Применение алгоритмов на реальном наборе данных

АС-24-04

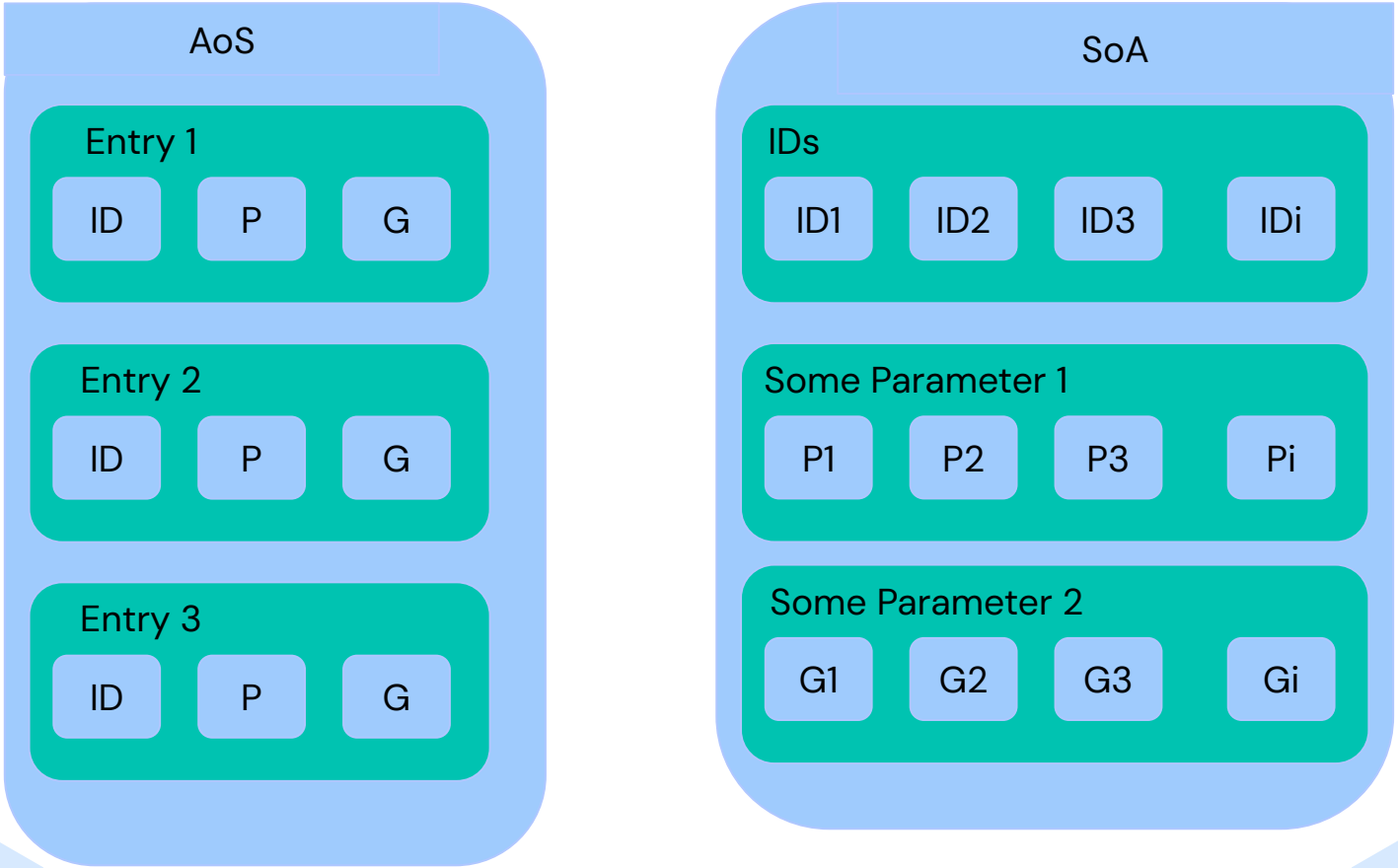
Дзуцев Д.Э.
Евтух А.К.
Киселев С. В.
Лазарев А. Н.
Межов И.А.
Сизёв Г. С.
Спицин К.М.
Салимов С.

```
A-7710198,Source1,2,2019-04-17 18:19:08,2019-04-17 18:48:39,42.87376,-72.222868,42.872692,-72.21600500000002,0.3
A-7710199,Source1,2,2019-04-17 18:03:31,2019-04-17 18:32:51,42.47595,-83.125962,42.475969,-83.124608,0.069,At 10
A-7710200,Source1,2,2019-04-17 20:43:51,2019-04-17 21:12:15,41.84094,-71.41111,41.83491,-71.4162,0.492000000000
A-7710201,Source1,2,2019-04-17 20:25:52,2019-04-17 20:54:04,42.86872,-73.80826,42.86872,-73.80826,0.0,At School
A-7710202,Source1,2,2019-04-17 20:25:52,2019-04-17 20:54:04,42.86872,-73.80826,42.86872,-73.80826,0.0,At School
A-7710203,Source1,3,2019-04-17 21:14:41,2019-04-17 21:43:33,42.36405,-83.07944,42.36967,-83.0834,0.438,At Grand
A-7710204,Source1,4,2019-04-17 21:59:41,2019-04-17 22:27:34,42.39879000000001,-83.10909000000002,42.3976,-83.111
A-7710205,Source1,4,2019-04-17 21:59:41,2019-04-17 22:27:34,42.40009000000001,-83.10521,42.39879000000001,-83.10
A-7710206,Source1,4,2019-04-18 00:46:11,2019-04-18 01:14:58,42.36421,-83.56,42.32589,-83.64296,4.996,Closed betw
A-7710207,Source1,4,2019-04-17 05:45:20,2019-04-17 06:14:35,39.10261,-84.49826,39.10378,-84.49818,0.081,Ramp clo
A-7710208,Source1,4,2019-04-17 05:45:20,2019-04-17 06:14:35,39.10378,-84.49818,39.10514000000001,-84.4963,0.138,
A-7710209,Source1,4,2019-04-17 05:45:20,2019-04-17 06:14:35,39.10182,-84.49698000000002,39.10261,-84.49826,0.088
A-7710210,Source1,2,2019-04-17 06:37:51,2019-04-17 07:06:59,41.63181,-72.87361999999997,41.67,-72.83931,3.178,Be
A-7710211,Source1,2,2019-04-17 05:39:26,2019-04-17 06:09:03,39.26321,-76.567739,39.263923,-76.588528,1.113,At I-
A-7710212,Source1,4,2019-04-17 06:07:51,2019-04-17 06:37:11,40.75265,-73.74579,40.76503,-73.7183,1.674000000000
A-7710213,Source1,2,2019-04-17 05:49:11,2019-04-17 06:18:30,39.00799,-76.40386,38.99074,-76.37165999999998,2.1,A
A-7710214,Source1,2,2019-04-17 09:49:11,2019-04-17 10:18:30,38.99074,-76.37165999999998,38.98333,-76.34004,1.774
A-7710215,Source1,3,2019-04-17 06:22:06,2019-04-17 06:51:13,40.60626,-74.03226,40.60359,-74.01873,0.733,At Bay 8
A-7710216,Source1,3,2019-04-17 07:03:21,2019-04-17 07:32:03,40.75609,-73.74051,40.76503,-73.7183,1.315999999999
A-7710217,Source1,2,2019-04-17 19:13:14,2019-04-17 19:41:50,40.18266,-74.61910999999998,40.18767,-74.61018,0.585
A-7710218,Source1,2,2019-04-17 07:37:07,2019-04-17 08:06:06,40.63171,-74.21167,40.64816,-74.20158,1.254,At I-278
A-7710219,Source1,3,2019-04-17 08:04:58,2019-04-17 08:34:25,40.043477,-85.994877,40.05857,-85.99055,1.068,Betwee
A-7710220,Source1,2,2019-04-17 07:45:22,2019-04-17 08:14:26,40.73065,-74.13205,40.73359,-74.12665,0.348,Ramp to
A-7710221,Source1,2,2019-04-17 07:45:22,2019-04-17 08:14:26,40.73256,-74.12328000000002,40.73195,-74.1236,0.045,
A-7710222,Source1,2,2019-04-17 07:45:22,2019-04-17 08:14:26,40.73359,-74.12665,40.73256,-74.12328000000002,0.19,
A-7710223,Source1,2,2019-04-17 07:45:22,2019-04-17 08:14:26,40.73002,-74.13318000000002,40.734515,-74.125081000
A-7710224,Source1,4,2019-04-17 07:34:58,2019-04-17 08:04:19,39.423254,-77.493762,39.42241,-77.48868,0.277,Closed
A-7710225,Source1,2,2019-04-17 07:57:29,2019-04-17 08:26:53,39.692059,-75.666962,39.69322,-75.66868000000002,0.1
A-7710226,Source1,2,2019-04-17 07:38:41,2019-04-17 08:08:11,38.81956,-76.92684,38.8187,-76.91388,0.7,At MD-5/Bra
A-7710227,Source1,2,2019-04-17 08:01:08,2019-04-17 08:30:32,38.958285,-85.84331999999998,38.968961,-85.8433,0.73
```

```
ID,Severity,City,County,State,Temperature(F),Wind_Chill(F),Humidity(%),Pressure(in),Wind_Speed(mph),Weather_Condition
0,3,Dayton,Montgomery,OH,36.9,,91.0,29.68,,Light Rain
1,2,Reynoldsburg,Franklin,OH,37.9,,100.0,29.65,,Light Rain
2,2,Williamsburg,Clermont,OH,36.0,33.3,100.0,29.67,3.5,Overcast
3,3,Dayton,Montgomery,OH,35.1,31.0,96.0,29.64,4.6,Mostly Cloudy
4,2,Dayton,Montgomery,OH,36.0,33.3,89.0,29.65,3.5,Mostly Cloudy
5,3,Westerville,Franklin,OH,37.9,35.5,97.0,29.63,3.5,Light Rain
6,2,Dayton,Montgomery,OH,34.0,31.0,100.0,29.66,3.5,Overcast
7,3,Dayton,Montgomery,OH,34.0,31.0,100.0,29.66,3.5,Overcast
8,2,Dayton,Montgomery,OH,33.3,,99.0,29.67,1.2,Mostly Cloudy
```

```c
struct accident {
    int id;                     // 0
    short int severity;           // 2
    char city[20];              // 12
    char county[20];            // 13
    char state[3];              // 14
    float temperature;          // 20
    float wind_temperature;     // 21
    float humidity_percent;       // 22
    float pressure;             // 23
    float wind_speed;           // 26
    char weather_condition[15];   // 28
};
```

