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»

Отчет по лабораторной работе №2 по курсу "Анализ алгоритмов"

Тема Алгоритмы сортировки

Студент Чепиго Д.С.

Группа ИУ7-54Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Содержание

Введение

Одной из важнейших процедур обработки структурированной информации является сортировка [**?**]. Сортировкой называют процесс перегруппировки заданной последовательности (кортежа) объектов в некотором определенном порядке. Определенный порядок (например, упорядочение в алфавитном порядке, по возрастанию или убыванию количественных характеристик, по классам, типам и.т.п.) в последовательности объектов необходимо для удобства работы с этим объектом. В частности, одной из целей сортировки является облегчение последующего поиска элементов в отсортированном множестве.

Любой алгоритм сортировки можно разбить на три основные части:

- сравнение элементов для определения их упорядоченности;
- перестановка элементов;
- сортирующий алгоритм, который осуществляет сравнение и перестановку элементов до тех пор, пока все элементы не будут упорядочены.

Важнейшей характеристикой любого алгоритма сортировки является скорость его работы, которая определяется функциональной зависимостью среднего времени сортировки последовательностей элементов данных, заданной длины, от этой длины. Время сортировки будет пропорционально количеству сравнений и перестановки элементов данных в процессе их сортировки.

Цель лабораторной работы – изучить и исследовать трудоемкость алгоритмов сортировки.

Задачи лабораторной работы:

- изучить и реализовать 3 алгоритма сортировки: слиянием, подсчетом, битонная;
- выбрать инструменты для процессорного времени выполнения реализаций алгоритмов;
- провести анализ затрат работы программы по времени;
- подготовить отчет по лабораторной работе.

1 Аналитическая часть

1.1 Сортировка слиянием

Алгоритм сортировки слиянием в большой степени соответствует парадигме метода разбиения. [1] Сортировка слиянием применяется для структур данных, доступ к элементам которых можно получать только последовательно, например, списки или массивы.

Приведем алгоритм сортировки для массива:

1. сортируемый массив разбивается на две части примерно одинакового размера;

2. каждая из получившихся частей сортируется отдельно тем же самым алгоритмом;

3. два упорядоченных массива половинного размера соединяются в один.

Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы. Эффективно заранее создать временный массив и передать его в качестве аргумента функции. Сортировка является устойчивой.

1.2 Сортировка подсчетом

Сортировка подсчетом применяется для структур данных, доступ к элементам которых только последовательно, например, списки или массивы. Используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих элементов.[2]

Приведем алгоритм сортировки для массива:

1. в сортируемом массиве находится максимальный и минимальный элемент;

2. создается дополнительный массив, размер которого – разница между максимальным и минимальным элементом;

3. считается колличество каждого числа сортируемого массива и результат записывается в дополнительный массив;

4. с помощью дополнительного массива формируется отсортированный массив.

Если в массиве используются только натуральные числа, то минимальный элемент находить не требуется, он будет равен 0. Применение сортировки подсчётом целесообразно когда сортируемые числа имеют диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел меньших 1000. Сортировка является устойчивой.

1.3 Битонная сортировка

Алгоритм применяется для массивов, размер которых степень двойки, так как он рекурсивно делит массив пополам. Из-за этого может понадобиться добавлять фиктивные элементы в сортируемый массив, что не влияет на асимптотику. Алгоритм основан на сортировке битонных последовательностей. Последовательность называется битонической, если она монотонно возрастает, а затем монотонно убывает, или если путем циклического сдвига ее можно привести к такому виду. [1]

Приведем алгоритм сортировки для массива:

1. сортируемый массив преобразуется в битонную последовательность;

2. сортируемый массив разбивается на две части одинакового размера и также преобразовывается в битонную последовательность;

3. два упорядоченных массива половинного размера соединяются в один.

Параллельный алгоритм сортировки применяется для создания сортировочных сетей. Сортировка является устойчивой.

2 Конструкторская часть

2.1 Разработка алгоритма сортировки слиянием

На рисунке 2.1 приведена схема алгоритма сортировки слиянием.

