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 № 4

по курсу «Анализ Алгоритмов»

на тему: «Параллельные вычисления на основе нативных потоков»

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

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

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Многопоточность — свойство кода программы выполняться параллельно (одновременно) на нескольких ядрах процессора или псевдопараллельно на одном ядре (каждый поток получает в свое распоряжение некоторое время, за которое он успевает исполнить часть своего кода на процессоре) [1].

Поток (thread) представляет собой независимую последовательность инструкций в программе. В приложениях, которые имеют пользовательский интерфейс, всегда есть как минимум один главный поток, который отвечает за состояние компонентов интерфейса. Кроме него в программе может создаваться множество независимых дочерних потоков, которые будут выполняться независимо [1].

Целью данной лабораторной работы является изучение принципов и получение навыков организации параллельного выполнения операций.

Для достижения поставленной цели необходимо решить следующие задачи:

1) изучить и описать алгоритм составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке;

2) разработать последовательную и параллельную версии данного алгоритма;

3) реализовать каждую версию алгоритма;

4) провести сравнительный анализ алгоритмов по времени работы реализаций;

5) обосновать полученные результаты в отчете к выполненной лабораторной работе.

1 Аналитический раздел

В данном разделе будет приведено теоретическое описание последовательной и параллельной версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

1.1 N-граммы

Пусть задан некоторый конечный алфавит

$$V = w_i \tag{1.1}$$

где w_i — символ.

Языком $L(V)$ называют множество цепочек конечной длины из символов w_i. N-граммой на алфавите V (1.1) называют произвольную цепочку из $L(V)$ длиной N, например последовательность из N букв русского языка одного слова, одной фразы, одного текста или, в более интересном случае, последовательность из грамматически допустимых описаний N подряд стоящих слов [2].

1.2 Последовательная версия алгоритма

Алгоритм составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке состоит из следующих шагов:

1) считывание текста в массив строк;
2) преобразование считанного текста (перевод букв в нижний регистр, удаление знаков препинания);
3) обработка каждой строки текста.

Обработка строки текста включает следующие шаги:

1) обработка каждого слова из строки текста;
2) выделение существующих в этом слове N-грамм;
3) увеличение количества выделенных N-грамм в словаре.

1.3 Параллельная версия алгоритма

В алгоритме составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке обработка строк текста происходит независимо, поэтому есть возможность произвести распараллеливание данных вычислений.

Для этого строки текста поровну распределяются между потоками. Каждый поток получает локальную копию словаря для N-грамм, производит вычисления над своим набором строк и после завершения работы всех потоков словари с количеством употреблений N-грамм каждого потока объединяются в один. Так как каждая строка массива передается в монопольное использование каждому потоку, не возникает конфликтов доступа к разделяемым ячейки памяти,следовательно, в использовании средства синхронизации в виде мьютекса нет необходимости.

Вывод

В данном разделе было приведено теоретическое описание последовательной и параллельной версии алгоритма составления файла словаря с количеством употреблений каждой N-граммы букв из одного слова в тексте на русском языке.

2 Конструкторский раздел

2.1 Разработка алгоритмов

2.1.1 Алгоритм блочной сортировки

На рисунке 2.1 представлена схема алгоритма блочной сортировки.

