

Содержание

1. Задание 1	2
1.1. Начало	2
1.2. Алгоритмы	3
1.2.1. Алгоритм №1	3
1.2.2. Алгоритм №2	9
1.3. Сравнение двух алгоритмов	15
1.4. Вывод из задания 1	17
1.4.1. Сравнение по времени выполнения	17
1.4.2. Сравнение по расходу памяти (Ёмкостная сложность)	18
2. Литература	19

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. Алгоритмы

1.2.1. 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.

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

п	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

п	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

✓ 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 $C_{\text{space}} = \text{const}$.

Построим график на основе полученных данных.

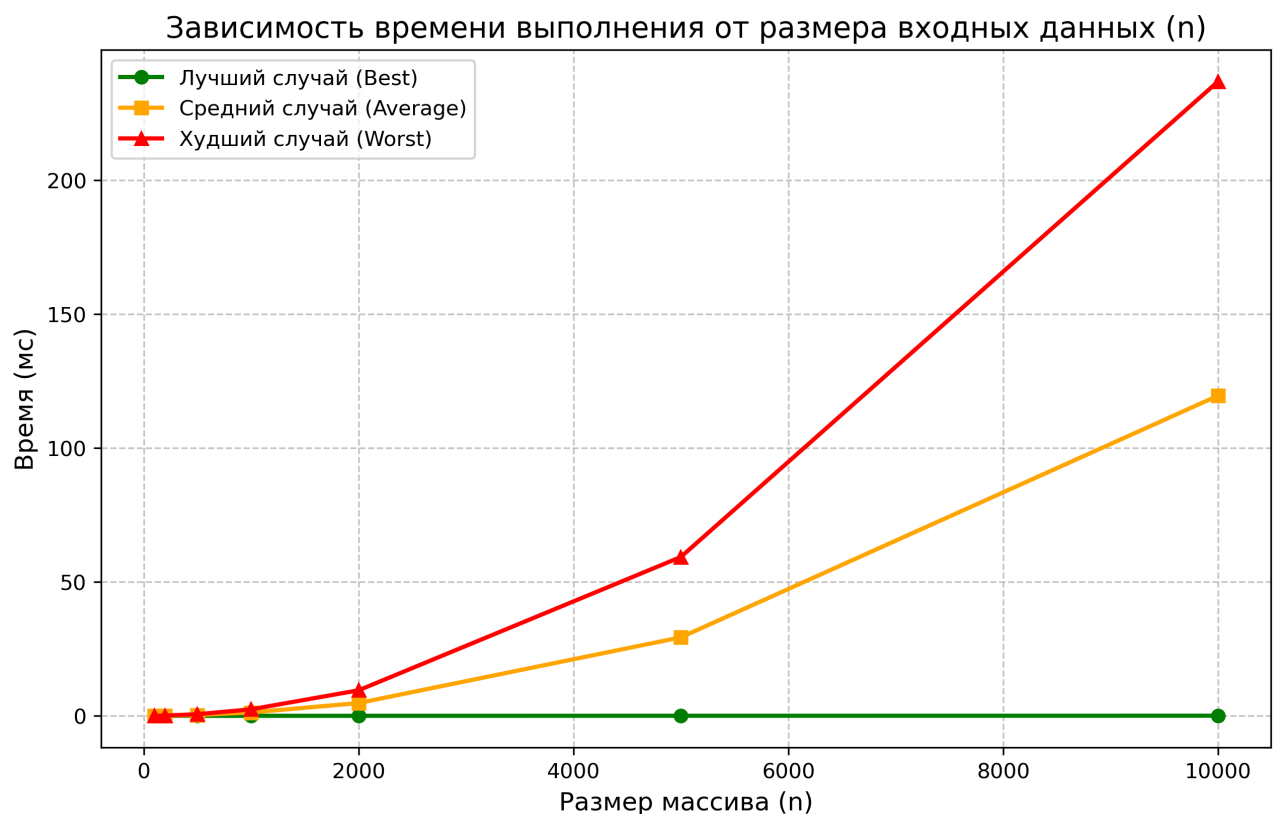


Рис. 1. Зависимость времени работы алгоритма от n . Видно квадратичное возрастание для среднего и худшего случаев.