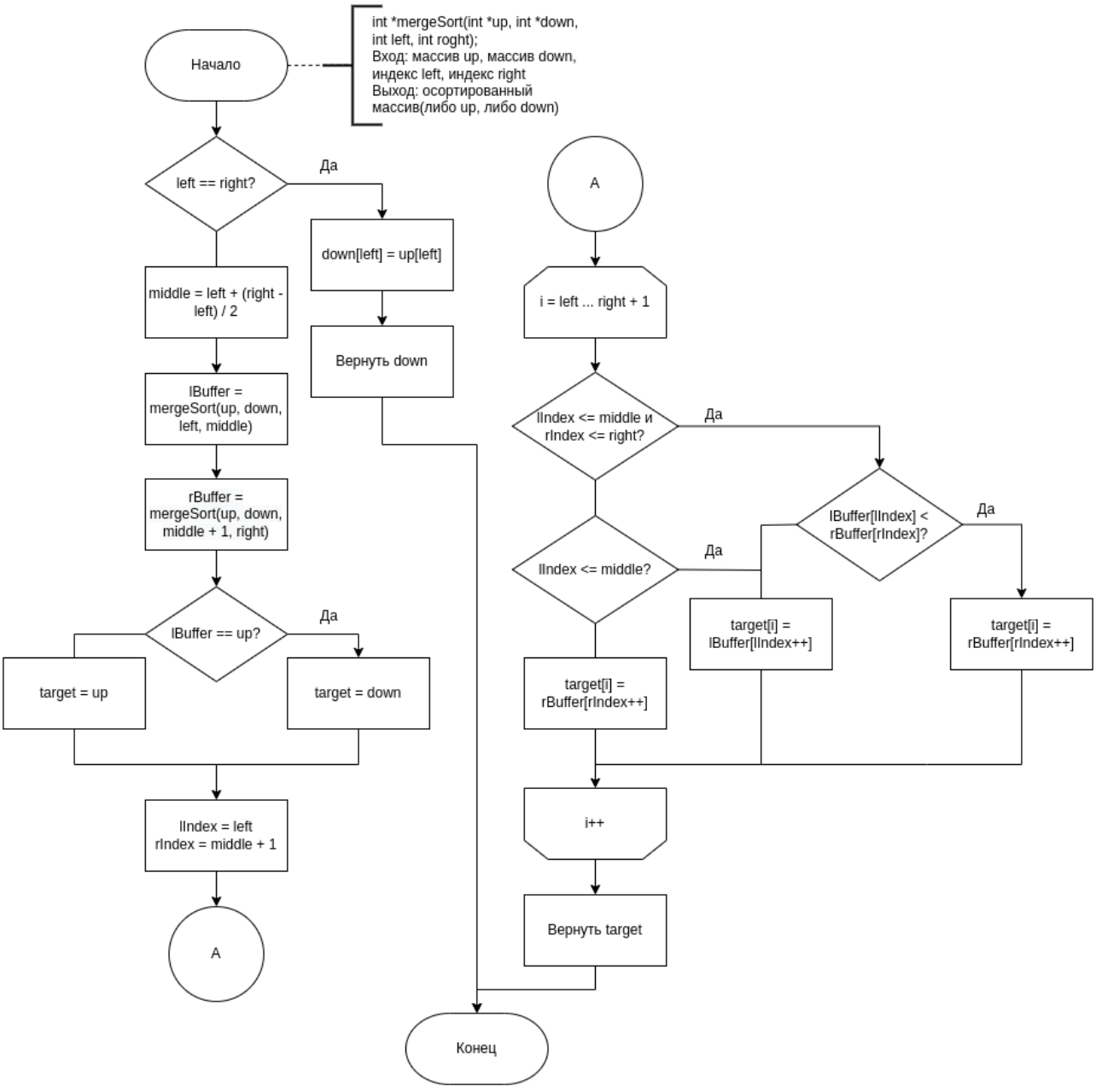

Рисунок 2.1 – Схема алгоритма сортировки слиянием.

На рисунке 2.2 приведена схема алгоритма сортировки подсчетом.

