Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

Кафедра информационных систем и технологий

Дисциплина: Основы алгоритмизации и программирования (ОАиП)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

на тему

«АНАЛИЗ АЛГОРИТМА БИНАРНОГО ПОИСКА»

Студент:

гр. 381574 Жгуновский О.Б.

Руководитель:

ассистент кафедры ИСиТ ИИТ БГУИР Потоцкий Д.С.

Минск 2024

Содержание

[Введение 3](#_Toc164968003)

[1. Моделирование программного продукта 4](#_Toc164968004)

[2. Программная реализация 8](#_Toc164968005)

[3. Тестирование 18](#_Toc164968006)

[Выводы 23](#_Toc164968007)

[Список используемой литературы 25](#_Toc164968008)

[Приложение А 26](#_Toc164968009)

# Введение

Бинарный поиск является одним из наиболее эффективных алгоритмов поиска в упорядоченном массиве данных. Этот метод, основанный на принципе деления пополам, широко применяется в различных областях информатики и программирования, где требуется быстрый поиск элементов в больших объемах данных.

Цель данной курсовой работы заключается в анализе алгоритма бинарного поиска, его моделировании, проектировании, а также оценке работы программного средства, реализующего данный алгоритм на языке программирования C#. В ходе работы будут рассмотрены основные принципы функционирования алгоритма, его преимущества и недостатки, а также проведено тестирование разработанного программного продукта с последующим анализом полученных результатов.

Задачами курсовой работы являются:

* Рассмотреть принципы работы алгоритма бинарного поиска;
* Реализация алгоритма бинарного поиска на языке программирования C#;
* Тестирование реализованного алгоритма;
* Понять суть работы алгоритма бинарного поиска.

Особое внимание будет уделено оценке работы программного средства и анализу результатов. Это включает в себя тестирование программы на различных наборах данных и анализ времени выполнения.

В конечном итоге, данная работа поможет получить глубокое понимание принципов работы алгоритма бинарного поиска, его применимости и эффективности, а также обеспечит практические навыки реализации и использования данного алгоритма для решения различных задач в области программирования и информационных технологий.

# Моделирование программного продукта

Алгоритм бинарного поиска — эффективный метод поиска элемента в отсортированном массиве данных. Данный алгоритм работает путём деления массива пополам, и последующего сравнения искомого значения с элементом в середине массива. Если значение совпадает с искомым — алгоритм завершается. Если же нет — алгоритм, путём сравнения искомого значения со значениями по бокам центрального элемента массива, определяет, в какой половине массива содержится искомое значение, и повторяет вышеописанные действия до тех пор, пока искомый элемент не будет найден или не останется больше элементов для проверки.

Если описывать алгоритм по шагам, это можно сделать следующим образом:

1. Получаем массив данных;
2. Сортируем массив данных;
3. Определяем левую и правую границы поиска;
4. Находим средний элемент, между границами поиска;
5. Сравниваем средний элемент с искомым значением;
6. Если искомое равно среднему элементу, заканчиваем поиск;
7. Если искомое больше среднего элемента, устанавливаем левую границу на значение среднего элемента;
8. Если искомое меньше среднего элемента, устанавливаем правую границу на значение среднего элемента;
9. Повторяем пункты 4-8 до тех пор, пока значение среднего элемента не будет совпадать с искомым (что отражено в пункте 6), либо же, когда левая граница становится больше правой (это означает, что элемент не найден в массиве);
10. Возвращаем результат.

Блок-схема алгоритма бинарного поиска, представлена на рисунке 1.1:

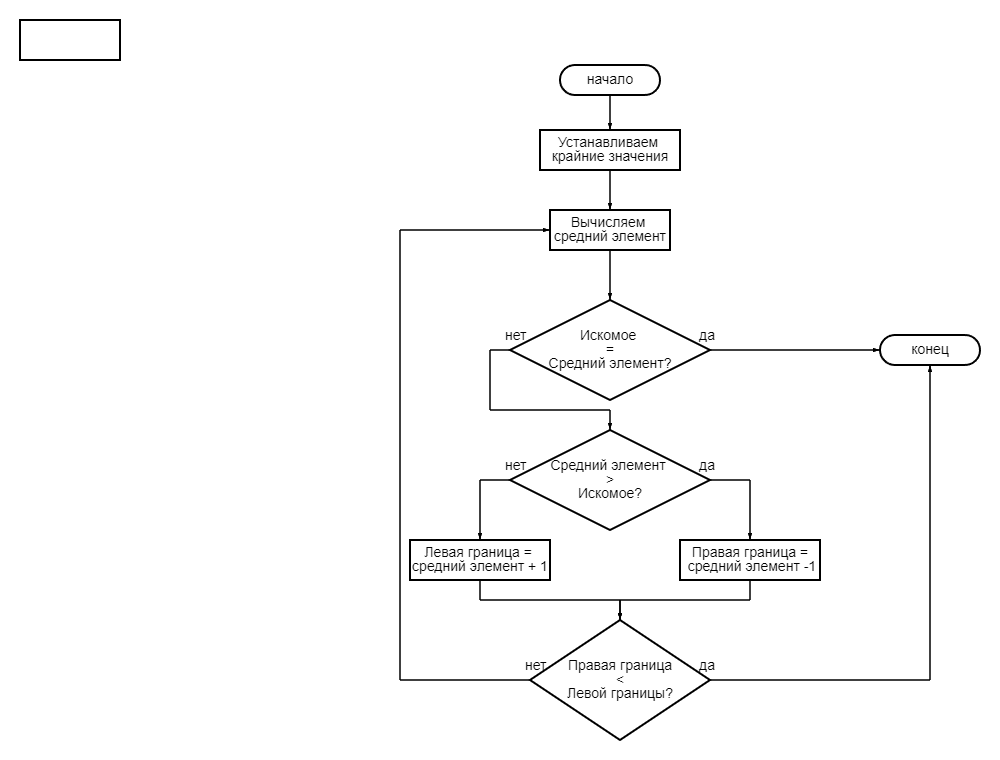


Рисунок 1.1 – блок-схема алгоритма бинарного поиска

Важной особенностью данного алгоритма является то, что он требует отсортированный массив. Соответственно, без выполнения данного условия, работать он вообще не будет.

Алгоритм обладает сложностью O(log n) в худшем и среднем случаях[1], где n — количество элементов в массиве. Что делает данный алгоритм, одним из самых эффективных при работе с большими объемами данных (если они, разумеется, отсортированы).

