



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"

Тема Многопоточное программирование

---

Студент Калашков П. А.

---

Группа ИУ7-56Б

---

Оценка (баллы)

---

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

---

# Содержание

<b>Введение</b>	<b>3</b>
<b>1 Аналитическая часть</b>	<b>4</b>
1.1 Многопоточность . . . . .	4
1.2 Паттерн thread pool . . . . .	5
<b>2 Конструкторская часть</b>	<b>7</b>
2.1 Требования к ПО . . . . .	7
2.2 Разработка алгоритмов . . . . .	7
<b>3 Технологическая часть</b>	<b>11</b>
3.1 Средства реализации . . . . .	11
3.2 Сведения о модулях программы . . . . .	11
3.3 Реализации алгоритмов . . . . .	12
3.4 Функциональные тесты . . . . .	15
<b>4 Исследовательская часть</b>	<b>16</b>
4.1 Технические характеристики . . . . .	16
4.2 Демонстрация работы программы . . . . .	16
4.3 Время выполнения алгоритмов . . . . .	18
<b>Заключение</b>	<b>22</b>
<b>Список литературы</b>	<b>23</b>

# Введение

По мере развития вычислительных систем программисты столкнулись с необходимостью производить параллельную обработку данных для улучшения отзывчивости системы, ускорения производимых вычислений и рационального использования вычислительных мощностей. Благодаря развитию процессоров стало возможным использовать один процессор для выполнения нескольких параллельных операций, что дало начало термину “многопоточность”.

**Цель работы:** изучение параллельного выполнения операций на примере сервера раздачи статической информации. Для достижения поставленной цели необходимо выполнить следующие задачи:

- 1) изучить основы распараллеливания вычислений;
- 2) разработать и реализовать программное обеспечение, позволяющее раздавать статическую информацию на локальном сервере согласно паттерну thread pool;
- 3) сравнить и проанализировать по времени обработки установленного количества поданных запросов с использованием многопоточности и без неё;
- 4) описать и обосновать полученные результаты в виде отчёта о выполненной лабораторной работе, выполненном как расчётно-пояснительная записка к работе.

# 1 Аналитическая часть

В этом разделе будет представлена информация о многопоточности, а также теоретически описан паттерн `thread pool`, реализация которого будет распараллелена в данной лабораторной работе.

## 1.1 Многопоточность

**Многопоточность** [1] (англ. *multithreading*) – это способность центрального процессора (ЦП) обеспечивать одновременное выполнение нескольких потоков в рамках использования ресурсов одного процессора. Поток – последовательность инструкций, которые могут исполняться параллельно с другими потоками одного и того же породившего их процесса.

Процессом [2] называют программу в ходе своего выполнения. Таким образом, когда запускается программа или приложение, создаётся процесс. Один процесс может состоять из одного или больше потоков. Таким образом, поток является сегментом процесса, сущностью, которая выполняет задачи, стоящие перед исполняемым приложением. Процесс завершается тогда, когда все его потоки заканчивают работу. Каждый поток в операционной системе является задачей, которую должен выполнить процессор. Сейчас большинство процессоров умеют выполнять несколько задач на одном ядре, создавая дополнительные, виртуальные ядра, или же имеют несколько физических ядер. Такие процессоры называются многоядерными.

При написании программы, использующей несколько потоков, следует учесть, что при последовательном запуске потоков и передаче управления исполняемому потоку не получится раскрыть весь потенциал многопоточности, т.к. большинство потоков будут существовать без дела. Необходимо создавать потоки для независимых, отнимающих много времени функций, и выполнять их параллельно, тем самым сокращая общее время выполнения процесса.

Одна из проблем, встающих при использовании потоков, является проблема совместного доступа к информации. Фундаментальным ограничением является запрет на запись из двух и более потоков в одну ячейку памяти

одновременно.

Из того следует, что необходим примитив синхронизации обращения к данным - так называемый мьютекс (англ. *mutex* - *mutual exclusion*). Он может быть захвачен для работы в монопольном режиме или освобождён. Так, если 2 потока одновременно пытаются захватить мьютекс, успевает только один, а другой будет ждать освобождения.

Набор инструкций, выполняемых между захватом и освобождением мьютекса, называется *критической секцией*. Поскольку в то время, пока мьютекс захвачен, остальные потоки, требующие выполнения критической секции, ждут освобождения мьютекса, требуется разрабатывать программное обеспечение таким образом, чтобы критическая секция был минимальной.