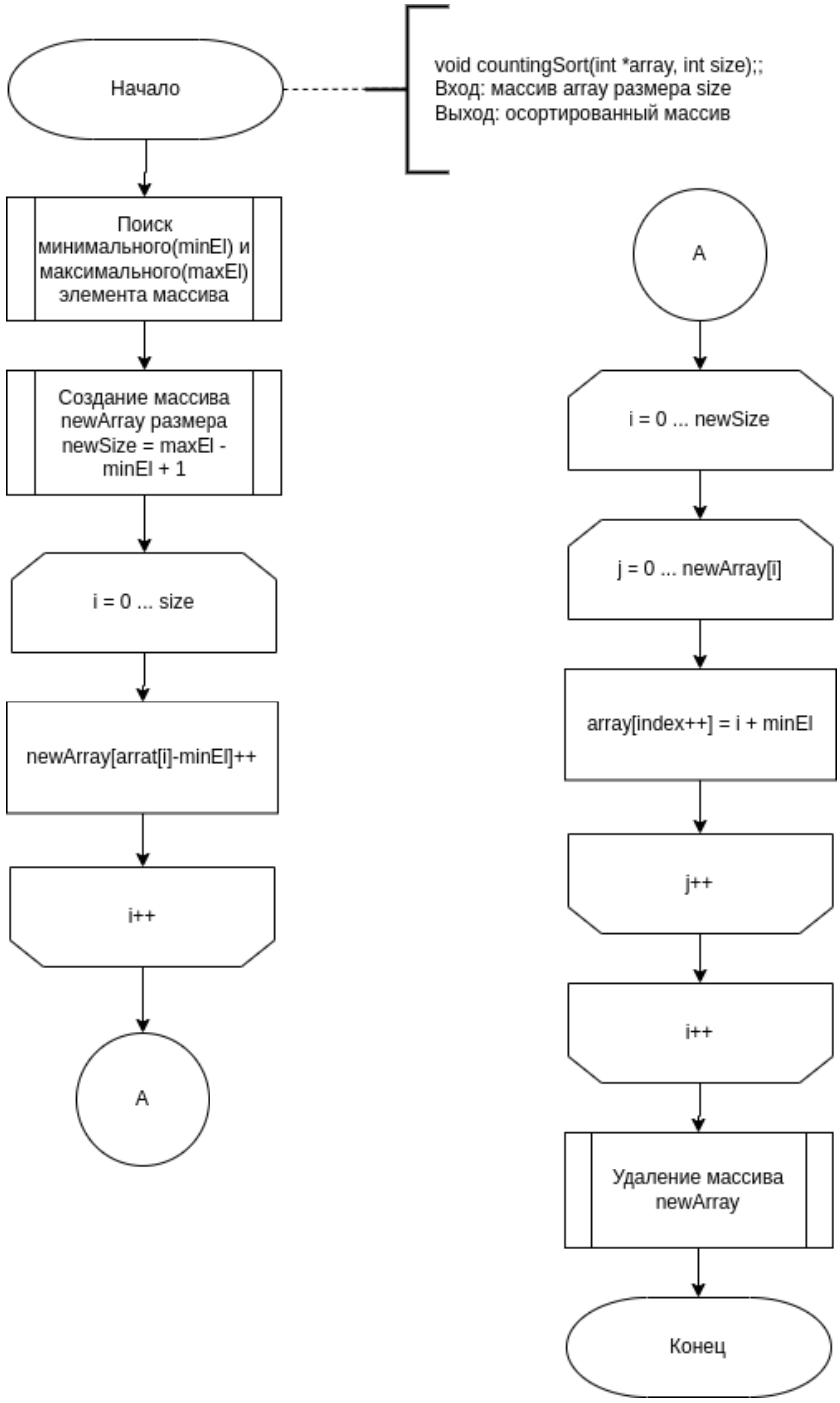

Рисунок 2.2 – Схема алгоритма сортировки подсчетом.

На рисунке 2.3 приведена схема алгоритма битонной сортировки.

