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 ИУ7-54Б (Группа) _____ (Подпись, дата) Булдаков М. (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

С развитием вычислительных систем появилась потребность в параллельной обработке данных для повышения эффективности систем, ускорения вычислений и более рационального использования имеющихся ресурсов. Благодаря совершенствованию процессоров стало возможно использовать их для выполнения множества одновременных задач, что привело к появлению понятия «многопоточность» [1].

Цель данной лабораторной работы — описать принципы параллельных вычислений на основе нативных потоков для исправления орфографических ошибок в тексте. Для достижения поставленной цели необходимо выполнить следующие задачи:

— описать алгоритм исправления орфографических ошибок в тексте;

— спроектировать программное обеспечение, реализующее алгоритм и его параллельную версию;

— выбрать инструменты для реализации и замера процессорного времени выполнения реализаций алгоритмов;

— проанализировать затраты реализаций алгоритмов по времени.

1 Аналитический раздел

В данном разделе будет представлена информация о многопоточности и исследуемом алгоритме исправления орфографических ошибок в тексте.

1.1 Многопоточность

Многопоточность [2] — это способность центрального процессора одновременно выполнять несколько потоков, используя ресурсы одного процессора. Каждый поток представляет собой последовательность инструкций, которые могут выполняться параллельно с другими потоками, созданными одним и тем же процессом.

Процессом называют программу в стадии выполнения [3]. Один процесс может иметь один или несколько потоков. Поток — это часть процесса, которая выполняет задачи, необходимые для выполнения приложения. Процесс завершается, когда все его потоки полностью завершены.

Одной из сложностей, связанных с использованием потоков, является проблема доступа к данным. Основным ограничением является невозможность одновременной записи в одну и ту же ячейку памяти из разных потоков. Это означает, что нужен механизм синхронизации доступа к данным, так называемый “мьютекс” (от англ. mutex - mutual exclusion, взаимное исключение). Мьютекс может быть захвачен одним потоком для работы в режиме монопольного использования или освобожден. Если два потока попытаются захватить мьютекс одновременно, то успех будет у одного потока, а другой будет блокирован, пока мьютекс не освободится.

1.2 Исправления орфографических ошибок в тексте

Для распознавания слов, написанных с ошибками, используется расстояние Левенштейна — минимальное количество ошибок, исправление которых приводит одно слово к другому [4]. Т. о. для введенного слова осуществляется проверка по корпусу, если данное слово не найдено в корпусе, то ищется ближайшее слово к данному по расстоянию Левенштейна.

Кроме того, следует вводить ограничение на количество ошибок, которые позволяется допустить. Как говорит поговорка: «Если в слове хлеб допустить всего четыре ошибки, то получится слово пиво». Если фиксируется число

ошибок, то для коротких слов оно может оказаться избыточным. Верхнюю границу числа ошибок обычно ограничивают как процентным соотношением, так и фиксированным числом. Например, не более 30% букв входного слова, но не более 3. При этом все равно стараются найти слова с минимальным количеством ошибок [4].

1.3 Использование потоков для исправления орфографических ошибок

Поскольку задача сводится к поиску слова в корпусе, можно распараллелить поиск по этому корпусу. В таком случае каждый поток будет вычислять расстояние Левенштейна между заданным словом и некоторым словом из корпуса и в случае, если расстояние будет удовлетворять требованиям, то данное слово будет записано в массив. Для определения наилучших соответствий необходимо хранить минимальное количество ошибок на текущий момент, т. е. возможна ситуация, когда в одном потоке минимальное количество ошибок будет считано, а в другом в тот же момент изменено, следовательно возникает конфликт. То же касается и записи подходящих слов в массив, требуется отбирать лучшие k слов, поэтому возможна ситуация, когда значение длины массива считывается в одном потоке и в тот же момент изменяется в другом потоке, т. е. возникает конфликт. Для решения проблем синхронизации необходимо использовать мьютекс, чтобы обеспечить монопольный доступ к длине массива и текущему минимальному количеству ошибок.

Вывод

В данном разделе была представлена информация о многопоточности и исследуемом алгоритме.

