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 № 3

по курсу «Анализ Алгоритмов»

на тему: «Трудоемкость сортировок»

Студент ИУ7-53Б (Группа) _____ (Подпись, дата) Н. Д. Лысцев (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сортировка – процесс перестановки предметов, при котором они располагаются в порядке возрастания или убывания [1].

Целью данной лабораторной работы является исследование трех алгоритмов сортировки: блочной сортировки, сортировки слиянием и поразрядной сортировки.

Для достижения поставленной цели необходимо решить следующие задачи:

1) описать три алгоритма сортировки: блочной, слиянием и поразрядной;
2) создать программное обеспечение, реализующее следующие алгоритмы:
   - алгоритм блочной сортировки;
   - алгоритм сортировки слиянием;
   - алгоритм поразрядной сортировки;
3) провести анализ эффективности реализаций алгоритмов по памяти и по времени;
4) провести оценку трудоемкости алгоритмов сортировки;
5) обосновать полученные результаты в отчете к выполненной лабораторной работе.

1 Аналитический раздел

В данном разделе будут рассмотрены алгоритм блочной сортировки, сортировки слиянием и поразрядной сортировки.

1.1 Алгоритм блочной сортировки

В алгоритме блочной сортировки элементы распределяются между конечным числом отдельных блоков так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем. Каждый блок затем сортируется отдельно: либо рекурсивно тем же методом, либо другим алгоритмом. Затем элементы помещаются обратно в массив.

1.2 Алгоритм сортировки слиянием

Алгоритм действий в сортировке слиянием:

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

2) каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;

3) два упорядоченных массива половинного размера соединяются в один.

1.3 Алгоритм поразрядной сортировки

Поразрядная сортировка [2] – это алгоритм сортировки, где массив несколько раз перебирается и элементы перераспределяются в зависимости от того, какая цифра находится в определённом разряде. После обработки всех разрядов массив оказывается упорядоченным.

Вывод

В данном разделе были рассмотрены алгоритм блочной сортировки, сортировки слиянием и поразрядной сортировки.

2 Конструкторский раздел

2.1 Разработка алгоритмов

2.1.1 Алгоритм блочной сортировки

На рисунке 2.1 представлена схема алгоритма блочной сортировки.

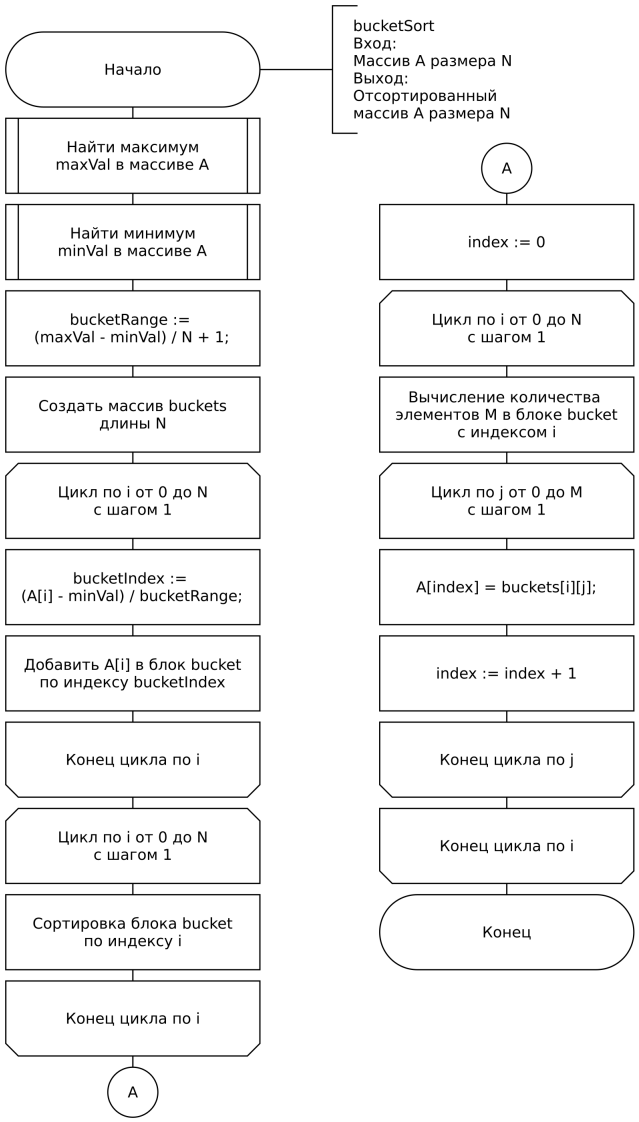

Рисунок 2.1 – Схема алгоритма блочной сортировки

2.1.2 Алгоритм сортировки слиянием

На рисунке 2.2 представлена схема алгоритма сортировки слиянием.

На рисунке 2.3 представлена схема алгоритма функции слияния двух отсортированных подмассивов.