Данный алгоритм относительно прост в реализации, и тратит мало памяти — O(1).

Для того, чтобы можно было провести анализа, необходимо выбрать другой алгоритм поиска. И мною был выбран алгоритм линейного поиска.

Линейный поиск — это простой алгоритм поиска элемента в списке, массиве или последовательности. Работает он, путем последовательного перебора каждого элемента в структуре данных, и сравнения его с целевым значением. Если элемент совпадает с целевым значением, то алгоритм возвращает его позицию, если нет - продолжает. Эффективность сильно зависит от размера списка: время выполнения линейного поиска пропорционально количеству элементов в списке. Иными словами, его сложность будет составлять О(n)[2].

Стоит отметить, что, он гораздо проще в реализации нежели бинарный поиск, и тратит тоже минимум памяти — O(1).

Так же, стоит отметить, что в отличии от алгоритма поиска, нет необходимости получать отсортированный массив. Однако, так как проверки теряют всякий смысл при разных входных данных, мы будем проводить поиск в отсортированном массиве.

Тип данных следует использовать тот же, что и для алгоритма бинарного поиска. По тем же причинам, а так же, для корректности результатов самого теста

Если описывать его по шагам, то выйдет следующее:

1. Получаем массив данных;
2. Сортируем массив данных;
3. Ставим счётчик в i = 0;
4. Проверяем, чтобы элемент массива с индексом i, был равен искомому числу;
5. Если элемент массива с индексом i равен искомому числу, завершаем алгоритм и выводим результат;
6. Если элемент массива с индексом i не равен искомому числу, то убеждаемся, что мы не дошли до конца массива, и увеличиваем значение i на 1, после чего, возвращаемся к шагу 4;
7. Если же мы дошли до конца массива, то возвращаем информацию о том, что искомого элемента нету в массиве.

Блок-схема алгоритма линейного поиска представлена на рисунке 1.2:

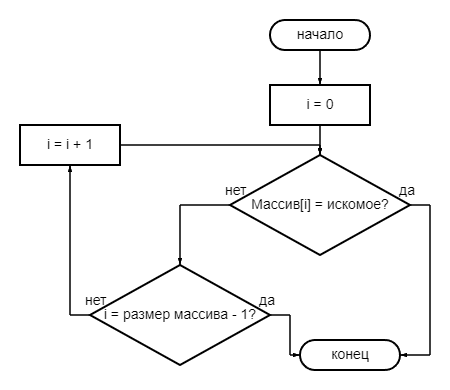


Рисунок 1.2 – блок-схема алгоритма линейного поиска

# Программная реализация

Для выполнения алгоритмов, программа должна получить следующие данные:

* Отсортированные массивы;
* Значения, которое необходимо найти.

А так же, если значение, которое необходимо найти в массиве отсутствует, то алгоритм должен это указать после своего выполнения.

Так как бинарный поиск может показать свою результативность только на больших массивах, и так как для получения каких-либо результатов, которые имеют значение требуется большой набор данных, предоставлять пользователю ручной ввод массивов просто отсутствует. А значит, стоит предоставить пользователю возможность выбрать размер массивов, и искомые значения.

Идя дальше, стоит отметить, что рандомизация значений в массивах, буквально, не является хоть сколько-нибудь необходимой. Реальная необходимость в рандомизации может возникнуть, если вы хотите имитировать работу со случайными данными или если природа данных предполагает случайный порядок. Однако, учитывая специфику работы алгоритма бинарного поиска, который предполагает работу с упорядоченными данными, рандомизация может оказаться излишней.

Вместо этого, массив проще заполнить числами от 1 до n, где n - номер ячейки массива, увеличенный на один. Такой подход обеспечит упорядоченный набор данных, что соответствует требованиям алгоритма бинарного поиска. Это также упростит процесс тестирования и анализа результатов, так как позволит легко отслеживать порядок элементов в массиве и оценивать производительность алгоритма на упорядоченных данных. Кроме того, это позволит нам упростить код программы, и уменьшить время её работы (иными словами – ускорить), так как в таком случае, пропадает необходимость в реализации алгоритма сортировки, как и естественно, его запуска.

Таким образом, в данном контексте, заполнение массива упорядоченными значениями, кажется, является более логичным, а главное – более эффективным подходом, нежели рандомизация значений. Это может способствовать более точному анализу производительности алгоритма бинарного поиска, как и ускорение работы программы.

Так как «сила» бинарного поиска в поиске по большим отсортированным массивам, а линейного – в маленьких и неотсортированных, стоит сделать массивы разной длины. Таким образом, будет создаваться набор массивов, с увеличивающимся размером от 100, до размера, который ввёл пользователь. Шаг будет составлять 100 позиций.

Так же, стоит пользователю дать возможность выбрать массив чисел для поиска, а не просто одно число.

После отработки алгоритма, необходимо показать пользователю минимальное, максимальное и среднее время для каждого алгоритма.

Для разработки был выбран объектно-ориентированный язык программирования C#[3], который имеет довольно много преимуществ, но главным и критическим для меня, является его удобный синтаксис, стабильная работа и соответствие требования технического задания.

И так, отмечая его положительные качества, хочется заметить, что C# имеет чистый и интуитивно понятный синтаксис, который делает ваш код более читабельным и простым в поддержке. Данный его аспект, особенно важен для разработчиков, работающих над большими проектами или в командах, где читаемость и ясность кода имеют первостепенное значение.

Ещё нельзя не заметить тот факт, что это крайне стабильный и надежный язык программирования, широко используемый для разработки различных типов приложений, включая веб-приложения, мобильные приложения, игры и другие. Это делает его привлекательным вариантом для разработки программного обеспечения любого размера и сложности. Что в свою очередь, для нас, означает если не желательность его изучения, то как минимум то, что его знание будет нам весьма полезно. А так как, глобально говоря, целью данной работы является всё же приобретение знаний, данный пункт можно смело относить к плюсам.

Пожалуй, последний фактор, который мне хочется выделить, это – соответствие языка C# требованиям технического задания. C# имеет широкий спектр возможностей и функций, которые позволяют реализовать различные требования проекта.

В качестве среды разработки будет использоваться Microsoft Visual Studio. Причина выбора кроется в том, что данная IDE является самой удобной и «правильной» средой разработки на языке программирования C#[4], из мне известных.

