Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Российский химико-технологический университет имени Д.И. Менделеева»

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

ПО КУРСУ

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

Ведущий преподаватель

Ассистент Крашенинников Р. С.

СТУДЕНТ группы КС-36 Лупинос А. В.

Москва

2025

Задание

В лабораторной работе предлагается изучить способ анализа алгоритма, связанный со временем. Рассмотреть для выбранного алгоритма сортировки наилучшие, наихудшее и среднее время и соотнести его с известным для алгоритма показателем эффективности О-большое.

Допускает реализация задания на любом языке программирования, кроме лиспоподобных. Преподаватель может не знать конкретного языка реализации, поэтому вы должны быть способны объяснить алгоритм и нарисовать его без демонстрации непосредственно вашего кода.

Задание:

Используя предыдущий код посерийного выполнения алгоритма сортировки и измерения времени, требуется реализовать метод сортировки слиянием.

1. Реализовать проведение тестирования алгоритма сериями расчётов для измерения параметров времени. За один расчет выполняются следующие операции:
    a. Генерируется массив случайных значений.
    b. Запоминается время начала расчета алгоритма сортировки.
    c. Выполняется алгоритм сортировки.
    d. Вычисляется время, затраченное на сортировку: текущее время минус время начала.
    e. Сохраняется время для одной попытки. После этого расчет повторяется до окончания серии.
        i. Алгоритм вычисляется 8 сериями по 20 раз за серию.
        ii. Алгоритм в каждой серии вычисляется для массива размером M (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000).
        iii. Массив заполняется числами с плавающей запятой в интервале от -1 до 1.
        iv. Для серии запоминаются все времена, которые были замерены.
2. При работе сортировки подсчитать:

a. Количество вызовов рекурсивной функции.
b. Глубину рекурсии.
c. Время выполнения сортировки.
d. Максимальное потребление дополнительной памяти (память за вычетом памяти до начала сортировки).

3. По полученным данным времени построить графики зависимости времени от числа элементов в массиве:
   a. Совмещенный график наихудшего времени выполнения сортировки и сложности алгоритма, указанной в нотации О большое[1].
   b. Совмещенный график среднего, наилучшего и наихудшего времени выполнения.
   c. Совмещенный график средней, наилучшей и наихудшей глубины рекурсии.
   d. Совмещенный график среднего, наилучшего и наихудшего потребления дополнительной памяти.
   e. Совмещенный график среднего, наилучшего и наихудшего количества вызовов рекурсивной функции.
4. По результатам расчетов оформляется отчет по предоставленной форме. В отчёте:
   a. Приводится описание алгоритма.
   b. Приводится описание выполнения задачи (описание кода и специфических элементов реализации).
   c. Приводятся выводы (графики и их анализ). Требуется ответить на вопрос о поведении алгоритма, изученного в процессе выполнения лабораторной работы, и зафиксировать его особенности.

Описание алгоритма

[1] Для построения графика вычисляется О большое для каждого размера массива. При этом при вычислении функции O(c * g(N)) подбирается такая константа c, чтобы при значении > 1000 график O(N) был выше графика наихудшего случая, но второй график на его фоне не превращался в прямую линию

Сортировка слиянием (англ. merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определенном порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Алгоритм был изобретен Джоном фон Нейманом в 1945 году.

Один из наиболее быстрых известных универсальных алгоритмов сортировки массивов обладает сложностью $O(n \cdot log_2 n)$ во всех случаях — лучшем, среднем и худшем — при упорядочивании n элементов. Однако из-за наличия ряда недостатков на практике он обычно применяется с определенными доработками.

Процесс сортировки включает следующие шаги:

1. Сортируемый массив разделяется на две части примерно одинакового размера.
2. Каждая из полученных частей сортируется отдельно с использованием того же алгоритма.
3. Два упорядоченных массива половинного размера объединяются в один.