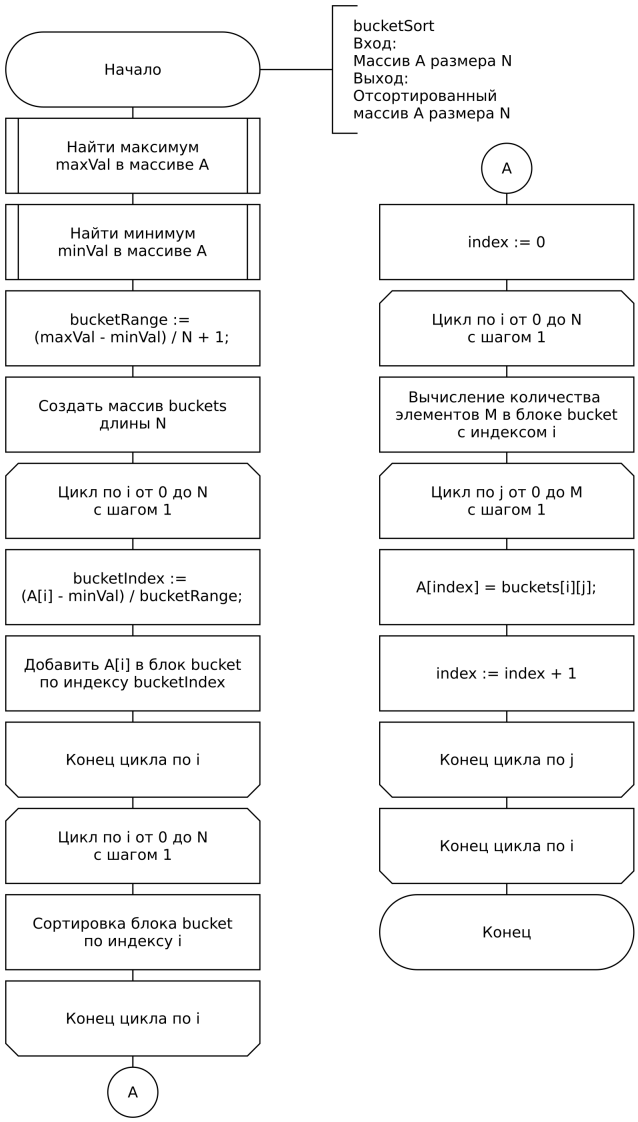

Рисунок 2.1 – Схема алгоритма блочной сортировки

2.1.2 Алгоритм сортировки слиянием

На рисунке 2.2 представлена схема алгоритма сортировки слиянием.

На рисунках 2.3 и 2.4 представлена схема алгоритма функции слияния двух отсортированных подмассивов.