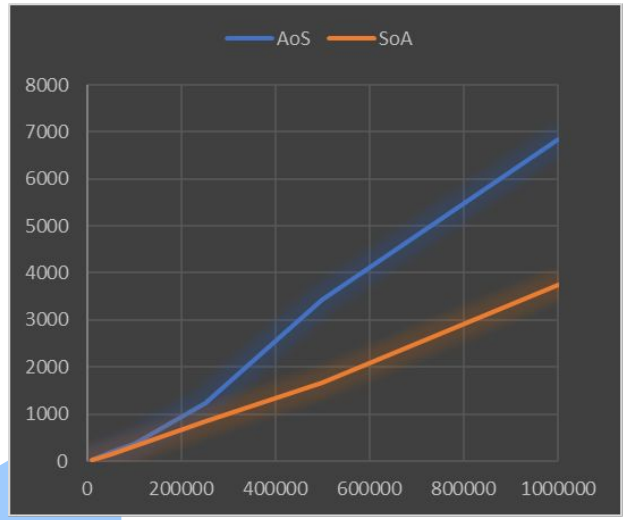
SoA и AoS

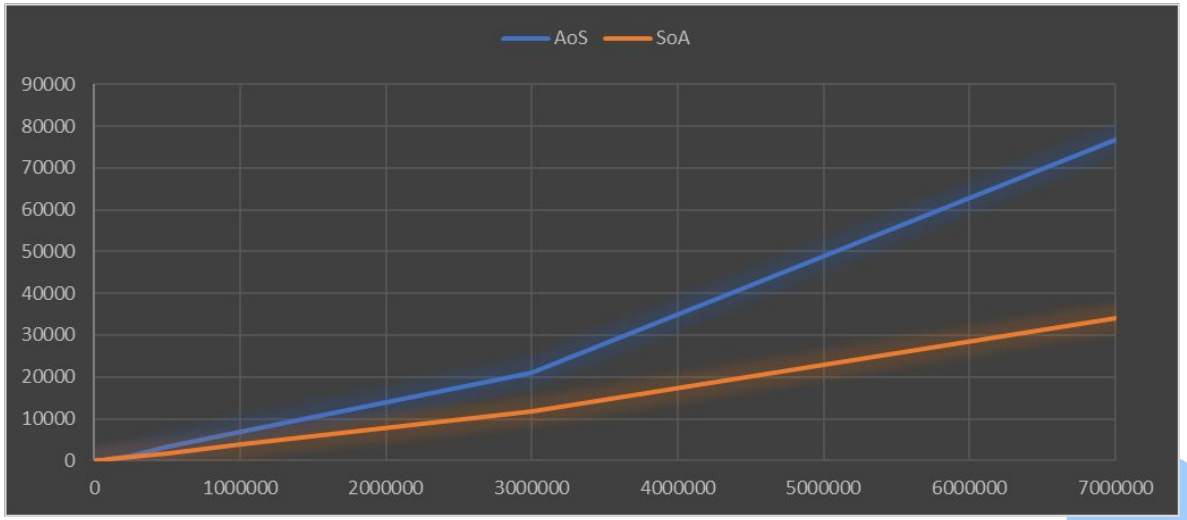

AoS
Entry 1
ID
P
G
Entry 2
ID
P
G
Entry 3
ID
P
G
SoA
IDs
ID1
ID2
ID3
IDi
Some Parameter 1
P1
P2
P3
Pi
Some Parameter 2
G1
G2
G3
Gi

SoA vs AoS сравнение

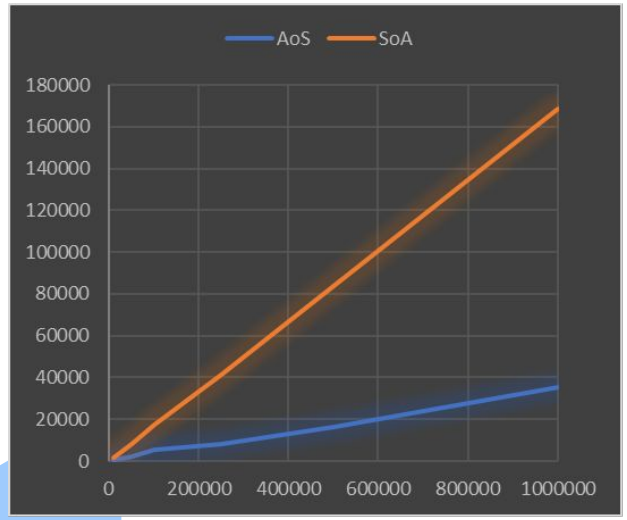
SoA быстрее в поиске/фильтрации, потому что при запросе .get в кэш-память процессора кэшируется больше значений соседних индексов. AoS быстрее в сортировке, потому что цельный объект в памяти перемещается быстрее, чем отдельные кусочки. Это демонстрируется на реальных тестах, по которым сделаны графики.