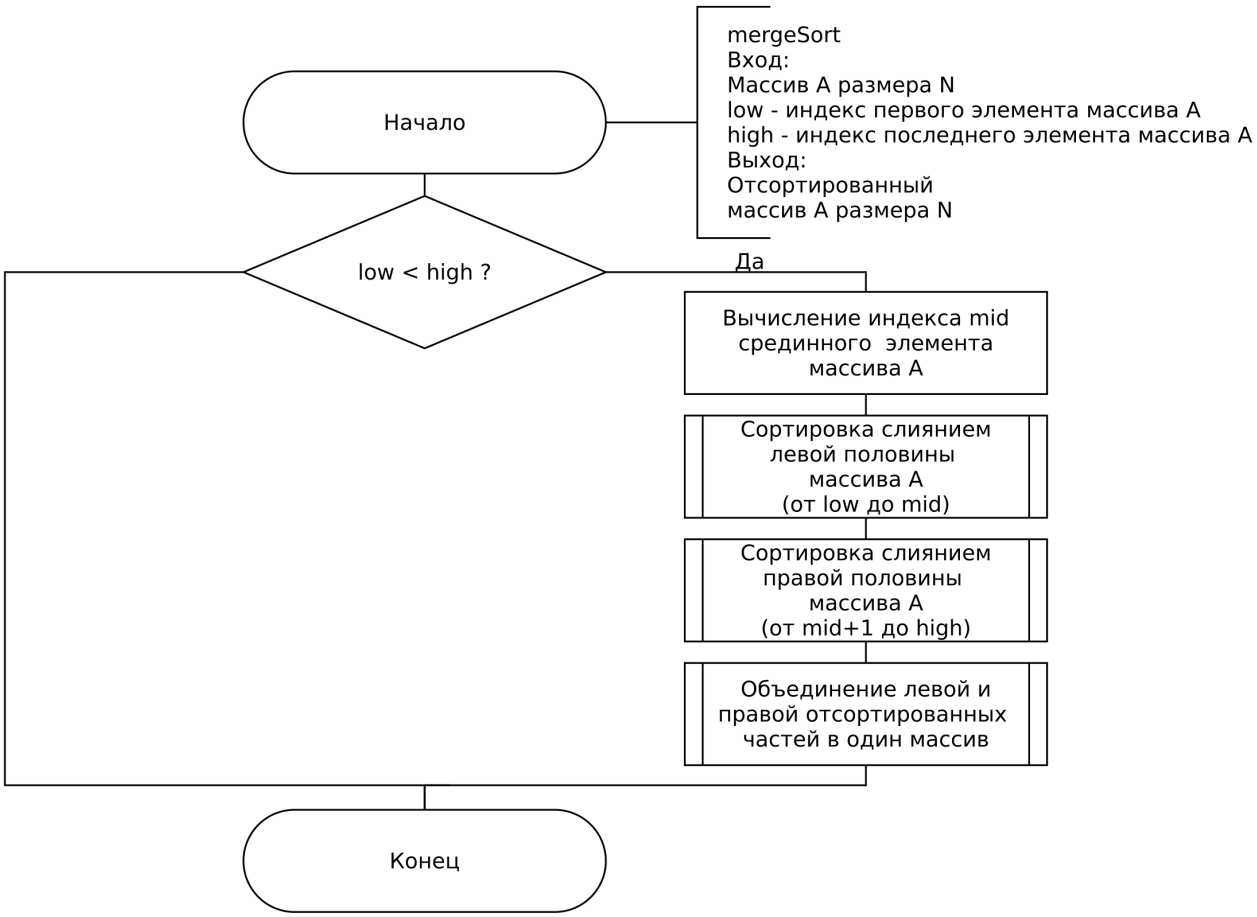

Рисунок 2.2 – Схема алгоритма сортировки слиянием

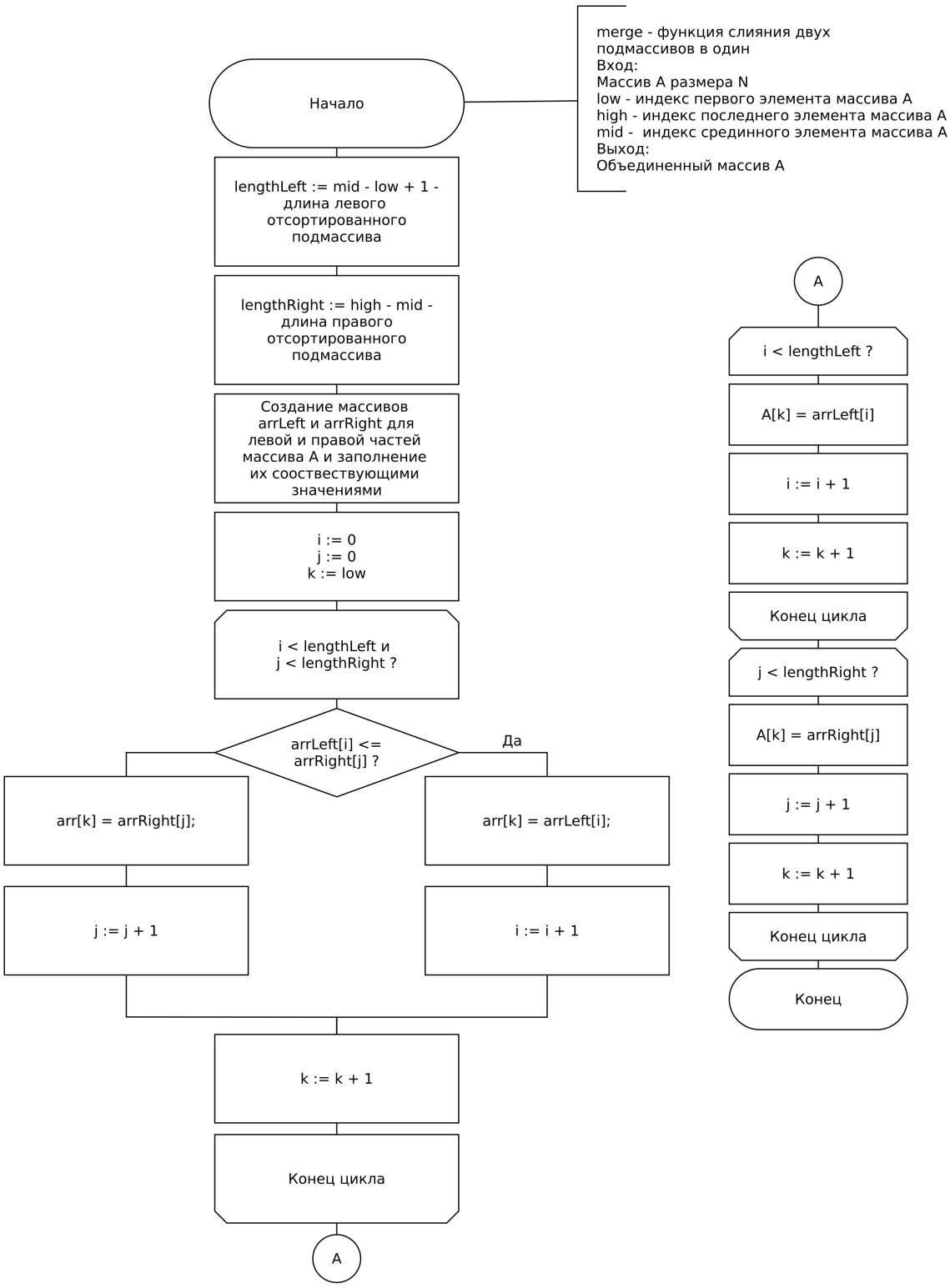

Рисунок 2.3 – Схема алгоритма функции слияния двух отсортированных подмассивов

2.1.3 Алгоритм поразрядной сортировки

На рисунке 2.4 представлена схема алгоритма сортировки слиянием.

На рисунке 2.5 представлена схема алгоритма функции сортировки по определенному разряду.