1.2.2. 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 }

```

Изменим код реализованного метода для возвращения метрик:

```

1  ComplexityMetrics del0therMethod(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 test0therMethod(size_t n, int runs = 100) {
2      std::cout << "\tttest0therMethod() for " << runs << " runs:" <<
std::endl;
3      std::cout << "\t\ttest0therMethodWorst() statistics: "
4                  << testTimes(
5                      [](size_t len) {

```

```

6         char x[len];
7         std::memset(x, 'A', len);
8         return delOtherMethod(x, len, '_');
9     },
10    runs, n)
11    .toString()
12    << std::endl;
13    std::cout << "\t\ttestOtherMethodBest() statistics: "
14    << testTimes(
15        [](size_t len) {
16            char x[len];
17            std::memset(x, '_', len);
18            return delOtherMethod(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         testOtherMethod(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   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
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.

n	время, мс	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)

n	время, мс	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)

n	время, мс	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)

✓ 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 $C_{\text{space}} = \text{const}$.

Построим график на основе полученных данных.

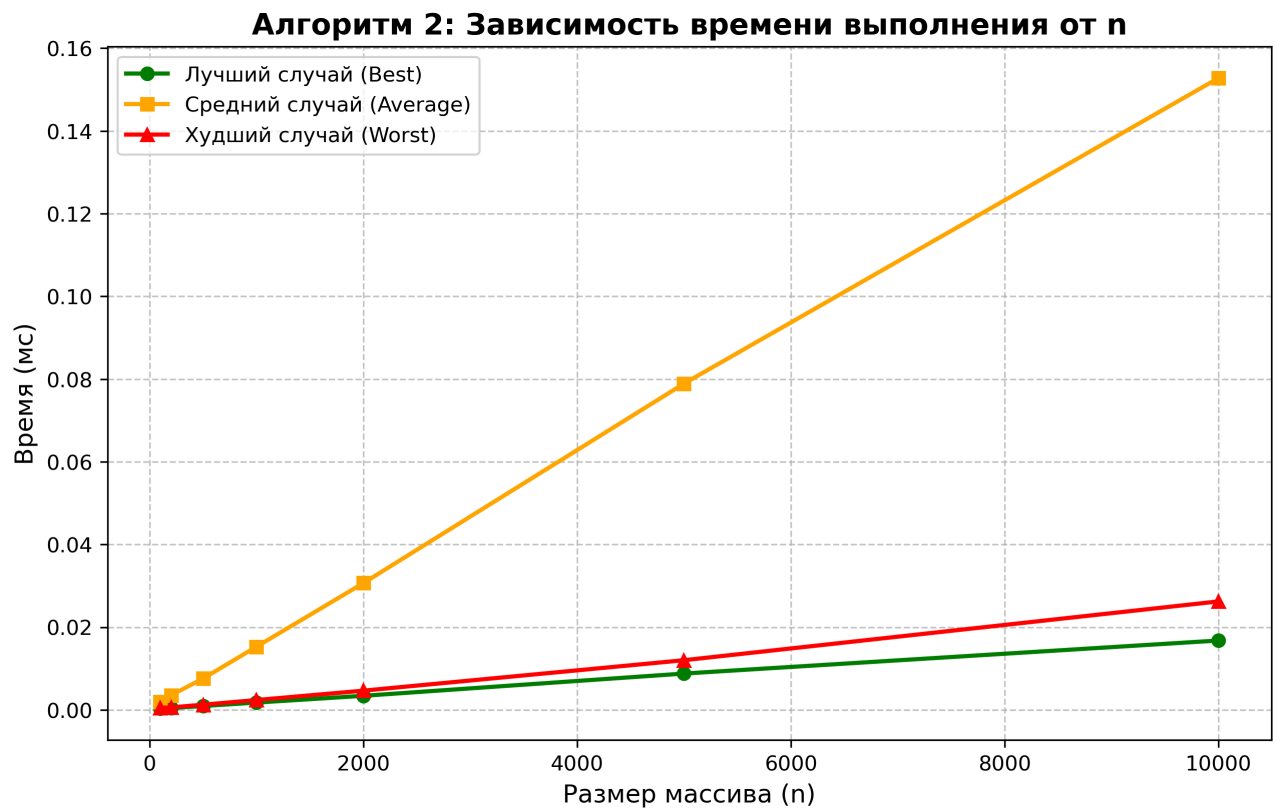


Рис. 2. Зависимость времени работы алгоритма от n . Видно квадратичное возрастание для среднего и худшего случаев.

1.3. 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.

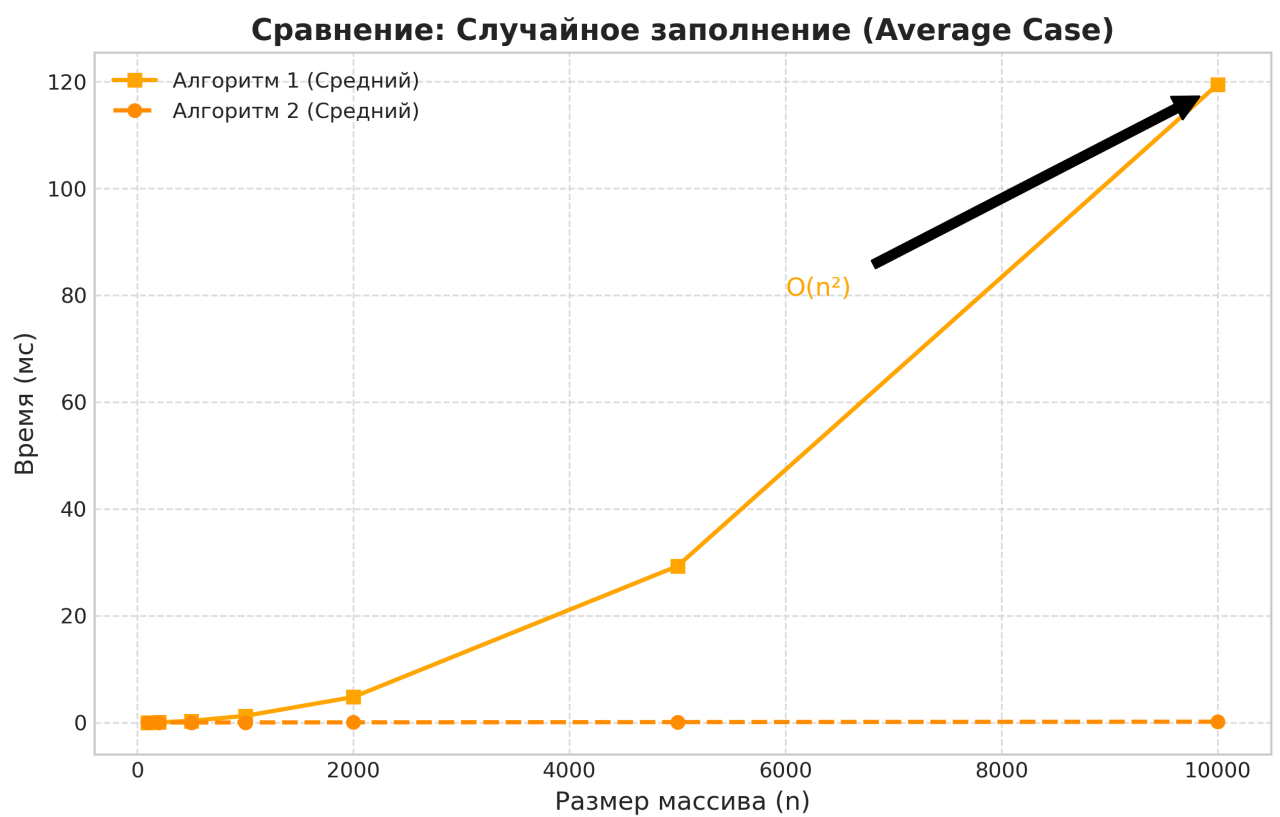


Рис. 4. Зависимость $T_{\Pi}(n)$ для случая со случайным заполнением.

