Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт

по лабораторной работе №3

Название: Исследование трудоемкости сортировок

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б (Группа) _____ (Подпись, дата) Н. В. Ляпина (И.О. Фамилия)

Преподователь _____ (Подпись, дата) Л.Л. Волкова (И.О. Фамилия)

Москва, 2022

Содержание

Введение

Алгоритм сортировки – это алгоритм для упорядочивания элементов в списке. Входом является последовательность из n элементов: $a_1, a_2, ..., a_n$. Результатом работы алгоритма сортировки является перестановка исходной последовательности $a'_1, a'_2, ..., a'_n$, такая что $a'_1 \leqslant a'_2 \leqslant ... \leqslant a'_n$, где $\leqslant$ – отношение порядка на множестве элементов списка. Поля, служащие критерием порядка, называются ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

В данной лабораторной работе рассмотриваются алгоритмы:

1) сортировка плавная;
2) сортировка расчёской;
3) сортировка слиянием.

Цель лабораторной работы:
изучить трудоемкости алгоритмов сортировки и их реализация.

Задачи лабораторной работы:

1) выбрать инструменты для замера процессорного времени выполнения реализации алгоритмов;
2) изучить алгоритмы сортировки расчёской, слияением и плавной сортировки;
3) реализовать:
    а) плавную сортировку;
    б) сортировку расчёской;
    в) сортировку слиянием.
4) дать оценку трудоёмкости в лучшем, произвольном и худшем случае;
5) провести замеры процессорного времени работы для лучшего, худшего и произвольного случая.

1 Аналитический раздел

1.1 Плавная сортировка (Smoothsort)

Плавная сортировка – алгоритм сортировки выбором, разновидность пирамидальной сортировки, разработанная Э. Дейкстрой. Но, в отличии от пирамидальной сортировки, в которой используется двоичная куча, здесь исползуется специальная куча, полученная с помощью чисел Леонардо.

Числа Леонардо

Числа Ленонардо – последовательность чисел, задаваемая зависимостью:

$$L(n) = \begin{cases} 1, & \text{если } n = 0; \\ 1, & \text{если } n = 1; \\ L(n-1) + L(n-2) + 1, & \text{если } n > 1. \end{cases} \tag{1.1}$$

где n – индекс в массиве чисел Леонардо.

Абсолютно любое целое число можно представить в виде суммы чисел Леонардо, имеющих разные порядковые номера. Массив из n элементов не всегда можно представить в виде одной кучи Леонардо, но любой массив можно разделить на несколько подмассивов, которые будут соответствовать разным числам Леонардо.

При разбиении массива важно учитывать, что:

1) Каждая куча Чисел Леонардо представляет собой несбалансированное бинарное дерево;

2) Корень каждой такой кучи – это последний элемент соответствующего подмассива;

3) Любой узел кучи со всеми своими потомками также представляет из себя леонардову кучу меньшего порядка.

Алгоритм

1) Создаем из массива кучу леонардовых куч, каждая из которых является сортирующим деревом;

   а) Перебираем элементы массива слева-направо;

   б) Проверяем, можно ли с помощью текущего элемента объединить две крайние слева кучи в уже имеющейся куче леонардовых куч:

   — Если да, то объединяем две крайние слева кучи в одну, текущий элемент становится корнем этой кучи, делаем просейку для объединённой кучи;

   — Если нет, то добавляет текущий элемент в качестве новой кучи (состоящей пока из одного узла) в имеющуюся кучу леонардовых куч.

2) Извлекает из куч текущие максимальные элементы, которые перемещакем в конец неотсортированной части массива:

а) Ищем максимумы в леонардовых кучах. Так как на предыдущем этапе для куч постоянно делалась просейка, максимумы находятся в корнях этих куч;

б) Найденный максимум (который является корнем одной из куч) меняем местами с последним элементом массива (который является корнем самой последней кучи);

в) После этого обмена куча, в который был найден максимум перестала быть сортирующим деревом. Поэтому делаем для неё просейку;

г) В последней куче удаляем корень, в результате чего эта куча распадается на две кучи;

д) После перемещения максимального элемента в конец, отсортированная часть массива увеличилась, а неотсортированная часть уменьшилась;

е) Повторить пункты а - б.

1.2 Сортировка расчёской

Основная идея сортировки расчёской заключается в том, чтобы изначально брать самое большое расстояние между сравниваемыми элементами. Затем, по мере упорядочивания массива сужать это расстояние до минимального. Таким образом, мы как бы расчёсываем массив сначала широким гребнем, потом гребнем поменьше. Этот принцип отражается в названии сортировки. Первоначальный разрыв между сравниваемыми элементами лучше брать с учётом специальной величины, называемой фактором уменьшения, оптимальное значение которой равно примерно 1,247.

Сначала расстояние между элементами максимально, то есть равно размеру массива минус один. Затем, пройдя массив с этим шагом, необходимо поделить шаг на фактор уменьшения и пройти по списку вновь. Так продолжается до тех пор, пока разность индексов не достигнет единицы. В этом случае сравниваются соседние элементы как и в сортировке пузырьком, но такая итерация одна.

1.3 Сортировка слиянием

Алгоритм использует принцип «разделяй и властвуй»: задача разбивается на подзадачи меньшего размера, которые решаются по отдельности, после чего их решения комбинируются для получения решения исходной задачи. Конкретно процедуру сортировки слиянием можно описать следующим образом:

1) Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.

2) Иначе массив разбивается на две части, которые сортируются рекурсивно.

3) После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

Вывод

Были рассмотрены алгоритмы сортировки расчёской, слиянием и плавная сортировка. Каждый имеет свою особенность, а конкретно сложность работы в лучшем и худшем случаях.

2 Констукторский раздел

В данном разделе будут рассмотрены схемы алгоритмов, требования к функциональности ПО, и определены способы тестирования.

2.1 Требования к функциональности ПО

В данной работе требуется обеспечить следующую функциональность.

1) Пользовательский режим:
   а) возможность подать на вход массив;
   б) вывод результата корректной сортировки.
2) Тестовый режим:
   а) возможность замера процессорного времени реализации каждой сортировки в худшем, лучшем и произвольном случае.

2.2 Схемы алгоритмов

Ниже будут представлены схемы сортировок:

1) плавная сортировка (рисунок 2.1);
2) сортировка расчёской (рисунок 2.5);
3) сортировка слияением (рисунок 2.6).