## 1.2 Паттерн thread pool

**Thread pool** [3] – архитектурный паттерн, предназначенный для достижения параллельности выполнения программы. Основная идея данного паттерна заключается в том, чтобы создавать набор (пул) потоков, из которого в случае появления задачи, которую можно выполнить параллельно с другими потоками, берётся поток. Задача выполняется в рамках забранного потока и после завершения освобождённый поток возвращается в пул.

Согласно поставленной задаче, реализации сервера для раздачи статической информации, можно сделать вывод, что задачами, которые можно выполнять параллельно, является обработка запроса. Поступившие запросы необходимо синхронно записывать в очередь запросов, после чего распределять задачи по имеющимся потокам.

## Вывод

В данной работе необходимо реализовать сервер раздачи статической информации с использованием архитектурного паттерна thread pool, а также провести сравнение времени обработки установленного количества запросов при использовании различного числа потоков. Требуется, чтобы сервер умел

отдавать информацию различного типа и размера, в том числе полноценные статические страницы, директории, вложенные файлы, а также пустые запросы.

## 2 Конструкторская часть

В данном разделе будут рассмотрены схема алгоритмов, реализующих сервер раздачи статической информации согласно паттерну thread pool с параллеливанием, а также без него.

### 2.1 Требования к ПО

Ряд требований к программе:

- сервер должен работать на локальном доменном имени на порту 8080;
- обращение к серверу происходит посредством браузера, запросы HTTP1.1;
- в ответ должен отдаваться статический файл, указанный в предыдущем пункте.

### 2.2 Разработка алгоритмов

На рисунках 2.1, 2.2 представлены схемы алгоритмов помещения запроса в очередь, а также обработки запроса в потоке. Заметим, что реализация сервера согласно данным схемам будет одинаковой как в случае использовании нескольких потоков, так и в случае отсутствия параллельности, в силу того, что даже непараллельная реализация является реализацией с использованием одного потока.