2 Конструкторский раздел

В этом разделе будет представлено описание используемых типов данных, а также схемы алгоритмов исправления орфографических ошибок.

2.1 Требования к программному обеспечению

Программа должна поддерживать два режима работы: режим массового замера времени и режим исправления введенного слова.

Режим массового замера времени должен обладать следующей функциональностью:

— генерировать корпус слов;

— осуществлять массовый замер, используя сгенерированные данные;

— результаты массового замера должны быть представлены в виде таблицы и графика.

К режиму исправления введенного слова выдвигается следующий ряд требований:

— возможность вводить слова, которые отсутствуют в корпусе;

— наличие интерфейса для выбора действий;

— на выходе программы, набор из самых близких к введенному слов.

2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры и типы данных:

— слово — массив букв;

— корпус — массив слов, отсортированный в лексикографическом порядке;

— мьютекс — примитив синхронизации.

2.3 Разработка алгоритмов

На рисунке 2.1 представлена схема поиска ближайших слов в корпусе без использования потоков. На рисунке 2.2 представлена схема поиска ближайших слов в корпусе с использованием потоков. На рисунке **??** представлена схема алгоритма программы, выполняющейся в потоке.

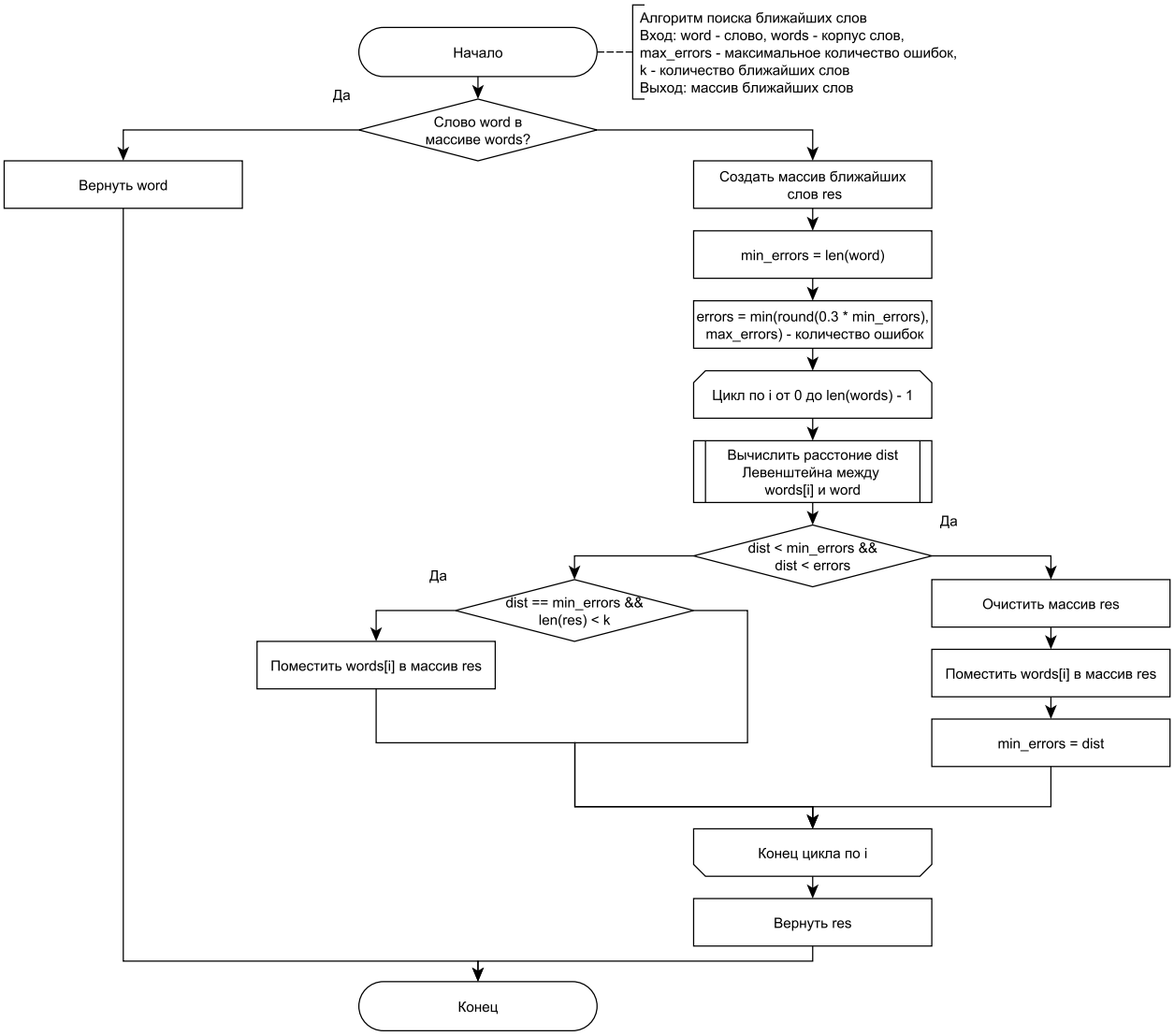

Рисунок 2.1 – Схема алгоритма поиска ближайших слов

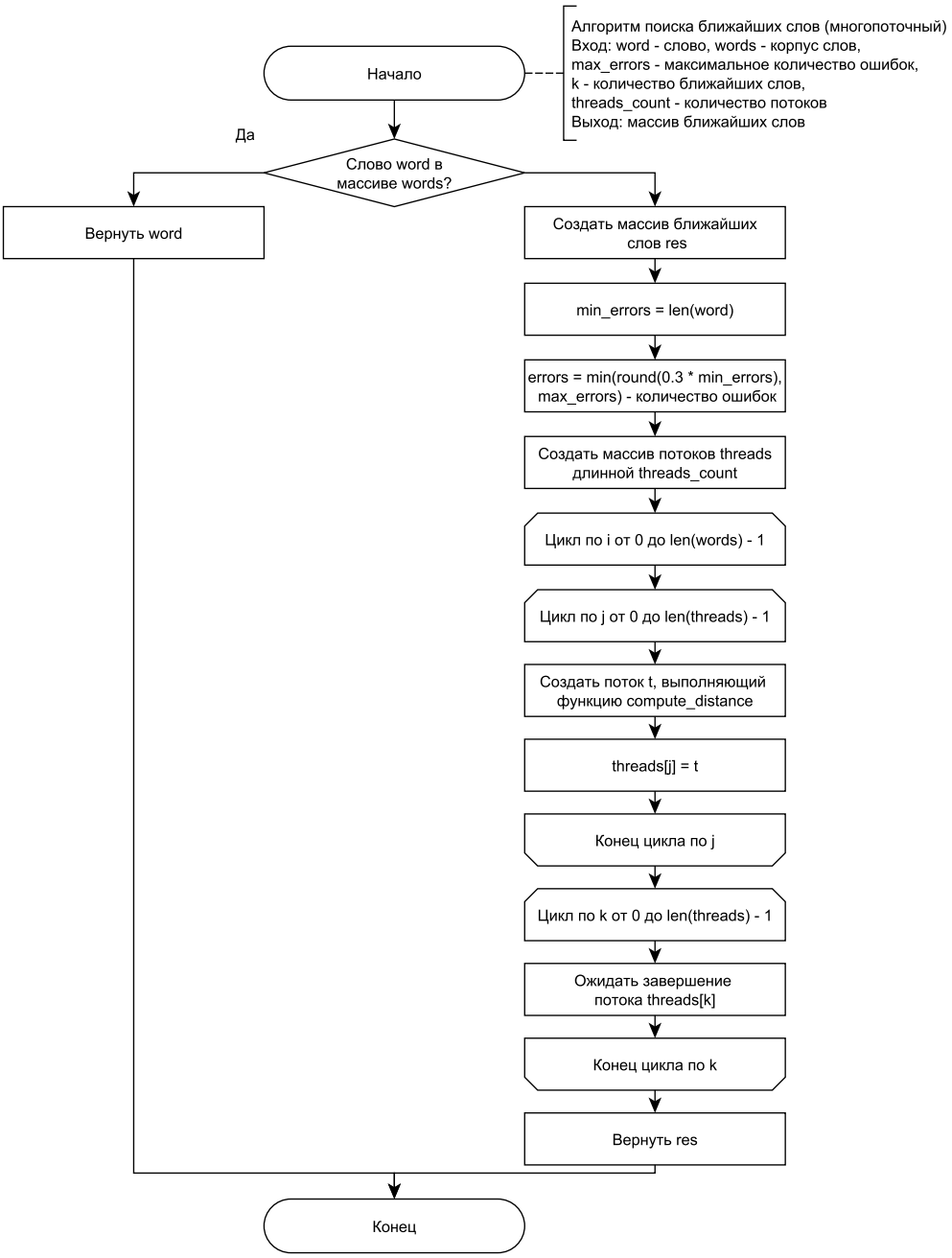

Рисунок 2.2 – Схема многопоточного алгоритма поиска ближайших слов

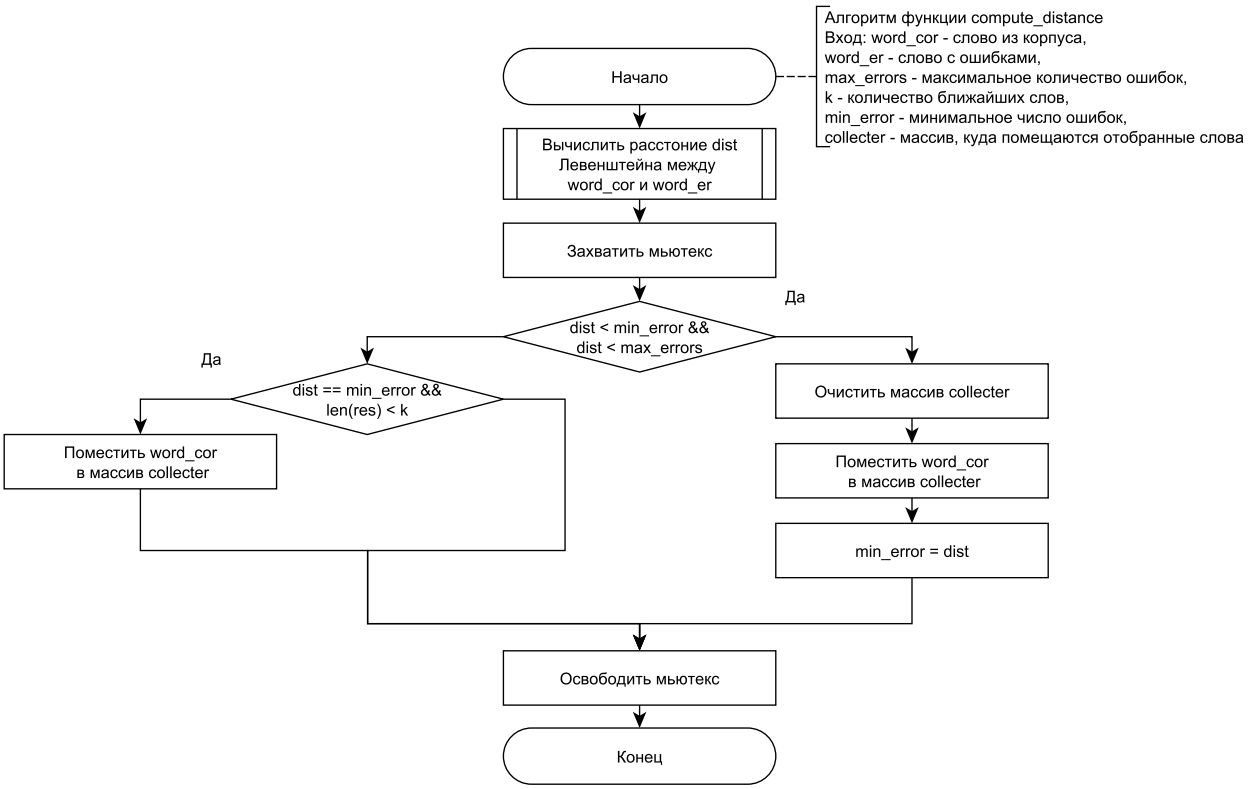

Рисунок 2.3 – Схема алгоритма потока

Поскольку в массиве в каждый момент времени содержатся слова, которые находятся на одинаковом расстоянии от введенного слова, то нет необходимости в какой-либо сортировке результатов, т. к. любое из найденных слов с одинаковой вероятностью может оказаться искомым. Поэтому шаг слияния результатов не требуется и достаточно помещать подходящее слово в массив.

Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

3 Технологический раздел

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

3.1 Средства реализации

Для реализации данной работы был выбран язык *C++* [5]. Данный выбор обусловлен следующим:

— язык поддерживает все структуры данных, которые выбраны в результате проектирования;

— язык позволяет реализовать все алгоритмы, выбранные в результате проектирования;

— язык позволяет работать с нативными потоками [6].

Время выполнения реализаций было замерено с помощью функции *clock* [7]. Для хранения слов использовалась структура данных *wstring* [8], в качестве массивов использовалась структура данных *vector* [9]. В качестве примитива синхронизации использовался *mutex* [10].

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— *main.cpp* — файл, содержащий функцию *main*;

— *correcter.cpp* — файл, содержащий код реализаций всех алгоритмов исправления ошибок;

— *measure_time.cpp* — файл, в котором содержатся функции для замера и вывода времени выполнения реализаций алгоритмов;

— *utils.cpp* — файл, в котором содержатся вспомогательные функции;

— *levenstein.cpp* — файл, в котором содержится реализация алгоритма поиска расстояния Левенштейна.

3.3 Реализация алгоритмов

В листинге 3.1 приведена реализация алгоритма исправления ошибок без дополнительных потоков. В листинге 3.2 приведена реализация алгоритма исправления ошибок с использованием дополнительных потоков. В листинге 3.3 приведена реализация функции, которая выполняется потоком.

Листинг 3.1 – Функция исправления ошибок

```
std::vector<std::wstring> get_closest_words(const
    std::vector<std::wstring> &words,
                                                const std::wstring
                                                    &word,
                                                size_t k,
                                                size_t max_errors){

    if (is_word_in_vec(words, word))
        return {word};

    std::vector<std::wstring> temp;
    size_t min = word.size();

    size_t errors = std::min(static_cast<size_t>(std::ceil(0.3 *
        word.size())), max_errors);

    for (const auto &cur_word: words){
        int dist = lev_mtr(cur_word, word);
        if (dist < min && dist <= errors){
            temp.clear();
            temp.push_back(cur_word);
            min = dist;
        }
        else if (dist == min && dist <= errors && temp.size() <
            k)
            temp.push_back(cur_word);
    }

    return temp;
}
```

Листинг 3.2 – Функция многопоточного исправления ошибок

```
std::vector<std::wstring> get_closest_words_mt(const
    std::vector<std::wstring> &words,
                                                   const
                                                      std::wstring
                                                      &word,
                                                   size_t k,
                                                   size_t
                                                      max_errors,
                                                  size_t
                                                     num_threads){

    if (is_word_in_vec(words, word))
        return {word};

    size_t min = word.size();
    size_t errors = std::min(static_cast<size_t>(std::ceil(0.3 *
       word.size())), max_errors);
    std::vector<std::wstring> collector;
    std::thread threads[num_threads];

    size_t l = 0;
    size_t created_threads = 0;

    while (l < words.size()) {
        created_threads = 0;
        for (size_t i = 0; i < num_threads; ++i) {
            threads[i] = std::thread(compute_distance,
                                     words[l],
                                     word,
                                     errors,
                                     k, std::ref(min),
                                     std::ref(collector));
            ++l;
            ++created_threads;
            if (l >= words.size())
                break;
        }

        for (size_t i = 0; i < created_threads; ++i) {
            threads[i].join();
```

```
35            }
36        }
37
38        return collector;
39    }
```