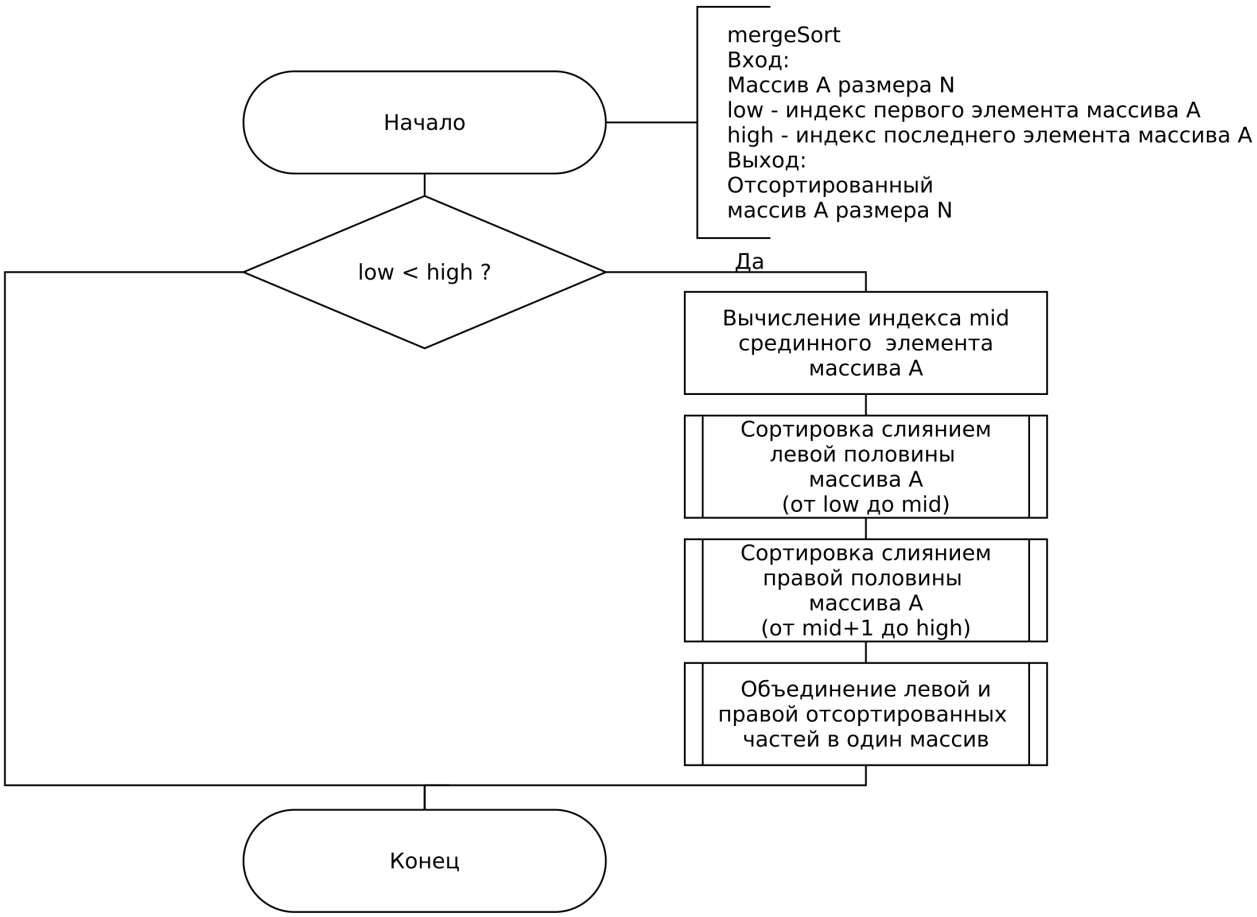

Рисунок 2.2 – Схема алгоритма сортировки слиянием

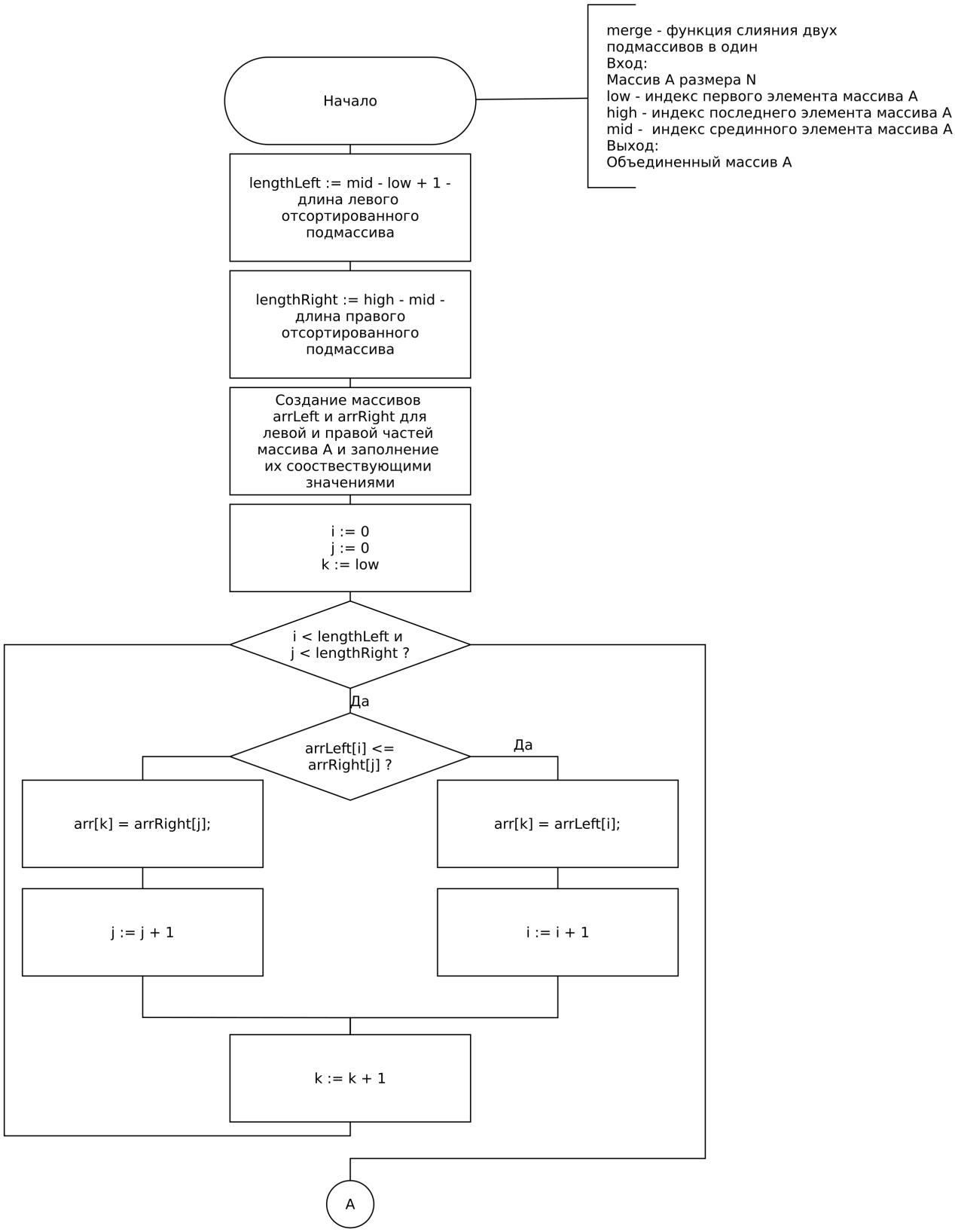

Рисунок 2.3 – Схема алгоритма функции слияния двух отсортированных подмассивов (начало)

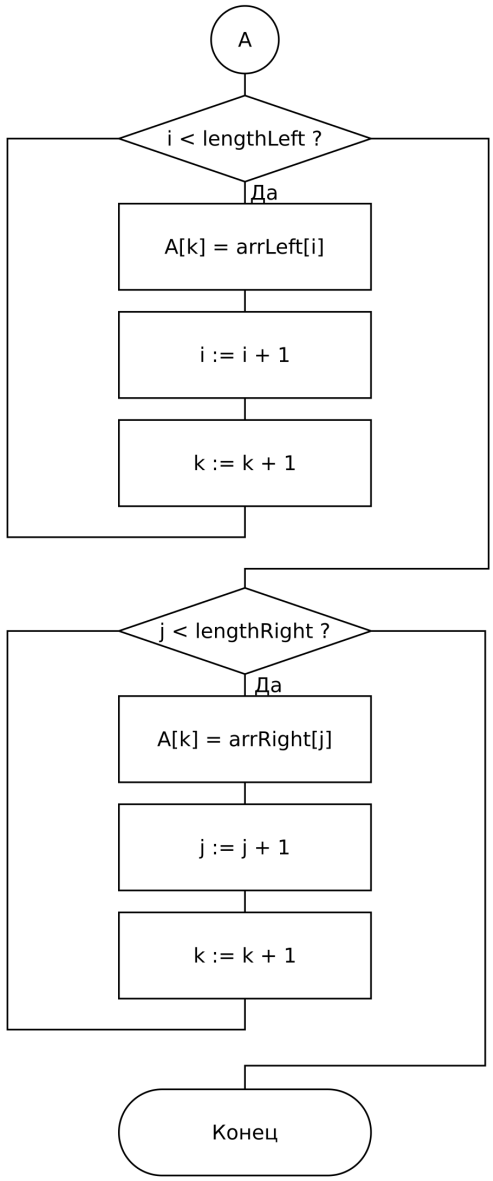

Рисунок 2.4 – Схема алгоритма функции слияния двух отсортированных подмассивов (конец)

2.1.3 Алгоритм поразрядной сортировки

На рисунке 2.5 представлена схема алгоритма сортировки слиянием.

На рисунке 2.6 представлена схема алгоритма функции сортировки по определенному разряду.