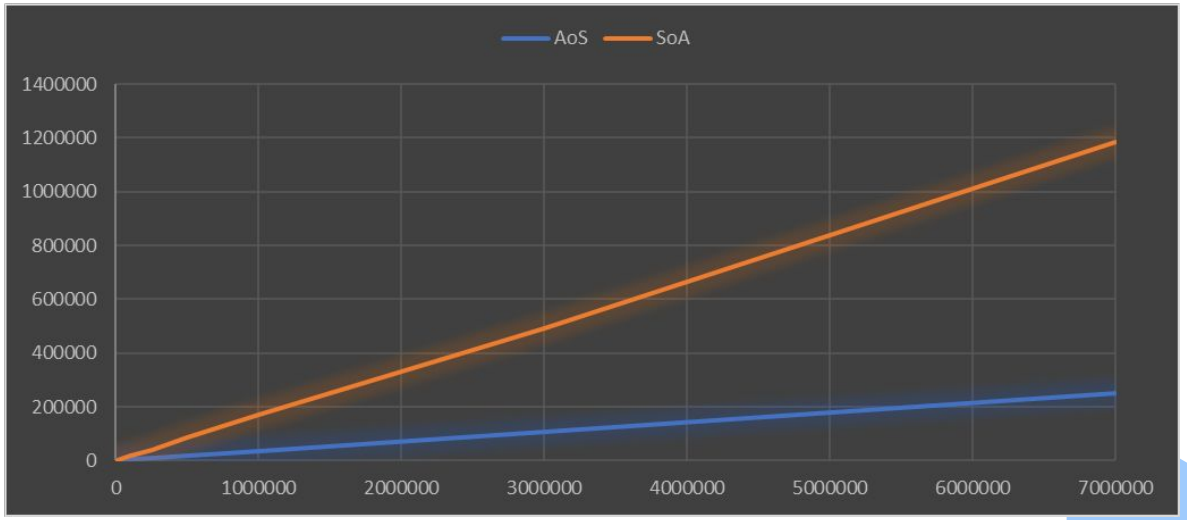
Тест 1: Фильтр по температуре



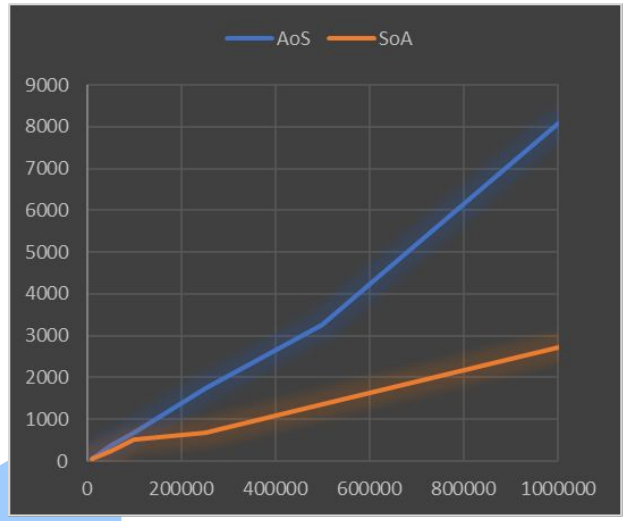


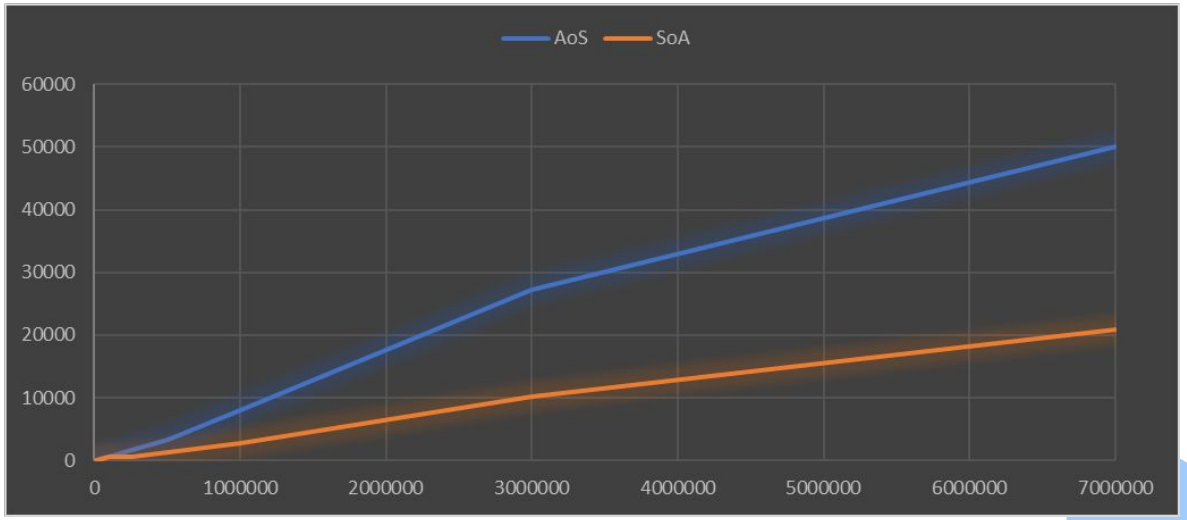
Тест 2: Сортировка по температуре



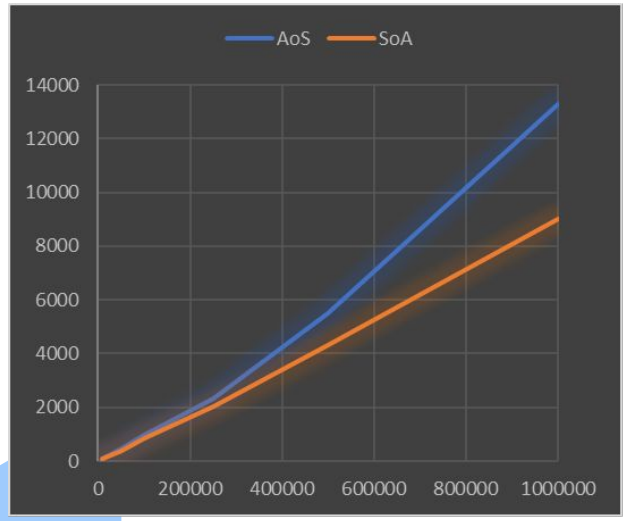


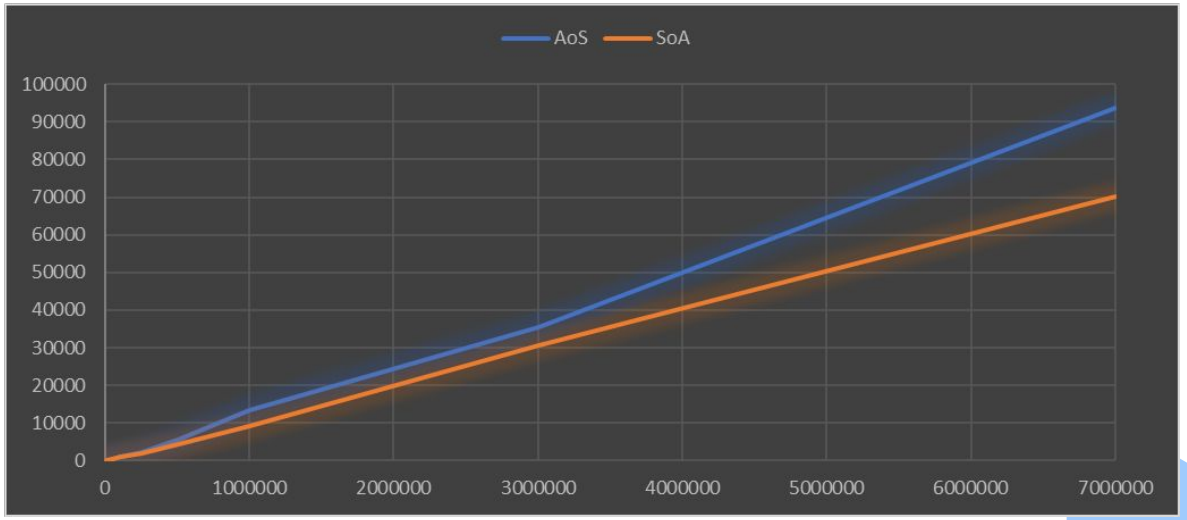
Тест 3: Поиск по температуре



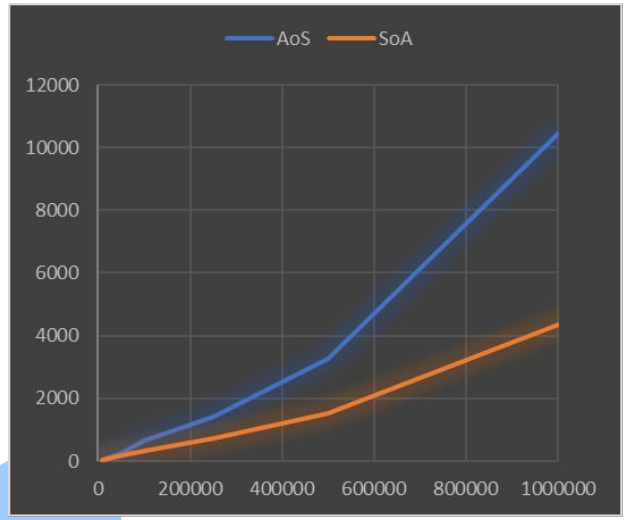


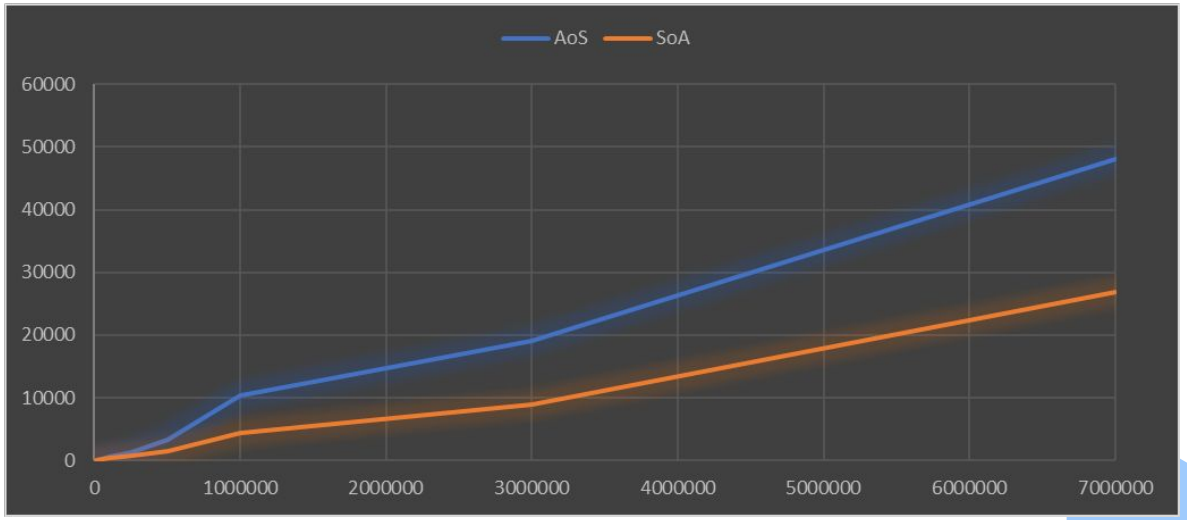
Тест 4: Фильтр по температуре и скорости ветра





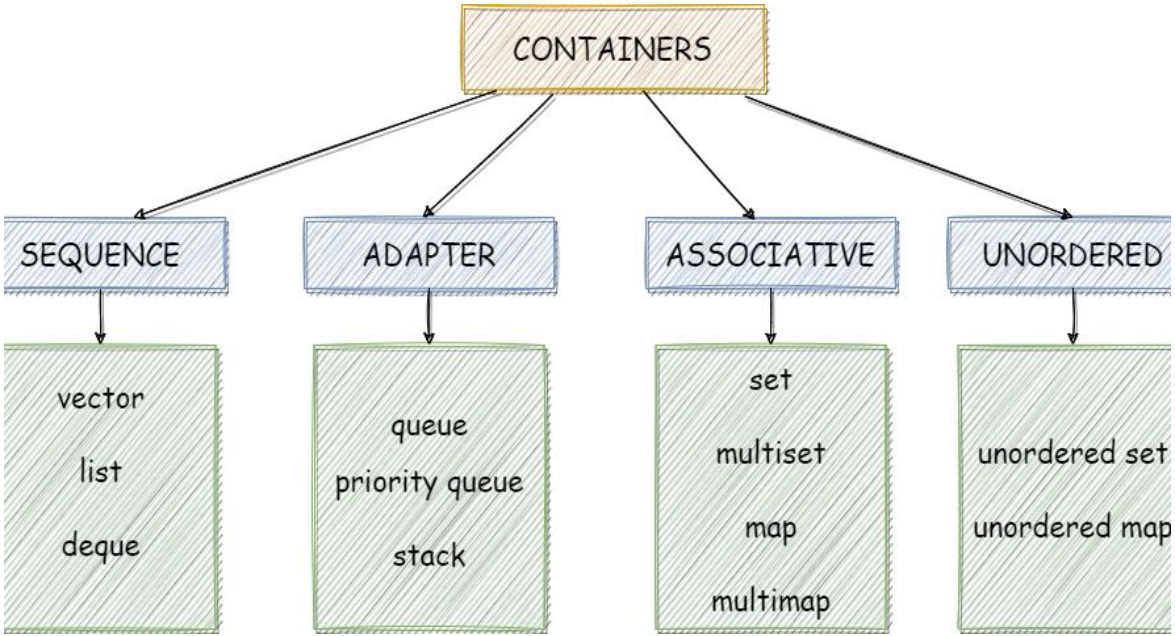
Тест 5: Поиск по температуре и скорости ветра