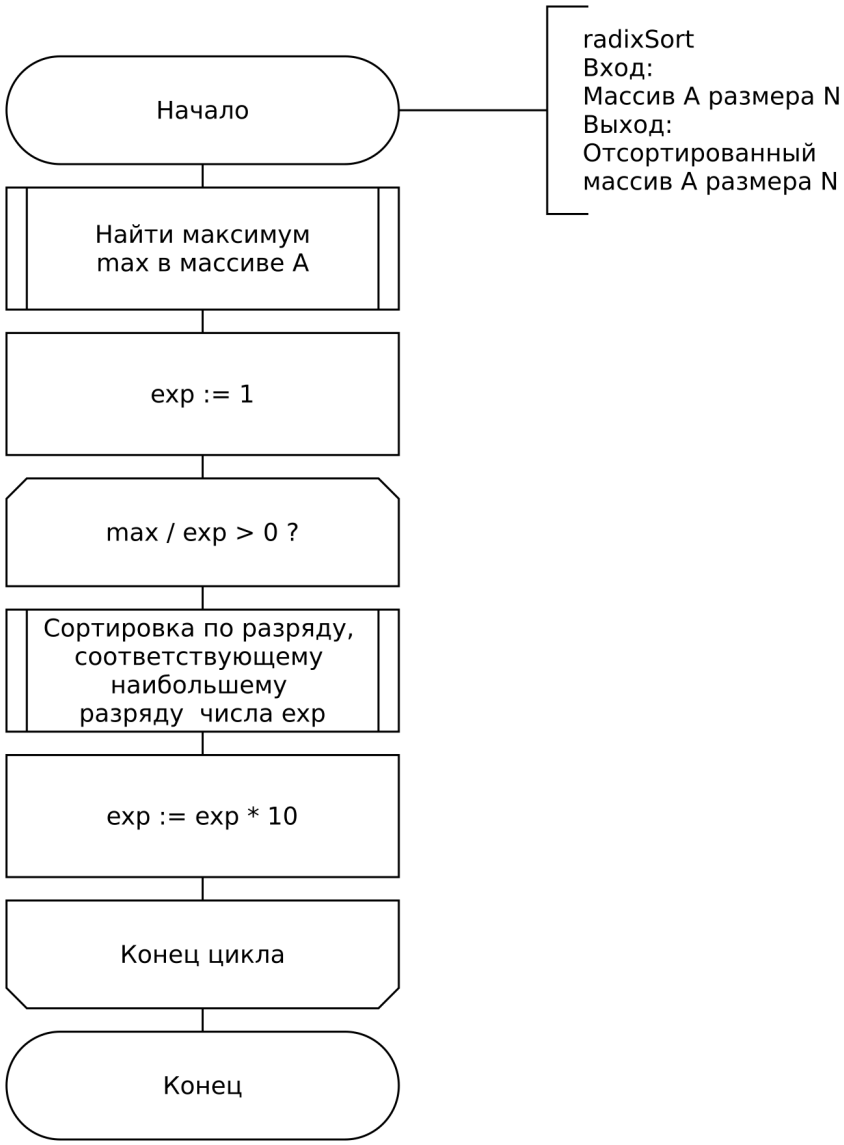

Рисунок 2.4 – Схема алгоритма поразрядной сортировки

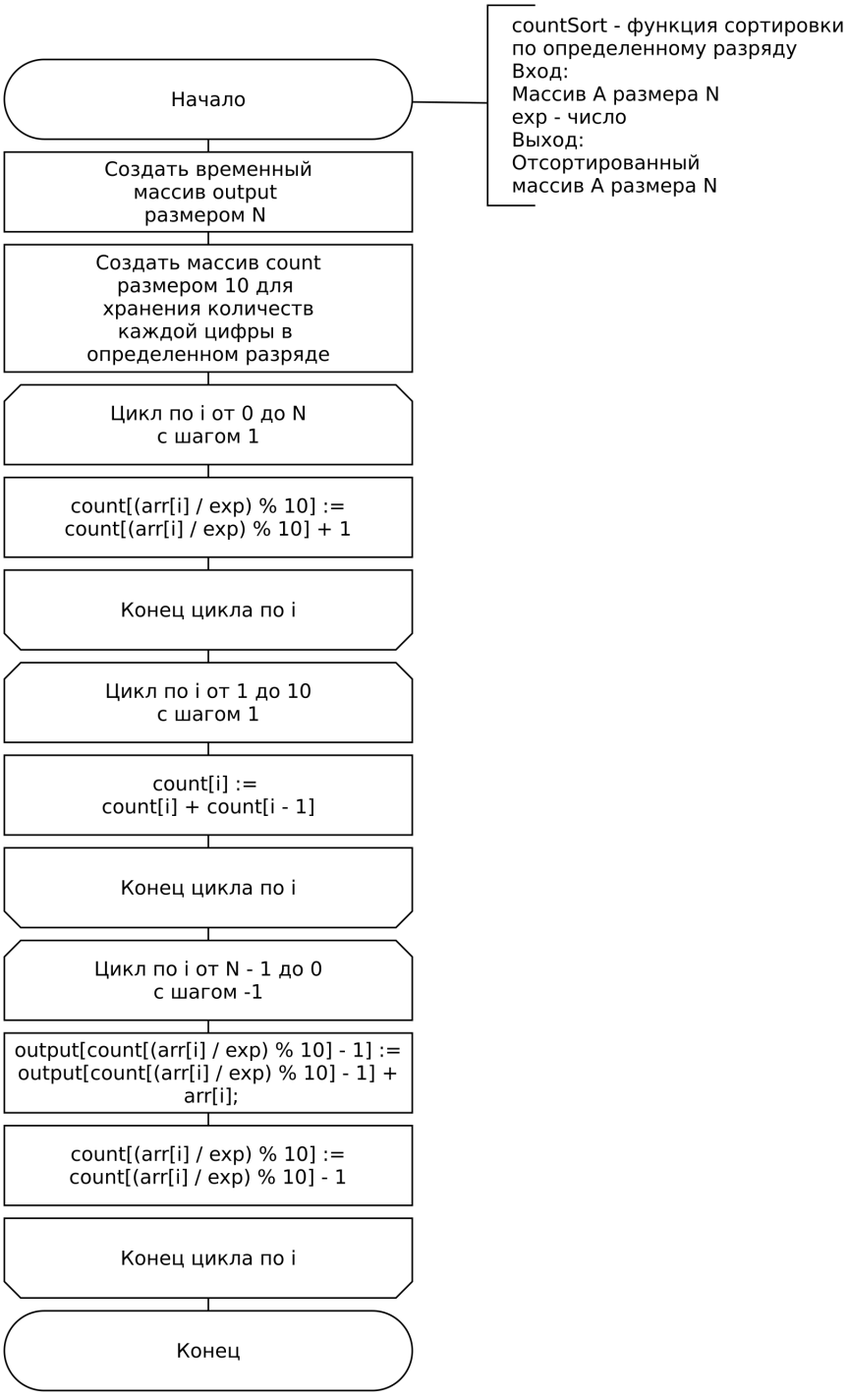

Рисунок 2.5 – Схема алгоритма функции сортировки по определенному разряду

2.2 Оценка трудоемкости алгоритмов

2.2.1 Модель вычислений для проведения оценки трудоемкости алгоритмов

Была введена модель вычислений для определения трудоемкости каждого отдельного взятого алгоритма сортировки.

1) Трудоемкость базовых операций имеет:

   — равную 1:

$$+, -, =, + =, - =, ==, ! =, <, >, <=, >=, [], ++, --, \\ \&\&, >>, <<, ||, \&, | \tag{2.1}$$

   — равную 2:

$$*, /, \%, * =, / =, \% = \tag{2.2}$$

2) Трудоемкость условного оператора:

$$f_{if} = f_{\text{условия}} + \begin{cases} min(f_1, f_2), & \text{лучший случай} \\ max(f_1, f_2), & \text{худший случай} \end{cases} \tag{2.3}$$

3) Трудоемкость цикла:

$$f_{for} = f_{\text{инициализация}} + f_{\text{сравнения}} + M_{\text{итераций}} \cdot (f_{\text{тело}} + \\ + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{2.4}$$

4) Трудоемкость передачи параметра в функции и возврат из функции равны 0.

2.2.2 Трудоемкость алгоритма блочной сортировки

Обозначим количество блоков за K. Алгоритм состоит из четырех последовательно идущих циклов:

1) Поиск минимума и максимума среди всех элементов массива.

2) Распределение элементов массива по соответствующим корзинам.

3) Сортировка элементов каждой корзины другим алгоритмом сортировки.

4) Соединение всех корзин воедино.

Для сортировки корзин на шаге 3) была использована функция $std :: sort$ из заголовочного файла «$algorithm$» библиотеки языка $C++$. Сложность данного алгоритма сортировки $O(N \cdot log(N))$.

Для поиска максимального и минимального элемента в массиве на шаге 1) были использованы функции $std :: max\_element$ и $std :: min\_element$ из заголовочного файла «$algorithm$» библиотеки языка $C++$. Сложность данных алгоритмов $O(N)$.

Трудоемкость инициализации пяти переменных:

$$f_1 = 5 \tag{2.5}$$

Цикл распределения элементов массива по соответствующим корзинам имеет следующую трудоемкость:

$$f_2 = 1 + 1 + N \cdot (8 + 1 + 1) = 2 + 10 \cdot N = O(N) \tag{2.6}$$

Цикл сортировки каждой корзины алгоритмом $std :: sort$ в лучшем случае (элементы распределены по блокам равномерно, асимптотика их просмотра $O(K)$, входной массив расположен так, что внутренняя сортировка работает за лучшее время – $O(N)$) имеет асимптотику:

$$f_3 = O(N + K) \tag{2.7}$$

В худшем случае (элементы не имеют математической разницы между собой и внутренняя сортировка работает за худшее время – $O(N^2)$) асимптотика данного цикла:

$$f_3 = O(N^2) \tag{2.8}$$

Так как элементы массива равномерно распределены по K корзинам, то цикл соединения всех корзин воедино имеет следующую трудоемкость:

$$\begin{gathered} f_4 = 1 + 1 + N \cdot (1 + 3 + \frac{N}{K} \cdot (5 + 1 + 3) + 1 + 1) = \\ 9 \cdot \frac{N^2}{K} + 6 \cdot N + 2 \end{gathered} \tag{2.9}$$

Итоговая трудоемкость f_{bucket} равна:

В лучшем случае:

$$\begin{gathered} f_{bucket} = f_1 + f_2 + f_3 + f_4 = 5 + 2 + 10 \cdot N + O(N + K) + \\ 9 \cdot \frac{N^2}{K} + 6 \cdot N + 2 = \\ 9 \cdot \frac{N^2}{K} + 16 \cdot N + 7 + O(N + K) = O(N + K) \end{gathered} \tag{2.10}$$

В худшем случае:

$$\begin{gathered} f_{bucket} = f_1 + f_2 + f_3 + f_4 = 5 + 2 + 10 \cdot N + O(N^2) + \\ 9 \cdot \frac{N^2}{K} + 6 \cdot N + 2 = \\ 9 \cdot \frac{N^2}{K} + 16 \cdot N + 7 + O(N^2) = O(N^2) \end{gathered} \tag{2.11}$$

2.2.3 Трудоемкость алгоритма сортировки слиянием

Пусть

— REC – трудоемкость рекурсивного алгоритма;

— DIR – трудоемкость прямого решения;

— DIV – трудоемкость разбиения ввода (N) на несколько частей;

— COM – трудоемкость объединения решений.

Тогда трудоемкость рекурсивного алгоритма считается по следующей формуле:

$$REC(N) = \begin{cases} DIR(N), & N \leq N_0 \\ DIV(N) + \sum_{i=1}^{n} REC(F[i]) + COM(N), & N > N_0 \end{cases} \tag{2.12}$$

где N – число входных элементов, N_0 – наибольшее число, определяющее тривиальный случай (прямое решение), n – число рекурсивных вызовов для данного N, $F[i]$ – число входных элементов для данного i.

