWorst statistics: Cп=999999, Mп=0, Tп=999999, took 1.836154 ms
---------

```

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

n	**время, мс**	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
100	0.0003	99	0	99
200	0.0004	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	0.3752	199999	0	199999
500000	0.9212	499999	0	499999
1000000	1.8362	999999	0	999999

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**	$C_{\text{п}}$	$M_{\text{п}}$	$T_{\text{п}} = C_{\text{п}} + M_{\text{п}}$
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

Анализ результатов тестирования сортировки пузырьком позволяет сделать следующий вывод:

Алгоритм сортировки пузырьком с оптимизацией (флаг `swapped`) является независимым от исходной упорядоченности входного массива.

Это подтверждается эмпирическими данными:

1. **В лучшем и худшем случаях** (когда массив уже отсортирован по возрастанию или по убыванию) алгоритм выполняет всего один проход по массиву.
   - Количество операций $T_{\text{п}}$ линейно зависит от n ($T_{\text{п}}(n)$).
   - Время выполнения минимально и растет пропорционально размеру массива.
   - Как видно из Таблица 9 и Таблица 8, время для $n = 1,000,000$ составляет всего ~1.8 мс.

Таким образом, несмотря на наличие оптимизации, которая спасает алгоритм в крайних случаях, его производительность в реальных сценариях (со случайными данными) остается крайне низкой из-за квадратичной сложности. Эта зависимость от входных данных делает сортировку пузырьком непрактичной для использования в задачах, где важна скорость обработки.

4. 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).