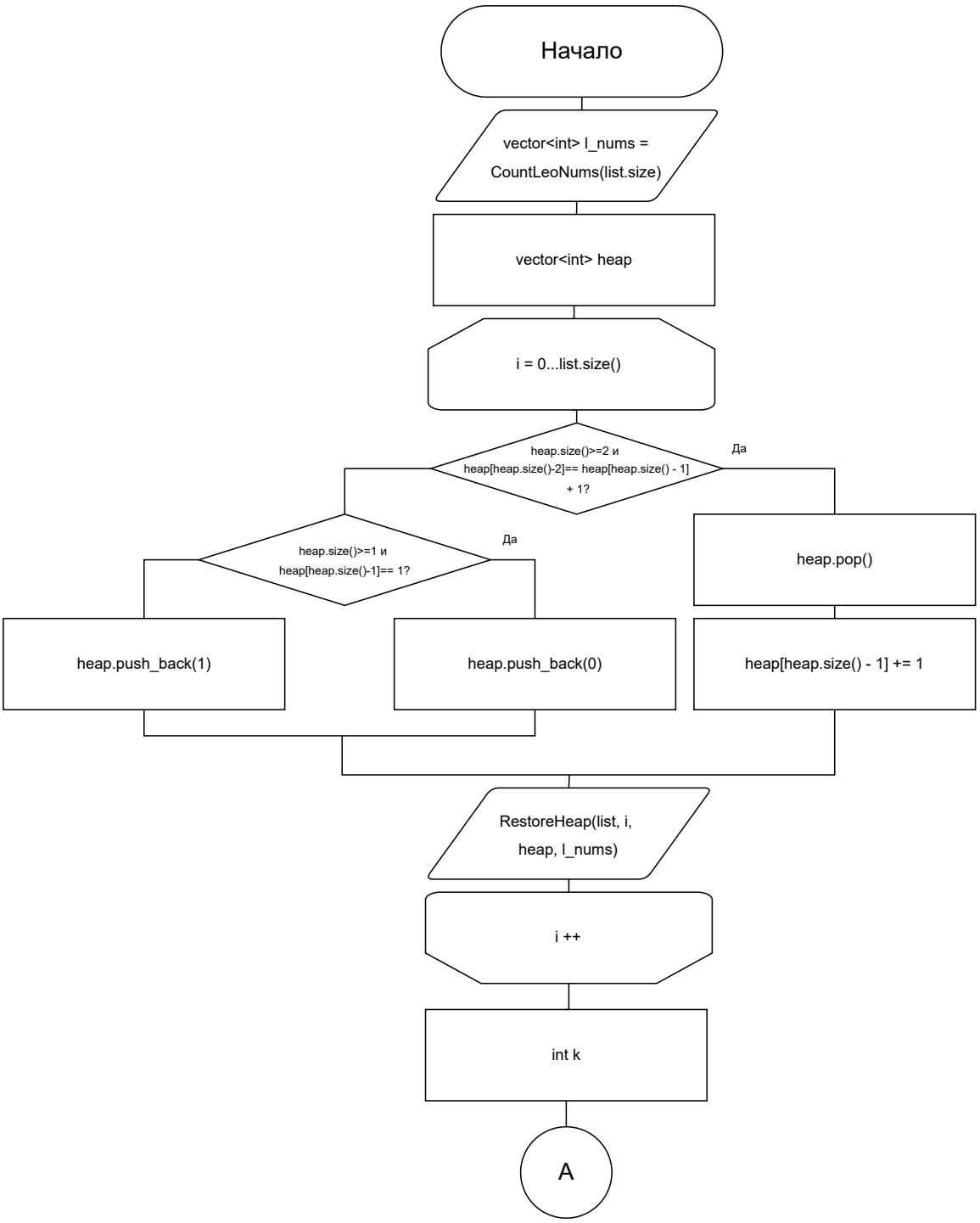

Рисунок 2.1 — Схема плавной сортировки Часть 1

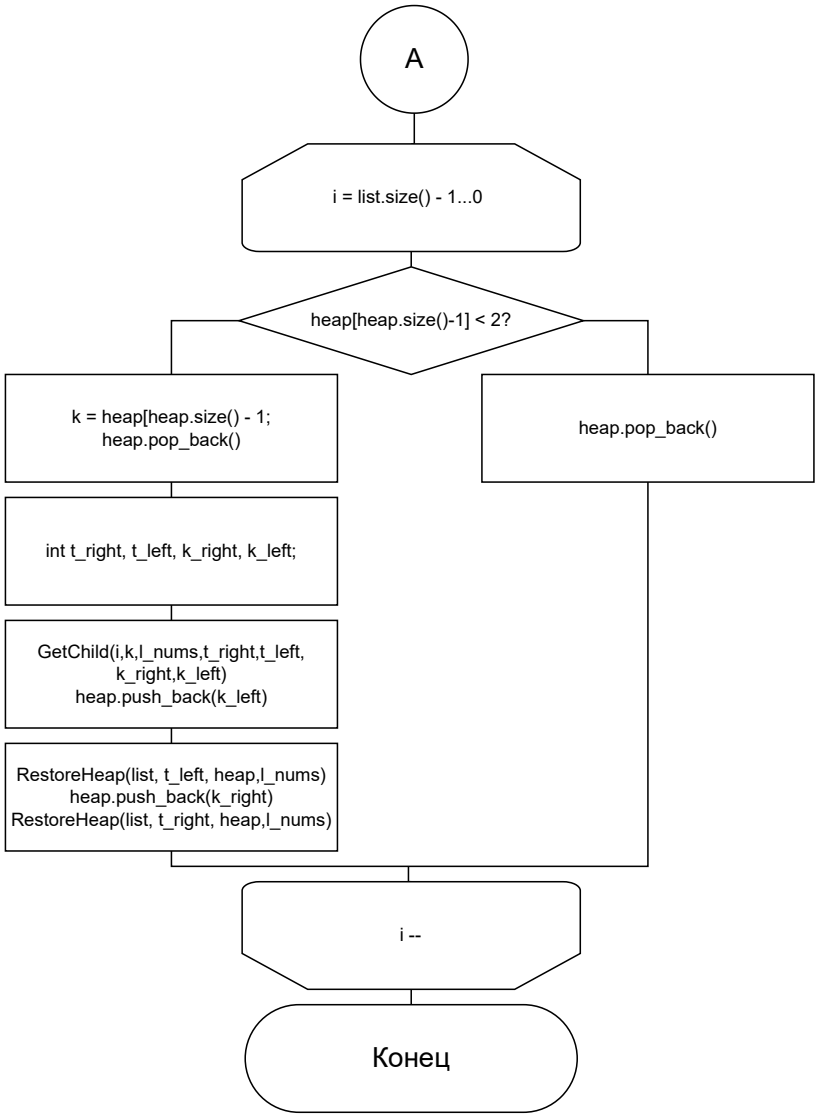

Рисунок 2.2 — Схема плавной сортировки Часть 2

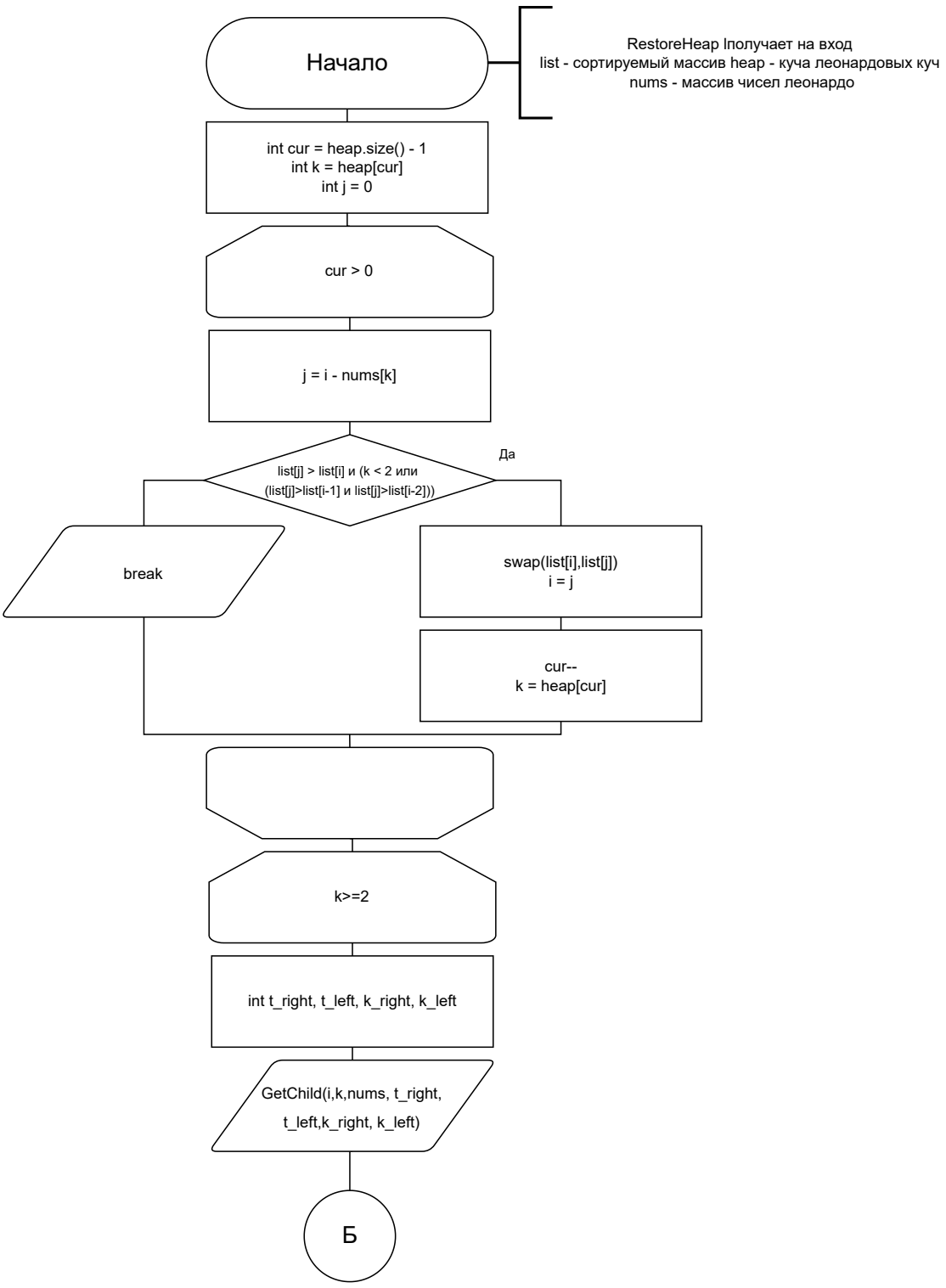

Рисунок 2.3 — Схема функции RestoreHeap, используемой в плавной сортировке Часть 1