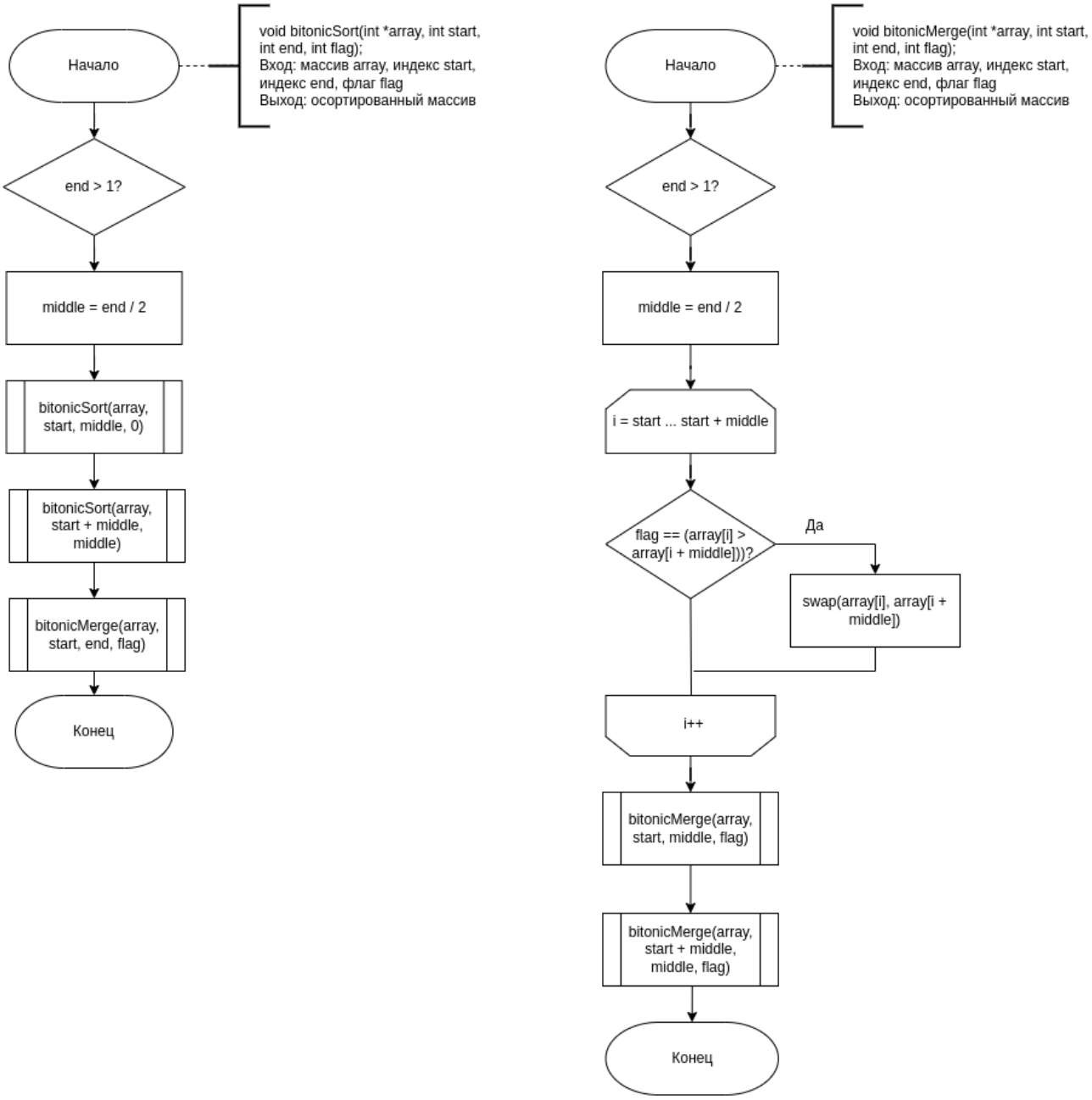

Рисунок 2.3 – Схема алгоритма битонной сортировки.

2.2 Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений.

Операции из списка (2.1) имеют трудоемкость 1:

$$=, +=, -=, +, -, *, /, \%, ==, !=, <, >, <=, >=, [], ++, -- \tag{2.1}$$

Трудоемкость оператора выбора `if условие then A else B` рассчитывается как:

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.2}$$

Трудоемкость цикла рассчитывается как:

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N \cdot (f_{\text{тела}} + f_{\text{инкремента}} + f_{\text{сравнения}}) \tag{2.3}$$

Трудоемкость вызова функции равна 0.

2.3 Трудоемкость алгоритмов

Далее размер массива обозначается как $size$.

2.3.1 Алгоритм сортировки подсчетом

Трудоемкость алгоритма сортировки подсчетом состоит из:

- трудоемкость поиска минимального элемента в массиве:

$$f_{min} = 2 + size \cdot (f_{\text{тела}} + 2) \tag{2.4}$$

- трудоемкость поиска максимального элемента в массиве:

$$f_{max} = 2 + size \cdot (f_{\text{тела}} + 2) \tag{2.5}$$

- трудоёмкость каждой итерации цикла поиска минимального или максимального элемента:

$$f_{\text{тела}} = 4 + \begin{cases} 0, & \text{в лучшем случае,} \\ 2, & \text{в худшем случае.} \end{cases} \tag{2.6}$$

- трудоемкость заполнения временного массива:

$$f_{\text{newArray}} = 2 + newSize \cdot (4 + 2) = 2 + 6 \cdot newSize \tag{2.7}$$