Рассмотрим два массива a и b (фактически это две части одного массива, но для удобства описания они рассматриваются как отдельные массивы). Требуется получить массив c с размером $|a| + |b|$. Для этого применяется процедура слияния. Она заключается в следующем: элементы массивов сравниваются, начиная с их начала, и меньший из них записывается в результирующий массив. Затем в том массиве, где оказался меньший элемент, происходит переход к следующему элементу, после чего сравнение продолжается. Если один из массивов заканчивается раньше, оставшиеся элементы другого массива просто дописываются в результирующий. В завершение результирующий массив заменяет два исходных, образуя отсортированный участок.

Затраты памяти при сортировке слиянием составляют $O(n)$, так как для слияния двух подмассивов требуется дополнительный массив размером n, пропорциональный исходному количеству элементов.

Глубина рекурсии — это то, насколько глубоко алгоритм "погружается" в деление массива на части. В сортировке слиянием массив каждый раз делится пополам, пока не получится много маленьких кусочков размером в один элемент. Чтобы понять, сколько раз можно поделить n пополам, используется логарифм по основанию 2 ($log_2 n$). Например, для массива из 8 элементов его делят 3 раза ($8 \rightarrow 4 \rightarrow 2 \rightarrow 1$), и глубина будет примерно $log_2 8 = 3$. В общем случае глубина рекурсии равна $log_2 n + 1$, где +1 — это начальный уровень.

Количество вызовов рекурсивной функции показывает, сколько раз алгоритм вызывает сам себя, чтобы отсортировать все части массива. Каждый раз, когда массив делится на две части, происходит два новых вызова — один для левой половины, другой для правой. В итоге для массива из n элементов таких вызовов будет ровно $2 \cdot n - 1$. Например, для 4 элементов: 1 вызов для всего массива, 2 для половин (2 и 2), и ещё 4 для отдельных элементов — итого 7 вызовов, что равно $2 \cdot 4 - 1$.