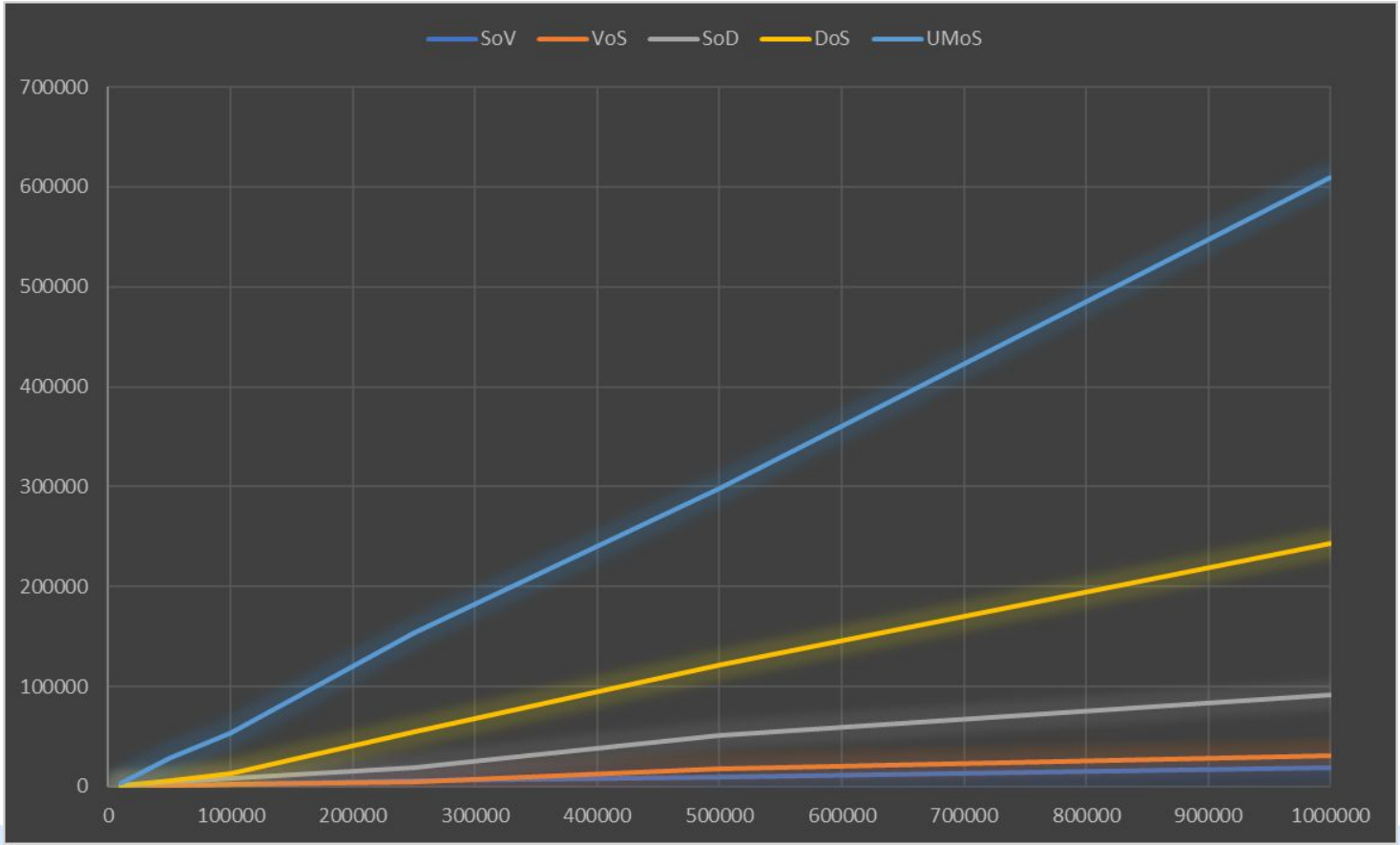


Хранение в STL контейнерах

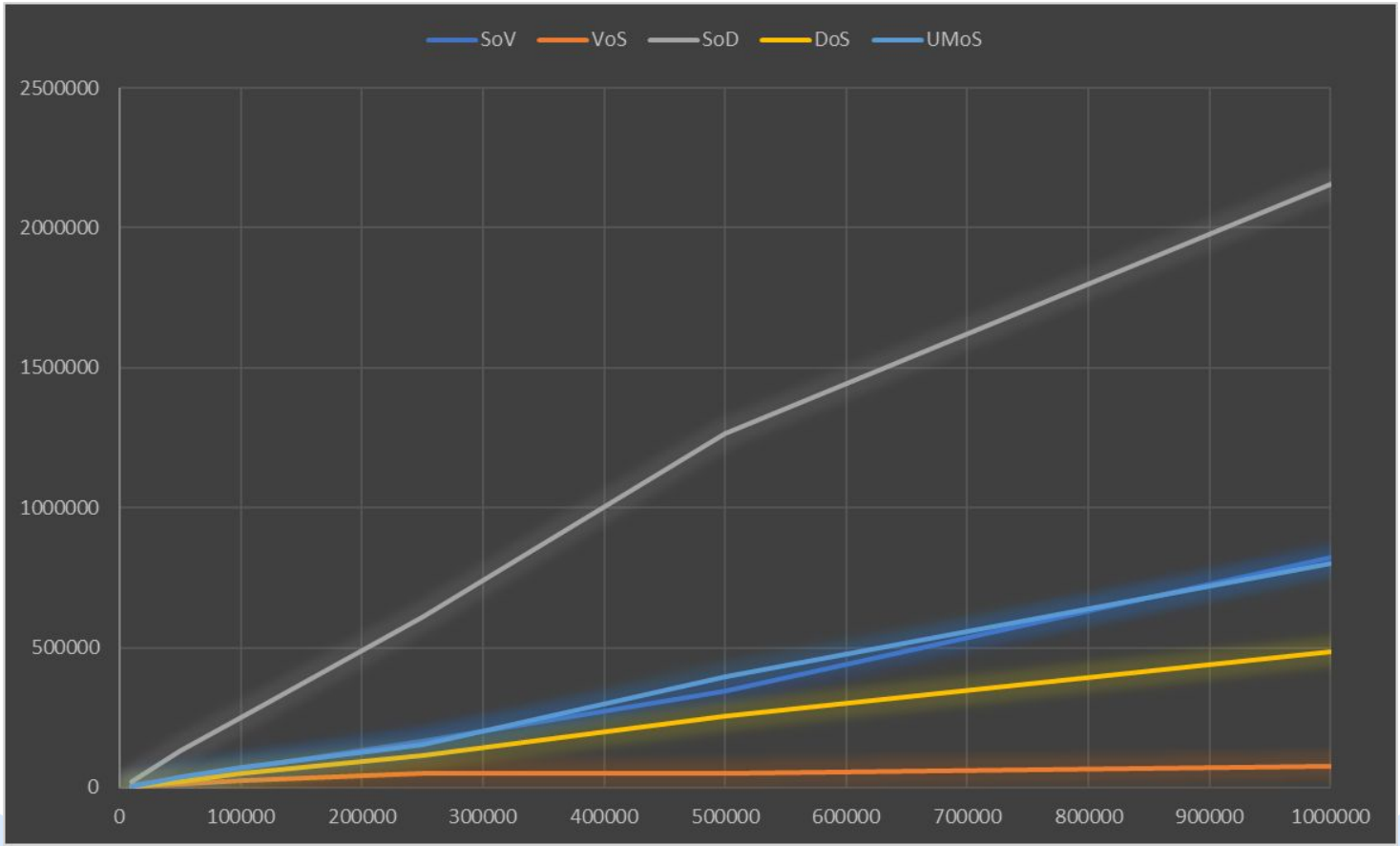
1. vector - динамический массив с непрерывным размещением элементов в памяти
2. deque - двусторонняя очередь в виде массива блоков фиксированного размера
3. unordered_map - неупорядоченный ассоциативный контейнер, который хранит пары