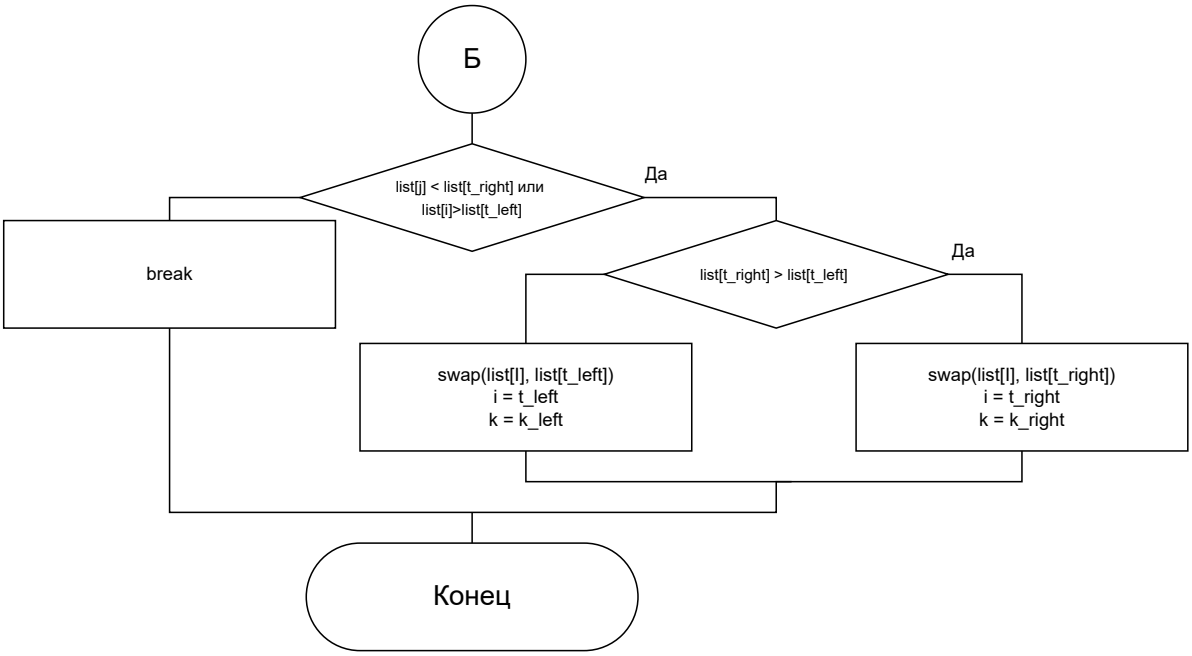

Рисунок 2.4 — Схема функции RestoreHeap, используемой в плавной сортировке Часть 2