Листинг 3.3 – Функция, выполняющаяся в потоке

```
1  std::mutex mutex;
2
3  void compute_distance(const std::wstring &word_cor,
4                        const std::wstring &word_er,
5                        size_t errors,
6                        size_t k,
7                        size_t &min,
8                        std::vector<std::wstring> &collector)
9  {
10     int dist = lev_mtr(word_cor, word_er);
11
12     mutex.lock();
13     if (dist < min && dist <= errors){
14         collector.clear();
15         collector.push_back(word_cor);
16         min = dist;
17     }
18     else if (dist == min && dist <= errors && collector.size() <
          k)
19         collector.push_back(word_cor);
20     mutex.unlock();
21 }
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для разработанных алгоритмов исправления ошибок. Для данных тестов максимальное количество ошибок равно двум и из массива выбираются 3 слова. Все тесты пройдены успешно. В таблице 3.1 пустое слово обозначается с помощью λ.

Таблица 3.1 – Функциональные тесты

Корпус	**Слово**	**Ожидаемый результат**
[Mama, Myla, Ramu]	Mama	[Mama]
[Mama, Myla, Ramu]	mamy	[mama]
[Mama, Myla, Ramu]	myma	[mama, myla]
[Mama, Myla, Ramu]	ahtung	[ ]
[ ]	ahtung	[ ]
[ ]	λ	[ ]
[Мама, Мыла, Раму]	λ	[ ]

Вывод

Были разработаны и протестированы спроектированные алгоритмы исправления ошибок.

4 Исследовательский раздел

В данном разделе будут приведены: пример работы программы, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного программного обеспечения, а именно показаны результаты сортировки массива $[1, 6, 3, 2, 1, 3, 4]$.

Рисунок 4.1 – Демонстрация работы программы при сортировке массива

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, следующие:

— процессор: AMD Ryzen 5 4600H 3 ГГц [**amd**];

— оперативная память: 16 ГБайт;

— операционная система: Windows 10 Pro 64-разрядная система версии 22H2 [**windows**].

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.3 Время выполнения реализаций алгоритмов

Результаты замеров времени выполнения реализаций алгоритмов сортировок приведены в таблицах 4.1 – 4.3. Замеры времени проводились на массивах одного размера и усреднялись для каждого набора одинаковых экспериментов.

В таблицах 4.1 – 4.3 используются следующие обозначения:

— Ш — реализация алгоритма сортировки Шелла;

— Г — реализация алгоритма гномьей сортировки;

— П — реализация алгоритма пирамидальной сортировки.

Таблица 4.1 – Время работы реализации алгоритмов на массивах, отсортированных в обратном порядке (в с)

Размер массива	Ш	Г	П
1000	$5.937 \cdot 10^{-4}$	$3.125 \cdot 10^{-4}$	$3.031 \cdot 10^{-3}$
2000	$1.313 \cdot 10^{-3}$	$1.187 \cdot 10^{-3}$	$6.781 \cdot 10^{-3}$
3000	$2.281 \cdot 10^{-3}$	$2.594 \cdot 10^{-3}$	$1.109 \cdot 10^{-2}$
4000	$3.125 \cdot 10^{-3}$	$4.500 \cdot 10^{-3}$	$1.578 \cdot 10^{-2}$
5000	$4.281 \cdot 10^{-3}$	$6.938 \cdot 10^{-3}$	$1.975 \cdot 10^{-2}$
6000	$5.281 \cdot 10^{-3}$	$9.938 \cdot 10^{-3}$	$2.409 \cdot 10^{-2}$
7000	$6.094 \cdot 10^{-3}$	$1.341 \cdot 10^{-2}$	$2.859 \cdot 10^{-2}$
8000	$6.875 \cdot 10^{-3}$	$1.734 \cdot 10^{-2}$	$3.347 \cdot 10^{-2}$
9000	$8.531 \cdot 10^{-3}$	$2.234 \cdot 10^{-2}$	$3.806 \cdot 10^{-2}$

Таблица 4.2 – Время работы реализации алгоритмов на отсортированных массивах (в с)

Размер массива	Ш	Г	П
1000	$5.937 \cdot 10^{-4}$	$9.375 \cdot 10^{-5}$	$3.063 \cdot 10^{-3}$
2000	$1.500 \cdot 10^{-3}$	$2.188 \cdot 10^{-4}$	$6.938 \cdot 10^{-3}$
3000	$2.250 \cdot 10^{-3}$	$2.812 \cdot 10^{-4}$	$1.109 \cdot 10^{-2}$
