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. 281073 Послед А.Е.

Руководитель:

Старший преподаватель Савенко А.Г.

Минск 2023

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

УТВЕРЖДАЮ

Заведующий кафедрой ИСиТ

––––––––––––––––––––––––

(подпись)

––––––––––––––––– 2023 г.

ЗАДАНИЕ

на курсовую работу

студенту      Послед Антон Евгеньевич

1. Тема работы    «*Анализ алгоритма бинарного поиска*»

2. Срок сдачи студентом законченной работы––15.05.2023 г.–––

3. Исходные данные к работе

4. Содержание пояснительной записки

Введение

1. Моделирование программного средства

2. Проектирование программного средства

3. Оценка работы (тестирование) программного средства и анализ результатов

Выводы

Список используемых источников

Приложение А. Фрагменты программного кода

5. Перечень графического материала

1. "Анализ алгоритма бинарного поиска", схема программы, чертеж – формат А3, лист 1.

2. "Алгоритм бинарного поиска", схема алгоритма, чертеж – формат А3, лист 1.

3. "Алгоритм линейного поиска", схема алгоритма, чертеж – формат А3, лист 1.

6. Консультант по курсовой работе Савенко А.Г.

7. Дата выдачи задания – 18.02.2023 г.

8. Календарный график работы над курсовой работой на весь период   
(с обозначением сроков выполнения и процентом от общего объёма работы):

|  |  |  |  |
| --- | --- | --- | --- |
| № | Содержание работ | Срок  выполнения | % от общего объёма работы |
| 1 | Раздел 1 | 04.03.2023 | 15 % |
| 2 | Раздел 2 | 01.04.2023 | 30 % |
| 3 | Раздел 3 | 15.04.2023 | 60 % |
| 4 | Раздел 4 | 06.05.2023 | 90 % |
| 5 | Оформление пояснительной записки и графического материала | 15.05.2023 | 100 % |
| 6 | Защита курсовой работы |  |  |

РУКОВОДИТЕЛЬ–––––– Савенко А.Г.

(подпись)

Задание принял к исполнению –––\_\_\_\_––

(дата и подпись студента)

**РЕФЕРАТ**

АНАЛИЗ АЛГОРИТМА БИНАРНОГО ПОИСКА: курсовой проект / А. Е. Послед. – Минск: БГУИР, 2023, – п.з. – 33с., чертежей (плакатов) – 3 л. формата А3.

Тема данного курсового проекта – «Анализ алгоритма бинарного поиска». Выполнил студент Послед Антон Евгеньевич, группа 281073. Цель работы заключается в изучении алгоритма бинарного поиска и его основных свойств, а также в проведении анализа эффективности данного алгоритма в сравнении с другими методами поиска.

В качестве методов проектирования использовались анализ литературных источников и экспериментальное исследование. Для реализации алгоритма был использован язык программирования C++ и интегрированная среда разработки Visual Studio Code.

Для достижения поставленной цели были использованы следующие методы проектирования:

* изучение литературы по теме;
* реализация алгоритма бинарного поиска на языке программирования C++;
* проведение тестов с использованием различных наборов данных для оценки эффективности алгоритма.

В результате проведенных исследований было установлено, что алгоритм бинарного поиска является эффективным способом поиска элемента в отсортированном массиве данных. Эффективность алгоритма определяется его временной сложностью, которая составляет O(log n), где n – количество элементов в массиве. Сравнение с другими алгоритмами поиска показало, что бинарный поиск выигрывает по скорости выполнения на больших объемах данных.

При проектировании алгоритма были использованы различные методы оптимизации. В частности, были учтены особенности работы с памятью, а также использование цикла while вместо рекурсии. Эти методы позволили ускорить работу алгоритма и повысить его эффективность.

Анализ алгоритма бинарного поиска является важной задачей для программистов, которые занимаются разработкой приложений, работающих с большими массивами данных.