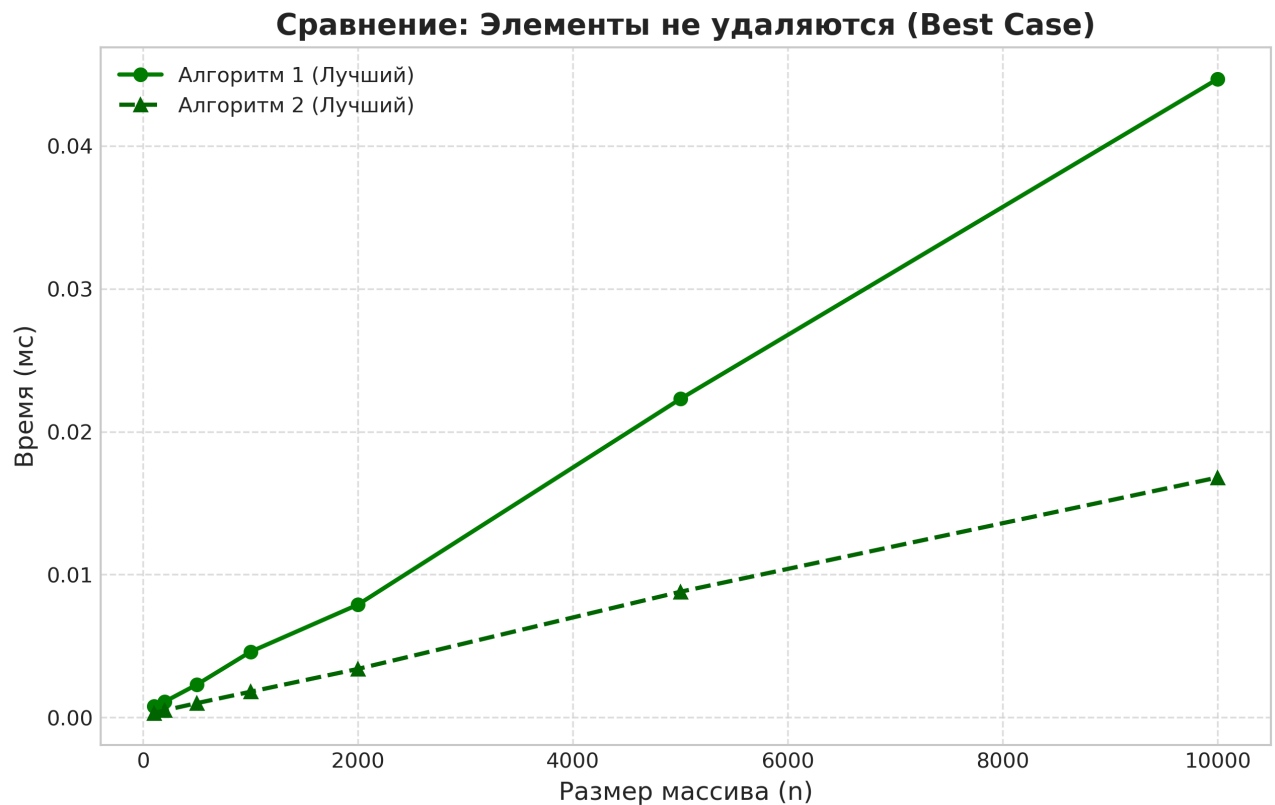


Рис. 5. Зависимость $T_{\Pi}(n)$ для случая, когда никакие элементы не подлежат удалению.

1.4. Вывод из задания 1

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

1.4.1. 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.

1.4.2. 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. 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).