Трудоемкость алгоритма сортировки слиянием определяется следующим образом:

1) Трудоемкость разбиения ввода (N) на части. Каждый следующий вызов берется размерность массива в 2 раза меньше предыдущей путем вычисления индекса срединного элемента массива.

$$DIV(N) = 1 + 2 + 1 = 4 \tag{2.13}$$

2) Трудоемкость сортировки левого и правого подмассива (обозначим ее буквой $G = G(N)$):

$$G(N) = 2 \cdot REC(\frac{N}{2}) \tag{2.14}$$

Число разбиений K массива размером N на подмассивы размера в два раза меньше в алгоритме сортировки слиянием определяется следующей формулой:

$$K = \log_2(N) \tag{2.15}$$

Поскольку выполняется сортировка массива размером N, то

$$REC(\frac{N}{2}) = \frac{N}{2} \cdot \log_2(\frac{N}{2}) = \frac{1}{2} \cdot N \cdot \log_2(N) - \frac{1}{2} \cdot N \tag{2.16}$$

Таким образом, трудоемкость сортировки левого и правого подмассива определяется следующей формулой:

$$G(N) = 2 \cdot (\frac{1}{2} \cdot N \cdot \log_2(N) - \frac{1}{2} \cdot N) = N \cdot \log_2(N) - N \tag{2.17}$$

3) Трудоемкость объединения решений, а именно слияние двух отсортированных подмассивов

$$\begin{gathered} COM(N) = 2 + 1 + \frac{N}{2} \cdot (4 + 1 + 1) + \frac{N}{2} \cdot (4 + 1 + 1) + \\ 3 + 1 + \frac{N}{2} \cdot (1 + 4 + 1 + 1) = \frac{19}{2} \cdot N + 7 \end{gathered} \tag{2.18}$$

Таким образом, трудоемкость алгоритма сортировки слиянием 2.12 определяется так:

$$\begin{gathered} f_{merge} = DIV(N) + G(N) + COM(N) = \\ 4 + N \cdot \log_2(N) - N + \frac{19}{2} \cdot N + 7 = \\ N \cdot \log_2(N) + \frac{17}{2} \cdot N + 11 = O(N \cdot \log_2(N)) \end{gathered} \tag{2.19}$$

2.2.4 Трудоемкость алгоритма поразрядной сортировки

Трудоемкость алгоритма поразрядной сортировки состоит из:

– цикла по всем разрядам наибольшего числа в массиве;

– сортировки массива по каждому из разрядов.

Асимптотика трудоемкости поиска наибольшего элемента массива равна:

$$f_1 = O(N) \tag{2.20}$$

Пусть число разрядов наибольшего числа равно K. Тогда трудоемкость цикла сортировки по всем разрядам наибольшего числа равна:

$$f_2 = 1 + 2 + K \cdot (f_{count} + 1 + 2) \tag{2.21}$$

где f_{count} – трудоемкость сортировки по одному разряду.

Трудоемкость сортировки по одному разряду равна:

$$\begin{gathered} f_{count} = 2 + 9 \cdot N + 47 + 2 + 18 \cdot N + 2 + 5 \cdot N = \\ 32 \cdot N + 53 \end{gathered} \tag{2.22}$$

Итоговая трудоемкость поразрядной сортировки в лучшем и худшем случае равна:

$$\begin{gathered} f_{radix} = 3 + K \cdot (32 \cdot N + 53 + 3) = \\ 32 \cdot K \cdot N + 56 \cdot K + 3 = O(K \cdot N) \end{gathered} \tag{2.23}$$

Вывод

В данном разделе были построены схемы алгоритмов блочной сортировки, сортировки слиянием и поразрядной сортировки, а также были выполнены теоретические расчеты трудоемкости этих алгоритмов.

Согласно расчетам трудоемкости, на равномерно распределенных данных самым эффективным алгоритмом сортировки будет алгоритм блочной сортировки со сложностью $O(N + K)$. Для сортировки же произвольных данных из всех трех алгоритмов лучше всего подошел бы алгоритм сортировки слиянием со сложностью $O(N \cdot \log_2(N))$.

3 Технологический раздел

В данном разделе будут перечислены требования к программному обеспечению, средства реализации, листинги кода и функциональные тесты.

3.1 Требования к программному обеспечению

К программе предъявляется ряд требований:

- на вход подаётся вектор элементов;
- все элементы вектора - целые неотрицательные числа (это необходимо для возможности сравнения сортировок между собой);
- на выходе в том же векторе находятся отсортированные по возрастанию элементы исходного.

3.2 Средства реализации

В качестве языка программирования для этой лабораторной работы был выбран $C++$ [3] по следующим причинам:

- в $C++$ есть встроенный модуль $ctime$, предоставляющий необходимый функционал для замеров процессорного времени;
- в стандартной библиотеке $C++$ есть оператор $sizeof$, позволяющий получить размер переданного объекта в байтах. Следовательно, $C++$ предоставляет возможности для проведения точных оценок по используемой памяти.

В качестве функции, которая будет осуществлять замеры процессорного времени, будет использована функция $clock\_gettime$ из встроенного модуля $ctime$ [4].

3.3 Сведения о модулях программы

Программа состоит из шести модулей:

1) `algorithms.cpp` — модуль, хранящий реализации алгоритмов сортировки;

2) `processTime.cpp` — модуль, содержащий функцию для замера процессорного времени;

3) `memoryMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования памяти в реализациях алгоритмов сортировки;

4) `timeMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования времени в реализациях алгоритмов сортировки;

5) `main.cpp` — файл, содержащий точку входа в программу;

6) `task7` — модуль, содержащий набор скриптов для проведения замеров программы по времени и памяти и построения графиков по полученным данным.

3.4 Реализации алгоритмов

В листингах 3.1 – 3.4 представлены реализации трех алгоритмов сортировки: блочной, сортировки слиянием и поразрядной.

Листинг 3.1 – Реализация алгоритма блочной сортировки

```
void bucketSort(vector<int> &arr)
{
    int n = arr.size();
    int minVal = *min_element(arr.begin(), arr.end());
    int maxVal = *max_element(arr.begin(), arr.end());
    int bucketRange = (maxVal - minVal) / n + 1;
    int bucketIndex, i, j, index = 0;

    vector<vector<int>> buckets(n);

    for (i = 0; i < n; i++)
    {
        bucketIndex = (arr[i] - minVal) / bucketRange;
        buckets[bucketIndex].push_back(arr[i]);
    }

    for (i = 0; i < n; i++)
        sort(buckets[i].begin(), buckets[i].end());

    for (i = 0; i < n; i++)
        for (j = 0; j < buckets[i].size(); j++)
            arr[index++] = buckets[i][j];
}
```

Листинг 3.2 – Реализация алгоритма сортировки слиянием (начало)

```
static void _merge(vector<int> &arr, int low, int high, int mid)
{
    int i, j, k, a;
    int lengthLeft = mid - low + 1;
    int lengthRight = high - mid;

    vector<int> arrLeft(lengthLeft), arrRight(lengthRight);

    for (a = 0; a < lengthLeft; a++)
        arrLeft[a] = arr[low + a];

    for (a = 0; a < lengthRight; a++)
        arrRight[a] = arr[mid + 1 + a];

    i = 0;
    j = 0;
    k = low;

    while (i < lengthLeft && j < lengthRight)
    {
        if (arrLeft[i] <= arrRight[j])
        {
            arr[k] = arrLeft[i];
            i++;
        }
        else
        {
            arr[k] = arrRight[j];
            j++;
        }
        k++;
    }
```

Листинг 3.3 – Реализация алгоритма сортировки слиянием (конец)

```
    while (i < lengthLeft)
    {
        arr[k] = arrLeft[i];
        k++;
        i++;
    }

    while (j < lengthRight)
    {
        arr[k] = arrRight[j];
        k++;
        j++;
    }
}

static void _mergeSort(vector<int> &arr, int low, int high)
{
    if (low < high)
    {
        _mergeSort(arr, low, (low + high) / 2);
        _mergeSort(arr, (low + high) / 2 + 1, high);

        _merge(arr, low, high, (low + high) / 2);
    }
}

void mergeSort(vector<int> &arr)
{
    _mergeSort(arr, 0, arr.size() - 1);
}
```

Листинг 3.4 – Реализация алгоритма поразрядной сортировки (конец)

```
static void countSort(vector<int> &arr, int exp)
{
    int n = arr.size();
    int i;

    vector<int> output(n);
    int count[10] = {0};

    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixSort(vector<int> &arr)
{
    int max = *max_element(arr.begin(), arr.end());
    int exp;

    for (exp = 1; max / exp > 0; exp *= 10)
        countSort(arr, exp);
}
```

3.5 Функциональные тесты

В таблице 3.1 приведены тестовые данные, на которых было протестировано разработанное ПО. Все тесты были успешно пройдены.