4000	$3.187 \cdot 10^{-3}$	$4.062 \cdot 10^{-4}$	$1.528 \cdot 10^{-2}$
5000	$4.313 \cdot 10^{-3}$	$5.000 \cdot 10^{-4}$	$1.984 \cdot 10^{-2}$
6000	$5.125 \cdot 10^{-3}$	$5.625 \cdot 10^{-4}$	$2.409 \cdot 10^{-2}$
7000	$6.031 \cdot 10^{-3}$	$6.563 \cdot 10^{-4}$	$2.853 \cdot 10^{-2}$
8000	$6.594 \cdot 10^{-3}$	$8.125 \cdot 10^{-4}$	$3.369 \cdot 10^{-2}$
9000	$8.500 \cdot 10^{-3}$	$9.063 \cdot 10^{-4}$	$3.825 \cdot 10^{-2}$

Таблица 4.3 – Время работы реализации алгоритмов на случайно упорядоченных массивах (в с)

Размер массива	Ш	Г	П
1000	$1.969 \cdot 10^{-3}$	$4.687 \cdot 10^{-4}$	$8.484 \cdot 10^{-3}$
2000	$4.484 \cdot 10^{-3}$	$1.359 \cdot 10^{-3}$	$1.891 \cdot 10^{-2}$
3000	$7.547 \cdot 10^{-3}$	$2.687 \cdot 10^{-3}$	$3.008 \cdot 10^{-2}$
4000	$1.003 \cdot 10^{-2}$	$4.469 \cdot 10^{-3}$	$4.127 \cdot 10^{-2}$
5000	$1.358 \cdot 10^{-2}$	$6.734 \cdot 10^{-3}$	$5.317 \cdot 10^{-2}$
6000	$1.648 \cdot 10^{-2}$	$9.312 \cdot 10^{-3}$	$6.509 \cdot 10^{-2}$
7000	$1.914 \cdot 10^{-2}$	$1.198 \cdot 10^{-2}$	$7.742 \cdot 10^{-2}$
8000	$2.178 \cdot 10^{-2}$	$1.555 \cdot 10^{-2}$	$8.947 \cdot 10^{-2}$
9000	$2.667 \cdot 10^{-2}$	$1.956 \cdot 10^{-2}$	$1.035 \cdot 10^{-1}$

На рисунках 4.2 – 4.4 изображены графики зависимостей времени выполнения реализаций сортировок от размеров массивов.

Рисунок 4.2 – Сравнение реализаций алгоритмов по времени выполнения на массивах, отсортированных в обратном порядке

Рисунок 4.3 – Сравнение реализаций алгоритмов по времени выполнения на отсортированных массивах

Рисунок 4.4 – Сравнение реализаций алгоритмов по времени выполнения на случайно упорядоченных массивах

4.4 Характеристики по памяти

Введем следующие обозначения:

— n — длина массива, который необходимо отсортировать arr;

— $size()$ — функция, вычисляющая размер в байтах;

— int — целочисленный тип данных;

— $float$ — вещественный тип данных.

Максимальное требование по памяти реализации алгоритма Шелла складывается из 5 локальных переменных типа int, адреса возврата int, возвращаемого значения (ссылки) и рассчитывается по формуле (4.1).

$$f_{memshell} = 6 \cdot size(int) + size(float*). \tag{4.1}$$

Аналогично, реализация алгоритма гномьей сортировки максимально требует памяти под 2 локальные переменные типа int, адрес возврата int, возвращаемое значение (ссылку), т. о. требования по памяти рассчитываются по формуле (4.2).

$$f_{memgnome} = 3 \cdot size(int) + size(float*). \tag{4.2}$$

Максимальное требование реализации алгоритма пирамидальной сортировки по памяти формируется из 3 локальных переменных типа int, адреса возврата int, возвращаемого по ссылке значения, при этом максимальная глубина рекурсии подпрограммы *heapify* равна $\log_2 N$. В подпрограмме *heapify* используются 3 локальных переменных типа int, а для вызова в нее необходимо передать массив по ссылке и 2 переменные типа int. Память требуемая реализацией алгоритма пирамидальной сортировки рассчитывается по формуле (4.3).

$$f_{heap} = 3 \cdot size(int) + size(float*) + \log_2 N(6 \cdot size(int) + size(float*)). \tag{4.3}$$

Вывод

В результате замеров времени выполнения реализаций различных алгоритмов было выявлено, что для массивов длины 9000, отсортированных в обратном порядке, реализация алгоритма Шелла по времени оказалась в 2.6 раза лучше, чем реализация гномьей сортировки, и в 4.5 раза лучше реализации пирамидальной сортировки. В свою очередь, реализация гномьей сортировки оказалась лучше в 1.7 раз по времени выполнения, чем реализация пирамидальной сортировки. Что соответствует теоретической оценке трудоемкости. Поскольку алгоритм Шелла по теоретической оценке трудоемкости в худшем случае выигрывает по константе гномью сортировку, $O(\frac{32}{3}N^2)$ и $O(\frac{23}{2}N^2)$ соответственно, при этом алгоритм пирамидальной сортировки, обладая теоретической оценкой трудоемкости $O(\frac{87}{2}N\log_2 N)$, из-за большой константы проигрывает остальным, хоть и обладает меньшей скоростью роста.

