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) Лысцев Н. Д. (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **4**

1 Аналитический раздел **5**

1.1 Алгоритм блочной сортировки 5

1.2 Алгоритм сортировки слиянием 5

1.3 Алгоритм поразрядной сортировки 5

2 Конструкторский раздел **7**

2.1 Оценка трудоемкости алгоритмов 7

2.1.1 Модель вычислений для проведения оценки трудоемкости алгоритмов 7

2.1.2 Трудоемкость алгоритма блочной сортировки 8

2.1.3 Трудоемкость алгоритма Винограда для умножения двух матриц 8

2.1.4 Трудоемкость оптимизированного алгоритма Винограда для умножения двух матриц 10

2.1.5 Трудоемкость алгоритма Штрассена для умножения двух матриц 11

2.1.6 Трудоемкость оптимизированного алгоритма Штрассена для умножения двух матриц 13

3 Технологический раздел **14**

3.1 Требования к программному обеспечению 14

3.2 Средства реализации 14

3.3 Сведения о модулях программы 15

3.4 Реализации алгоритмов 16

3.5 Функциональные тесты 20

4 Исследовательский раздел **21**

4.1 Технические характеристики 21

4.2 Время выполнения алгоритмов 21

4.3 Использование памяти 21

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **22**

ВВЕДЕНИЕ

Сортировка – процесс перегруппировки последовательности объектов в некотором порядке. Это одна из фундаментальных операций в алгоритмике и компьютерных науках, играющая ключевую роль в эффективной обработке данных.

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) Изучить и 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

Блочная сортировка – алгоритм сортировки, в котором сортируемые 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. Затем элементы помещаются обратно в массив [1].

1.2 Алгоритм сортировки слиянием

Сортировка слиянием – алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй» [2].

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 – алгоритм сортировки, который выполняется за линейное время. Сравнение производится поразрядно: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца [3].

Вывод

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

2 Конструкторский раздел

2.1 Оценка трудоемкости алгоритмов

2.1.1 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:

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

— 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:

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

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

2.1.2 Трудоемкость алгоритма блочной сортировки

Трудоемкость алгоритма блочной сортировки будет слагаться из:

— инициализации пяти переменных, суммарная трудоемкость которых равна 5;

— цикла по $j \in [1 \ldots P]$, трудоёмкость которого: $f = 2 + 2 + P \cdot (2 + f_{body})$;

— цикла по $k \in [1 \ldots M]$, трудоёмкость которого: $f = 2 + 2 + 14M$;

Поскольку трудоемкость стандартного алгоритма равна трудоемкости внешнего цикла, то:

$$\begin{aligned} f_{standart} &= 2 + N \cdot (2 + 2 + P \cdot (2 + 2 + M \cdot (2 + 8 + 1 + 1 + 2))) = \\ &= 2 + 4N + 4NP + 14NMP \approx 14NMP = O(N^3) \end{aligned} \tag{2.5}$$

2.1.3 Трудоемкость алгоритма Винограда для умножения двух матриц

При вычислении трудоемкости алгоритма Винограда учитывается следующее:

— создание и инициализация массивов $rowFactor$ и $colFactor$, трудоёмкость которых указана в формуле (2.6);

$$f_{init} = N + M \tag{2.6}$$

— заполнение массива $rowFactor$, трудоёмкость которого указана в формуле (2.7);

$$\begin{gathered} f_{rowFactor} = 2 + N \cdot (4 + \frac{M}{2} \cdot (4 + 6 + 1 + 2 + 3 \cdot 2)) = \\ = 2 + 4N + \frac{19NM}{2} = 2 + 4N + 9,5NM \end{gathered} \tag{2.7}$$

— заполнение массива $colFactor$, трудоёмкость которого указана в формуле (2.8);

$$\begin{aligned} f_{colFactor} = 2 + P \cdot (4 + \frac{M}{2} \cdot (4 + 6 + 1 + 2 + 3 \cdot 2)) = \\ = 2 + 4P + \frac{19PM}{2} = 2 + 4P + 9,5PM \end{aligned} \tag{2.8}$$

— цикл заполнения для чётных размеров, трудоёмкость которого указана в формуле (2.9);

$$\begin{aligned} f_{cycle} = 2 + N \cdot (4 + P \cdot (2 + 7 + 4 + \frac{M}{2} \cdot (4 + 28))) = \\ = 2 + 4N + 13NP + \frac{32NPM}{2} = 2 + 4N + 13NP + 16NPM \end{aligned} \tag{2.9}$$