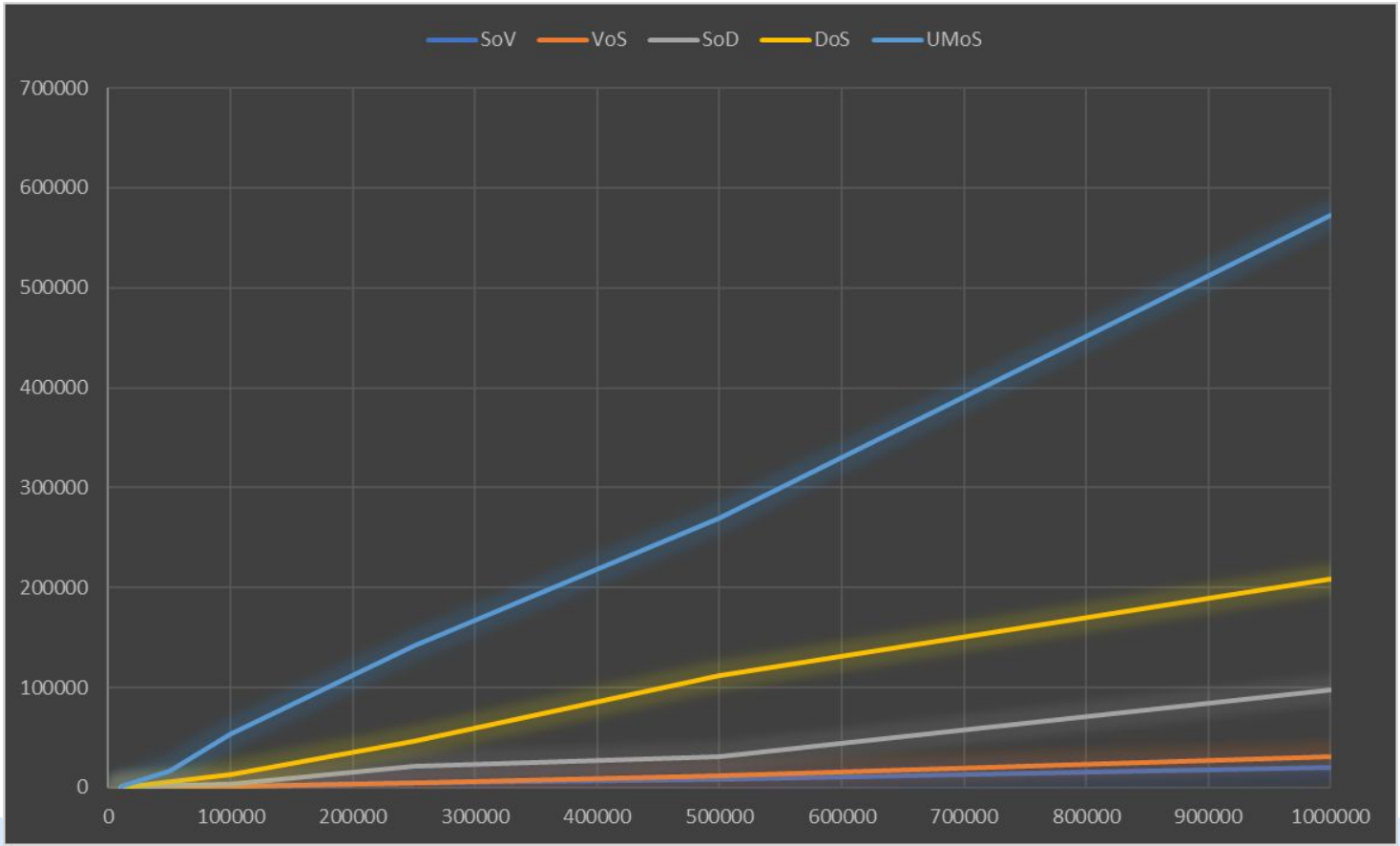
Тест 1: Сравнение одного



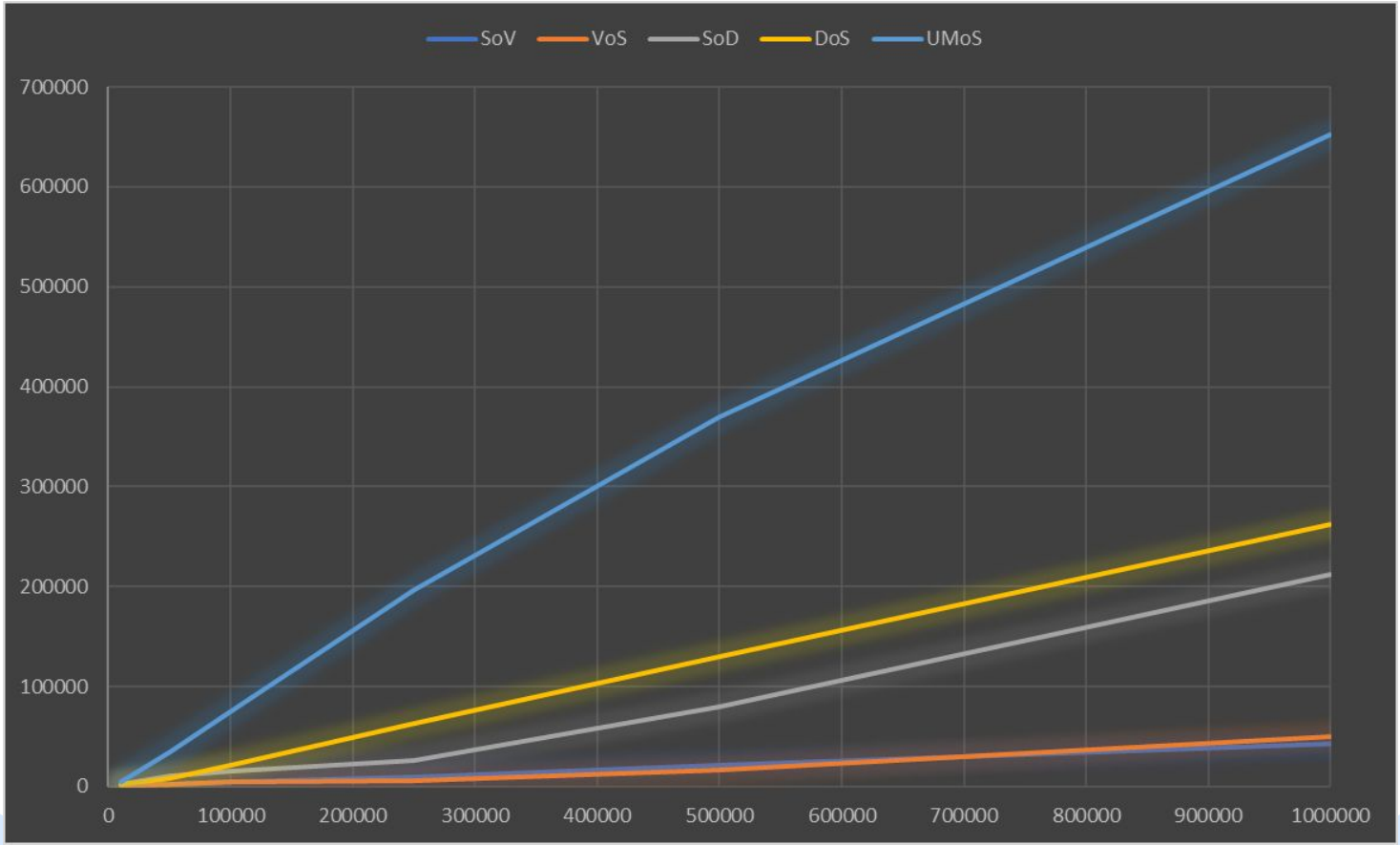
Тест 2: Сравнение +



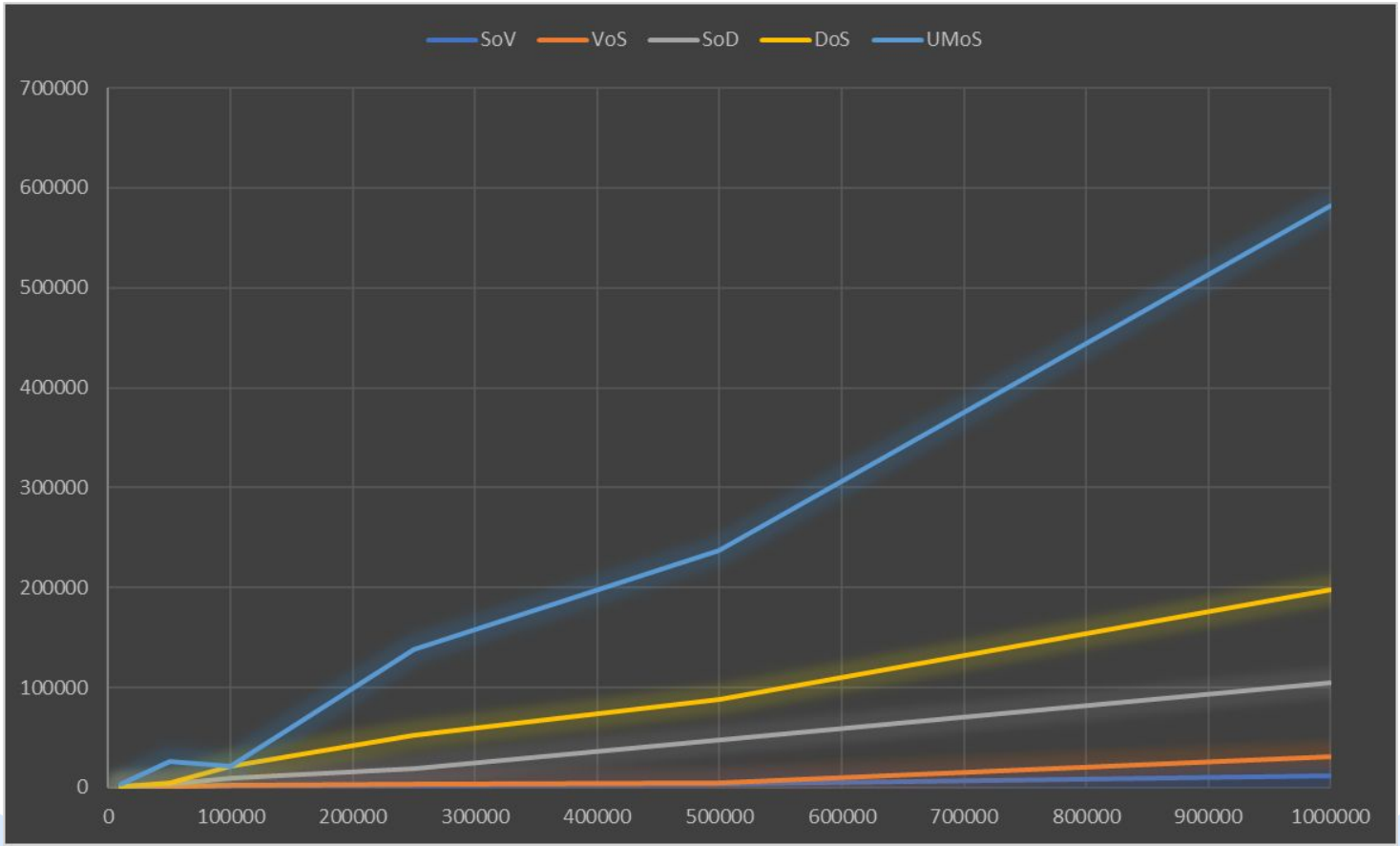
Тест 3: Поиск точного



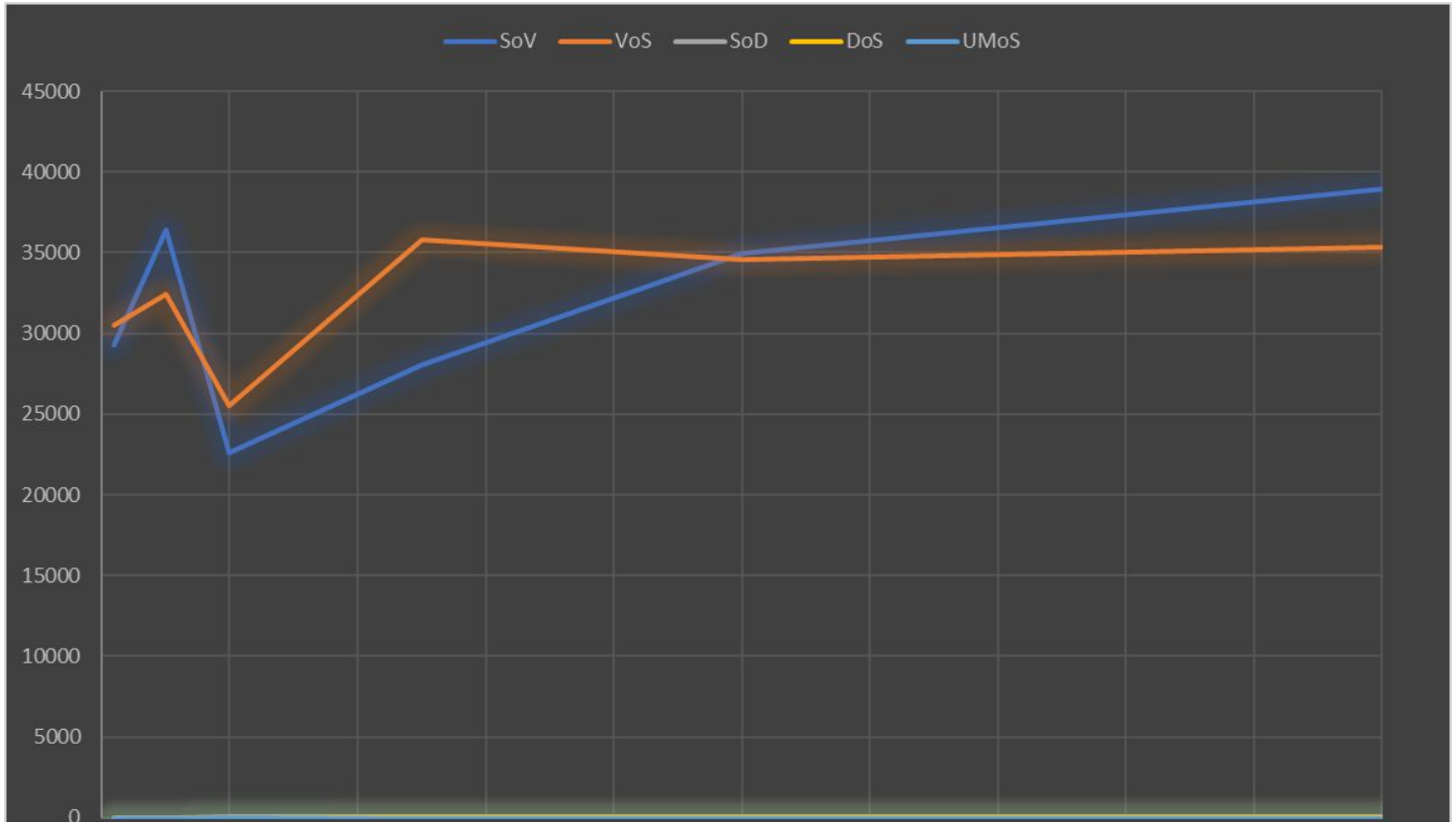
Тест 4: Сравнение двух



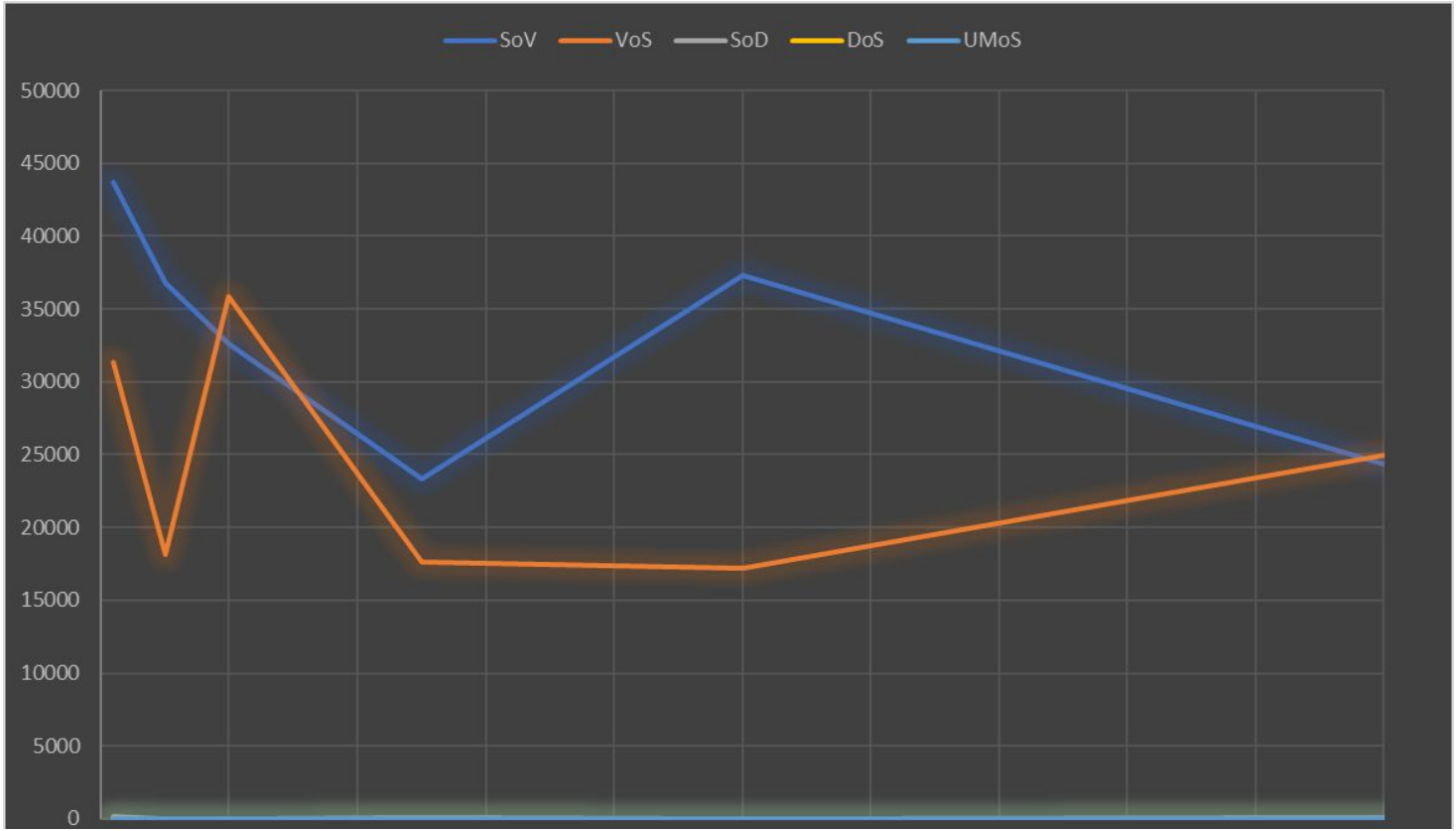
Тест 5: Поиск двух точных



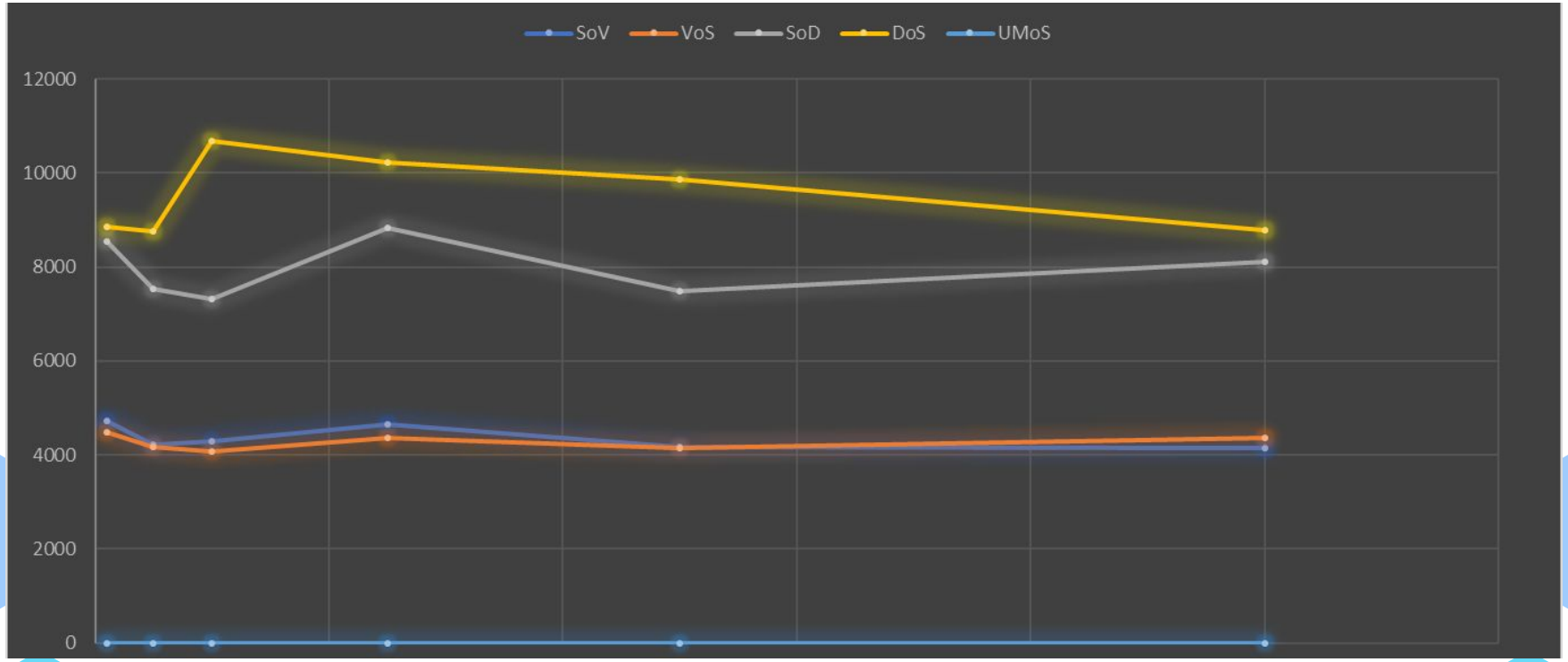
Тест 6: Вставка в начало



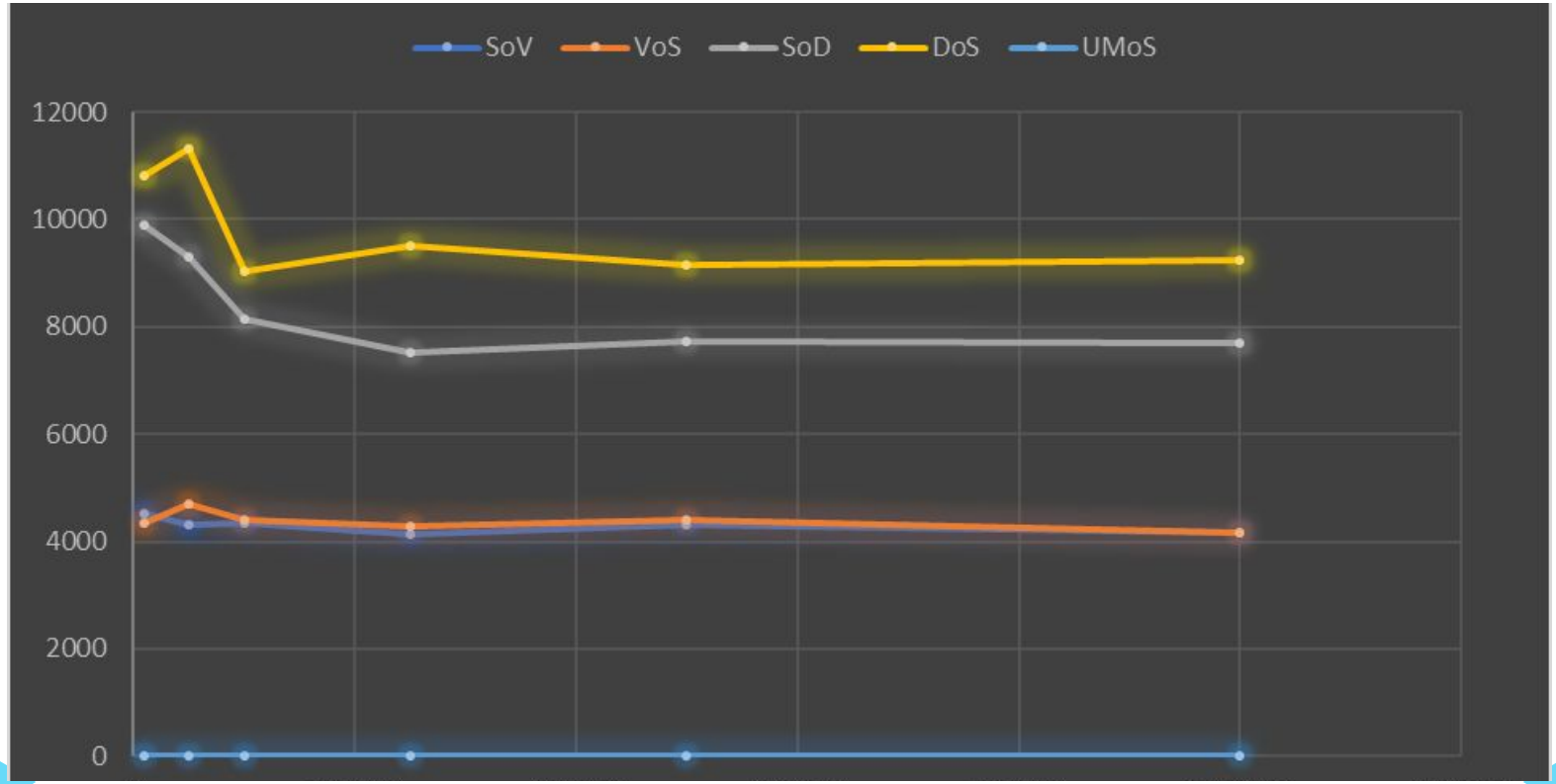
Тест 7: Удаление из начала



Тест 8: Вставка в середину



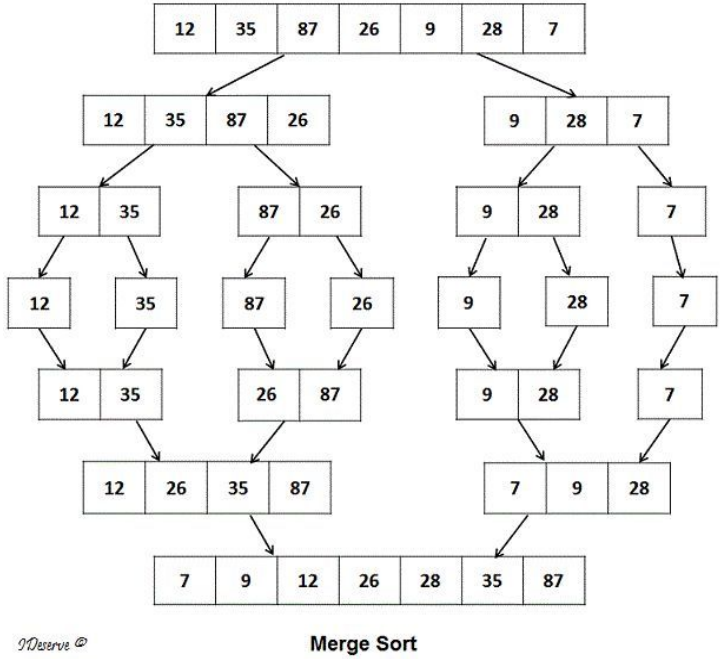
Тест 9: Удаление из середины



Сортировка слиянием. Описание алгоритма

Merge Sort — это стабильный алгоритм сортировки с гарантированной временной сложностью O(n log n). Алгоритм состоит из трёх этапов:

1. **Индексная сортировка** — вместо перемещения 11 массивов SoA, сортируются только индексы (экономия памяти и операций)
2. **Итеративное слияние** — bottom-up подход с удвоением размера подмассивов
3. **Цикловая перестановка** — реальные данные переставляются один раз в конце, используя алгоритм циклов для минимизации swap операций



Почему Merge Sort?

- Merge Sort требует намного меньше операций обмена, чем Quick Sort ($O(n \log n)$ vs $O(n^2)$ в худшем случае)
- Гарантированная производительность — $O(n \log n)$ в лучшем, среднем и худшем случаях (в отличие от Quick Sort)
- Индексная сортировка — ключевая оптимизация. Пока сортируются int индексы, реальные данные неподвижны. Затем переставляются один раз через циклы
- Детерминированность — для набора данных (100k записей) алгоритм показывает стабильное время выполнения

```cpp
// Инициализация индексов
std::vector<int> indices(size);
for (int i = 0; i < size; ++i) indices[i] = i;

// Bottom-up слияние
for (int width = 1; width < size; width *= 2) {
    for (int left = 0; left < size; left += 2 * width) {
        int mid = std::min(left + width, size);
        int right = std::min(left + 2 * width, size);

        if (mid < right) {
            // Слияние подмассивов indices[left..mid) и indices[mid..right)
            // Сравнения: get(indices[i]) <= get(indices[j])
            merge(left, mid, right);
        }
    }
}
```

```
MergeSort took 4196519mcs.
MergeSort finished correctly
```

Тест на 1 млн записей

Выбор алгоритма КМП и его преимущества

Алгоритм	Сложность	Преимущества	Недостатки	Почему не подошел
Наивный поиск	O(n×m)	Простая реализация	Медленный на больших данных	Недопустимо медленно для 50к+ записей
Rabin-Karp	O(n+m) в среднем	Хорош для множества паттернов	Хеш-коллизии, ложные срабатывания	Сложность реализации, нужна проверка коллизий