Таблица 3.1 – Функциональные тесты

Массив	**Блочная**	**Слиянием**	**Поразрядная**
1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6
6 5 4 3 2 1	1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6
41 56 67 10 34 2	2 10 34 41 56 67	2 10 34 41 56 67	2 10 34 41 56 67
54 33 0 55 33 7 14	0 7 14 33 33 54 55	0 7 14 33 33 54 55	0 7 14 33 33 54 55
4 4 4 4 4 4	4 4 4 4 4 4	4 4 4 4 4 4	4 4 4 4 4 4
10	10	10	10
{}	Сообщение об ошибке	Сообщение об ошибке	Сообщение об ошибке

Вывод

В данном разделе были реализованы и протестированы 3 алгоритма сортировки: алгоритм блочной сортировки, алгоритм сортировки слиянием и алгоритм поразрядной сортировки.

4 Исследовательский раздел

В данном разделе будут проведены сравнения реализаций алгоритмов сортировки по времени работы и по затрачиваемой памяти.

4.1 Технические характеристики

Технические характеристики устройства, на котором проводились исследования:

– операционная система: Ubuntu 22.04.3 LTS x86_64 [5];

– оперативная память: 16 Гб;

– процессор: 11th Gen Intel® Core™ i7-1185G7 @ 3.00 ГГц × 8.

4.2 Время выполнения алгоритмов

Время работы алгоритмов измерялось с использованием функции *clock_gettime* из встроенного модуля *ctime*.

Замеры времени для каждого размера 1000 раз.

4.2.1 Массив случайно сгенерированных чисел заданного размера

На вход подавались случайно сгенерированные векторы заданного размера.

Исходя из полученных данных, самыми быстрыми алгоритмами сортировки из всех трех являются алгоритм поразрядной сортировки и блочной сортировки: средняя разница во времени сортировки этих двух алгоритмов составляет порядка 5 мкс, на некоторых значениях быстрее один алгоритм, на других – другой. Это связано с тем, что в этих двух алгоритмах создается всего по одному динамическому подмассиву для хранения данных, в то время как в алгоритме сортировки слиянием при каждом рекурсивном вызове создается два подмассива для хранения левой и правой отсортированных частей при слиянии.

Также важную роль в результатах времени работы блочной сортировки играет алгоритм, с помощью которого сортируется каждый сформированный блок.

Для сортировки блоков была использована функция $std :: sort$ из заголовочного файла «$algorithm$» библиотеки языка $C++$, что существенно уменьшает время работы блочной сортировки.

Данные представлены в таблице 4.1. Их графическое отображение представлено на рисунке 4.1.

Таблица 4.1 – Замер времени для массивов размеров от 100 до 1000 элементов

Линейный размер, штуки	**Время, мкс**		
	Блочная	**Слиянием**	**Поразрядная**
100	20.914	24.570	22.775
200	44.529	55.438	44.095
300	65.092	83.636	69.413
400	90.714	117.511	95.950
500	109.428	145.115	114.910
600	134.144	179.441	134.590
700	162.905	218.916	162.367
800	186.143	255.873	180.051
900	204.253	299.945	213.199
1000	226.394	319.310	225.048

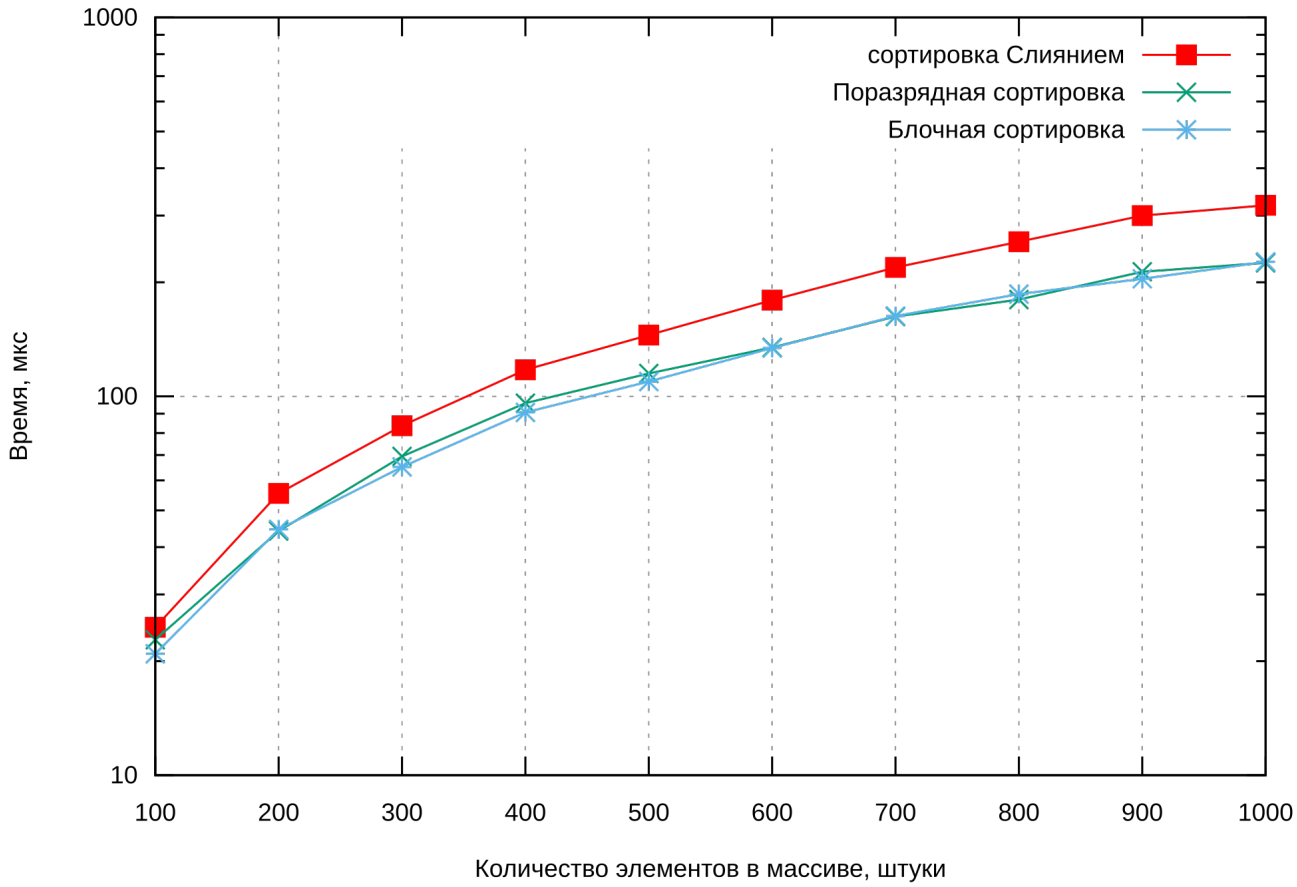

Рисунок 4.1 – Результаты замеров времени работы алгоритмов для массивом размеров от 100 до 1000 элементов

4.2.2 Массив случайно сгенерированных чисел заданного размера, отсортированный по возрастанию элементов

На вход подавались случайно сгенерированные векторы заданного размера, отсортированные по возрастанию элементов.

В отличие от данных, полученных в предыдущем исследовании, эти данные позволяют точно определить самый быстрый алгоритм сортировки из всех трех – алгоритм блочной сортировки. Хоть средняя разница во времени между алгоритмом блочной сортировки и поразрядной сортировки сохранила свое значение порядка 5 мкс, но на каждой из рассмотренных размерностей массива алгоритм блочной сортировки оказывался быстрее. Это связано с тем, что в

алгоритме поразрядной сортировки сортировка начинается с младших разрядов, что значения которых в отсортированном массиве могут быть не упорядочены, а в алгоритме блочной сортировки все каждое значение уже отсортированного массива попадает в свою собственный блок, где сортируется за $O(1)$ и возвращается в исходный массив в том же порядке.