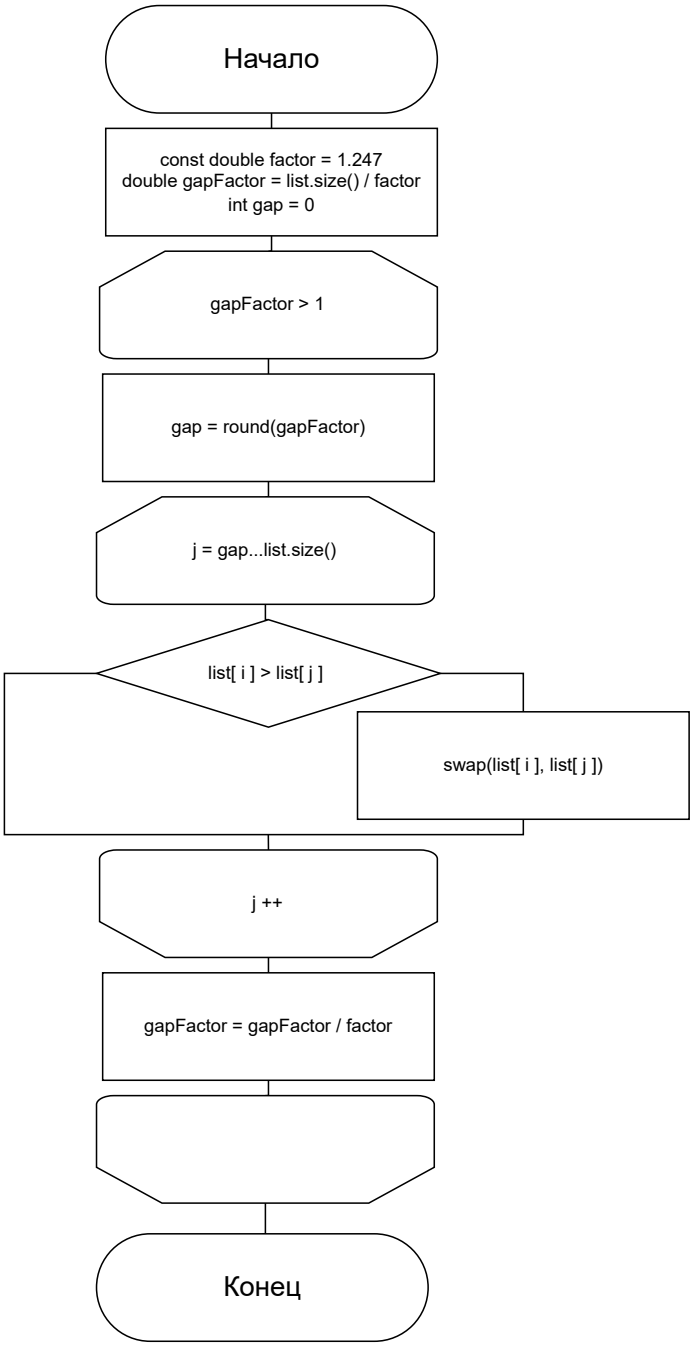

Рисунок 2.5 — Схема сортировки расчёской

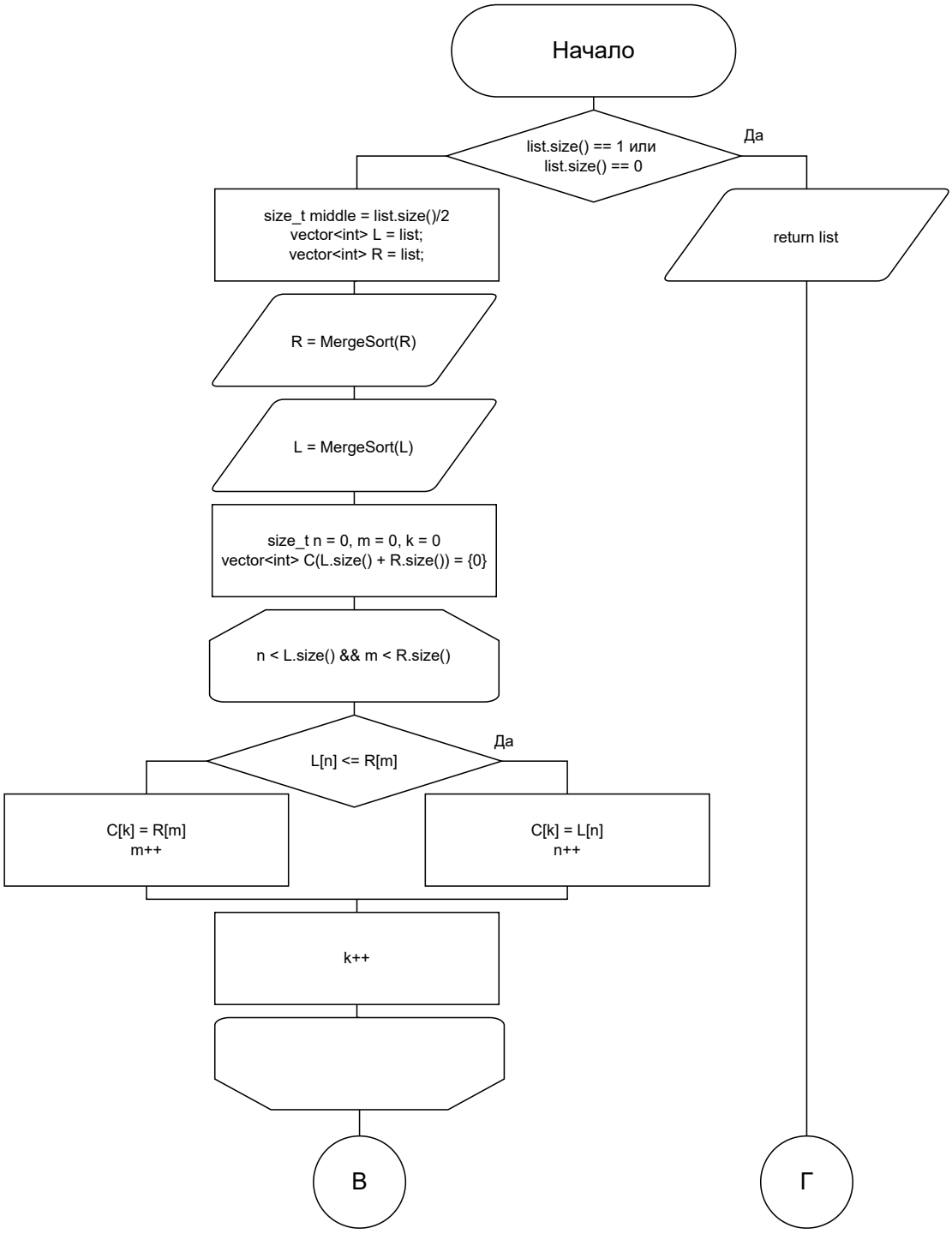

Рисунок 2.6 — Схема сортировки слиянием Часть 1

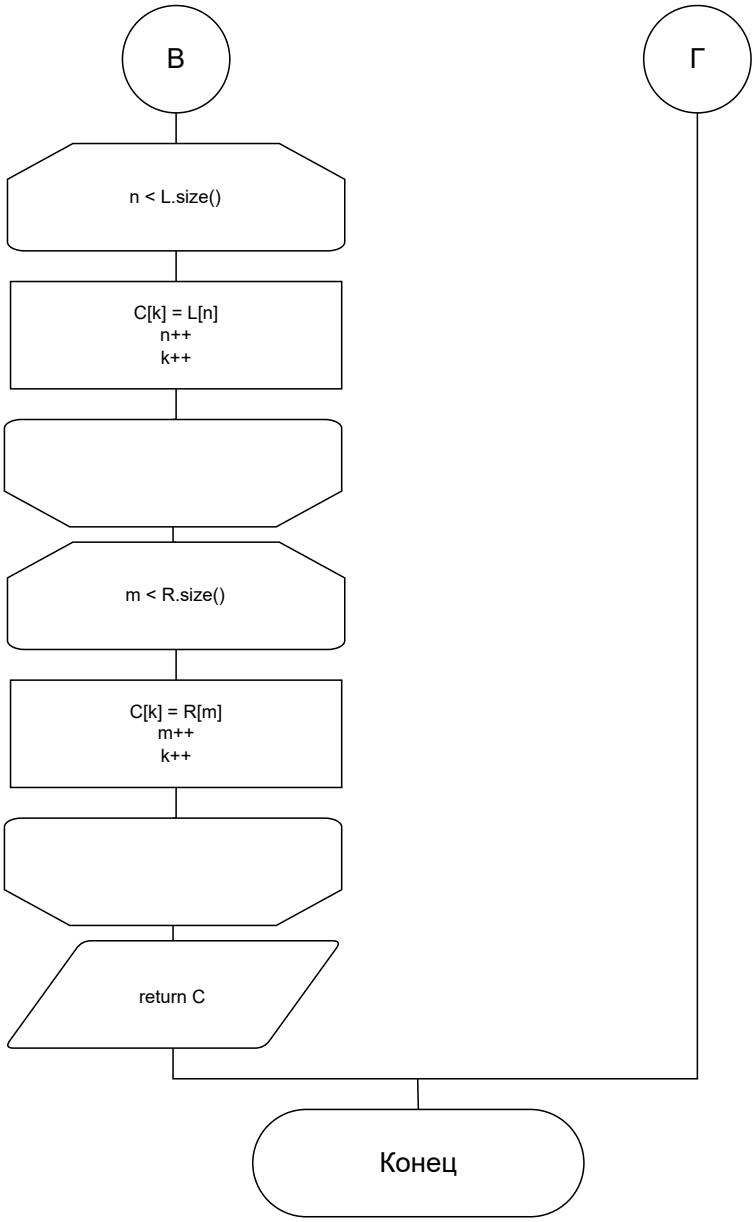

Рисунок 2.7 — Схема сортировки слиянием Часть 2

2.3 Тесты

Тестирование ПО будет проводиться методом чёрного ящика. Необходимо проверить работу системы на массивах различной длины.

2.4 Трудоемкость алгоритма

Трудоёмкость – количество работы, которую алгоритм затрачивает на обработку данных. Является функцией от длины входов алгоритма и позволяет оценить количество работы.

Введём модель вычисления трудоёмкости.

2.4.1 Базовые операции

Стоимость представленных ниже операций единична:

1) $=, +, + =, -, - =, *, * =, /, / =, ++, --, \%$
2) $<, \leqslant, >, \geqslant, ==, \neq$
3) []

2.4.2 Условный оператор

if(условие){
// Тело A
}
$else${
// Тело B
}

Пусть трудоемкость тела A равна f_A, а тела B f_B, тогда трудоемкость условного оператора можно найти по формуле (2.1):

$$f_{if} = f_{uslovie} + \begin{cases} min(f_A, f_B), & \text{– лучший случай,} \\ max(f_A, f_B), & \text{– худший случай.} \end{cases} \tag{2.1}$$

2.4.3 Цикл со счетчиком

$for(int\ i = 0; i < n; i++)${
// Тело цикла
}

Начальная инициализация цикла $int\ i = 0$ выполняется один раз. Условие $i < n$ проверяется перед каждой итерацией цикла и при входе в цикл – $n + 1$ операций. Тело цикла выполняется ровно n раз. Счётчик $i++$ выполняется на каждой итерации, перед проверкой условия, т.е. n раз. Тогда, если трудоёмкость тела цикла равна f, трудоёмкость всего цикла определяется формулой (2.2):

$$f_{for} = 2 + n(2 + f) \tag{2.2}$$

2.4.4 Сортировка расчёской

Лучший случай

Массив отсортирован, обмены между элементами не происходят.