Для отсортированных массивов длинной 9000 реализация гномьей сортировки оказалась лучше по времени в 9 раза, чем реализация алгоритма Шелла, и в 42 раза лучше, чем реализация пирамидальной сортировки. В свою очередь, реализация пирамидальной сортировки на отсортированных массивах, оказалась хуже в 4.5 раза, чем реализации алгоритма Шелла по времени выполнения. Что соответствует теоретической оценке трудоемкости. Поскольку в лучшем случае алгоритм гномьей сортировки обладает наименьшей асимптотической оценкой и малой константой $O(7N)$. А алгоритм Шелла выигрывает алгоритм пирамидальной сортировки по константе, $O(10N\log_2 N)$ и $O(29N\log_2 N)$ соответственно.

Для случайно упорядоченных массивов длинной 9000 реализация гномьей сортировки оказалась лучше по времени в 1.4 раза, чем реализация алгоритма Шелла, и в 5.3 раза лучше, чем реализация пирамидальной сортировки. В свою очередь, реализация пирамидальной сортировки на случайно упорядоченных массивах, оказалась хуже в 3.9 раз, чем реализации алгоритма Шелла по времени выполнения.

При этом меньше всего памяти требует реализация гномьей сортировки, а больше всего — реализация пирамидальной сортировки.

Стоит заметить, что для обратно упорядоченных массивов длинной менее 2500, реализация гномьей сортировки показывала лучшие результаты. На случайно упорядоченных массивах, гномья сортировка хоть и оказалась

эффективней, но обладая большей скоростью роста, при больших длинах массивов она окажется менее эффективной, чем сортировка Шелла и пирамидальная. То же касается и пирамидальной сортировки, за счет большой константы, она показала худший результат во всех случаях, но поскольку асимптотическая оценка этого алгоритма меньше остальных для случайно упорядоченных и обратно упорядоченных массивов, данная реализация покажет лучшую эффективность по времени при гораздо больших длинах массивов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Гладышев Е.И. М. А.* Многопоточность в приложениях // Актуальные проблемы авиации и космонавтики. — 2012. — № 8.
2. *Stoltzfus J.* Multithreading. — — Режим доступа: `https://www.techopedia.com/definition/24297/multithreading-computer-architecture` (дата обращения: 07.12.2023).
3. *У. Ричард Стивенс С. А. Р.* UNIX. Профессиональное программирование. 3-е издание //. — — СПб.: Питер, 2018. — С. 994.
4. *Большакова Е.И. К. Э.* Автоматическая обработка текстов на естественном языке и компьютерная лингвистика //. — М.: МИЭМ, 2011. — С. 122—124.
5. C++ reference [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/` (дата обращения: 20.12.2023).
6. Concurrency support library [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/thread` (дата обращения: 20.12.2023).
7. std::clock [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/chrono/c/clock` (дата обращения: 19.09.2023).
8. Strings library [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/string` (дата обращения: 19.09.2023).
9. std::vector [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/string` (дата обращения: 19.09.2023).
10. std::mutex [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/thread/mutex` (дата обращения: 19.09.2023).