Кроме того, стоит отметить, что Microsoft Visual Studio предоставляет пользователям весьма широкий набор инструментов и функций для разработки приложений на языке C#. Она обеспечивает удобный, а так же, интуитивно понятный интерфейс, что значительно упрощает процесс разработки и увеличивает продуктивность разработчика.

Более того, Visual Studio интегрируется с множеством других продуктов и сервисов от Microsoft, такими как Azure, GitHub и другими, что обеспечивает удобство работы над проектами, интеграцию с системами контроля версий и облачными сервисами. Безусловно, некоторые, я бы даже сказал, что большинство из продуктов и сервисов в процессе разработки данной программы использоваться не будут, но тем не менее, опыт разработки в таком мощном инструменте никогда не будет лишним.

Visual Studio, также, имеет весьма мощные инструменты для отладки кода, создания профилирования производительности и автоматизации процесса разработки, что значительно упрощает решение сложных задач и обеспечивает как качественную разработку программного обеспечения, так и высокое качество итогового продукта.

Наконец, Microsoft Visual Studio – одна из самых популярных сред разработки, которая используется для работы с языком программирования C#. Присоединившись к большому сообществу разработчиков, вы сможете найти поддержку, учебные материалы и решения своих проблем. В результате чего, можно снова отметить, что навыки работы с данным мощным и востребованным инструментом крайне необходимы, и тренировать их – не будет лишним.

Для реализации данного алгоритма, мною было выбран тип данных «массив» (array), и не просто так, на это имеются три причины.

Во-первых, это наиболее простой и понятный тип данных, включённый в стандартную библиотеку C#. Его простота в использовании делает его привлекательным выбором для реализации алгоритмов бинарного и линейного поиска, особенно для разработчиков, не имеющих как опыта работы с более сложными структурами данных, так и желания погружаться в их изучение без явной необходимости в конкретной ситуации.

Во-вторых, он обеспечивает прямой доступ к элементам по индексу, что означает, что мы можем легко обращаться к любому элементу массива, зная его индекс. Это важно опять-таки для обеих алгоритмов, так как можно в случае с линейным поиском нам не нужно «городить огород», и придумывать что возвращать в результате поиска, а в случае с бинарным поиском нам важно не только предыдущее утверждение (про придумывание разных маркеров для возврата в случае нахождения, или не нахождения значения), но и тот факт что так как он оперирует с отсортированным набором он требует доступа к элементам по их индексу для сравнения с целевым значением, а так же для удобного движения по самому массиву и поиску центральных элементов.

В третьих, массивы имеют фиксированную длину, которая задаётся при их инициализации. С одной стороны это сильно ограничивает нас, с другой - этого достаточно для реализации данных алгоритмов. Тем более, что алгоритм работает с отсортированным набором данных (бинарный делает это по причине того, что иначе он работать просто не будет, а линейный – за компанию. В некотором роде, для достоверности результатов) и не требует динамического изменения размера массива в процессе выполнения. Таким образом, фиксированная длина массива полностью удовлетворяет требованиям алгоритма.

Короче говоря, выбор массива как типа данных, для реализации алгоритма бинарного и линейного поиска, обоснован его простотой в использовании, доступом к элементам по индексу и, в некотором роде, его фиксированной длиной, что делает его как минимум подходящим, я бы даже сказал эффективным типом данных для данной задачи.

В данном проекте используется 2 пространства имён: System и System.Diagnostics.

System даёт нам возможность использовать важнейший для консольного приложения объект – System.Console, так же, необходимый в данном проекте System.Math.

System.Console — это класс, предоставляющий возможность работы с консолью. Данный класс содержит методы для чтения ввода с консоли, вывода информации на консоль, управления цветом фоном и цветом текста в консоли, а также другие функции для взаимодействия с пользователем через консольное окно. В данном проекте используется множество функций данного класса, среди которых есть:

* Write() – выводит текст в консоль без перехода на новую строку;
* WriteLine() – выводит текст на консоль с переходом на новую строку;
* ReadLine() – считывает строку текста;
* ForegroundColor() – устанавливает цвет текста;
* ResetColor() – сбрасывает цвет как текста, так и фона текста на значения по умолчанию;
* Clear() – очищает содержимое консоли.

System.Math – это класс, который предоставляет методы для выполнения математических операций. Этот класс содержит ряд стандартных математических функций, из которых в данном проекте используется только метод Abs().

Метод System.Math.Abs() возвращает абсолютное значение числа, то есть, его значение без учета знака. Например, абсолютное значение числа -5 равно - 5, а абсолютное значение числа 7 равно - 7.

Естественно, в данное пространство имён входят и другие объекты, такие как[5]:

* Threading – для работы с потоками;
* Text – для работы с текстом;
* IO – для работы с файловой системой.

Но в данном проекте используется они не используются.

System.Diagnostics предоставляет инструменты для отладки программного кода. Как и в случае System, мы не используем все возможности данного пространства имён. И не смотря на то, что данное пространство имён предоставляет, например, следующие классы[6]:

* Process – для работы с процессами;
* PerformanceCounter – для получения статистики о производительности системы;
* EventLog – для работы с журналом событий ОС Windows;
* Debug и Trace – для вывода отладочной информации во время исполнения приложения;
* TraceListener – для перенаправления вывода отладочной информации.

Из всего содержимого System.Diagnostics, нам необходим только Stopwatch, для замера производительности. Выбор Stopwatch в качестве таймера обоснован его высокой точностью, а так же, соответствия основной цели существования инструмента – измерения времени выполнения операций. То бишь, не вижу смысла изгаляться и использовать функционал непредназначенный для чего-то, если есть что-то, что полностью соответствует требованиям.

А если рассказывать более подробно, то использование данного инструмента можно дополнительно подчеркнуть тем, что нам необходимы достоверные данные о времени выполнения операций. А в отличие от других изученных мною методов замера времени, System.Diagnostics.Stopwatch не зависит от различных факторов, таких как частота обновления системного таймера или, например, нагрузка на процессор. Это обеспечивает надежность и стабильность измерений.

В общем, специализация Stopwatch делает его для нас идеальным инструментом для данной задачи. Использование иных методов или инструментов, не предназначенных специально для сей задачи, может привести к несовершенным, неточным результатам. Поэтому использование Stopwatch является логичным, правильным и рациональным выбором для данной ситуации.