Описание выполнения задачи

Для выполнения алгоритма сортировки слиянием была реализована программа на языке программирования C++:

```
#include <iostream>
#include <random>
#include <vector>
#include <chrono>
#include <limits>

using namespace std;

struct SortStats {
    int recursion_calls;
    int max_depth;
    int max_extra_memory;
};
```

```
void merge(vector<double> &arr, int left, int mid, int right, SortStats &stats) {
    int temp_mem = right - left;
    if (temp_mem > stats.max_extra_memory) {
        stats.max_extra_memory = temp_mem;
    }

    vector<double> result(right - left);
    int it1 = 0, it2 = 0;
    while ((left + it1) < mid && (mid + it2) < right) {
        if (arr[left + it1] < arr[mid + it2]) {
            result[it1 + it2] = arr[left + it1];
            ++it1;
        } else {
            result[it1 + it2] = arr[mid + it2];
            ++it2;
        }
    }
    while ((left + it1) < mid) {
        result[it1 + it2] = arr[left + it1];
        ++it1;
    }
    while ((mid + it2) < right) {
        result[it1 + it2] = arr[mid + it2];
        ++it2;
    }
    for (int i = 0; i < (it1 + it2); ++i) {
        arr[left + i] = result[i];
    }
}

void mergeSort(vector<double> &arr, int left, int right, int depth, SortStats &stats) {
    stats.recursion_calls++;
    if (depth > stats.max_depth) {
        stats.max_depth = depth;
    }

    if ((left + 1) >= right) {
        return;
    }

    int mid = (left + right) / 2;

    mergeSort(arr, left, mid, depth + 1, stats);
    mergeSort(arr, mid, right, depth + 1, stats);

    merge(arr, left, mid, right, stats);
}

int main() {
    int sizes[8] = {1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000};
    random_device rd;
    mt19937 gen(rd());
```

```
    uniform_real_distribution<> distribution(-1, 1);

    // Векторы для хранения итоговых данных по всем размерам
    vector<double> max_times(8), avg_times(8), min_times(8);
    vector<int> max_rec_calls(8), avg_rec_calls(8), min_rec_calls(8);
    vector<int> max_depths(8), avg_depths(8), min_depths(8);
    vector<int> max_memories(8), avg_memories(8), min_memories(8);

    for (int s = 0; s < 8; ++s) {
        int size = sizes[s];
        vector<double> times(20);
        vector<int> recursion_counts(20);
        vector<int> depths(20);
        vector<int> memories(20);

        for (int k = 0; k < 20; ++k) {
            vector<double> arr(size);
            for (auto &element : arr) {
                element = distribution(gen);
            }

            SortStats stats = {0, 0, 0};
            auto start = chrono::high_resolution_clock::now();
            mergeSort(arr, 0, size, 0, stats);
            auto end = chrono::high_resolution_clock::now();

            // Измеряем время в миллисекундах напрямую
            chrono::duration<double, milli> ms_diff = end - start;
            double time_ms = ms_diff.count();

            times[k] = time_ms;
            recursion_counts[k] = stats.recursion_calls;
            depths[k] = stats.max_depth;
            memories[k] = stats.max_extra_memory;
        }

        // Вычисление средних, минимальных и максимальных значений
                double avg_time = 0, min_time = numeric_limits<double>::max(), max_time =
numeric_limits<double>::lowest();
                    int avg_rec = 0, min_rec = numeric_limits<int>::max(), max_rec =
numeric_limits<int>::lowest();
                int avg_depth = 0, min_depth = numeric_limits<int>::max(), max_depth =
numeric_limits<int>::lowest();
            int avg_memory = 0, min_memory = numeric_limits<int>::max(), max_memory =
numeric_limits<int>::lowest();

        for (int i = 0; i < 20; ++i) {
            avg_time += times[i];
            min_time = min(min_time, times[i]);
            max_time = max(max_time, times[i]);

            avg_rec += recursion_counts[i];
            min_rec = min(min_rec, recursion_counts[i]);
```

```
            max_rec = max(max_rec, recursion_counts[i]);

            avg_depth += depths[i];
            min_depth = min(min_depth, depths[i]);
            max_depth = max(max_depth, depths[i]);

            avg_memory += memories[i];
            min_memory = min(min_memory, memories[i]);
            max_memory = max(max_memory, memories[i]);
        }

        avg_time /= 20;
        avg_rec /= 20;
        avg_depth /= 20;
        avg_memory /= 20;

        // Сохранение данных для текущего размера
        max_times[s] = max_time;
        avg_times[s] = avg_time;
        min_times[s] = min_time;
        max_rec_calls[s] = max_rec;
        avg_rec_calls[s] = avg_rec;
        min_rec_calls[s] = min_rec;
        max_depths[s] = max_depth;
        avg_depths[s] = avg_depth;
        min_depths[s] = min_depth;
        max_memories[s] = max_memory;
        avg_memories[s] = avg_memory;
        min_memories[s] = min_memory;
    }

    // Вывод данных в табличной форме для каждого графика

    // 1. График наихудшего времени
    cout << "\nWorst Time" << endl;
    cout << "Size,Worst Time (ms)" << endl;
    for (int s = 0; s < 8; ++s) {
        cout << sizes[s] << "," << max_times[s] << endl;
    }

    // 2. График среднего, наилучшего и наихудшего времени
    cout << "\nExecution Time (ms)" << endl;
    cout << "Size,Average Time,Best Time,Worst Time" << endl;
    for (int s = 0; s < 8; ++s) {
        cout << sizes[s] << "," << avg_times[s] << "," << min_times[s] << "," << max_times[s] <<
endl;
    }

    // 3. График среднего, наилучшего и наихудшего числа вызовов
    cout << "\nRecursion Calls" << endl;
    cout << "Size,Avg Calls,Min Calls,Max Calls" << endl;
    for (int s = 0; s < 8; ++s) {
```

```
            cout << sizes[s] << "," << avg_rec_calls[s] << "," << min_rec_calls[s] << "," <<
max_rec_calls[s] << endl;
        }

        // 4. График средней, наилучшей и наихудшей глубины
        cout << "\nRecursion Depth" << endl;
        cout << "Size,Avg Depth,Min Depth,Max Depth" << endl;
        for (int s = 0; s < 8; ++s) {
            cout << sizes[s] << "," << avg_depths[s] << "," << min_depths[s] << "," << max_depths[s]
<< endl;
        }

        // 5. График среднего, наилучшего и наихудшего потребления памяти
        cout << "\nExtra Memory" << endl;
        cout << "Size,Avg Memory,Min Memory,Max Memory" << endl;
        for (int s = 0; s < 8; ++s) {
            cout << sizes[s] << "," << avg_memories[s] << "," << min_memories[s] << "," <<
max_memories[s] << endl;
        }

        return 0;
    }
```