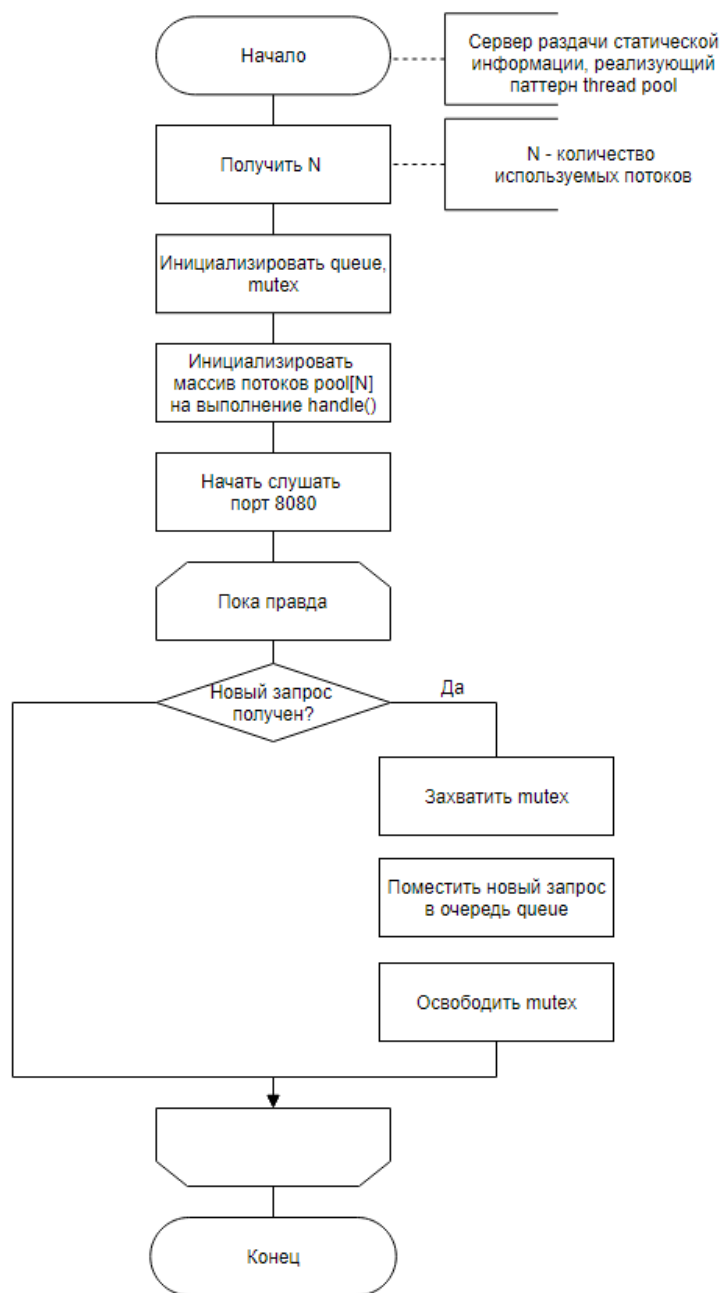


Рисунок 2.1 – Схема алгоритма сервера помещения запроса в очередь



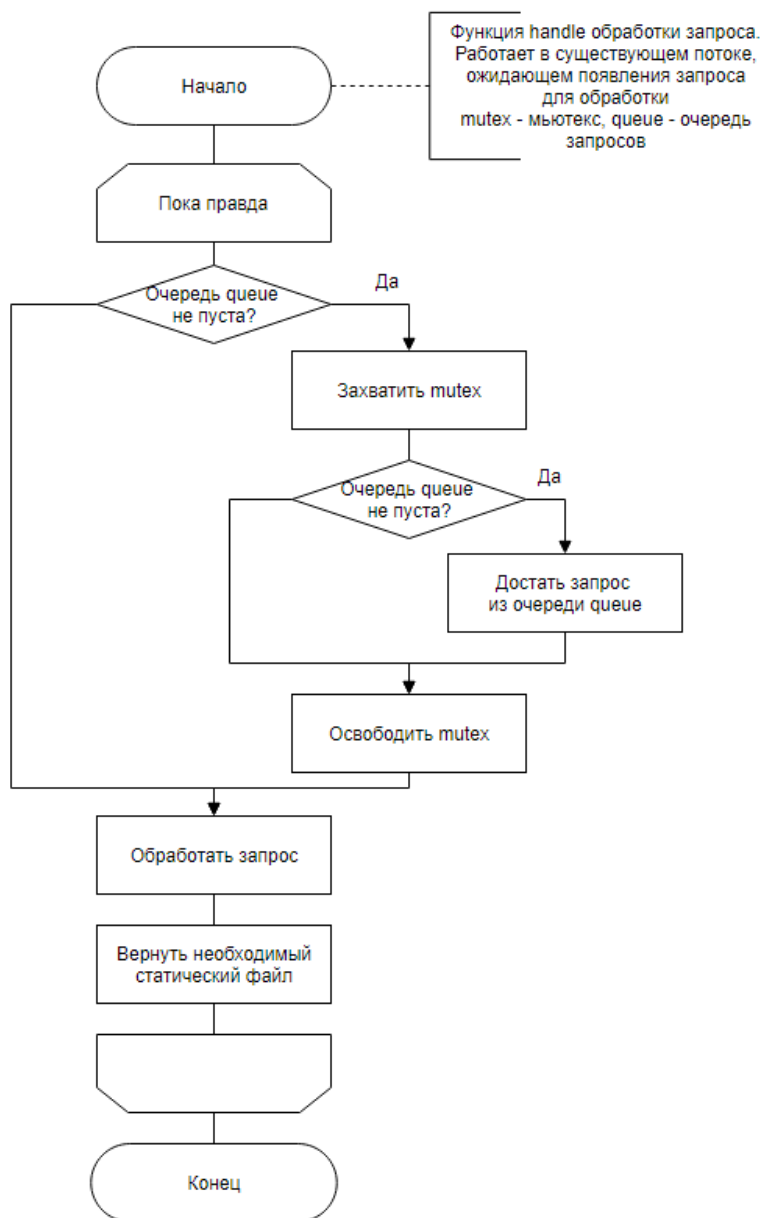


Рисунок 2.2 – Схема алгоритма обработки запроса в потоке

## Вывод

Были разработаны схемы всех алгоритмов помещения запроса в очередь, а также обработки и получения необходимого запроса из очереди в потоке. Критическая секция была снижена настолько, насколько было возможно, что позволит наиболее эффективно использовать многопоточность.