— цикла, который дополнительно нужен для подсчёта значений при нечётном размере матрицы, трудоемкость которого указана в формуле (2.10);

$$f_{check} = 3 + \begin{cases} 0, & \text{чётная} \\ 2 + M \cdot (4 + P \cdot (2 + 14)), & \text{иначе} \end{cases} \tag{2.10}$$

Тогда для худшего случая (нечётный общий размер матриц) имеем:

$$f_{worst} = f_{init} + f_{rowFactor} + f_{colFactor} + f_{cycle} + f_{check} \approx 16NMP = O(N^3) \tag{2.11}$$

Для лучшего случая (чётный общий размер матриц) имеем:

$$f_{best} = f_{init} + f_{rowFactor} + f_{colFactor} + f_{cycle} + f_{check} \approx 16NMP = O(N^3) \tag{2.12}$$

2.1.4 Трудоемкость оптимизированного алгоритма Винограда для умножения двух матриц

Трудоемкость оптимизированного алгоритма Винограда состоит из:

- кэширования значения $\frac{M}{2}$ в циклах, которое равно 3;
- создания и инициализации массивов $rowFactor$ и $colFactor$ (2.6);
- заполнения массива $rowFactor$, трудоёмкость которого (2.7);
- заполнения массива $colFactor$, трудоёмкость которого (2.8);
- цикла заполнения для чётных размеров, трудоёмкость которого указана в формуле (2.13);

$$\begin{aligned} f_{cycle} &= 2 + N \cdot (4 + P \cdot (4 + 7 + \frac{M}{2} \cdot (2 + 10 + 5 + 2 + 4))) = \\ &= 2 + 4N + 11NP + \frac{23NPM}{2} = 2 + 4N + 11NP + 11,5 \cdot NPM \end{aligned} \tag{2.13}$$

- условия, которое нужно для дополнительных вычислений при нечётном размере матрицы, трудоемкость которого указана в формуле (2.14);

$$f_{check} = 3 + \begin{cases} 0, & \text{чётная} \\ 2 + N \cdot (4 + P \cdot (2 + 10)), & \text{иначе} \end{cases} \tag{2.14}$$

Тогда для худшего случая (нечётный общий размер матриц) имеем:

$$f_{worst} = 3 + f_{init} + f_{atmp} + f_{btmp} + f_{cycle} + f_{check} \approx 11NMP = O(N^3) \tag{2.15}$$

Для лучшего случая (чётный общий размер матриц) имеем:

$$\begin{aligned} f_{best} = 3 + f_{init} + f_{rowFactor} + f_{colFactor} \\ + f_{cycle} + f_{check} \approx 11NMP = O(N^3) \end{aligned} \tag{2.16}$$

2.1.5 Трудоемкость алгоритма Штрассена для умножения двух матриц

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:

$$REC(N) = \begin{cases} DIR(N), & N \leq N_0 \\ DIV(N) + \sum\limits_{i=1}^{n} REC(F[i]) + COM(N), & N > N_0 \end{cases} \tag{2.17}$$

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.

Для расчета трудоемкости алгоритма Штрассена предположим, что размеры переданных матриц – степени двойки.

Тогда трудоемкость алгоритма Штрассена определяется следующим образом:

- Для матрицы, размером $N \leq 2$ трудоемкость определяется как и в случае классического алгоритма умножения матриц, то есть согласно формуле 2.5
- Для матриц размером $N > 2$ определяется так:

  1) Трудоемкость разбиения ввода (N) на части. Каждый следующий вызов берется размерность матрицы в 2 раза меньше предыдущей,

и происходит создание соответствующих подматриц и заполнение их значениями.

$$DIV(N) = 1 + 8 \cdot (3 + \frac{N}{2} \cdot ((3 + \frac{N}{2} \cdot (5 + 2 + 1)) + 2 + 1) = \\ 16 \cdot N^2 + 24 \cdot N + 25 \tag{2.18}$$

2) Трудоемкость вычисления матриц $M_i,\ \ i = \overline{1,7}$ (обозначим ее буквой $G = G(N)$):

$$G(N) = 10 \cdot (2 + \frac{N}{2} \cdot (2 + \frac{N}{2} \cdot (8 + 1 + 1) + 1 + 1)) + \\ +7 \cdot REC(\frac{N}{2}) \tag{2.19}$$