Данная программа реализует алгоритм сортировки слиянием (merge sort) и тестирует его производительность на массивах различных размеров. Алгоритм многократно выполняется для каждого размера массива, измеряются время работы, количество вызовов рекурсивной функции, глубина рекурсии и потребления дополнительной памяти.

Структура кода:

- Библиотеки: Подключены <iostream>, <random>, <vector>, <chrono>, <limits> для вывода, генерации чисел, массивов, времени и границ типов.
- Структура SortStats: Хранит recursion_calls (вызовы), max_depth (глубина), max_extra_memory (память) с типом int.
- Функция merge: Сливает два подмассива, подсчитывает память как right - left, создает временный вектор и объединяет элементы.
- Функция mergeSort: Рекурсивно делит массив, увеличивает recursion_calls и max_depth, вызывает merge для слияния.

- Функция main: Обрабатывает 8 размеров (1000–128000), выполняет 20 попыток на серию, генерирует числа в [-1, 1], замеряет время, подсчитывает параметры и выводит результаты.

Специфические элементы реализации:

- Время: Измеряется в миллисекундах через <chrono> с типом chrono::duration<double, milli>.
- Память: Считается в элементах $O(n)$, фиксируется максимум в merge.
- Рекурсия: Вызовы $(2 \cdot n - 1)$ и глубина $(log_2 n + 1)$ отслеживаются через stats.

Выводы

В ходе выполнения лабораторной работы была реализована сортировка слиянием на языке C++ и проведено ее тестирование на массивах различного размера. Для анализа эффективности алгоритма были измерены параметры времени, количества рекурсивных вызовов, глубины рекурсии и потребления дополнительной памяти.

Для наглядного представления результатов одного тестового запуска программы были построены пять диаграмм:

1. Совмещенный график наихудшего времени выполнения сортировки и сложности алгоритма в нотации *O* большое (рис. 1). Методом подбора найдена константа $c = 0.0001$, при которой график $O(n \cdot log_2 n)$ лежит выше линии наихудшего времени для $n > 1000$, сохраняя читаемость зависимости "Worst Time".
2. График среднего, наилучшего и наихудшего времени выполнения (рис. 2).
3. График среднего, минимального и максимального количества рекурсивных вызовов (рис. 3).
4. График средней, минимальной и максимальной глубины рекурсии (рис. 4).
5. График среднего, минимального и максимального потребления дополнительной памяти (рис. 5).