$$f_{CombSort} = 3 + 1 + M \cdot (1 + 2 + 1 + 3 + N \cdot (3 + 3)) = O(N \cdot \log N) \tag{2.3}$$

Худший случай

Массив неотсортирован.

$$f_{CombSort} = 3 + 1 + M \cdot (1 + 2 + 1 + 3 + N \cdot (3 + 3 + 3 + 4)) = O(N^2) \tag{2.4}$$

2.4.5 Сортировка слиянием

Лучший случай и худший случай

Так как сортировка происходит в любом случае, то трудоемкость в лучшем и худшем случае одинакова.

$$f_{MergeSort} = \frac{N}{2} \cdot (2+2+2 \cdot N \cdot (2+1)+2+3+3+M \cdot (2+3+4)+K \cdot (1+3+2)+L \cdot (1+2+3)) = O(N \cdot \log N) \tag{2.5}$$

2.4.6 Плавная сортировка

Лучший случай

Массив отсортирован, обмены между элементами не происходят.

$$f_{CountLeoNums} = 2 + 2 + N \cdot (1 + 3 + 2) = 4 + N \cdot 6 \tag{2.6}$$

$$f_{RestoreHeap} = 2 + 2 + 3 + H_{size} \cdot (2 + 2 + 12) + 1 + M \cdot (1 + 8 + 3 + 3) \tag{2.7}$$

$$f_{SmoothSort} = 1 + f_{CountLeoNums} + 2 + N \cdot (2 + 7 + 3 + f_{RestoreHeap}) + 2 + N \cdot (2 + 3) = O(N) \tag{2.8}$$

Худший случай

Массив неотсортирован.

$$f_{CountLeoNums} = 2 + 2 + N \cdot (1 + 3 + 2) = 4 + N \cdot 6 \tag{2.9}$$

$$f_{RestoreHeap} = 2 + 2 + 3 + H_{size} \cdot (2 + 2 + 12 + 11) + 1 + M \cdot (1 + 8 + 3 + 3 + 3 + 4 + 5) \tag{2.10}$$

$$f_{SmoothSort} = f_{CountLeoNums}+3+N \cdot (2+7+4+f_{RestoreHeap})+2+N \cdot (16+2 \cdot f_{RestoreHeap}) = O(N \cdot \log N) \tag{2.11}$$

Вывод

Сортировка расчёской: лучший – $O(N \cdot \log N)$, худший – $O(N^2)$.

Сортировка слиянием: лучший – $O(N \cdot \log N)$, худший – $O(N \cdot \log N)$.

Плавная сортировка: лучший – $O(N)$, худший – $O(N \cdot \log N)$.

3 Технологический раздел

В данном разделе представлены средства, использованные в процессе разработки для реализации задачи, а также листинг кода программы. Кроме того показаны результаты тестирования и анализа затрачиваемой памяти.

3.1 Средства реализации

Для реализации поставленной задачи был использован язык C++, так как имеется большой опыт работы с ним. Для измерения процессорного времени была использована ассемблерная вставка.

3.2 Сведения о модулях программы

Программа состоит из:

— main.cpp – главный файл программы, в нем располагается точка входа;
— algs.cpp – содержит алгоритмы сортировок;
— time.cpp – содержит ассемблерную вставку для замера времени.

3.3 Листинг программы

В приведенных ниже листингах представлены следующие реализации:

1) плавная сортировка(листинг 3.5 3.6, листинг используемых функций 3.1 3.2 3.4);
2) сортировка расчёской (листинг 3.7);
3) сортировка слиянием (листинг 3.9).

Листинг 3.1 — Функция создания массива чисел Леонардо

```
1  static vector<int> CountLeoNums(int len)
2  {
3      int a = 1, b = 1;
4      int tmp = 0;
5      vector<int> nums;
6      while(a <= len)
7      {
8          nums.push_back(a);
9          tmp = a;
10         a = b;
11         b = tmp + b + 1;
12     }
13     return nums;
14 }
```

Листинг 3.2 — Функция получения идексов в куче чисел Леонардо

```
static void GetChild(int i, int k, vector<int> nums, int &t_r, int &t_l, int &k_r,
    int &k_l)
{
    t_r = i-1;
    k_r = k-2;
    t_l = t_r - nums[k_r];
    k_l = k - 1;
}
```

Листинг 3.3 — Функция пересоздания кучи куч чисел Леонардо

```
static void RestoreHeap(vector<int> &list, int i, vector<int> &heap, vector<int>
    &nums)
{
    int cur = heap.size() - 1;
    int k = heap[cur];
    int j = 0;
    int tmp = 0;
    while (cur > 0)
    {
        j = i - nums[k];
        if ((list[j] > list[i]) && (k < 2 || (list[j] > list[i-1] && list[j] >
            list[i-2])))
        {
            tmp = list[i];
            list[i] = list[j];
            list[j] = tmp;
            i = j;
            cur--;
            k = heap[cur];
        }
        else
            break;
    }

    while (k>=2)
    {
        int t_right, t_left, k_right, k_left;
        GetChild(i, k, nums, t_right, t_left, k_right,k_left);
        if (list[i] < list[t_right] || list[i] < list[t_left])
        {
            if (list[t_right] > list[t_left])
            {
                tmp = list[i];
                list[i] = list[t_right];
```

Листинг 3.4 — Функция пересоздания кучи куч чисел Леонардо

```
1
2                   list[t_right] = tmp;
3                   i = t_right;
4                   k = k_right;
5               }
6               else
7               {
8                   tmp = list[i];
9                   list[i] = list[t_left];
10                  list[t_left] = tmp;
11                  i = t_left;
12                  k = k_left;
13              }
14          }
15          else
16              break;
17      }
18  }
```

Листинг 3.5 — Алгоритм плавной сортировки

```
1  void SmoothSort(vector<int> &list)
2  {
3      vector<int> l_nums = CountLeoNums(list.size());
4
5      vector<int> heap;
6      for (int i = 0; i < (int)list.size(); i++)
7      {
8          if (heap.size() >=2 && heap[heap.size() - 2] == heap[heap.size() - 1] + 1)
9          {
10             heap.pop_back();
11             heap[heap.size() - 1] += 1;
12         }
13         else
14         {
15             if (heap.size() >= 1 && heap[heap.size() - 1] == 1)
16                 heap.push_back(0);
17             else
18                 heap.push_back(1);
19         }
20
21         RestoreHeap(list, i, heap, l_nums);
22
23     }
```