## 3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги реализаций рассматриваемых сортировок.

### 3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *C++*[4]. В текущей лабораторной работе требуется замерить процессорное время работы выполняемой программы и визуализировать результаты при помощи графиков. Инструменты для этого присутствуют в выбранном языке программирования.

Построение графиков и функциональное тестирование было осуществлено при помощи языка программирования *Python*[5], нагрузочное тестирование при помощи утилиты *wrk*[6], а время работы было замерено с помощью функции *process\_time(...)* из библиотеки *time*[7].

### 3.2 Сведения о модулях программы

Программа состоит из шести модулей:

- 1) *main* – файл, содержащий точку входа;
- 2) *Server* – модуль, содержащий реализацию сервера;
- 3) *utils* – модуль, содержащий служебные алгоритмы;
- 4) *constants.hpp* – файл, содержащий константы программы;
- 5) *responses.hpp* – файл, содержащий шаблоны ответов.

### 3.3 Реализации алгоритмов

В листингах 3.1, 3.2, 3.3 представлены реализации алгоритмов создания пула потоков, помещения запросов в очередь и обработки запроса в потоке

Листинг 3.1 – Заголовочный файл реализации сервера

```

1 class Server {
2     public:
3     Server() = delete;
4
5     explicit Server(size_t poolSize = POOL_SIZE, int port = PORT,
6         int countConn = CONN_COUNT)
7     : poolSize(poolSize), serverPort(port),
8       queueConnections(CONN_COUNT) {
9         this->serverSocket = 0, this->clientSocket = 0;
10        memset(&this->serverAddress, 0, sizeof(struct sockaddr_in));
11        memset(&this->clientAddress, 0, sizeof(struct sockaddr_in));
12        this->threadPool.resize(this->poolSize);
13        for(size_t i = 0; i < this->poolSize; i++) {
14            this->threadPool[i] =
15                std::thread([this]() { this->handleRequest(); });
16        }
17    };
18
19    int init();
20    void run();
21
22    private:
23    int serverSocket;
24    int clientSocket;
25    struct sockaddr_in serverAddress;
26    struct sockaddr_in clientAddress;
27    int serverPort;
28
29    std::vector<std::thread> threadPool;
30    size_t poolSize;
31    std::queue<int> requestQueue;
32
33    int queueConnections;
34    std::mutex queueMutex;
35    std::condition_variable cv;
36
37    void handleRequest();
38    int loadConfig();
39};

```

Листинг 3.2 – Запуск сервера и алгоритм помещения запросов в очередь

```

1 void Server::run() {
2     while (true) {
3         socklen_t addr_size = sizeof(this->clientAddress);
4         this->clientSocket = accept(serverSocket, (struct sockaddr
5             *) &this->clientAddress, &addr_size);
6         char peerIP[INET_ADDRSTRLEN] = {0};
7         if (inet_ntop(AF_INET, &this->clientAddress.sin_addr,
8             peerIP, sizeof(peerIP))) {
9             std::cout << "Accepted connection with " << peerIP <<
10                "\n";
11         } else {
12             return;
13         }
14         this->queueMutex.lock();
15         this->requestQueue.push(clientSocket);
16         this->cv.notify_one();
17         this->queueMutex.unlock();
18     }
19 }

```

Листинг 3.3 – Алгоритм обработки запроса в потоке

```

1 void Server::handleRequest() {
2     std::unique_lock<std::mutex> lock(this->queueMutex,
3         std::defer_lock);
4     int client_socket = -1;
5     while (true) {
6         lock.lock();
7         this->cv.wait(lock, [this]() { return
8             !this->requestQueue.empty(); });
9         client_socket = this->requestQueue.front();
10        this->requestQueue.pop();
11        lock.unlock();
12        char req[2 * REQ_SIZE];
13        recv(client_socket, req, sizeof(req), 0);
14        std::string reply = handle(req);
15        send(client_socket, reply.c_str(), reply.size(), 0);
16        close(client_socket);
17    }
18 }

```

### 3.4 Функциональные тесты

В таблице ?? приведены тесты для функций, реализующих алгоритмы сортировки. Тесты для всех сортировок пройдены *успешно*.