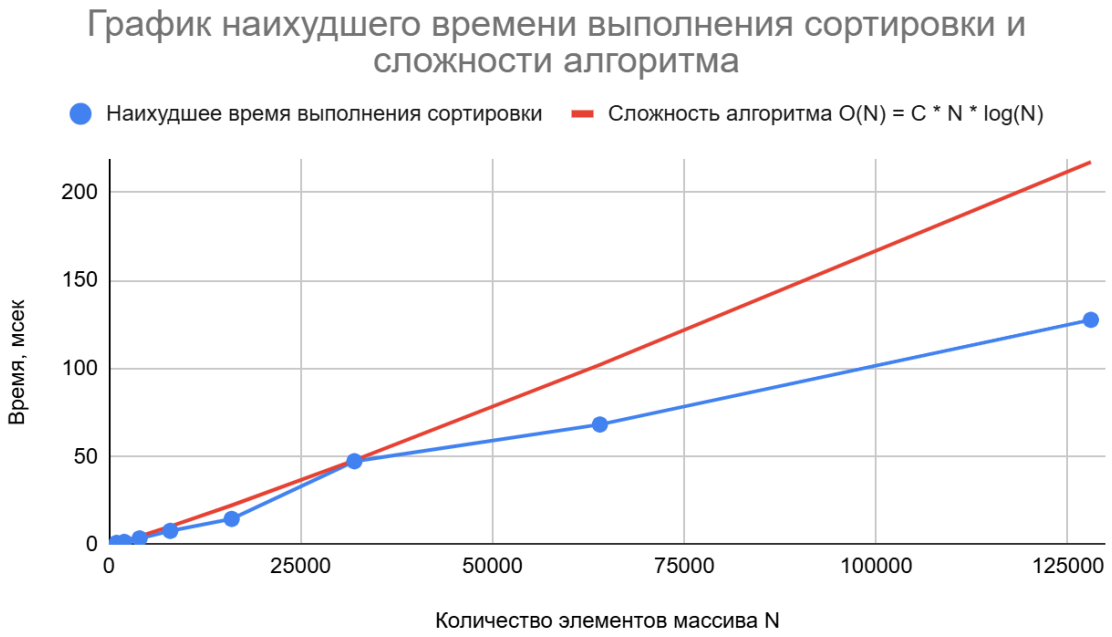

Рисунок 1 - График наихудшего времени выполнения сортировки и сложности алгоритма (п. 1)

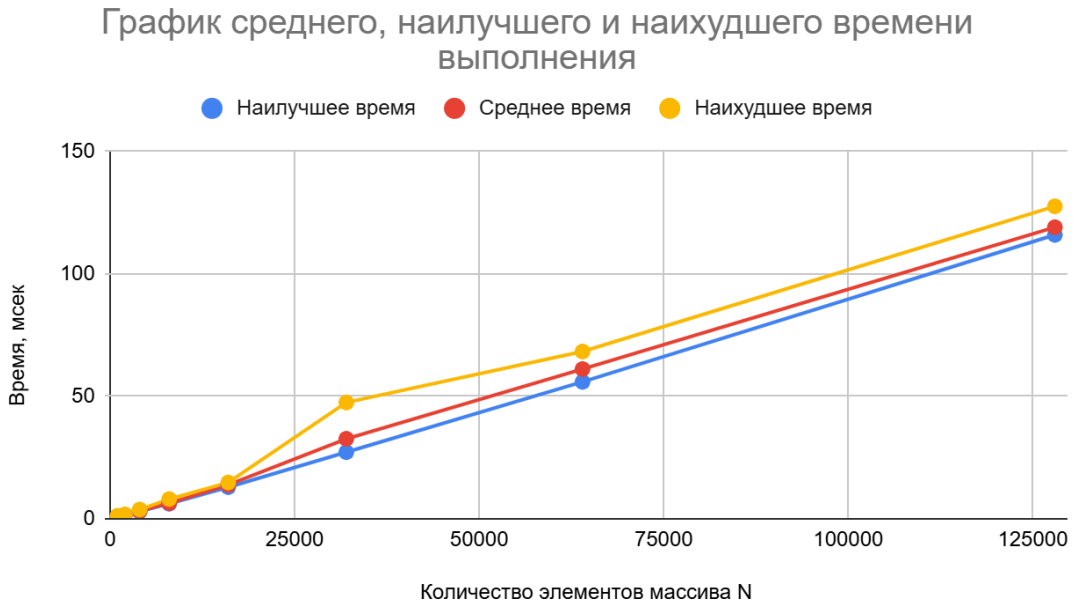

Рисунок 2 - График среднего, наилучшего и наихудшего времени выполнения (п. 2)