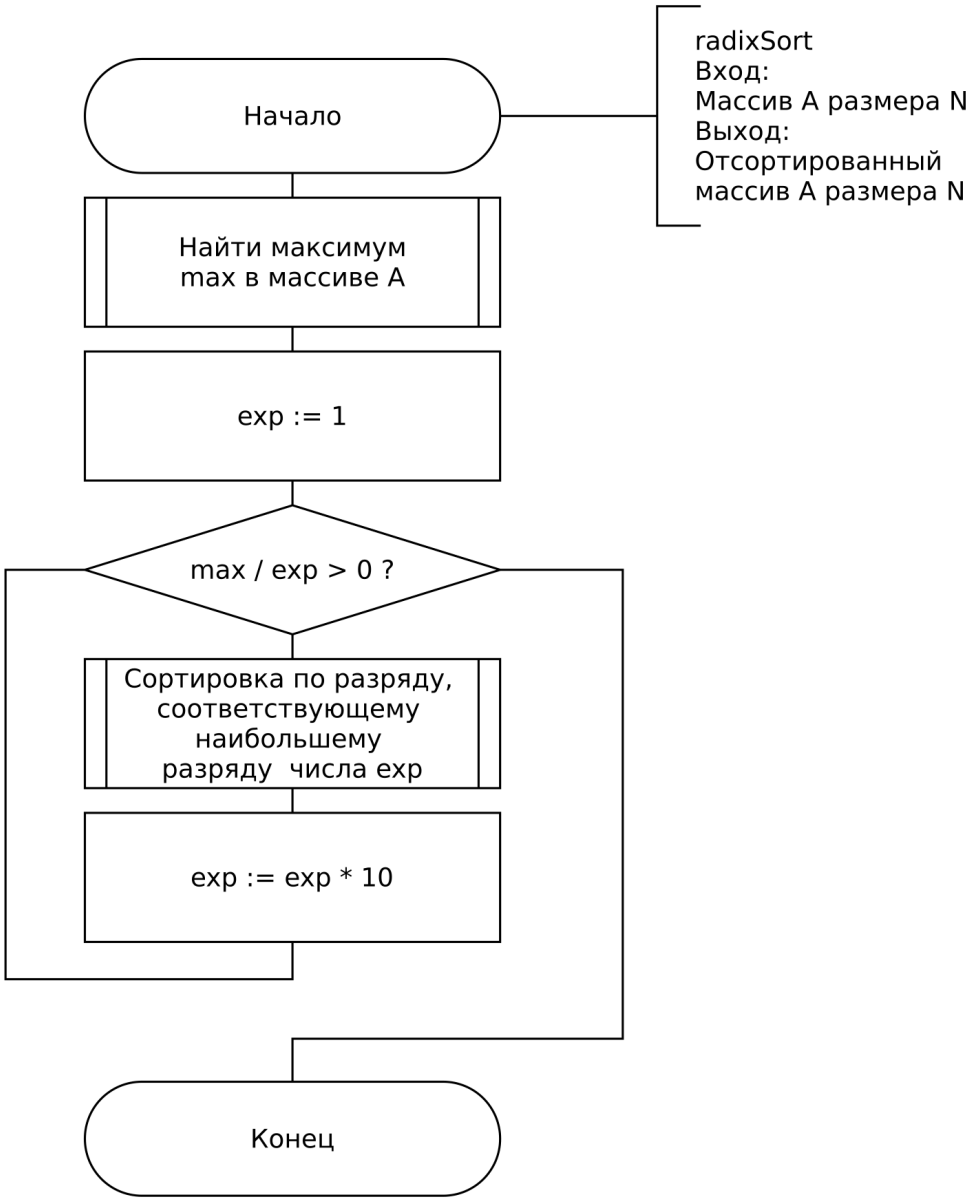

Рисунок 2.5 – Схема алгоритма поразрядной сортировки

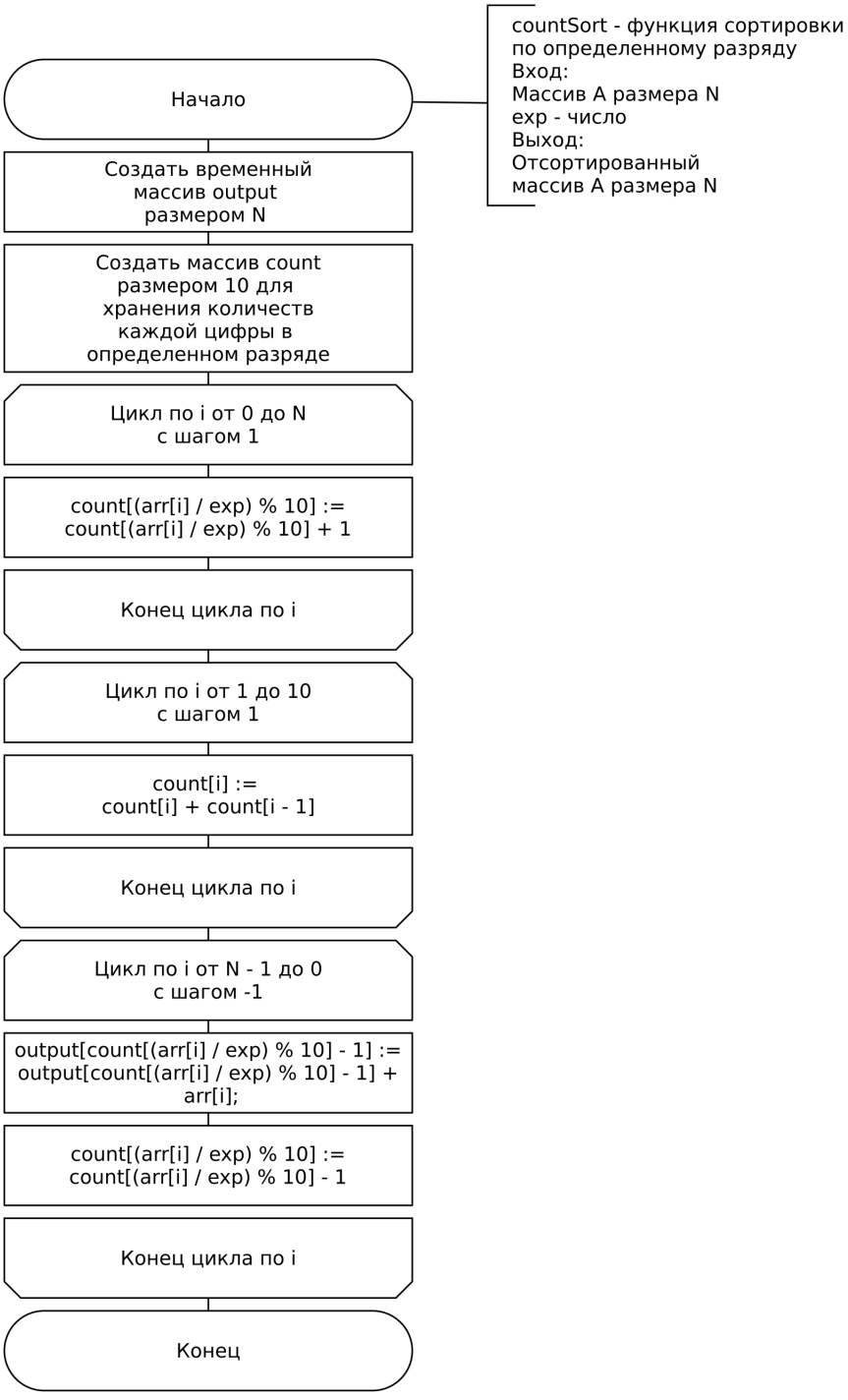

Рисунок 2.6 – Схема алгоритма функции сортировки по определенному разряду

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. На вход подавались случайно сгенерированные векторы заданного размера.

Исходя из полученных данных, наиболее быстрым алгоритмом сортировки из всех трех является алгоритм блочной сортировки: на больших размерах он работает в 1.5 раза быстрее алгоритма сортировки слиянием, в 1.04 раз быстрее алгоритма поразрядной сортировки. Алгоритм сортировки слиянием оказался самым неэффективным по времени среди всех алгоритмов.