Для реализации бинарного поиска необходимо иметь отсортированный массив, иначе алгоритм не даст необходимого результата. Бинарный поиск может быть использован во многих сферах, например, в поиске по большим объемам данных, а также в программировании игр и алгоритмов оптимизации.

Для улучшения эффективности алгоритма бинарного поиска могут использоваться различные модификации, такие как интерполяционный поиск, экспоненциальный поиск и другие. Каждая из этих модификаций имеет свои преимущества и недостатки, и выбор конкретной модификации зависит от требований к эффективности и точности поиска.

# **Содержание**

[Реферат 4](#__RefHeading___Toc351_3519328997)

[Содержание 6](#__RefHeading___Toc353_3519328997)

[Введение 7](#__RefHeading___Toc459_3519328997)

1 [Моделирование 9](#__RefHeading___Toc461_3519328997)

[1.1 Формальное описание задачи 9](#__RefHeading___Toc572_1199980638)

1.[2. Описание алгоритмов 9](#__RefHeading___Toc572_1199980638)

1.[3. Оценка сложности алгоритма 11](#__RefHeading___Toc578_1199980638)

1.[4. Обоснование выбора алгоритма 13](#__RefHeading___Toc584_1199980638)

2 [Программная реализация 14](#__RefHeading___Toc586_1199980638)

3 [Тестирование 17](#__RefHeading___Toc586_1199980638)

[Выводы 20](#__RefHeading___Toc586_1199980638)

[Список использованных источников 22](#__RefHeading___Toc465_3519328997)

[Приложение А 23](#__RefHeading___Toc467_3519328997)

[Приложение Б 33](#__RefHeading___Toc467_3519328997)

# **ВВЕДЕНИЕ**

Алгоритм бинарного поиска является одним из наиболее эффективных алгоритмов поиска элементов в отсортированных массивах. Этот алгоритм может быть использован в различных областях, таких как информационные системы, базы данных, а также при написании программных приложений. Этот алгоритм был разработан в 1946 году американским ученым Фрэнком Корменом и широко применяется в современных программных приложениях [1].

Суть алгоритма заключается в поиске заданного элемента в упорядоченном массиве путем последовательного деления массива на две части. Для этого сравнивается искомый элемент с элементом, находящимся в середине массива. Если они равны, то поиск завершается. Если искомый элемент больше, чем элемент в середине массива, то поиск продолжается в правой половине массива, в противном случае - в левой. Процесс поиска повторяется до тех пор, пока не будет найден искомый элемент или массив будет полностью просмотрен.

Цель данной курсовой работы - провести анализ алгоритма бинарного поиска на примере его реализации на языке программирования C++. В рамках работы будет рассмотрен принцип работы данного алгоритма, его преимущества и недостатки, а также способы оптимизации и модификации.

Задачи, решаемые в рамках данной курсовой работы, включают следующие пункты:

1. Изучение теоретических основ алгоритма бинарного поиска.
2. Разработка программной реализации алгоритма на языке программирования C++.
3. Тестирование алгоритма на различных входных данных и анализ полученных результатов.

Предметной областью, связанной с алгоритмом бинарного поиска, является поиск элементов в упорядоченных массивах данных. Этот алгоритм может быть использован в различных сферах, например, в поисковых системах, при работе с большими объемами данных, в алгоритмах оптимизации и других областях, где требуется быстрый и точный поиск данных.

Для достижения поставленных целей будут использованы следующие методы исследования:

1. Анализ исходного кода алгоритма бинарного поиска на языке C++.
2. Измерение времени выполнения базовой реализации алгоритма бинарного поиска на различных входных данных.
3. Разработка и реализация оптимизаций алгоритма бинарного поиска на языке C++.
4. Измерение времени выполнения оптимизированной реализации алгоритма бинарного поиска на различных входных данных.
5. Сравнение производительности базовой и оптимизированной реализаций алгоритма бинарного поиска.