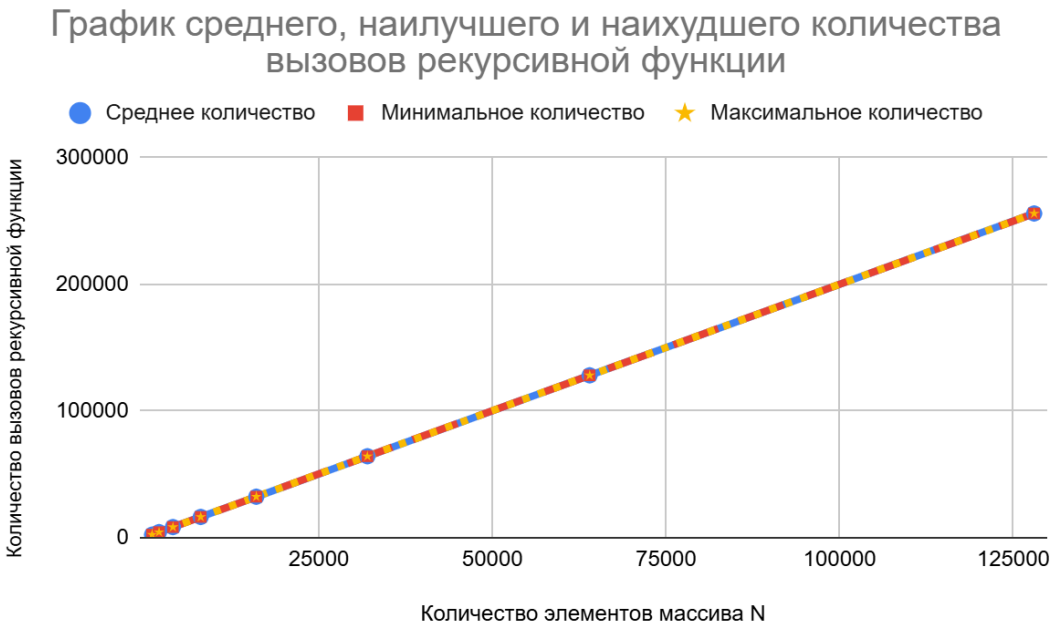

Рисунок 3 - График среднего, минимального и максимального количества рекурсивных вызовов (п. 3)