- трудоемкость внешнего цикла заполнения массива:

$$f_{\text{массив}} = 2 + newSize \cdot (f_{\text{внутренний}} + 2) \tag{2.8}$$

- трудоемкость внутреннего цикла сортировки исходного массива:

$$f_{\text{внутренний}} = 3 + size \cdot (3 + 4) = 3 + 7 \cdot newArray[i] \tag{2.9}$$

- суммарная трудоёмкость внешнего и внутреннего циклов:

$$f_{\text{суммарная}} = 2 + newSize \cdot (5 + 7 \cdot newArray[i]) \tag{2.10}$$

Трудоёмкость в лучшем случае (массив отсортирован, все элементы совпадают, диапазон значений не превышает размер массива):

$$f_{\text{лучший}} = 8 + 10 \cdot size + 12 \cdot newSize \approx size + newSize \tag{2.11}$$

Трудоёмкость в худшем случае (массив отсортирован в обратном порядке, минимум и максимум находятся в конце массива, все элементы разные, диапазон значений превышает размер массива):

$$f_{\text{худший}} = 16 + 20 \cdot size + 16 \cdot newSize \approx size + newSize \tag{2.12}$$

Трудоемкость в худшем и лучшем случае совпадает и равна $O(size + newSize)$, где $newSize$ – диапазон значений элементов массива.

2.3.2 Алгоритм битонной сортировки

Трудоемкость алгоритма битонной сортировки состоит из:

- рекурсивное разбиение массива на массивы половинной длины:

$$f_{\text{разбиение}} = log_2(size) \tag{2.13}$$

- трудоемкость соединения двух массивов половинной длины:

$$f_{\text{соединение}} = log_2(size) \cdot f_{\text{построение}} \tag{2.14}$$

- трудоемкость построения битонной последовательности:

$$f_{\text{построение}} = 1 + 3 + (start + size/2) \cdot (3 + f_{\text{перестановка}}) \tag{2.15}$$

- трудоемкость перестановки элементов при построении битонной последовательности

$$f_{\text{перестановка}} = 5 + \begin{cases} 0, & \text{в лучшем случае,} \\ 9, & \text{в худшем случае.} \end{cases} \tag{2.16}$$

Трудоёмкость в лучшем случае (массив отсортирован):

$$\begin{aligned} f_{\text{лучший}} = (log_2(size))^2 \cdot (4 + (3 + 5) \cdot (start + size/2)) \approx \\ size \cdot (log_2(size))^2 = O(size \cdot (log_2(size))^2) \end{aligned} \tag{2.17}$$

Трудоёмкость в худшем случае (массив отсортирован в обратном порядке):

$$\begin{aligned} f_{\text{худший}} = (log_2(size))^2 \cdot (4 + (3 + 5 + 9) \cdot (start + size/2)) \approx \\ size \cdot (log_2(size))^2 = O(size \cdot (log_2(size))^2) \end{aligned} \tag{2.18}$$

Трудоемкость в худшем и лучшем случае совпадает и равна $O(size \cdot (log_2(size))^2)$. Заполнение массива фиктивными элементами до размерности степени двойки не влияет на асимптотику.

2.3.3 Алгоритм сортировки слиянием

Трудоемкость алгоритма сортировки слиянием может быть найдена аналогичным образом, как и в битонной сортировке. Она состоит из:

- рекурсивное разбиение массива на массивы половинной длины:

$$f_{\text{разбиение}} = log_2(size) \tag{2.19}$$

- трудоемкость соединения двух массивов половинной длины:

$$f_{\text{соединение}} = 2 + size \cdot (3 + \begin{cases} 4, & \text{в лучшем случае,} \\ 10, & \text{в худшем случае.} \end{cases}) \tag{2.20}$$

Трудоёмкость в лучшем случае (массив отсортирован):

$$f_{\text{лучший}} = log_2(size) \cdot (2 + size \cdot 7) \approx size \cdot log_2(size) = O(size \cdot log_2(size)) \tag{2.21}$$

Трудоёмкость в худшем случае (массив отсортирован в обратном порядке):

$$f_{\text{худший}} = log_2(size) \cdot (2 + size \cdot 13) \approx size \cdot log_2(size) = O(size \cdot log_2(size)) \tag{2.22}$$

Трудоемкость в худшем и лучшем случае совпадает и равна $O(size \cdot log_2(size))$.

3 Технологическая часть

3.1 Требования к ПО

К программе предъявляется ряд требований:

- входными данными являются размер массива и непосредственно элементы массива;
- на выходе – отсортированный массив.