Также важную роль в результатах времени работы блочной сортировки играет алгоритм, с помощью которого сортируется каждый сформированный блок. Для сортировки блоков была использована функция $std :: sort$ из заголовочного файла «$algorithm$» библиотеки языка $C++$, что существенно уменьшает время работы блочной сортировки.

Алгоритм сортировки слиянием оказался медленнее рассмотренных выше алгоритмов. Это связано с тем, что в этих двух алгоритмах создается всего по одному динамическому подмассиву для хранения данных, в то время как в алгоритме сортировки слиянием при каждом рекурсивном вызове создается два подмассива для хранения левой и правой отсортированных частей при слиянии.

Данные представлены в таблице 4.2. Их графическое отображение представлено на рисунке 4.2.

Таблица 4.2 – Замер времени для массивов размеров от 100 до 1000 элементов

Линейный размер, штуки	**Время, мкс**		
	Блочная	**Слиянием**	**Поразрядная**
100	20.878	21.533	22.429
200	40.743	46.217	44.154
300	60.475	69.919	64.169
400	83.173	98.500	90.431
500	104.820	125.650	112.199
600	122.446	151.872	135.148
700	142.568	175.570	155.219
800	164.111	201.655	169.880
900	190.310	232.209	199.635
1000	211.530	253.790	215.852

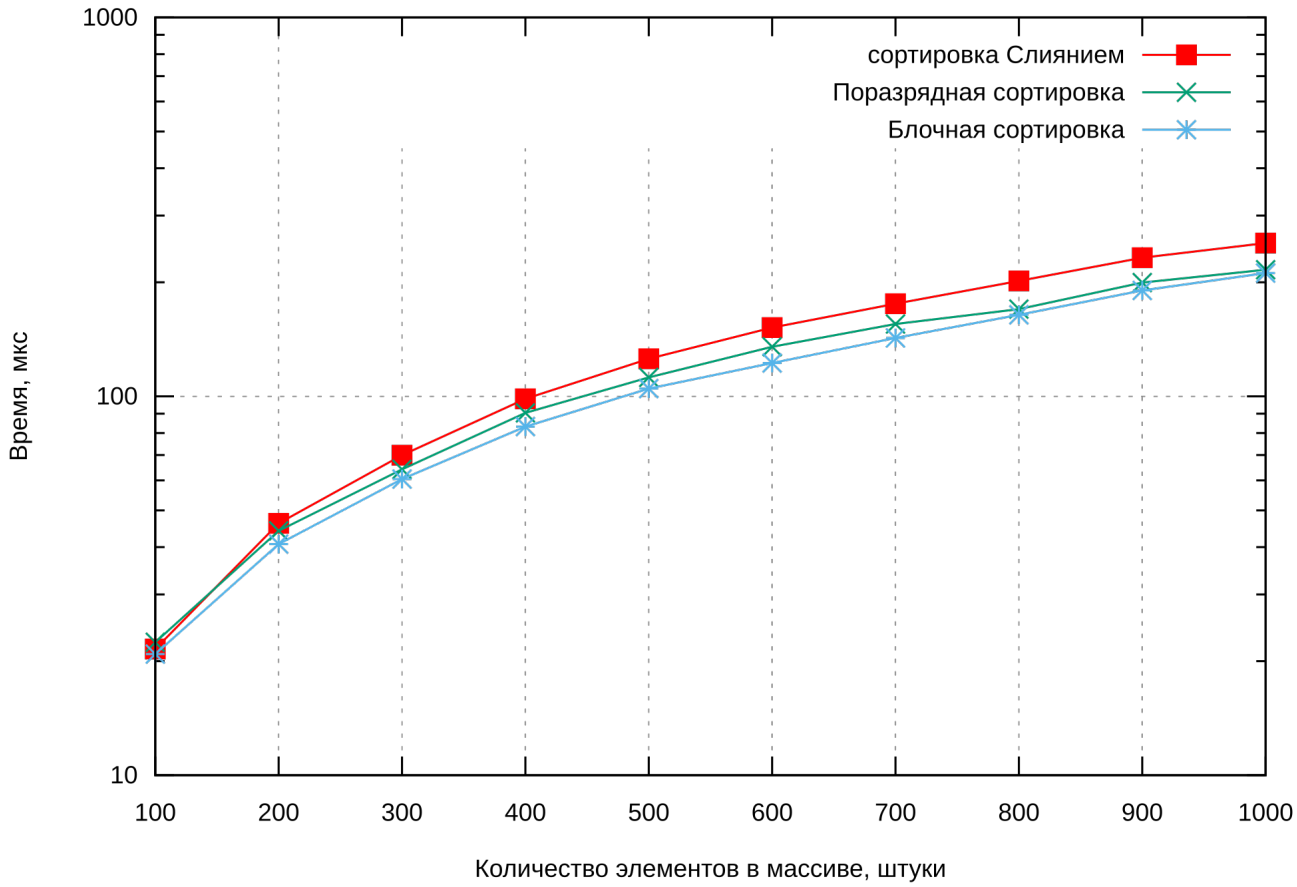

Рисунок 4.2 – Результаты замеров времени работы алгоритмов для массивом размеров от 100 до 1000 элементов

4.2.3 Массив случайно сгенерированных чисел заданного размера, отсортированный по убыванию элементов

На вход подавались случайно сгенерированные векторы заданного размера, отсортированные по убыванию элементов.

Исходя из полученных данных, самыми быстрыми алгоритмами сортировки из всех трех являются алгоритм поразрядной сортировки и блочной сортировки: средняя разница во времени сортировки этих двух алгоритмов составляет порядка 5 мкс, на некоторых значениях быстрее один алгоритм, на других – другой. Это связано с тем, что в этих двух алгоритмах создается всего по одному динамическому подмассиву для хранения данных, в то время как в алгоритме

сортировки слиянием при каждом рекурсивном вызове создается два подмассива для хранения левой и правой отсортированных частей при слиянии.

Также важную роль в результатах времени работы блочной сортировки играет алгоритм, с помощью которого сортируется каждый сформированный блок. Для сортировки блоков была использована функция $std :: sort$ из заголовочного файла «$algorithm$» библиотеки языка $C++$, что существенно уменьшает время работы блочной сортировки.

Алгоритм сортировки слиянием на размерах массива, больших ста, оказался медленнее рассмотренных выше алгоритмов. Это связано с тем, что в этих двух алгоритмах создается всего по одному динамическому подмассиву для хранения данных, в то время как в алгоритме сортировки слиянием при каждом рекурсивном вызове создается два подмассива для хранения левой и правой отсортированных частей при слиянии.

Однако, на размере массива в 100 элементов можно увидеть, что алгоритм сортировки слиянием работает быстрее остальных алгоритмов.

Данные представлены в таблице 4.3. Их графическое отображение представлено на рисунке 4.2.