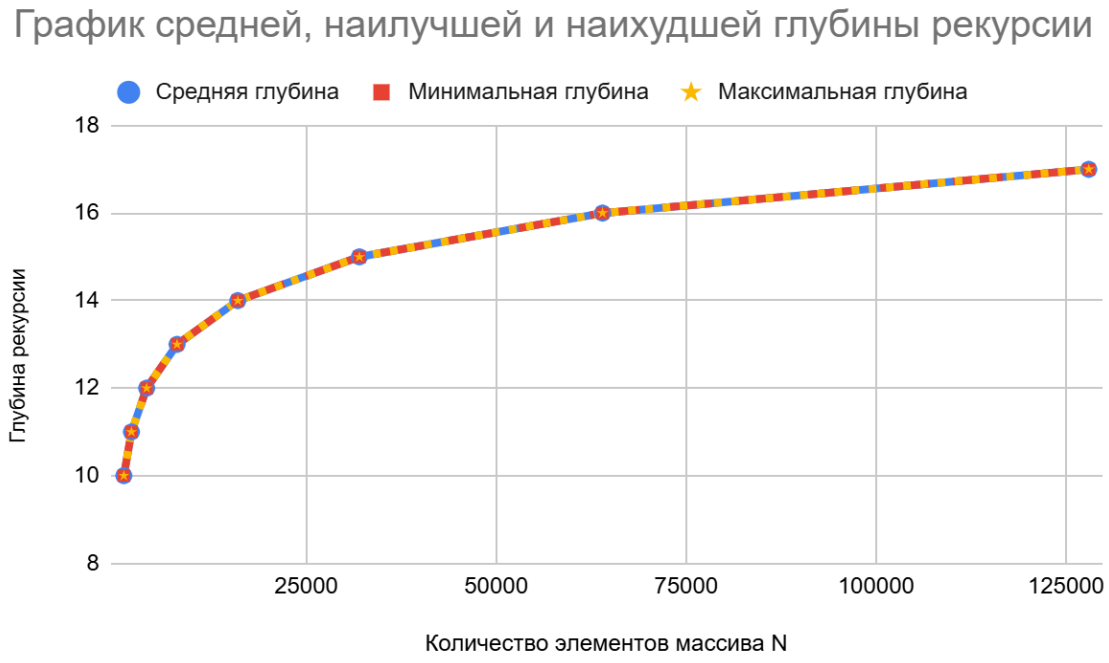

Рисунок 4 - График средней, минимальной и максимальной глубины рекурсии (п. 4)

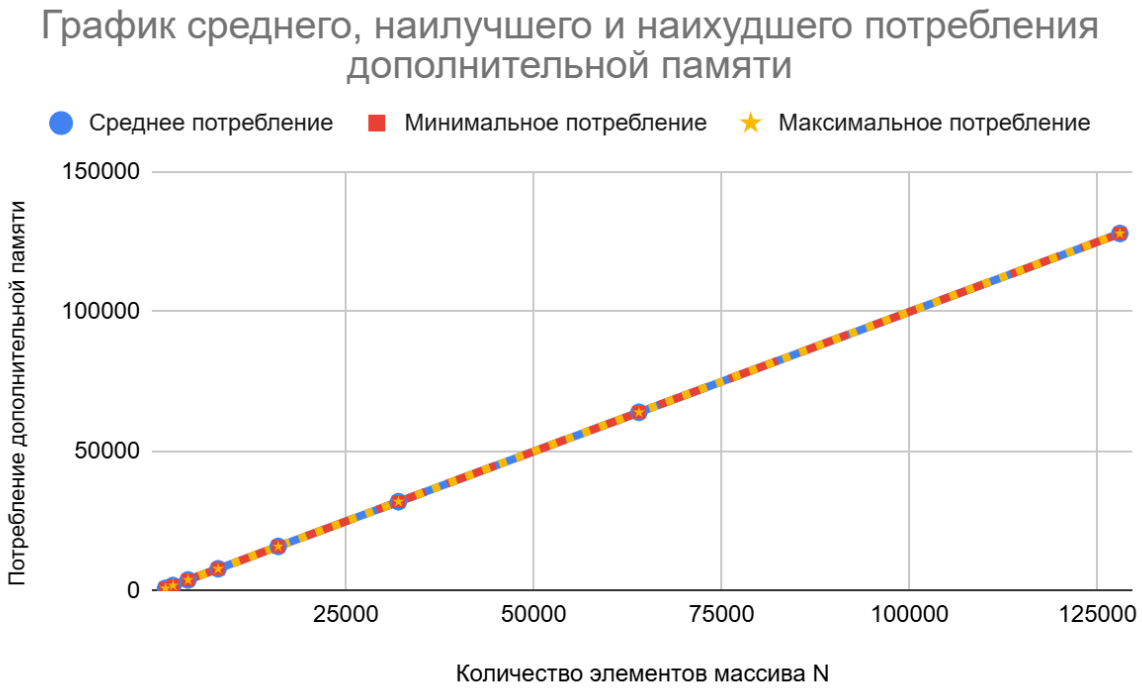

Рисунок 5 - График среднего, минимального и максимального потребления дополнительной памяти (п. 5)

На основании построенных графиков подтверждается теоретическая оценка сложности алгоритма $O(n \cdot log_2 n)$ для сортировки слиянием. Диаграммы демонстрируют, что время выполнения растёт пропорционально $n \cdot log_2 n$ с увеличением размера массива. Количество рекурсивных вызовов $2 \cdot n - 1$и глубина рекурсии $log_2 n + 1$ остаются стабильными и не зависят от исходного порядка элементов, что указывает на отсутствие влияния начальной упорядоченности массива на эффективность алгоритма.

Таким образом, сортировка слиянием показывает высокую стабильность и предсказуемость по сравнению с алгоритмами вроде шейкерной сортировки.