В коде мною было реализовано несколько классов, на которые необходимо обратить внимание, а так же, разобрать их.

Класс Program, содержит точку входа и 3 функции.

Функция Init, предназначена для инициализации массива, в котором происходит поиск, и его заполнения. Делает она это с помощью класса JaggedArray, речь о котором пойдёт ниже.

Функция SelectSearchableValues, предназначена для инициализации массива, который содержит данные, которые алгоритм будет искать во время тестов.

И функция StartBenchmark, запускает сами алгоритмы поиска, а так же, выводит результаты. Делает она это с помощью класса Benchmark.

JaggedArray представляет из себя класс, который ответственен за создание и наполнение массива.

В конструкторе происходит оповещение о создании массива, вывод его содержимого, если количество элементов меньше 100, и вызов методов для его создания.

ValidateSize предназначен для определения размеров самого большого массива, а так же, для ограничения размера в 700000 ячеек. Сделано это по той причине, что во время теста обнаруживалось, что значения больше, вызывают переполнение оперативной памяти, в результате чего – персональный компьютер намертво зависает.

ArraysAmount, считает количество массивов, которые нужно создать.

FillArray делает именно то, что указано в его названии – заполняет массивы числами начиная с единицы, и увеличивая значение на один.

А функция Print, предназначена для вывод в консоль содержимого массива в случае, если его размер меньше 100.

Цель класса ConsoleManipulator. – выводить в консоль сообщения в цветном виде. То бишь, он предназначен для подсветки текста в определённых отладочных сообщениях.

Так, методы ShowPositiveMessage, ShowInfoMessage и ShowNegativeMessage, выводят сообщения с зелёным, синим и красным шрифтом соответственно. А метод ShowMessage – вызывается предыдущими методами, выводит сообщение, и ставит стандартный цвет.

Особняком стоит метод CLSAfterKeydown. Он предназначен для вывода сообщения «Нажмите кнопку чтобы продолжить...», и очистки окна консоли после, соответственно, нажатия любой кнопки.

Класс Search предназначен для запроса и хранения искомых данных в виде массива, и предоставляет методы по поиску.

Метод getSearchValues является эдаким интерфейсом для запроса искомых значений вне данного класса.

Метод setSearchValuesViaString, позволяет преобразовать введённую пользователем строку, в массив искомых значений. После чего, вывести отладочную информацию.

Для вывода в консоль значений, по которым будет производиться поиск, используется метод Print.

Бинарный поиск реализует метод FindElPositionViaBinarySearch, а линейный – FindElPositionViaBinarySearch. В случае удачи, данные методы возвращают индекс элемента, иначе они возвращают -1.

Остался объективно самый большой класс – Benchmark. Мало того что он содержит большое количество приватных полей, так ещё и кучу методов.

Метод Start, используется для входа в данный класс, и он производит замеры времени исполнения линейного и бинарного поиска, путём вызова других классов.

Методы TestLinear и TestBinar вызывают соответственно линейный и бинарный поиск, а так же, записывают время, затраченное на исполнение поиска.

Метод PrintResults используется для выводов результатов. Сначала для каждого промежутка (массива), а после считает выводит среднее время и максимальные минимальные значения затраченного времени.

Метод PerformansePrint позволяет выводить результаты в более красивом виде.

Метод SetExtremes проверяет затраченное время, и устанавливает его в качестве максимального или минимального.

Метод CountAverage считает среднее время для каждого из алгоритмов поиска.

# Тестирование

Данный раздел позволяет убедиться в корректной работе программы в различных сценариях её использования. Тестирование программы для определения скорости работы алгоритма бинарного поиска, и сравнения результатов оного со скоростью работы алгоритма линейного поиска.

Так окно консоли очищается после каждого этапа, первый тест будет расписан полностью, а в остальных – будут опущены все окна кроме последнего.

Первый тест. Размер 1, искомое 1.

При вводе 1, выводит созданный массив. Что видно на рисунке 3.1:



Рисунок 3.1 – ввод единицы в размерах

После нажатия кнопки, открывается окно, изображённое на рисунке 3.2:

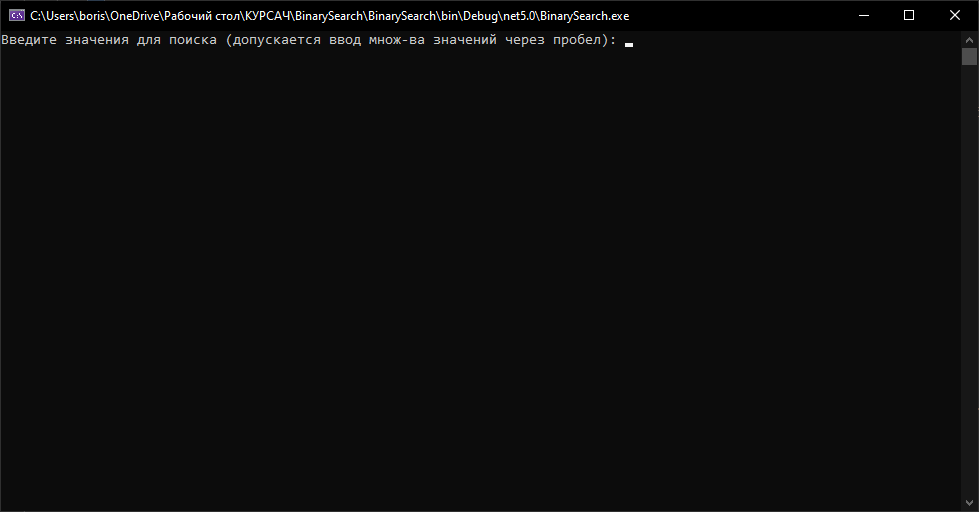


Рисунок 3.2 – окно запроса ввода значений для поиска

Вводим «1». Результат ввода продемонстрирован на рисунке 3.3:

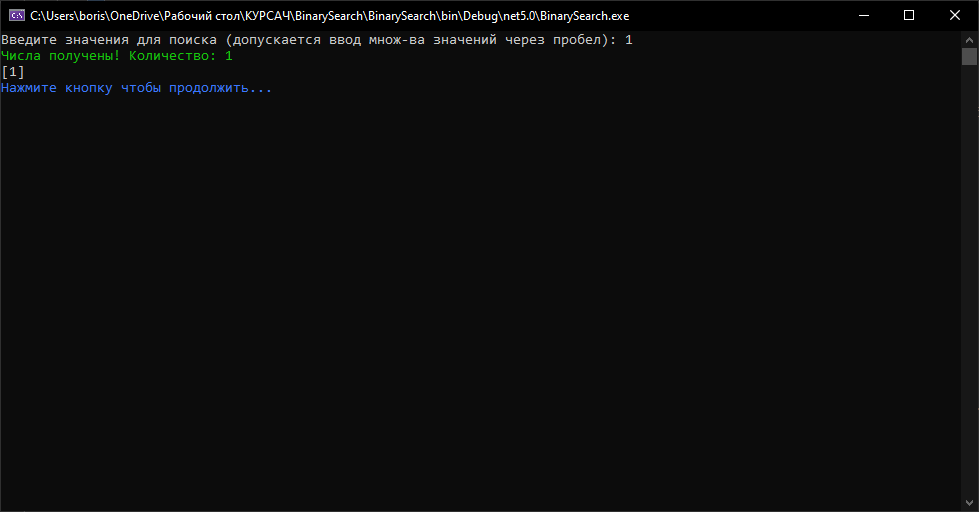


Рисунок 3.3 – результат ввода числа «1»

Результат можно увидеть на рисунке 3.3:

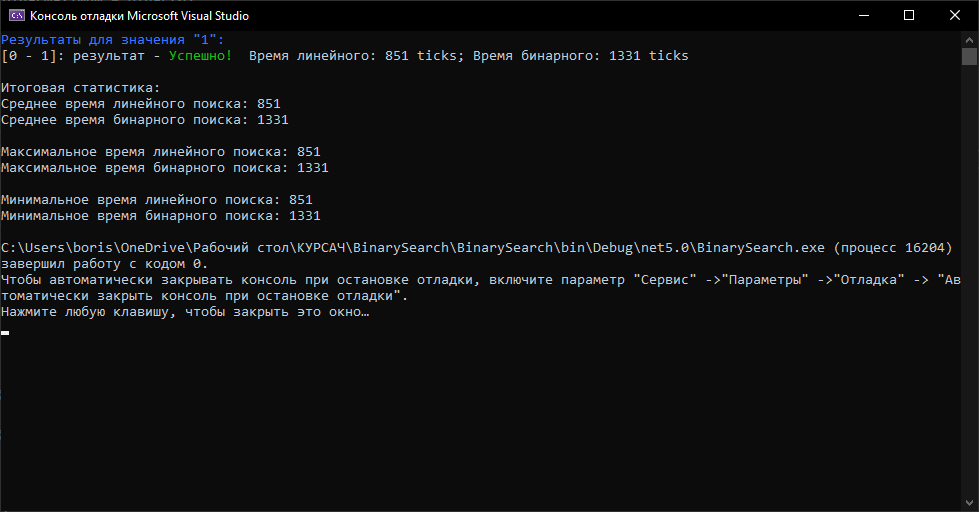


Рисунок 3.3 – результат поиска «1» в массиве размера «1»

Второй тест: размер стандартный в текущей программе, иными словами – 100. Искомое – 1. Результат виден на рисунке 3.4:

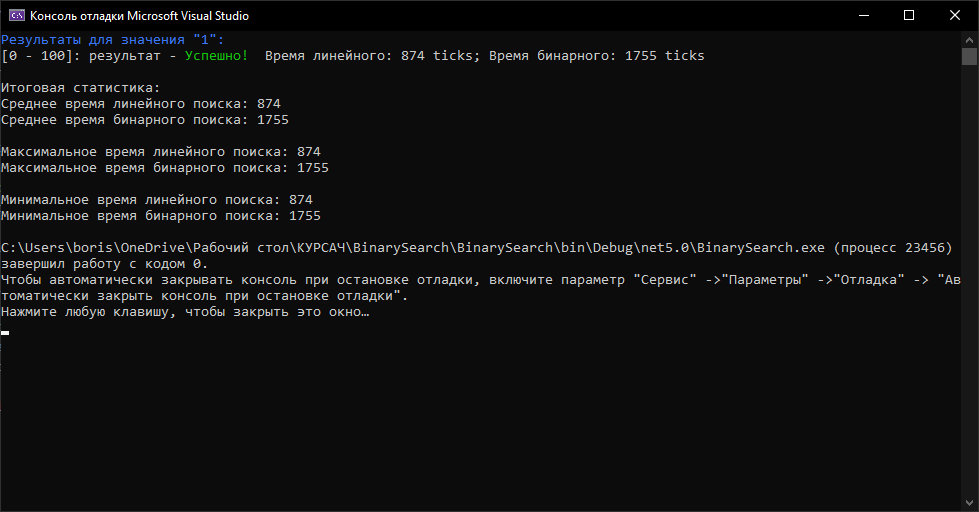


Рисунок 3.4 – результат поиска «1» в массиве размера «100»

Третий тест: размер стандартный, иными словами – 100. Искомое – 100. Результат можно увидеть на рисунке 3.5:

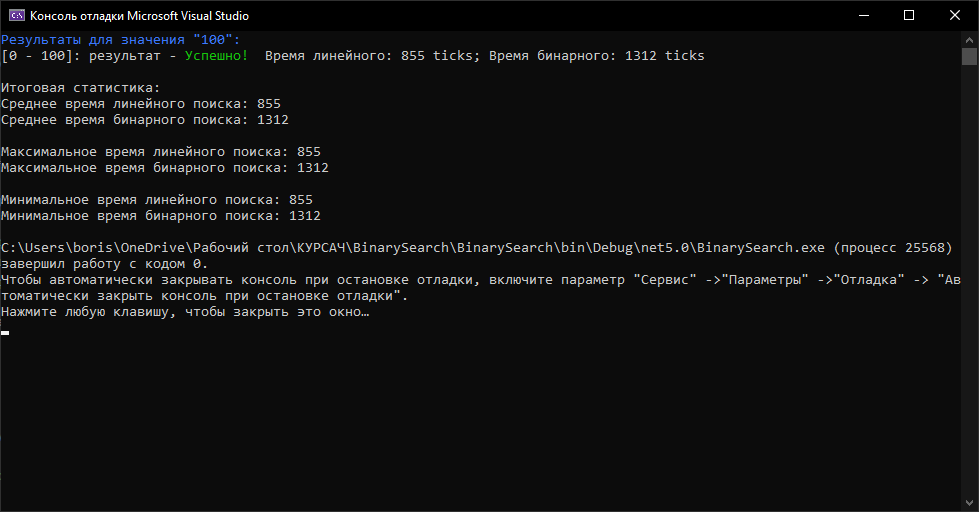


Рисунок 3.5 – результат поиска «100» в массиве размера «100»