### Вывод

Были разработаны схемы всех трех алгоритмов сортировки. Для каждого алгоритма была вычислена трудоёмкость и оценены лучший и худший случаи.

## 4 Исследовательская часть

В данном разделе будут приведён пример работы программа, а также проведён сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

### 4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

- операционная система: Mac OS Monterey Версия 12.5.1 (21G83) [8] x86\_64;
- память: 16 GB;
- процессор: 2,7 GHz 4-ядерный процессор Intel Core i7 [9].

При тестировании ноутбук был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

### 4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.



```
Меню
1. Блинная сортировка
2. Пирамидная сортировка
3. Сортировка бинарным деревом
4. Замер времени
0. Выход

Выбор: 2
Введите массив поэлементно в одной строке (окончание - Enter):
1 2 3 4 5 5 4 3 2 1
[1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
Меню
1. Блинная сортировка
2. Пирамидная сортировка
3. Сортировка бинарным деревом
4. Замер времени
0. Выход

Выбор: █
```

Рисунок 4.1 – Пример работы программы

## 4.3    **Время                      выполнения                      алгоритмов**

Как было сказано выше, используется функция замера процессорного времени `process_time(...)` из библиотеки `time` на Python. Функция возвращает пользовательское процессорное время типа `float`.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Результаты замеров времени работы алгоритмов сортировки на различных входных данных (в мс) приведены в таблицах 4.1, 4.2 и 4.3.

Таблица 4.1 – Отсортированные данные

Размер	Блинная	Поразрядная	Бинарным деревом
100	0.1662	0.0714	0.8730
200	0.5113	0.2058	3.3267
300	1.1026	0.3131	7.6354
400	2.0140	0.4364	13.6751
500	3.3046	0.5591	21.5524
600	5.0567	0.6798	31.3052
700	6.6944	0.7852	43.0406
800	8.5163	0.8766	56.4318

Таблица 4.2 – Отсортированные в обратном порядке  
данные

Размер	Блинная	Поразрядная	Бинарным деревом
100	0.1606	0.1048	0.7138
200	0.5005	0.2008	2.7633
300	1.0747	0.3110	6.3060
400	1.9383	0.4312	11.3831
500	3.1148	0.5427	18.0577
600	4.6409	0.6693	26.0260
700	6.7969	0.8317	36.7397
800	8.7922	0.9583	47.2628

Также на рисунках 4.2, 4.3, 4.4 приведены графические результаты замеров работы сортировок в зависимости от размера входного массива.

Таблица 4.3 – Случайные данные

Размер	Блинная	Поразрядная	Бинарным деревом
100	0.2734	0.1043	0.1560
200	0.8321	0.2090	0.3756
300	1.6837	0.3142	0.6025
400	2.8938	0.4281	0.9785
500	4.4438	0.5419	1.1784
600	6.4153	0.6704	1.5523
700	8.6692	0.7678	1.9018
800	11.3752	0.8992	2.2986