Листинг 3.6 — Алгоритм плавной сортировки

```
    int k;
    for (int i = list.size() - 1; i >= 0; i--)
    {
        if (heap[heap.size() - 1] < 2)
            heap.pop_back();
        else
        {
            k = heap[heap.size() - 1];
            heap.pop_back();
            int t_right, t_left, k_right, k_left;
            GetChild(i, k, l_nums, t_right, t_left, k_right,k_left);
            heap.push_back(k_left);
            RestoreHeap(list, t_left, heap, l_nums);
            heap.push_back(k_right);
            RestoreHeap(list, t_right, heap, l_nums);
        }
    }
}
```

Листинг 3.7 — Алгоритм сортировки расчёской

```
void CombSort(vector<int> &list)
{
    const double factor = 1.247;
    double gapFactor = list.size() / factor;
    int gap = 0;
    while (gapFactor > 1)
    {
        gap = round(gapFactor);

        for(int i = 0, j = gap;j < (int)list.size(); i++, j++)
        {
            if (list[i] > list[j])
            {
                int tmp = list[i];
                list[i] = list[j];
                list[j] = tmp;
            }
        }
        gapFactor = gapFactor / factor;
    }
}
```

Листинг 3.8 — Алгоритм сортировки слиянием

```
vector<int> MergeSort(vector<int> list)
{
    if (list.size() == 1 || list.size() == 0)
        return list;
    size_t middle = list.size()/2;

    vector<int> L;
    for(size_t i = 0; i < middle; i++)
        L.push_back(list[i]);
    vector<int> R;
    for(size_t i = middle; i < list.size() ; i++)
        R.push_back(list[i]);

    R = MergeSort(R);
    L = MergeSort(L);

    size_t n = 0, m = 0, k = 0;
    vector<int> C (L.size() + R.size());
    for(int &l:C)
        l = 0;

    while(n < L.size() && m < R.size())
    {
        if (L[n] <= R[m])
        {
            C[k] = L[n];
            n++;
        }
        else
        {
            C[k] = R[m];
            m++;
        }
        k++;
    }
    while (n < L.size())
    {
        C[k] = L[n];
        n++;
        k++;
    }
```

Листинг 3.9 — Алгоритм сортировки слиянием

```
1
2       while (m < R.size())
3       {
4           C[k] = R[m];
5           m++;
6           k++;
7       }
8       return C;
9   }
```

3.4 Тестирование

В таблице 3.1 отображён возможный набор тестов для тестирования методом чёрного ящика, в соответствующих столбцах таблицы представлены результаты тестирования.

Таблица 3.1 — Тесты проверки корректности программы

№	Input	Smooth	Comb	Merge
1	1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
2	4 3 2 1	1 2 3 4	1 2 3 4	1 2 3 4
3	2 3 1 4	1 2 3 4	1 2 3 4	1 2 3 4
4	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1
5	1	1	1	1

Вывод

Были разработаны и протестированы спроектированные алгоритмы и указаны средства реализации поставленной задачи.

4 Экспериментальный раздел

В данном разделе будут проведены эксперименты для проведения сравнительного анализа алгоритмов по затрачиваемому процессорному времени в зависимости от длины массива и степени его отсортированности.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

В рамках данного проекта были проведёны следующие эксперименты:

1) сравнение времени работы сортировок в лучшем случае(график 4.1);

2) сравнение времени работы сортировок в худшем случае(график 4.2).

3) сравнение времени работы сортировок в произвольном случае(график 4.3).

Тестирование проводилось на компьютере с процессором Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz под управлением Windows 11 с 8 Гб оперативной памяти.

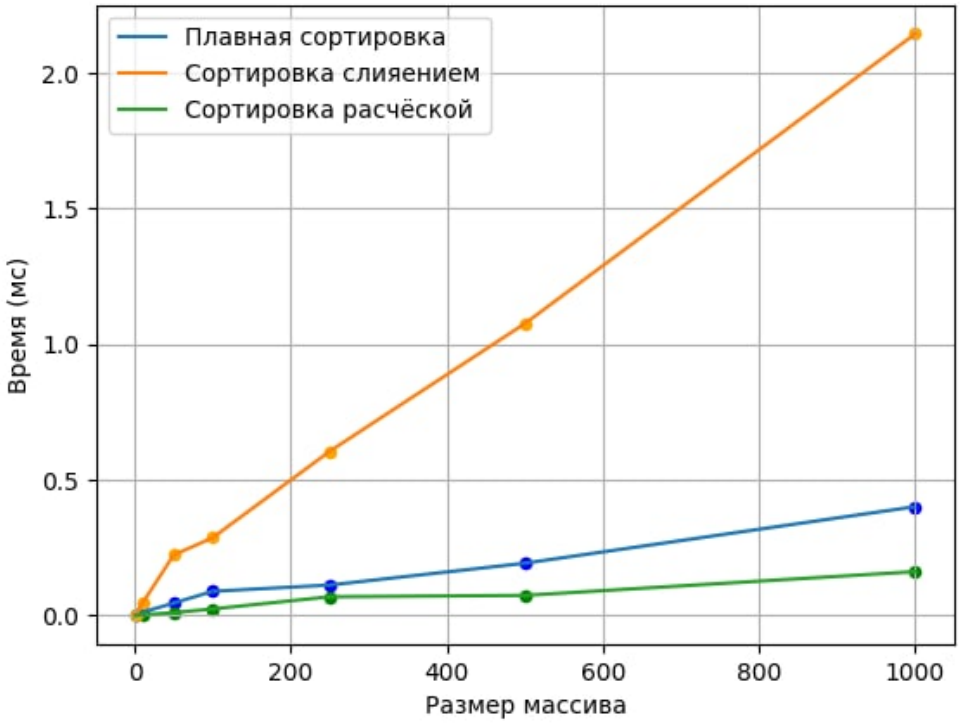

Рисунок 4.1 — Зависимость времени работы алгоритмов от размера массива в лучшем случае