Четвёртый тест: размер стандартный, иными словами – 100. Искомое – 101. Целью данного теста является получение информации об обработки ситуации, когда результат нельзя обнаружить. Результат можно увидеть на рисунке 3.6:

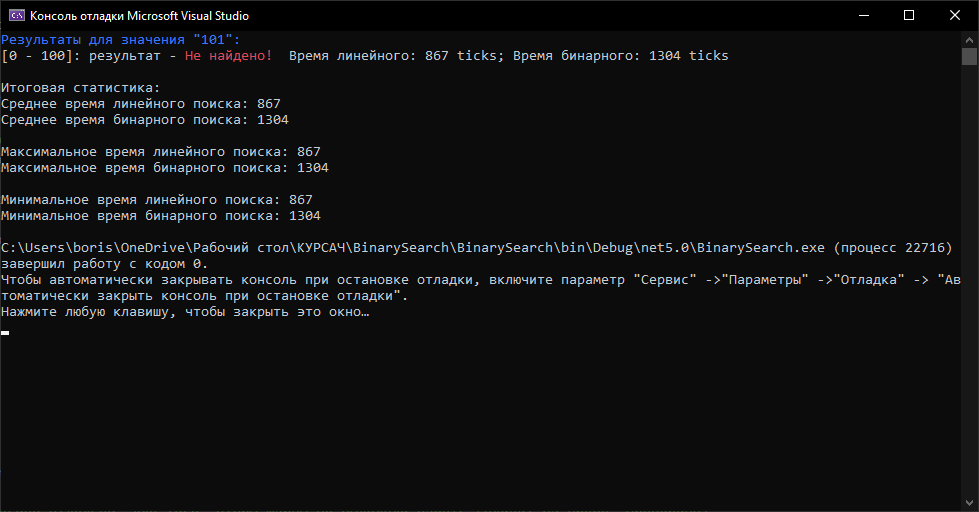


Рисунок 3.6 – результат поиска «101» в массиве размера «100»

Пятый тест: целью теста является проверка максимального размера массива. Размер выставляем максимальный, иными словами – 700000. Ищем 1. Результат можно увидеть на рисунке 3.7:

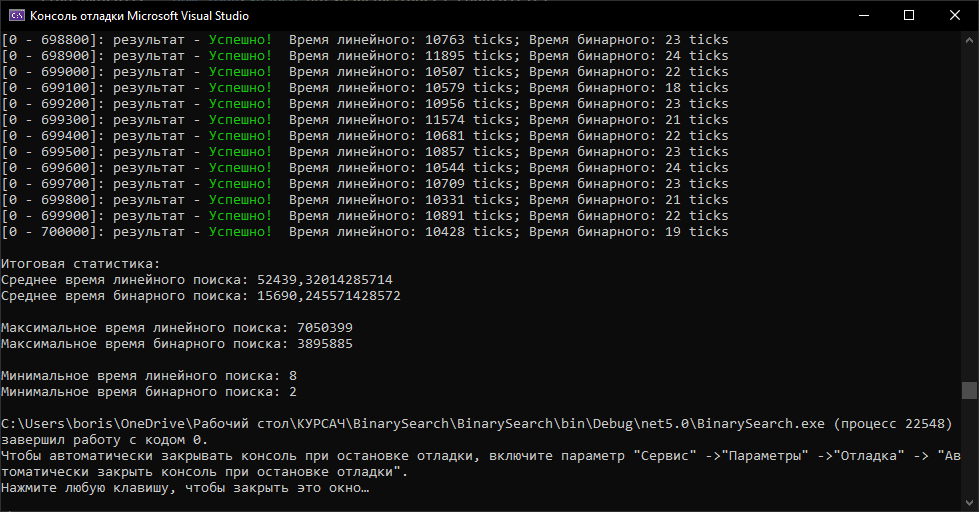


Рисунок 3.7 – результат поиска «1» в массиве размера «700000»

Шестой тест. Выставляем размер в 500000, ищем массив из 15, 1, 35, 1234, 123434444. Результат виден на рисунке 3.8:

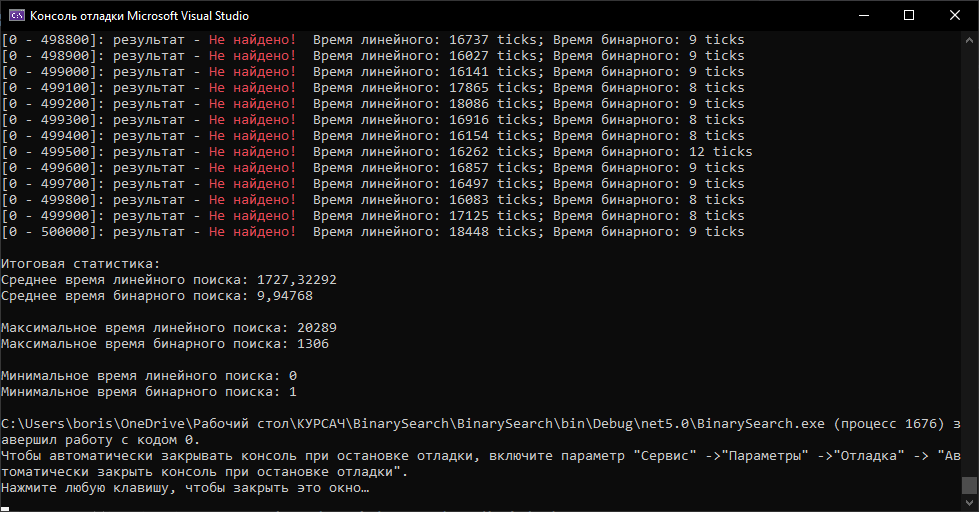


Рисунок 3.8 – результат поиска «15, 1, 35, 1234, 123434444» в массиве размера «700000»

# Выводы

В результате, данной курсовой работы, мы рассмотрели алгоритм бинарного поиска, и произвели его сравнение с алгоритмом линейного поиска. После чего, данные алгоритмы были реализованы на языке программирования C# в консольном приложении. Были рассмотрены используемые пространства имён, а так же функции, которые они содержат. Была рассмотрена написанная программа, включая функции и классы, которые были написаны.