Boyer-Moore	O(n/m) в лучшем	Очень быстрый на практике	Сложный, плох для маленьких алфавитов	Оптимизирован для английского текста
КМП (мой выбор)	O(n+m) гарантировано	Надежный, простой, линейный	Требует предобработки паттерна	Идеален для нашего случая

Как работает КМП— принцип работы

Шпаргалка" **КМР** (префикс-функция)**:**
Текст**:** АБАБАБЦ
Ищем слово**:** А Б А Б Ц
Шпаргалка**: 0 0 1 2 0**

Строим шпаргалку один раз → **Читаем текст один раз без возвратов** → **Каждую букву проверяем один раз**

Реализация алгоритма КМП

```cpp
#pragma once

#include <vector>
#include <string>
#include <functional>
#include <algorithm>

namespace substring_search {

    template<typename DatasetType, typename Getter>
    std::vector<int> countSubstringOccurrences(
        DatasetType& dataset,
        Getter getter,
        const std::string& pattern
    ) {
        int size = dataset.get_size();
        std::vector<int> results(size, 0);

        if (pattern.empty()) {
            return results;
        }

        // 1. Вычисление префикс-функции (LPS)
        int m = pattern.length();
        std::vector<int> lps(m, 0);
        int len = 0;
        int i = 1;

        while (i < m) {
            if (pattern[i] == pattern[len]) {
                len++;
                lps[i] = len;
                i++;
            }
            else {
                if (len != 0) {
                    len = lps[len - 1];
                }
                else {
                    lps[i] = 0;
                    i++;
                }
            }
        }
```

```cpp
        // 2. КМР поиск для каждой записи
        for (int idx = 0; idx < size; idx++) {
            const char* fieldValue = getter(dataset, idx);
            if (fieldValue != nullptr) {
                std::string text(fieldValue);

                int n = text.length();
                if (m <= n) {
                    int count = 0;
                    int textIdx = 0;
                    int patternIdx = 0;

                    while (textIdx < n) {
                        if (pattern[patternIdx] == text[textIdx]) {
                            textIdx++;
                            patternIdx++;
                        }

                        if (patternIdx == m) {
                            count++;
                            patternIdx = lps[patternIdx - 1];
                        }
                        else if (textIdx < n && pattern[patternIdx] != text[textIdx]) {