3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран C++ – компилируемый, статически типизированный язык программирования общего назначения [3].

Данный выбор обусловлен поддержкой языком парадигмы объектно – ориентированного программирования и наличием методов для замера процессорного времени.

Время работы реализованных алгоритмов было замерено с помощью библиотеки chrono[4].

3.3 Реализация алгоритмов

В листинге 3.1 приведена реализация алгоритма сортировки слиянием.

Листинг 3.1 – Реализация алгоритма сортировки слиянием

```
1  int *mergeSort(int *up, int *down, int left, int right)
2  {
3      if (left == right)
4      {
5          down[left] = up[left];
6          return down;
7      }
8
9      int middle = left + (right - left) / 2;
10     int *lBuffer = mergeSort(up, down, left, middle);
```

```
    int *rBuffer = mergeSort(up, down, middle + 1, right);
    int *target;

    if (lBuffer == up)
        target = down;
    else
        target = up;

    int lIndex = left, rIndex = middle + 1;

    for (int i = left; i <= right; i++)
    {
        if (lIndex <= middle && rIndex <= right)
        {
            if (lBuffer[lIndex] < rBuffer[rIndex])
                target[i] = lBuffer[lIndex++];
            else
                target[i] = rBuffer[rIndex++];
        }
        else if (lIndex <= middle)
            target[i] = lBuffer[lIndex++];
        else
            target[i] = rBuffer[rIndex++];
    }

    return target;
}
```

В листинге 3.2 приведена реализация алгоритма сортировки подсчетом.

Листинг 3.2 – Реализация алгоритма сортировки подсчетом

```
void countingSort(int *array, int size)
{
    int minEl = array[0];
    int maxEl = array[0];

    for (int i = 0; i < size; i++)
        if (array[i] > maxEl)
            maxEl = array[i];

    for (int i = 0; i < size; i++)
        if (array[i] < minEl)
            minEl = array[i];

    int newSize = maxEl - minEl + 1;
    int *newArray = new int[newSize]();

    for (int i = 0; i < size; i++)
```

```
18          newArray[array[i] - minEl]++;
19
20      int index = 0;
21
22      for (int i = 0; i < newSize; i++)
23          for (int j = 0; j < newArray[i]; j++)
24              array[index++] = i + minEl;
25
26      delete[] newArray;
27  }
```

В листинге 3.3 приведена реализация алгоритма битонной сортировки.

Листинг 3.3 – Реализация алгоритма битонной сортировки

```
1  void bitonicMerge(int *array, int start, int end, int flag)
2  {
3      if (end > 1)
4      {
5          int middle = end / 2;
6
7          for (int i = start; i < start + middle; i++)
8              if (flag == (array[i] > array[i + middle]))
9                  swap(array[i], array[i + middle]);
10
11         bitonicMerge(array, start, middle, flag);
12         bitonicMerge(array, start + middle, middle, flag);
13     }
14 }
15
16 void bitonicSort(int *array, int start, int end, int flag)
17 {
18     if (end > 1)
19     {
20         int middle = end / 2;
21
22         bitonicSort(array, start, middle, 0);
23         bitonicSort(array, start + middle, middle, 1);
24
25         bitonicMerge(array, start, end, flag);
26     }
27 }
```

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Ubuntu 22.04.1 LTS Linux x86_64 [5];
- память: 8 ГБ;
- процессор: Intel® Core™ i3-7130U.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Пример работы программы

На рисунке 4.1 представлен пример работы программы. Вводится размерность массива и элементы массива. Далее выводится отсортированный массив и время выполнения каждой сортировки в микросекундах. Все сортировки верно отсортировали исходный массив. В данном примере:

- присутствовали отрицательные элементы (дополнительный массив в сортировке подсчетом был размерности 16, что больше размера самого массива);
- размер массива не равен степени числа два (для битонной сортировки массив дополнялся фиктивными элементами);
- максимальный и минимальный элементы находились в конце (худший случай поиска минимального и максимального элементов в сортировке подсчетом).

```
dashori@fossa ~/P/A/b/l/code (main)> ./a.out
Введите размерность массива: 9
Введите массив: -1 8 -2 3 7 -4 6 -5 9

Сортировка слиянием:
Время: 4 (микросекунды)
-5 -4 -2 -1 3 6 7 8 9

Сортировка подсчетом:
Время: 16 (микросекунды)
-5 -4 -2 -1 3 6 7 8 9

Битонная сортировка:
Время: 18 (микросекунды)
-5 -4 -2 -1 3 6 7 8 9
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения реализованных алгоритмов

Замеры времени работы реализованных алгоритмов для массива каждой размерности проводились 2000 раз. Все три алгоритма сортировки имеют одинаковую трудоемкость в лучшем, худшем и среднем случаях, поэтому каждый раз элементы массива генерировались случайно – числа из диапазона (-INT_MAX, INT_MAX) [6]. Данный диапазон исключает превосходство сортировки подсчетом из-за входных данных, так как дополнительный массив явно содержал больше элементов, чем исходный. В качестве результата, представленного в таблице 4.1, взято среднее время на каждой длине массива.

Таблица 4.1 – Результаты замеров времени реализованных сортировок в микросекундах

Длина массива	Слиянием	Подсчетом	Битонная
10	1	1	2
30	2	1	8
64	2	1	8
128	6	1	23
200	10	2	56
254	12	3	57
255	12	3	57
256	12	3	55
257	13	3	131
258	13	3	131
300	16	3	136
400	22	4	134
500	27	5	136
512	27	5	133
1024	59	11	316
2048	128	22	736
4096	272	45	1696
8192	573	90	3856
16384	1197	181	8720
32768	2502	363	19570
65536	5229	726	43674

На рисунке 4.2 представлена зависимость времени работы реализованных алгоритмов сортировки от длины массива. На рисунке 4.3 представлена зависимость времени работы реализованных алгоритмов сортировки от длины массива, где длина массива – степень числа два.

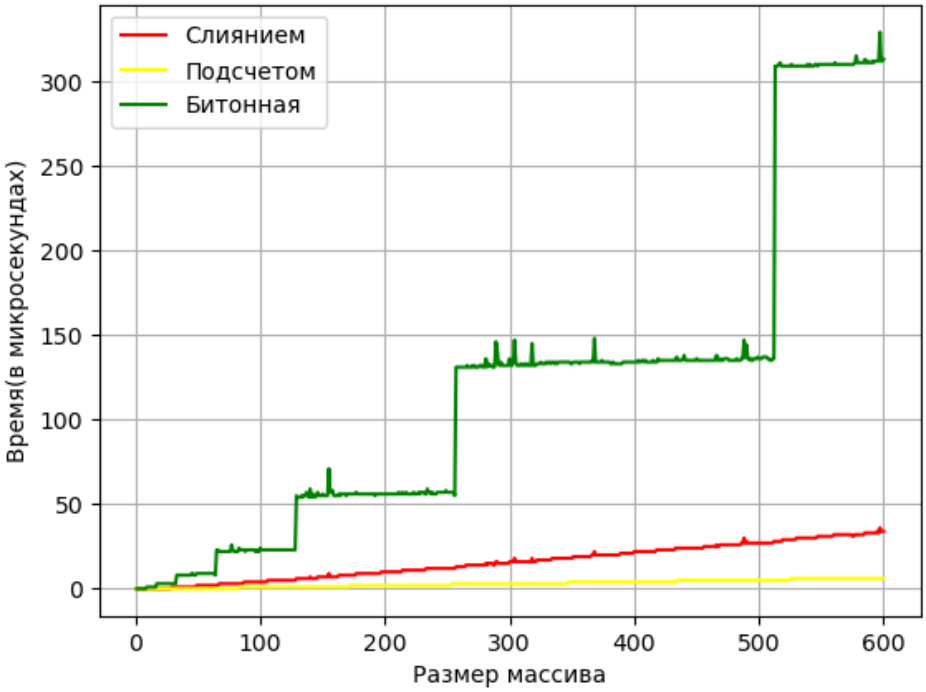

Рисунок 4.2 – Зависимость времени работы реализованных алгоритмов сортировки от длины массива.

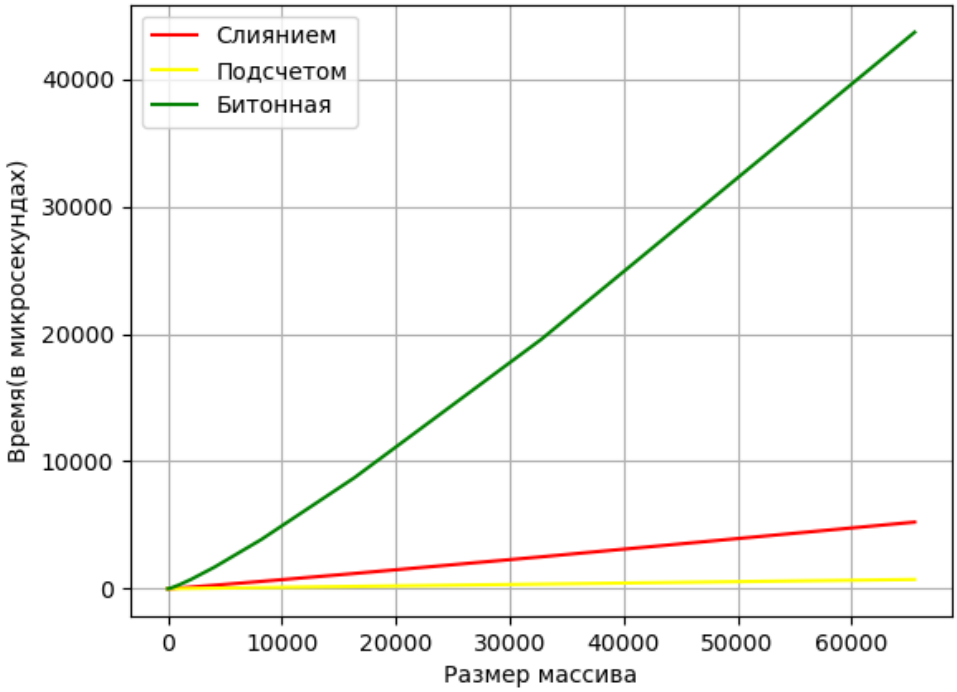

Рисунок 4.3 – Зависимость времени работы реализованных алгоритмов сортировки от длины массива, где длина – степень числа два.

Из рисунка 4.2 видно особенность битонной сортировки, которая сортирует только массивы, длина которых является степенью числа два. Все массивы, имеющую иную длину, дополняются фиктивными элементами до ближайшей степени числа два, что видно на рисунке 4.2.