где, так как $N = 2^k$ и согласно с **??**

$$REC(\frac{N}{2}) = REC(2^{k-1}) = 7 \cdot M(2^{k-2}) = \dots 7^{i-1} M(2^{k-i}) = \dots \\ 7^{k-1} M(2^{k-k}) = 7^{k-1} \tag{2.20}$$

подставляя $k = \log_2(N)$ получаем, что

$$REC(\frac{N}{2}) = \frac{N^{\log_2(7)}}{7} \tag{2.21}$$

Таким образом, трудоемкость вычисления матриц $M_i,\ \ i = \overline{1,7}$ определяется следующей формулой:

$$G(N) = 10 \cdot (10 \cdot (\frac{N}{2})^2 + 4 \cdot \frac{N}{2} + 2) + N^{\log_2(7)} = \\ 25 \cdot N^2 + 20 \cdot N + 20 + N^{\log_2(7)} \tag{2.22}$$

3) Трудоемкость объединения решений, а именно формирование результирующей матрицы из вычисленных матриц $M_i,\ \ i = \overline{1,7}$

$$
\begin{gathered}
COM(N) = 8 \cdot (2 + \frac{N}{2} \cdot (2 + \frac{N}{2} \cdot (8 + 1 + 1) + 1 + 1)) + \\
4 \cdot (3 + \frac{N}{2} \cdot ((3 + \frac{N}{2} \cdot (5 + 2 + 1)) + 2 + 1) = \\
28 \cdot N^2 + 28 \cdot N + 28
\end{gathered} \tag{2.23}
$$

Таким образом, для матриц размером $N > 2$ трудоемкость алгоритма Штрассена согласно 2.17 определяется так:

$$
\begin{gathered}
f_{strassen}(N) = DIV(N) + G(N) + COM(N) = \\
16 \cdot N^2 + 24 \cdot N + 25 + 25 \cdot N^2 + 20 \cdot N + 20 + N^{\log_2(7)} + \\
28 \cdot N^2 + 28 \cdot N + 28 = \\
N^{\log_2(7)} + 69 \cdot N^2 + 72 \cdot N + 73 \approx N^{\log_2(7)} = O(N^{\log_2(7)})
\end{gathered} \tag{2.24}
$$

2.1.6 Трудоемкость оптимизированного алгоритма Штрассена для умножения двух матриц

При программной реализации алгоритма Штрассена не нашлось мест для применения предложенных по варианту оптимизаций, поэтому трудоемкость алгоритма Штрассена осталасть такой же, как и в предыдущем пункте.

Вывод

В данном разделе были построены схемы алгоритмов классического умножения матриц, умножения матриц с использованием алгоритма Винограда и алгоритма Штрассена. Также были приведены оценки трудоемкости этих алгоритмов.

Согласно расчетам трудоемкости, наиболее эффективным оказался алгоритм Штрассена. Трудоемкость оптимизированной версии алгоритма Винограда в 1.5 раза меньше, чем у его неоптимизированной версии и в 1.27 раз маньше, чем у классического алгоритма.

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 $C++$ [4] 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 $ctime$ [5].

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 протестированно 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 [6];
- 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 матрицы проводились 1000 раз. На вход подавались случайно сгенерированные матрицы заданного размера. z

4.3 Использование памяти

Вывод

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

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Блочная сортировка [Электронный ресурс]. — Режим доступа: https://en.wikipedia.org/wiki/Bucket_sort (дата обращения: 21.11.2023).
2. Сортировка слиянием [Электронный ресурс]. — Режим доступа: https://en.wikipedia.org/wiki/Merge_sort (дата обращения: 21.11.2023).
3. Поразрядная сортировка [Электронный ресурс]. — Режим доступа: https://en.wikipedia.org/wiki/Radix_sort (дата обращения: 21.11.2023).
4. 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).
5. clock_getres [Электронный ресурс]. — Режим доступа: https://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html (дата обращения: 28.09.2022).
6. Ubuntu 22.04.3 LTS (Jammy Jellyfish) [Электронный ресурс]. — Режим доступа: https://releases.ubuntu.com/22.04/ (дата обращения: 28.09.2022).