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	21.984	24.079	23.367
200	43.961	52.015	47.727
300	66.580	90.222	69.285
400	92.634	116.973	93.435
500	113.289	148.954	116.621
600	134.604	187.587	141.475
700	157.763	216.882	165.108
800	179.392	250.836	187.108
900	205.149	292.716	211.348
1000	221.511	332.679	231.352

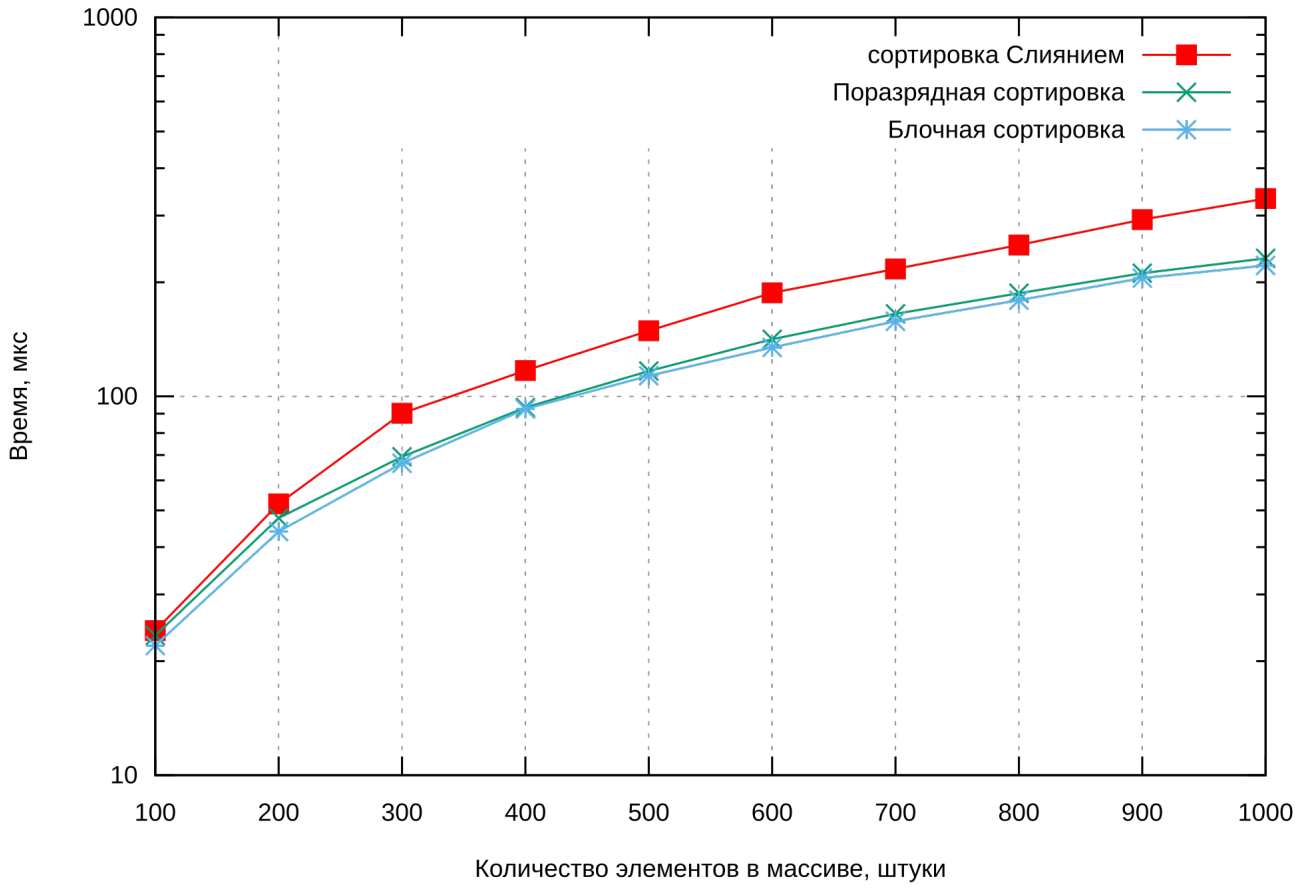

Рисунок 4.1 – Результаты замеров времени работы алгоритмов для массивом размеров от 100 до 1000 элементов

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 4.2. Их графическое отображение представлено на рисунке 4.1.