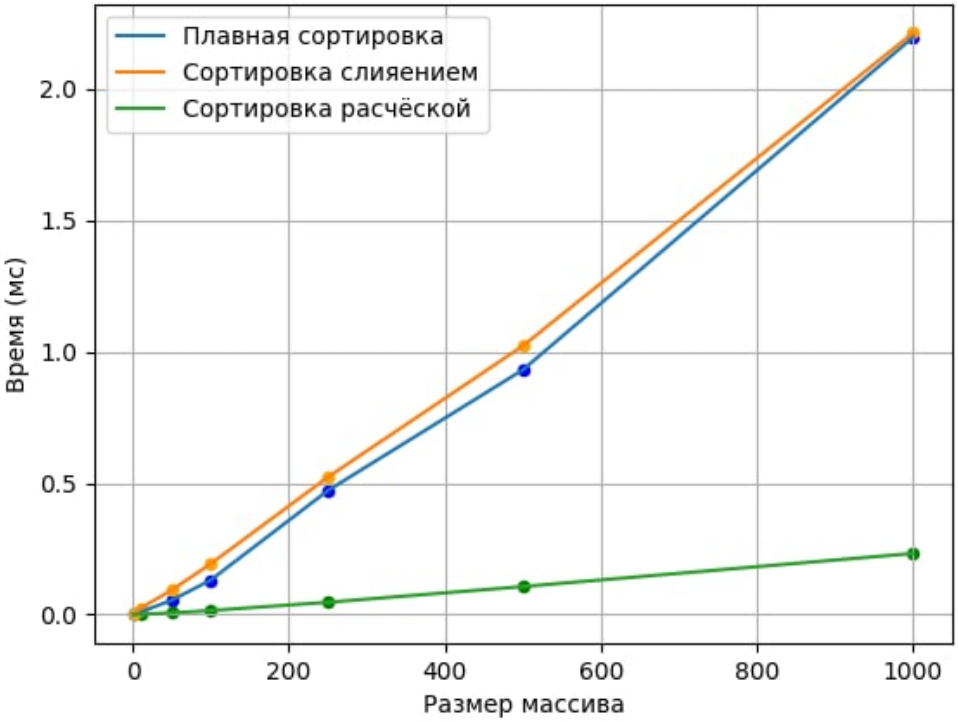

Рисунок 4.2 — Зависимость времени работы алгоритмов от размера массива в худшем случае

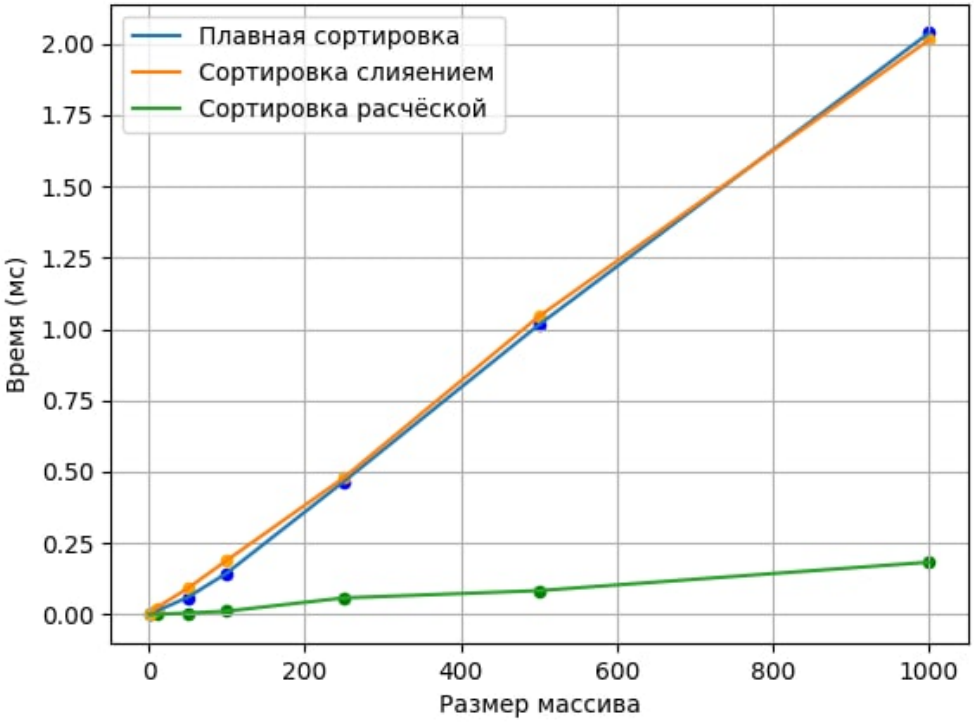

Рисунок 4.3 — Зависимость времени работы алгоритмов от размера массива в произвольном случае

4.2 Вывод

В ходе экспериментов по замеру времени работы было установлено, что в лучшем случае, когда массив отсортирован, сортировка слиянием оказалась самой медленной.

В худшем случае сортировка слиянием и плавная сортировка работают одинаково хуже сортировки расчёской.

В произвольном случае время работы сортировок слиянием и плавной сортировки сопоставимо. Сортировка расчёской работает одинаково хорошо во всех случаях.

Заключение

В ходе работы были изучены и реализованы сортировки слиянием, расчёской и плавная сортировка. А также дана оценка их трудоемкости и замерено процессорное время работы алгоритмов.

В ходе экспериментов по замеру времени работы было установлено, что в лучшем случае, когда массив отсортирован, сортировка слиянием дает наихудший результат. Алгоритм сортировки расчёской оказался быстрее всех остальных во всех случаях.

В худшем случае самой медленной сортировкой является сортировка слиянием и плавная сортировка, а сортировка расчёской является самой быстрой.

В произвольном случае время работы сортировоки слиянием и плавной сортировки. Самой медленной является сортировка слиянием.

Список источников

1) Ассемблерные вставки в AVR-GCC. // [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/275937/ дата обращения: 20.09.2022);

2) C/C++: как измерять процессорное время. // [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/282301/ (дата обращения: 20.09.2022);

3) Sorting algorithm. // [Электронный ресурс]. Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=\T2A\CYRS\T2A\cyro\T2A\cyrr\T2A\cyrt\T2A\cyri\T2A\cyrr\T2A\cyro\T2A\cyrv\T2A\cyrk\T2A\cyri (дата обращения: 20.09.2022).