Таблица 4.3 – Замер времени для массивов размеров от 100 до 1000 элементов

Линейный размер, штуки	**Время, мкс**		
	Блочная	**Слиянием**	**Поразрядная**
100	23.647	22.608	24.341
200	44.744	47.338	46.256
300	67.371	73.604	69.102
400	89.571	103.810	93.015
500	111.640	126.591	111.817
600	133.385	157.268	145.395
700	162.672	186.137	160.111
800	178.832	211.646	181.307
900	204.869	239.801	204.348
1000	227.685	266.782	224.748

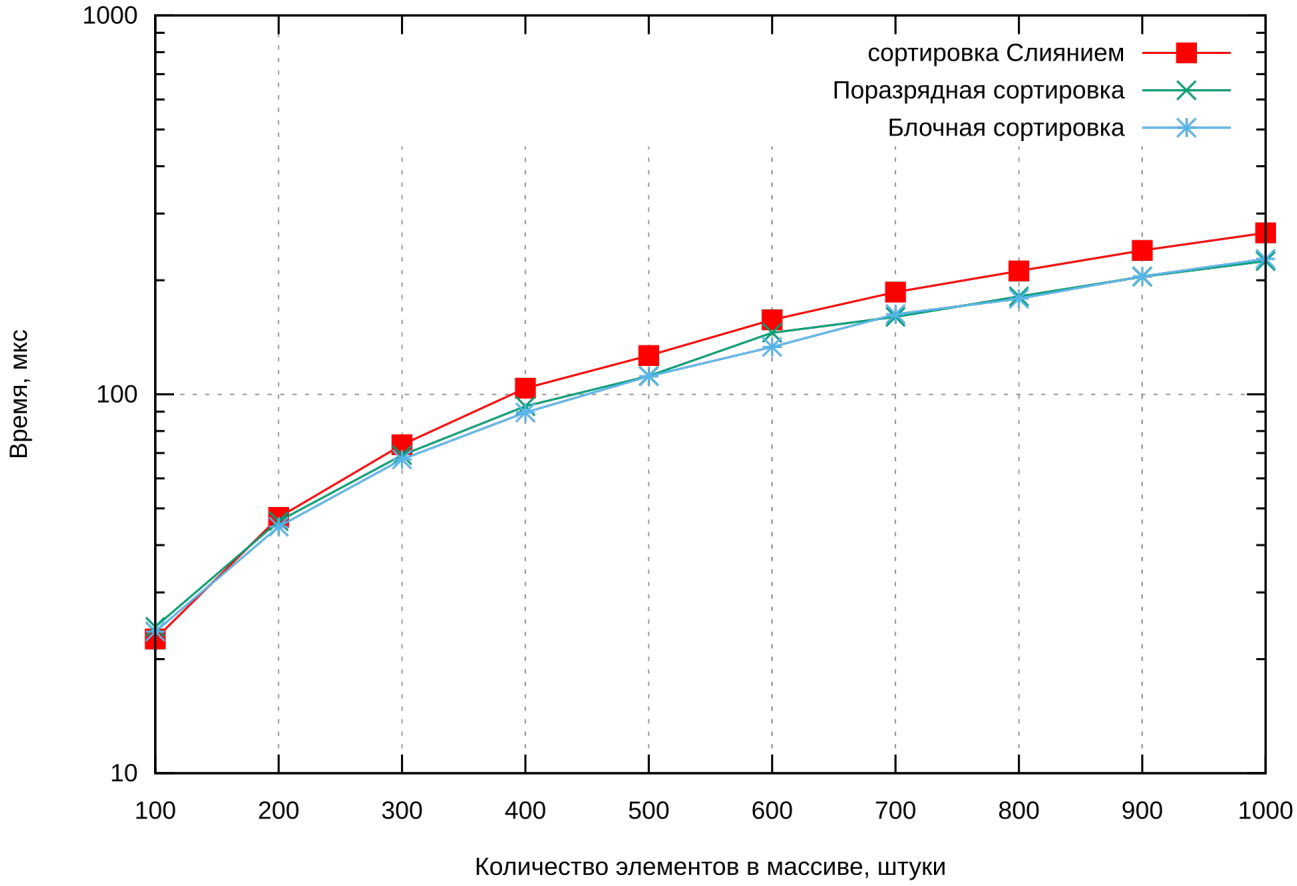

Рисунок 4.3 – Результаты замеров времени работы алгоритмов для массивом размеров от 100 до 1000 элементов

4.2.4 Массив, состоящий из одинаковых элементов

На вход подавались массивы заданного размера, состоящие из одинаковых элементов.

Исходя из полученных данных, можно увидеть, что самый быстрый алгоритм сортировки из всех трех – алгоритм блочной сортировки. Это связано с тем, что в алгоритме блочной сортировки все значения массива попадают в один блок, где сортируются функцией $std :: sort$ из заголовочного файла «$algorithm$» библиотеки языка $C++$, что существенно уменьшает время работы блочной сортировки.

Алгоритм сортировки слиянием оказался медленнее рассмотренных выше алгоритмов. Это связано с тем, что в этих двух алгоритмах создается всего по одному динамическому подмассиву для хранения данных, в то время как в алгоритме сортировки слиянием при каждом рекурсивном вызове создается два подмассива для хранения левой и правой отсортированных частей при слиянии.

Однако, на размере массива в 100 элементов можно увидеть, что алгоритм сортировки слиянием работает быстрее алгоритма поразрядной сортировки.

Данные представлены в таблице 4.4. Их графическое отображение представлено на рисунке 4.2.

Таблица 4.4 – Замер времени для массивов размеров от 100 до 1000 элементов

Линейный размер, штуки	**Время, мкс**		
	Блочная	**Слиянием**	**Поразрядная**
100	13.632	22.066	23.332
200	28.923	46.919	44.327
300	49.952	77.532	73.044
400	68.537	106.991	92.550
500	82.382	130.768	112.892
600	110.334	165.079	139.385
700	121.641	184.348	166.654
800	154.355	221.561	186.757
900	157.712	253.447	212.145
1000	176.284	268.412	231.275

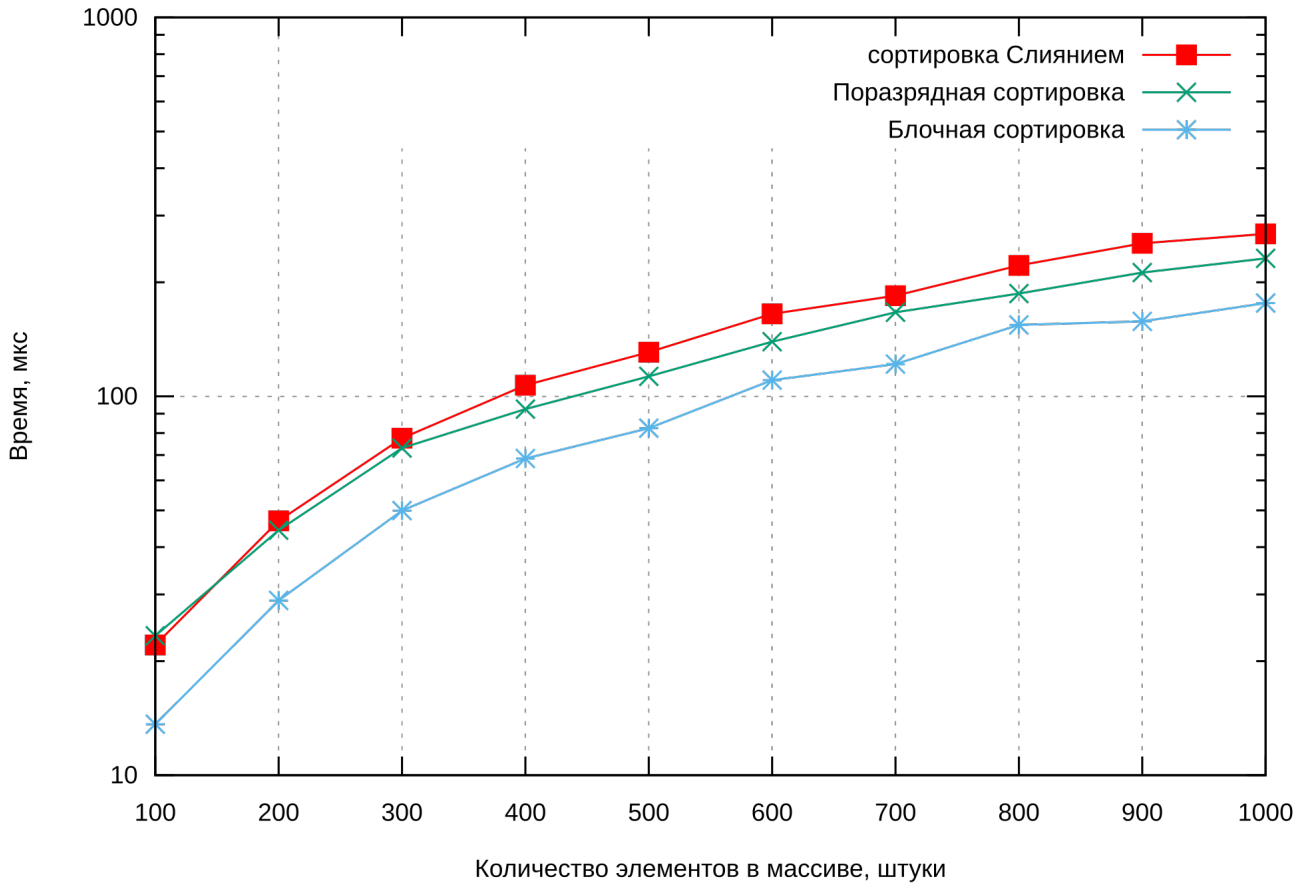

Рисунок 4.4 – Результаты замеров времени работы алгоритмов для массивом размеров от 100 до 1000 элементов

4.3 Использование памяти

Анализируя полученные данные можно увидеть, что самым эффективным по памяти является является алгоритм блочной сортировки. Это обусловлено тем, что в этом алгоритме создается всего один дополнительный массив блоков, в который по которым равномерно распределяются элементы массива, в то время как для работы других алгоритмов сортировки необходимо минимум два дополнительных массива данных.