Консольное приложение позволяет вводить в себя следущий набор данных:

* Размер самого большого массива, в котором будет производиться поиск;
* Массив искомых элементов (которых должно быть больше одного).

В результате работы программы, пользователь получает следующие выходные данные:

* Среднее время линейного поиска;
* Среднее время бинарного поиска;
* Минимальное время линейного поиска;
* Минимальное время бинарного поиска;
* Максимальное время линейного поиска;
* Максимальное время бинарного поиска;
* Результаты каждого прогона тестов каждого массива с каждыми искомыми значениями.

Были произведены тесты данной программы с разными входными данными. Результаты данных тестов были зафиксированы. Тестирование показало, что программа корректно обрабатывает ввод, включая обработку некорректного ввода, и заменяет некорректные данные на корректными, о чём, уведомляет пользователя.

В результате анализа полученных результатов, можно сделать вывод, что линейный алгоритм и является более универсальным алгоритмом, который, к тому же, хорошо работает с неотсортированными наборами данных. К тому же, он является максимально простым в реализации.

В свою очередь, начиная с размера в 100000 элементов, при условии, что массив отсортирован (иначе, он и вовсе работать не будет), бинарный поиск показывает среднее время поиска лучше, нежели бинарный поиск.

Иными словами, и тот и тот алгоритм имеет свои плюсы и недостатки, которые необходимо учитывать при реализации функции поиска. Ведь нужно учитывать как тип данных, который будет использоваться для поиска, так и его размеры.

# Список используемой литературы

[1] Бинарный поиск. [Электронный ресурс]. - 2024. - Режим доступа: https://blog.skillfactory.ru/glossary/binarnyj-poisk/

[2] Какая временная сложность поиска в обычном массиве. [Электронный ресурс]. - 2024. - Режим доступа: https://uchet-jkh.ru/i/kakaya-vremennaya-sloznost-poiska-v-obycnom-massive/#:~:text=Сложность%20алгоритма%20линейного%20поиска%20составляет,является%20его%20простота%20и%20универсальность.

[3] Краткий обзор языка C#. [Электронный ресурс]. - 2024. - Режим доступа: https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/

[4] Лучшие IDE для разработки на C# [Электронный ресурс]. - 2024. - Режим доступа: https://gb.ru/blog/luchshie-ide-dlya-razrabotki-na-c/

[5] System Пространство имен. [Электронный ресурс]. - 2024. - Режим доступа: https://learn.microsoft.com/ru-ru/dotnet/api/system?view=net-8.0

[6] System.Diagnostics Пространство имен. [Электронный ресурс]. - 2024. - Режим доступа: https://learn.microsoft.com/ru-ru/dotnet/api/system.diagnostics?view=net-8.0

# Приложение А

using System;

using System.Diagnostics;

namespace BinarySearch

{

internal class Program

{

private static JaggedArray array;

private static void Main(string[] args)

{

Init();

SelectSearchableValues();

StartBenchmark();

}

private static void Init()

{

Console.Write("Для начала работы программы, введите размер массива (по умолчанию 100): ");

string input = Console.ReadLine();

int.TryParse(input, out int size);

array = new JaggedArray(size);

ConsoleManipulator.CLSAfterKeydown();

}

private static void SelectSearchableValues()

{

Console.Write("Введите значения для поиска (допускается ввод множ-ва значений через пробел): ");

Search.setSearchValuesViaString(Console.ReadLine());

ConsoleManipulator.CLSAfterKeydown();

}

private static void StartBenchmark()

{

ConsoleManipulator.ShowNegativeMessage("Начинается бенчмарк. Ожидайте.");

Benchmark.Start(array);

Benchmark.PrintResults(array);

}

}

internal class JaggedArray

{

public int[][] array;

private const int step = 100;

private const int defaultSize = 100;

private const int maxSize = 700000; //узнал путём эксперементов

public JaggedArray(int size)

{

size = ValidateSize(size);

array = new int[ArraysAmount(size)][];

FillArray(size);

ConsoleManipulator.ShowPositiveMessage("Создание успешно! Количество: " + array.Length);

// Не будем отображать, если массив занимает более одной ячейки

// То бишь, размер больше чтем step

if (array.Length < 2)

{

Print();

}

else

{

ConsoleManipulator.ShowInfoMessage("Так как размер больше " + defaultSize + ", отображатся массивы не будут.");

}

}

private int ValidateSize(int size)

{

if (size == 0)

{

ConsoleManipulator.ShowInfoMessage("Установлено значение по умолчанию: " + defaultSize);

return defaultSize;

}

else if(size < 0)

{

size = Math.Abs(size);

ConsoleManipulator.ShowNegativeMessage("Размер массива не может быть отрицательным!");

ConsoleManipulator.ShowInfoMessage("Установлено положительное значение: " + size);

}

if (size > maxSize)

{

ConsoleManipulator.ShowNegativeMessage("Значения выше " + maxSize + " приводят к заполнению оперативной памяти и зависанию ПК!");

ConsoleManipulator.ShowInfoMessage("Установлено безопастное значение (выяснено эксперементальным путём. Кушает ~10.2 гигов оперативы): " + maxSize);

return maxSize;

}

return size;

}

public int ArraysAmount(int size)

{

return ((size == 0 ? defaultSize : size) + Math.Abs(1-defaultSize)) / step;

}

private void FillArray(int size)

{

int currentSize = 0;

for (int i = 0; i < array.Length; i++)

{

currentSize += step;

array[i] = new int[(size - currentSize) > 0 ? currentSize : size];

for (int j = 0; j < array[i].Length; j++)

{

array[i][j] = j + 1;

}

}

}

public void Print()

{

for (int i = 0; i < array.Length; i++)

{

Console.Write('[');

for (int j = 0; j < array[i].Length; j++)

{

Console.Write(j != array[i].Length - 1 ? (array[i][j] + ",") : array[i][j]);

}

Console.WriteLine(']');

}

}

}

internal static class ConsoleManipulator

{

public static void ShowPositiveMessage(string message)

{

Console.ForegroundColor = ConsoleColor.Green;

ShowMessage(message);

}

public static void ShowInfoMessage(string message)

{

Console.ForegroundColor = ConsoleColor.Blue;

ShowMessage(message);