Самая быстрая сортировка из исследуемых – сортировка подсчетом. На размерах массива до 64 она превосходит в 8 раз битонная сортировка и в 3 раза сортировка слиянием. На размерах от 100 до 600 она работает быстрее в 30-50 чем битонная и в 6 раз быстрее чем сортировка слиянием. На самом максимальном тестируемом размере – 65536 сортировка подсчетом быстрее битонной сортировки в 60 раз, а сортировки слиянием в 7 раз.

Быстрая скорость сортировки подсчетом объясняется тем фактом, что она проигрывает по памяти двум другим алгоритмам сортировки, так как хранит дополнительный массив длина которого – диапазон значений. Так как значения элементов массива генерировались произвольно в диапазоне (-INT_MAX, INT_MAX) [6], исключается вариант, что сортировка подсчетом работает быстрее из-за тестовых данных. Также алгоритм битонной сортировки был реализован не параллельно, что ухудшает его трудоемкость.

Заключение

В ходе выполнения лабораторной работы поставленная цель была достигнута: были изучены и исследованы трудоемкости алгоритмов сортировки.

В ходе выполнения лабораторной работы были решены все задачи:

- изучены и реализованы 3 алгоритма сортировки: слиянием, подсчетом и битонная;
- проведен анализ затрат работы реализованных алгоритмов по памяти;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчет по лабораторной работе.

Результат замерных экспериментов реализованных алгоритмов показал, что самым быстрым алгоритмом сортировки является алгоритм сортировки подсчетом. При размерах массива до 64 элементов он превосходит алгоритм сортировки слияния и алгоритм битонной сортировки в 3 и 8 раз соответственно. На размерах массива больше 100 превосходство увеличивается до 6 и 40 раз.

Список использованных источников

[1] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. / М.: Издательский дом «Вильямс», 2011.— 1296 с.

[2] Кнут Д. Сортировка и поиск. Вильямс, 2000. Т. 3 из Искусство программирования. с. 834.

[3] Язык программирования C++ [Электронный ресурс]. Режим доступа: `https://isocpp.org/`(дата обращения: 20.10.2022)

[4] Стандарт языка C++ [Электронный ресурс]. Режим доступа: `https://isocpp.org/files/papers/N4860.pdf`(дата обращения: 20.10.2022)

[5] Операционная система Ubuntu 22.04 LTS [Электронный ресурс]. Режим доступа: `https://releases.ubuntu.com/jammy/`(дата обращения: 20.10.2022)

[6] Стандарт языка Си [Электронный ресурс]. Режим доступа: `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`(дата обращения: 20.10.2022)