Алгоритм сортировки слиянием, как и в случае с оценкой алгоритмов по времени, является самым не эффективным: при размере массива в 100 элементов он расходует памяти в среднем в 1.4 раза больше, чем любой другой алгоритм. Это связано стем, что при каждом рекурсивном вызове при слиянии двух отсортированных подмассивов выделяется память под их хранение.

Данные представлены в таблице 4.5. Их графическое отображение представлено на рисунке **??**.

Таблица 4.5 – Замер используемой памяти для массивов размеров от 100 до 1000 элементов

Линейный размер, штуки	**Память, байты**		
	Блочная	**Слиянием**	**Поразрядная**
100	880	1 280	932
200	1 680	2 164	1 732
300	2 480	3 040	2 532
400	3 280	3 848	3 332
500	4 080	4 624	4 132
600	4 880	5 524	4 932
700	5 680	6 308	5 732
800	6 480	7 132	6 532
900	7 280	7 924	7 332
1000	8 080	8 708	8 132

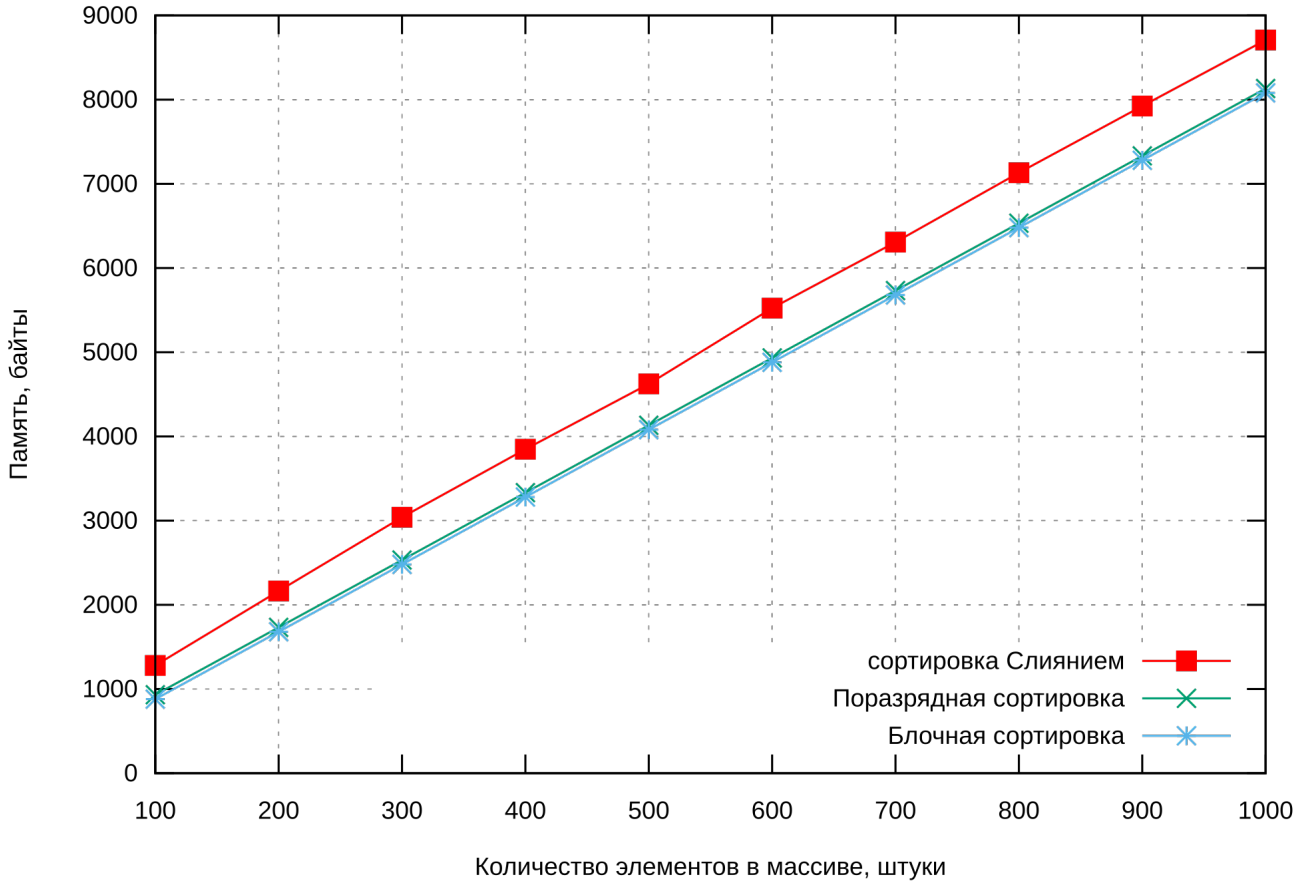

Рисунок 4.5 – Результаты замеров работы используемой памяти алгоритмов для массивом размеров от 100 до 1000 элементов

Вывод

В данном разделе были проведены замеры времени работы, а также расчеты используемой памяти реализаций алгоритмов сортировки.

Самым эффективным по обоим параметрам оказался алгоритм блочной сортировки, самым неэффективным по обоим параметрам оказался алгоритм сортировки слиянием. Полученные результаты для алгоритма поразрядной сортировки показали, что он выполняет сортировку почти также быстро, как алгоритм блочной сортировки. Алгоритм поразрядной сортировки немного проигрывает по расходу памяти алгоритму блочной сортировки, но выигрывает у алгоритма сортировки слиянием.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) Изучены и описаны три алгоритма сортировки: блочной, слиянием и поразрядной.

2) Создано программное обеспечение, реализующее следующие алгоритмы:

   – алгоритм блочной сортировки;

   – алгоритм сортировки слиянием;

   – алгоритм поразрядной сортировки.

3) Проведен анализ эффективности реализаций алгоритмов по памяти и по времени.

4) Проведена оценка трудоемкости алгоритмов сортировки.

5) Подготовлен отчет по лабораторной работе.

Цель данной лабораторной работы, а именно исследование трех алгоритмов сортировки: блочной сортировки, сортировки слиянием и поразрядной сортировки, также была достигнута.

Согласно теоретическим расчетам трудоемкости алгоритмов сортировки наименее трудоемким на равномерно распределенных данных оказался алгоритм блочной сортировки, наиболее трудоемким – алгоритм поразрядной сортировки.

Результаты проведенного исследования практически подтвердили теоретические расчеты трудоемкости: наиболее эффективным по времени работы и по используемой памяти является алгоритм блочной сортировки, но наиболее трудоемким оказался алгоритм сортировки слиянием.

На равномерно распределенных данных лучше всего использовать алгоритм блочной сортировки, а для сортировки любых элементов, которые можно поделить на разряды, подошел бы алгоритм поразрядной сортировки.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Кнут Д. Э.* Искусство програмирования. — М. : Издательский дом Вильямс, 2007.
2. Сортировка по разрядам [Электронный ресурс]. — Режим доступа: `:http://algolab.valemak.com/radix` (дата обращения: 04.12.2023).
3. Справочник по языку C++ [Электронный ресурс]. — Режим доступа: `https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=msvc-170` (дата обращения: 28.09.2022).
4. clock_getres [Электронный ресурс]. — Режим доступа: `https://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html` (дата обращения: 28.09.2022).
5. Ubuntu 22.04.3 LTS (Jammy Jellyfish) [Электронный ресурс]. — Режим доступа: `https://releases.ubuntu.com/22.04/` (дата обращения: 28.09.2022).