                            if (patternIdx != 0) {
                                patternIdx = lps[patternIdx - 1];
                            }
                            else {
                                textIdx++;
                            }
                        }
                    }

                    results[idx] = count;
                }
            }
        }

        return results;
    }
```

Сортировка подсчётом

- Алгоритм: counting sort (bucket для дискретных ключей).
- Подходит для малокардинальных полей: severity, enum, bool.
- Реализация через getter: значение берётся как get(i).
- Совместима с AoS и SoA через общий интерфейс swapitems().

```
template <class Dataset, typename getter>
inline void bucket_sort_by_severity(Dataset& ds, getter get) {
    const int n = ds.get_size();
    if (n <= 1) return;

    auto mn = get(0);
    auto mx = mn;
    for (int i = 1; i < n; ++i) {
        auto s = get(i);
        if (s < mn) mn = s;
        if (s > mx) mx = s;
```

Почему он подходит

- Не всегда нужны универсальные алгоритмы: данные часто повторяются.
- Counting sort работает за O(n + k), где k — число уникальных значений.
- Для bool: k = 2, для severity: k ≈ 4 → почти линейное время.
- Идеален для колонок true/false в исходном датасете (флаги/признаки).

```
// Severity sorting with bucket sort
bucket_sort_by_severity(AoS, [&AoS](int i) {return AoS.get_severity(i); });
bool flag = true;
for (int i = 0; i < 99999; i++) {
    if (AoS.get_severity(i) > AoS.get_severity(i + 1)) flag = false;
    //cout << AoS.get_severity(i);
}
cout << ((flag) ? "Bucket sorting finished correct\n" : "Bucket sorting is incorrect\n");
return 0;
```

Ограничения

1)Только дискретные значения: алгоритм работает, когда ключ — целое/категория с небольшим числом вариантов (например bool, severity, enum).

2)Не подходит для float: для float диапазон и количество уникальных значений большие → k = max-min+1 становится огромным, память/время растут, выигрыш пропадает.

3)Не подходит для string: нельзя напрямую построить массив частот по строкам без дополнительного этапа (кодирование/хеш/словари), что уже меняет алгоритм и может не дать ускорения.

4)Зависимость от диапазона (k): сложность O(n + k): если k большой, counting sort может стать хуже обычной сортировки (O(n log n)).

5)Память: требуется дополнительная память под cnt[k], start[k], order[n] → при

Замеры (пример)

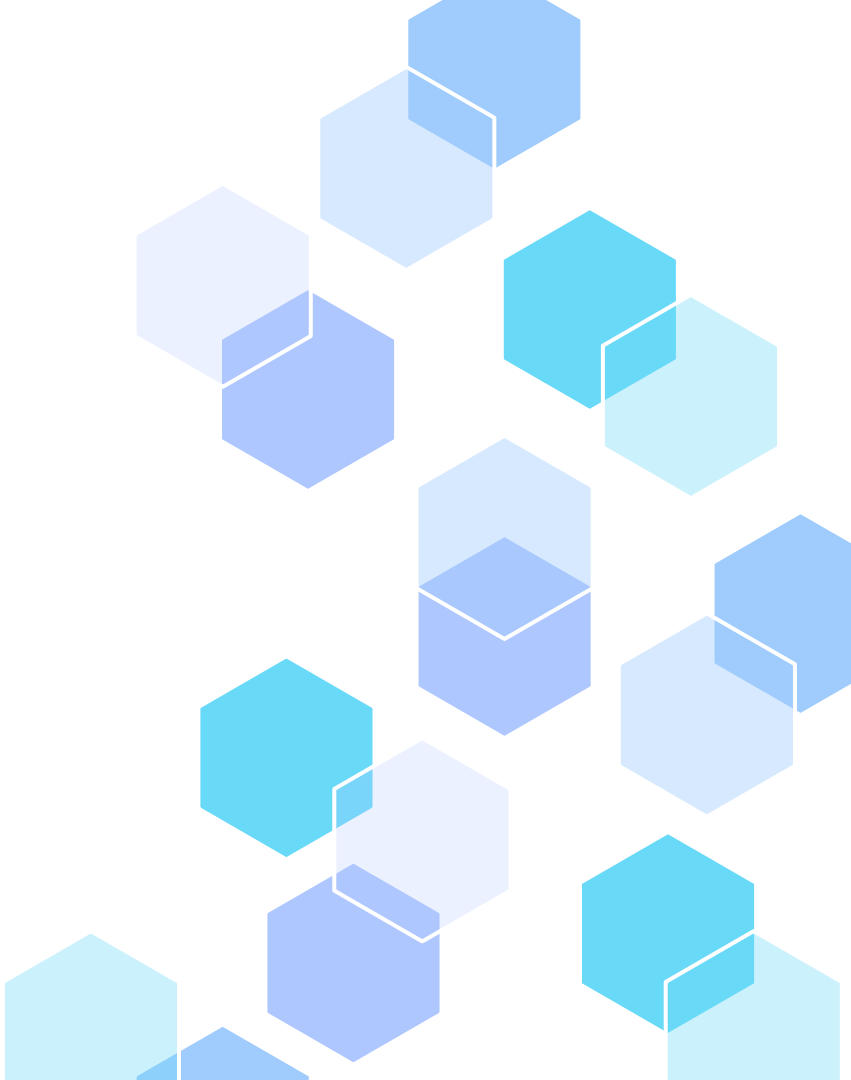
Ниже — пример измерения времени (Test) в microseconds на 1 млн записей severity.