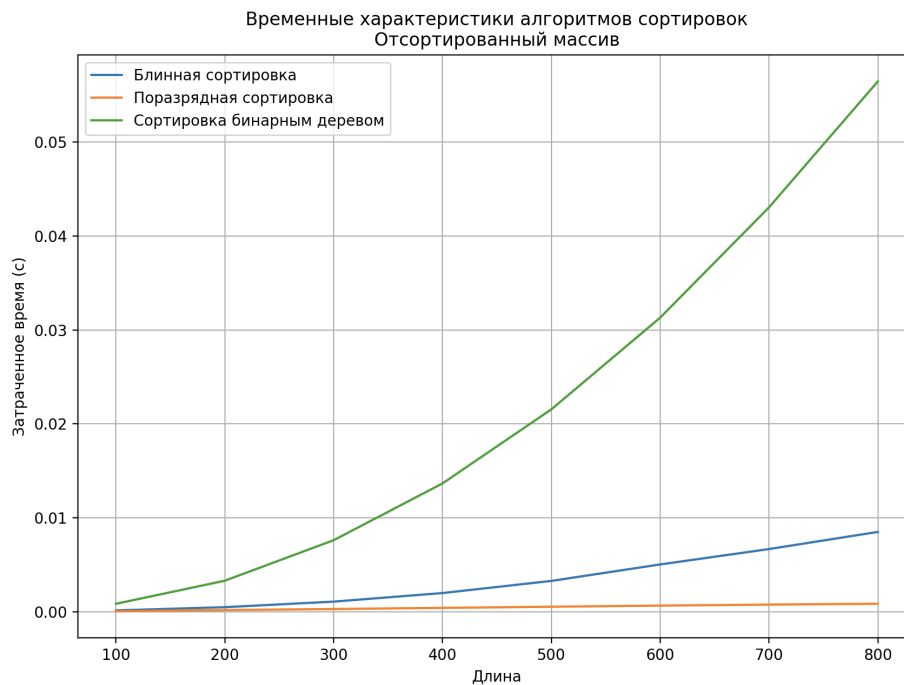


Рисунок 4.2 – Отсортированный массив

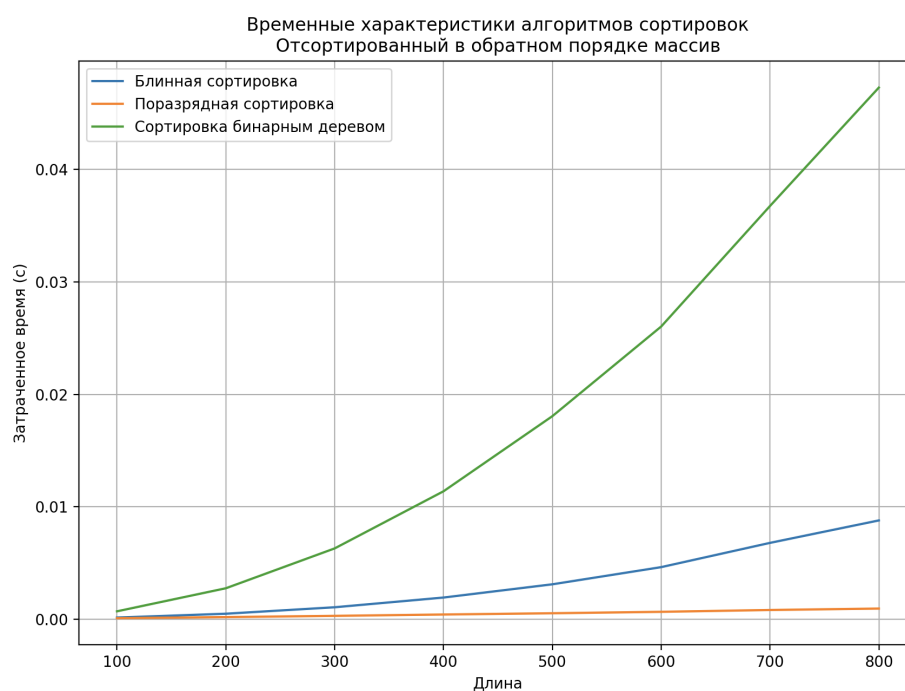


Рисунок 4.3 – Отсортированный в обратном порядке массив

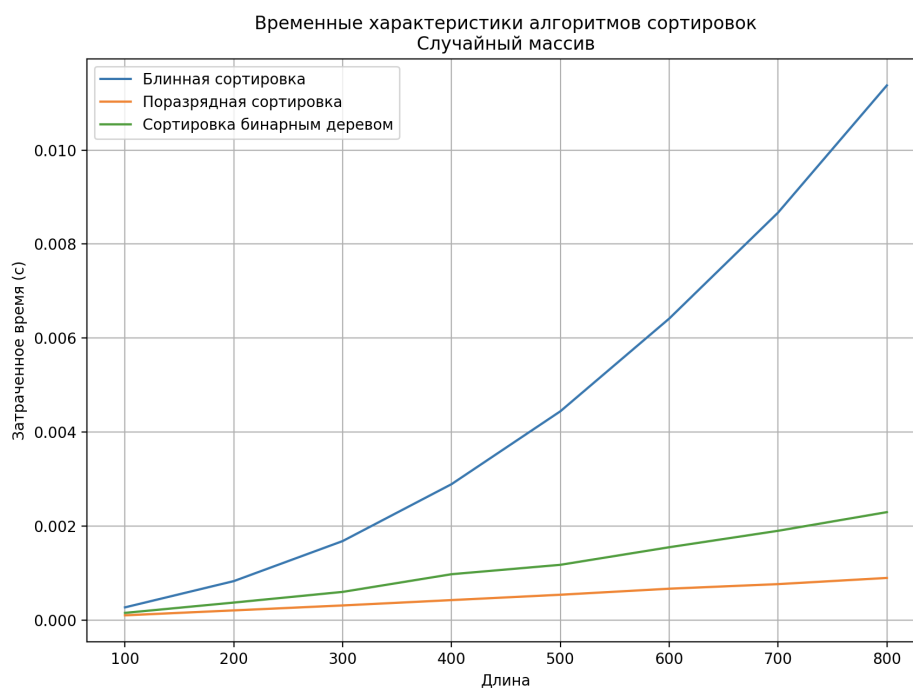


Рисунок 4.4 – Случайный массив

## Вывод

Исходя из полученных результатов, сортировка бинарным деревом на отсортированных массивах и блинная сортировка на случайном массиве работают дольше всех (примерно в 40 раз дольше, чем поразрядная сортировка), при этом поразрядная сортировка показала себя лучше всех на любых данных. Можно сделать вывод, что использование сортировки бинарным деревом показывает наилучший результат при случайных, никак не отсортированных данных, т.к. при отсортированных данных обычное бинарное дерево вырождается в связный список, из-за чего вырастает высота дерева. Поразрядная сортировка же эффективнее в том случае, когда приблизительно известно максимальное количество разрядов в сортируемых данных.

Теоретические результаты замеров и полученные практически результаты совпадают.

# Заключение

Цель, которая была поставлена в начале лабораторной работы была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи:

- 1) были изучены и реализованы алгоритмы сортировки: блонная, поразрядная и бинарным деревом;
- 2) была выбрана модель вычисления и проведен сравнительный анализ трудоёмкостей выбранных алгоритмов сортировки;
- 3) на основе экспериментальных данных проведено сравнение выбранных алгоритмов сортировки;
- 4) подготовлен отчёт о лабораторной работе, представленный как расчётно-пояснительная записка к работе.