}

public static void ShowNegativeMessage(string message)

{

Console.ForegroundColor = ConsoleColor.Red;

ShowMessage(message);

}

private static void ShowMessage(string message)

{

Console.WriteLine(message);

Console.ResetColor();

}

public static void CLSAfterKeydown()

{

ShowInfoMessage("Нажмите кнопку чтобы продолжить...");

Console.ReadKey();

Console.Clear();

}

}

internal static class Search

{

private static int[] searchValues;

public static int[] getSearchValues()

{

return searchValues;

}

public static void setSearchValuesViaString(string str)

{

string[] stringArray = str.Split(' ');

searchValues = new int[stringArray.Length];

for (int i = 0; i < stringArray.Length; i++)

{

int.TryParse(stringArray[i], out searchValues[i]);

}

ConsoleManipulator.ShowPositiveMessage("Числа получены! Количество: " + searchValues.Length);

if (searchValues.Length < 100)

{

Print();

}

else

{

ConsoleManipulator.ShowInfoMessage("Так как размер больше 100, отображатся массивы не будут.");

}

}

public static void Print()

{

Console.Write('[');

for (int i = 0; i < searchValues.Length; i++)

{

Console.Write(i != searchValues.Length - 1 ? (searchValues[i] + ",") : searchValues[i]);

}

Console.WriteLine(']');

}

public static int FindElPositionViaBinarySearch(int searchValuePosition, int[] array)

{

int left = 0;

int right = array.Length - 1;

int index = 0;

while (left <= right)

{

index = (right + left) / 2;

if (array[index] == searchValues[searchValuePosition])

{

return index;

}

if (array[index] < searchValues[searchValuePosition])

{

left = index + 1;

}

else

{

right = index - 1;

}

}

return -1;

}

public static int FindElPositionByLinearSearch(int searchValuePosition, int[] array)

{

for (int i = 0; i < array.Length; i++)

{

if (array[i] == searchValues[searchValuePosition])

{

return i;

}

}

return -1;

}

}

internal static class Benchmark

{

//[номер искомого в Search][При размере][Результаты - ticks,result(bool)]

private static long[][][] binarResults;

private static long binarMinimum = long.MaxValue;

private static long binarMaximum = 0;

private static double binarAverage;

//[номер искомого в Search][При размере][Результаты - ticks,result(bool)]

private static long[][][] linearResults;

private static long linearMinimum = long.MaxValue;

private static long linearMaximum = 0;

private static double linearAverage;

public static void Start(JaggedArray array)

{

binarResults = new long[Search.getSearchValues().Length][][];

for (int i = 0; i < binarResults.Length; i++)

{

binarResults[i] = new long[array.array.Length][];

}

TestBinar(array.array);

linearResults = new long[Search.getSearchValues().Length][][];

for (int i = 0; i < linearResults.Length; i++)

{

linearResults[i] = new long[array.array.Length][];

}

TestLinear(array.array);

}

private static void TestLinear(int[][] array)

{

//вынесено отдельно, для того, чтобы шарпы не выделяли память (влияет на время. Проверено)

int result;

for (int i = 0; i < Search.getSearchValues().Length; i++)

{

for (int j = 0; j < array.Length; j++)

{

Stopwatch stopwatch = Stopwatch.StartNew();

result = Search.FindElPositionByLinearSearch(i, array[j]);

stopwatch.Stop();

linearResults[i][j] = new long[] { stopwatch.ElapsedTicks, result };

}

}

}

private static void TestBinar(int[][] array)

{

//вынесено отдельно, для того, чтобы шарпы не выделяли память (влияет на время. Проверено)

int result;

for (int i = 0; i < Search.getSearchValues().Length; i++)

{

for (int j = 0; j < array.Length; j++)

{

Stopwatch stopwatch = Stopwatch.StartNew();

result = Search.FindElPositionViaBinarySearch(i, array[j]);

stopwatch.Stop();

binarResults[i][j] = new long[] { stopwatch.ElapsedTicks, result };

}

}

}

public static void PrintResults(JaggedArray array)

{

Console.Clear();

for (int i = 0; i < binarResults.Length; i++)

{

ConsoleManipulator.ShowInfoMessage("Результаты для значения \"" + Search.getSearchValues()[i] + "\":");

for (int j = 0; j < binarResults[i].Length; j++)

{

PerformansePrint(binarResults[i][j], linearResults[i][j], array.array[j]);

SetExtremes(binarResults[i][j], linearResults[i][j]);

}

}

Console.WriteLine("\nИтоговая статистика:");

CountAverage();

Console.WriteLine("Среднее время линейного поиска: " + linearAverage);

Console.WriteLine("Среднее время бинарного поиска: " + binarAverage);

Console.WriteLine();

Console.WriteLine("Максимальное время линейного поиска: " + linearMaximum);

Console.WriteLine("Максимальное время бинарного поиска: " + binarMaximum);

Console.WriteLine();

Console.WriteLine("Минимальное время линейного поиска: " + linearMinimum);

Console.WriteLine("Минимальное время бинарного поиска: " + binarMinimum);

}

private static void PerformansePrint(long[] binar, long[] linear, int[] array)

{

Console.Write("[0 - " + array.Length + "]: результат - ");

Console.ForegroundColor = binar[1] >= 0 ? ConsoleColor.Green : ConsoleColor.Red;

Console.Write(binar[1] >= 0 ? "Успешно! " : "Не найдено! ");

Console.ResetColor();

Console.WriteLine(" Время линейного: " + linear[0] + " ticks; Время бинарного: " + binar[0] + " ticks");

}

private static void SetExtremes(long[] binar, long[] linear)

{

if (binar[0] > binarMaximum)

{

binarMaximum = binar[0];

}

if (binar[0] < binarMinimum)

{

binarMinimum = binar[0];

}

if (linear[0] > linearMaximum)

{

linearMaximum = linear[0];

}

if (linear[0] < linearMinimum)

{

linearMinimum = linear[0];

}

}

private static void CountAverage()

{

long bSum = 0;

int counter = 0;

long lSum = 0;

for (int i = 0; i < binarResults.Length; i++)

{

for (int j = 0; j < binarResults[i].Length; j++)

{

bSum += binarResults[i][j][0];

lSum += linearResults[i][j][0];

counter++;

}

}

binarAverage = (double)bSum / counter;

linearAverage = (double)lSum / counter;

}

}

}