Для low-cardinality полей (bool, severity) counting sort даёт O(n + k).

```
BucketSort took 441929mcs.
BucketSort finished correctly
```

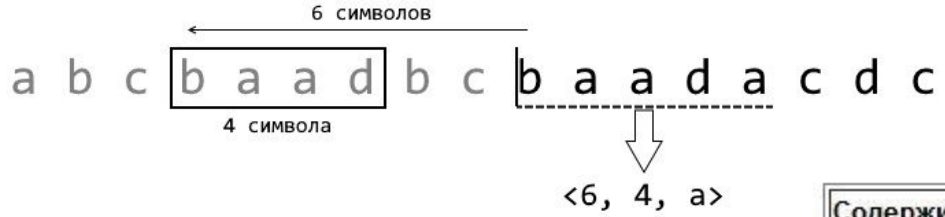
Алгоритмы сжатия

- LZ77
- LZ78
- LZW



LZ77

В кодируемых строках часто содержатся совпадающие длинные подстроки. Идея, лежащая в основе LZ77, заключается в замене повторений на ссылки на позиции в тексте, где такие подстроки уже встречались.Информацию о повторении можно закодировать парой чисел — смещением назад от текущей позиции (offset) и длиной совпадающей

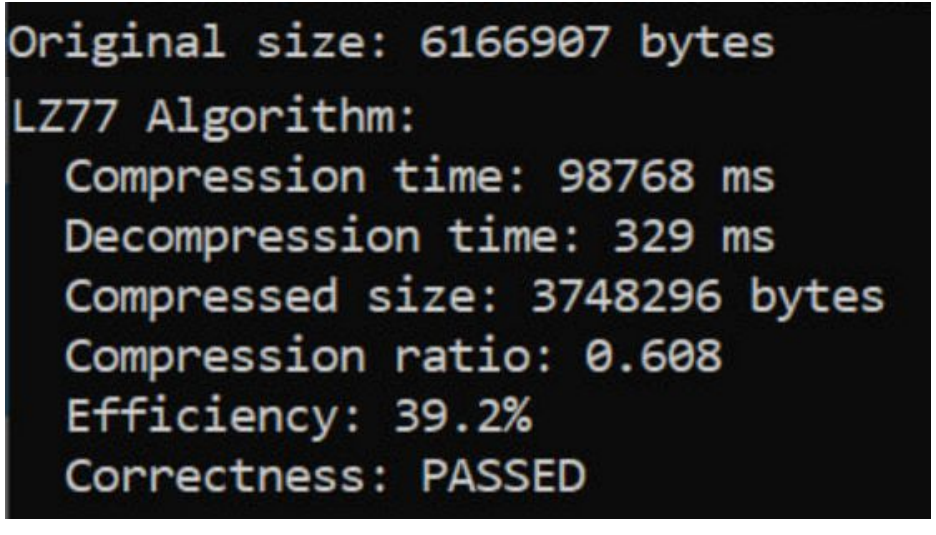


Содержимое окна	Содержимое буфера	КОД
$kababababab z$	k	<0,0,k>
$kababababab z$	a	<0,0,a>
$kababababab z$	b	<0,0,b>
k ab $ababababz$	aba	<2,2,a>
$kaba$ babab abz	$bababz$	<2,5,z>

LZ77-реализация

Принцип работы:

- Скользящее окно
- Поиск совпадений в буфере
- Токены (смещение, длина, символ)




```cpp
// Токен LZ77
struct LZ77Token {
    uint16_t offset;    // Смещение назад
    uint16_t length;    // Длина совпадения
    char next_char;     // Следующий символ

    LZ77Token(uint16_t o, uint16_t l, char n)
        : offset(o), length(l), next_char(n) {}
};

// Сжатие LZ77
vector<LZ77Token> LZ77Compressor::compress(
    const string& input,
    size_t window_size = 4096,
    size_t lookahead_buffer = 18) {

    vector<LZ77Token> output;
    size_t input_len = input.length();
    size_t pos = 0;

     while (pos < input_len) {
        // Поиск наилучшего совпадения в скользящем окне
        size_t match_offset = 0;
        size_t match_length = 0;
        char next_char = input[pos];

        // ... алгоритм поиска совпадений ...

        if (match_length > 0) {
            output.emplace_back(match_offset, match_length, next_char);
            pos += match_length + 1;
        } else {
            output.emplace_back(0, 0, next_char);
            pos++;
        }
    }

    return output;
}
```

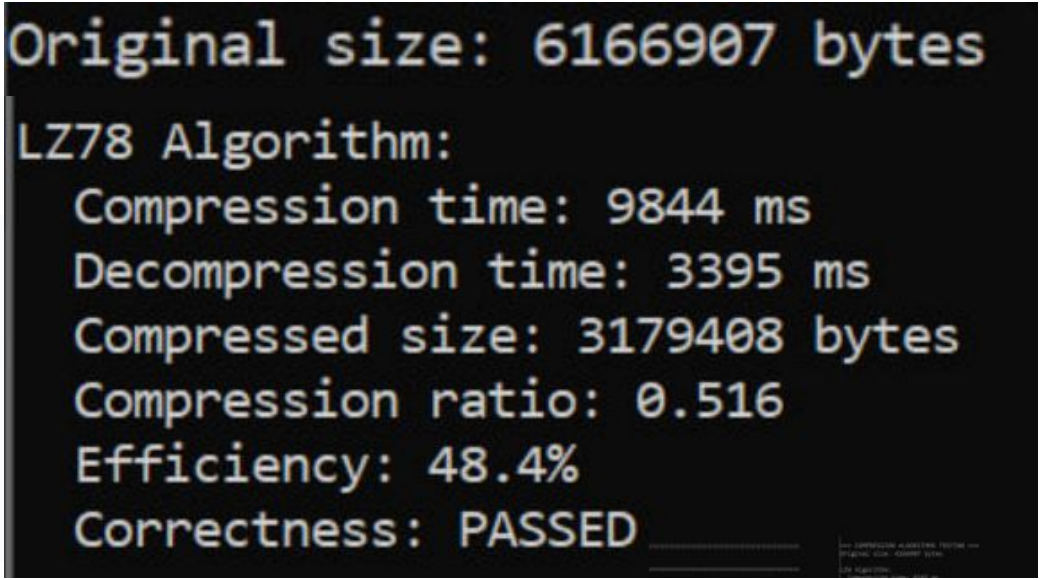
LZ78

LZ78 имеет немного другую идею: этот алгоритм в явном виде использует словарный подход, генерируя временный словарь во время кодирования и декодирования. Изначально словарь пуст, а алгоритм пытается закодировать первый символ. На каждой итерации мы пытаемся увеличить кодируемый префикс, пока такой префикс есть в словаре. Кодовые слова такого алгоритма будут состоять из двух частей — номера в словаре самого длинного найденного префикса (pos) и символа, который идет за этим префиксом (next). При этом после кодирования такой пары префикс с приписанным символом добавляется в словарь, а алгоритм продолжает кодирование со следующего символа.

Словарь	Осталось обработать	Найденный префикс	Код	Примечание
$\varnothing$	$abacababacabc$	—	$\langle 0, a\rangle$	В словаре ничего не нашлось, вместо номера в словаре указываем 0
a	$bacababacabc$	—	$\langle 0, b\rangle$	
a, b	$acababacabc$	a	$\langle 1, c\rangle$	Нашелся префикс a (слова в словаре нумеруются с 1), добавляем ac
a, b, ac	$ababacabc$	a	$\langle 1, b\rangle$	
a, b, ac, ab	$abacabc$	ab	$\langle 4, a\rangle$	
a, b, ac, ab, aba	$cabc$	—	$\langle 0, c\rangle$	
a, b, ac, ab, aba, c	abc	ab	$\langle 4, c\rangle$	

LZ78-реализация

```cpp
// Токен LZ78
struct LZ78Token {
    uint16_t index;     // Индекс в словаре
    char next_char;     // Новый символ

    LZ78Token(uint16_t i, char n) : index(i), next_char(n) {}
};

// Сжатие LZ78
vector<LZ78Token> LZ78Compressor::compress(const string& input) {
    unordered_map<string, uint16_t> dictionary;
    vector<LZ78Token> output;

    dictionary[""] = 0;
    uint16_t next_code = 1;
    string current;

    for (char c : input) {
        string combined = current + c;
        if (dictionary.find(combined) != dictionary.end()) {
            current = combined;
        } else {
            output.emplace_back(dictionary[current], c);
            dictionary[combined] = next_code++;
            current = "";
        }
    }

    return output;
}
```



LZW

```
Original size: 6166907 bytes

LZW Algorithm:
  Compression time: 9265 ms
  Decompression time: 1353 ms
  Compressed size: 1909324 bytes
  Compression ratio: 0.310
  Efficiency: 69.0%
  Correctness: PASSED
```

```
=== COMPRESSION ALGORITHMS BENCHMARK ===
========================================
Algorithm Compress(ms)Decompress(ms)    OriginalCompressed     RatioEfficiency%
------------------------------------------------------------------------------
LZW              10141          1616     6166907   1909324     0.310      69.0%
LZ77            106455           273     6166907   3748296     0.608      39.2%
LZ78              8737          3420     6166907   3179408     0.516      48.4%

BEST ALGORITHM: LZW (efficiency: 69.0%)
```

Ссылка на репозиторий

https://github.com/JustStorm/Dataset_Hw_AC-24-04