Исходя из полученных результатов, сортировка бинарным деревом на отсортированных массивах и блонная сортировка на случайном массиве работают дольше всех (примерно в 40 раз дольше, чем поразрядная сортировка), при этом поразрядная сортировка показала себя лучше всех на любых данных. Можно сделать вывод, что использование сортировки бинарным деревом показывает наилучший результат при случайных, никак не отсортированных данных, т.к. при отсортированных данных обычное бинарное дерево вырождается в связный список, из-за чего вырастает высота дерева. Поразрядная сортировка же эффективнее в том случае, когда приблизительно известно максимальное количество разрядов в сортируемых данных.

# Список

# литературы

- [1] Stoltzfus J. Multithreading. – Techopedia - Janalta Interactive Inc., 2022. URL: <https://www.techopedia.com/definition/24297/multithreading-computer-architecture>.
- [2] У. Ричард Стивенс Стивен А. Раго. UNIX. Профессиональное программирование. 3-е издание. – СПб.: Питер., 2018. С. – 994.
- [3] Christudas B. Query by Slice, Parallel Execute, and Join: A Thread Pool Pattern in Java. – Infosys Technologies, 2008. URL: <https://web.archive.org/web/20080207124322/http://today.java.net/pub/a/today/2008/01/31/query-by-slice-parallel-execute-join-thread-pool-pattern.html>.
- [4] Программирование на C/C++ [Электронный ресурс]. Режим доступа: <http://www.c-cpp.ru/> (дата обращения: 11.10.2021).
- [5] Welcome to Python [Электронный ресурс]. Режим доступа: <https://www.python.org> (дата обращения: 11.10.2022).
- [6] Утилита wrk [Электронный ресурс]. Режим доступа: <https://linuxcommandlibrary.com/man/wrk> (дата обращения: 11.10.2022).
- [7] time — Time access and conversions [Электронный ресурс]. Режим доступа: <https://docs.python.org/3/library/time.html#functions> (дата обращения: 04.10.2021).
- [8] macOS Monterey [Электронный ресурс]. Режим доступа: <https://www.apple.com/macos/monterey/> (дата обращения: 11.10.2022).
- [9] Процессор Intel® Core™ i7 [Электронный ресурс]. Режим доступа: <https://www.intel.com/processors/core/i7/docs> (дата обращения: 17.09.2022).