Таблица 4.2 – 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

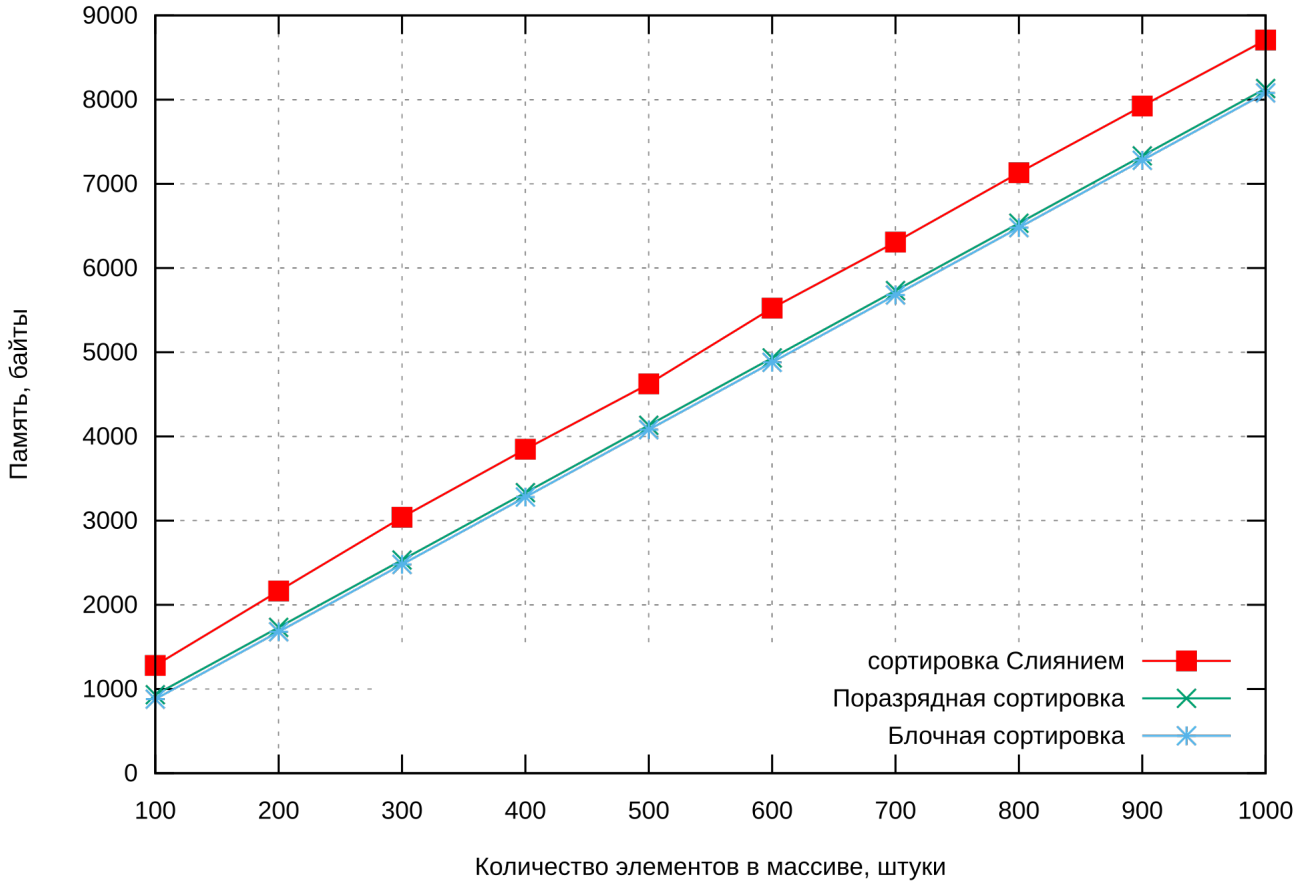

Рисунок 4.2 – Результаты замеров работы используемой памяти алгоритмов для массивом размеров от 100 до 1000 элементов

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. Многопоточность в языке программирования C# [Электронный ресурс]. — Режим доступа: https://cyberleninka.ru/article/n/mnogopotochnost-v-yazyke-programmirovaniya-c (дата обращения: 27.01.2024).
2. N-граммы в лингвистике [Электронный ресурс]. — Режим доступа: https://cyberleninka.ru/article/n/n-grammy-v-lingvistike (дата обращения: 27.01.2024).
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).