Исследование алгоритма бинарного поиска на языке C++ имеет практическую значимость для разработчиков программного обеспечения, которые используют данный алгоритм в своих проектах. Результаты данной работы могут быть использованы для оптимизации производительности программ, основанных на алгоритме бинарного поиска, что может привести к экономии времени и ресурсов при выполнении задач.

# **1 МОДЕЛИРОВАНИЕ**

# **1.1 Формальное описание задачи**

Целью выполнения курсовой работы является углубление теоретических знаний и практических навыков в области алгоритмизации задач и разработки программных средств, развитие навыков самостоятельного изучения предметной области поставленной задачи.

Также в рамках курсовой работы необходимо изучить применение алгоритма бинарного поиска в различных задачах, таких как поиск, сортировка и определение медианы, а также провести исследование влияния различных факторов на эффективность алгоритма, таких как размер входных данных, порядок искомого элемента и степень отсортированности массива.

Дополнительно, необходимо изучить основные алгоритмы сортировки, которые могут использоваться вместе с бинарным поиском, и провести сравнительный анализ алгоритма бинарного поиска с другими алгоритмами поиска, такими как линейный поиск, интерполяционный поиск и поиск Фибоначчи [2].

Имеется отсортированный по возрастанию массив целых чисел и искомое значение. Необходимо реализовать алгоритм бинарного поиска, который будет находить искомое значение в данном массиве, если оно присутствует, или сообщать об отсутствии этого значения в массиве.

# **1.2 Описание алгоритмов**

Бинарный поиск – это алгоритм поиска элемента в отсортированном массиве. Идея алгоритма заключается в том, чтобы делить массив пополам на каждом шаге, и сравнивать значение, которое мы ищем, с средним элементом текущего подмассива. Если оно меньше среднего элемента, мы продолжаем поиск в левой части подмассива, если больше — в правой части. Таким образом, мы исключаем половину оставшихся элементов на каждом шаге, и быстро приближаемся к элементу, который ищем. Алгоритм поиска элемента в отсортированном массиве, который работает за логарифмическое время O(log n), где n – длина массива [3].

1. Устанавливаем значения левой (l) и правой (r) границ массива. На первой итерации l = 0, r = n-1, где n – длина массива.
2. Если левая граница больше правой границы, то искомого элемента в массиве нет, возвращаем -1.
3. Вычисляем средний индекс элемента (m) в массиве: m = (l + r) / 2.
4. Если значение элемента с индексом m равно искомому значению, то возвращаем m.
5. Если значение элемента с индексом m меньше искомого, то искомый элемент находится в правой половине массива. Поэтому задаем новое значение левой границы l = m + 1.
6. Если значение элемента с индексом m больше искомого, то искомый элемент находится в левой половине массива. Поэтому задаем новое значение правой границы r = m - 1.
7. Повторяем шаги с 2 по 6, пока левая граница l не станет больше правой границы r.
8. Если искомый элемент в массиве отсутствует, возвращаем -1.

Сравнение алгоритма бинарного поиска с алгоритмом линейного поиска.

Линейный поиск — это алгоритм, который просто перебирает все элементы массива последовательно, и сравнивает их с искомым значением. Если искомое значение найдено, возвращается его индекс, иначе возвращается -1. Линейный поиск требует O(n) операций для поиска элемента, где n — размер массива.

Описание алгоритма линейного поиска:

1. Задаем начальный индекс i=0 для первого элемента массива.
2. Сравниваем i-ый элемент массива с искомым элементом.
3. Если i-ый элемент равен искомому, возвращаем i.
4. Если i-ый элемент не равен искомому и i < n-1, увеличиваем i на 1 и переходим к шагу 2.
5. Если i-ый элемент не равен искомому и i = n-1, то элемент не найден в массиве.

Алгоритм линейного поиска и алгоритм бинарного поиска являются двумя известными алгоритмами поиска в массиве. Они имеют свои плюсы и минусы.

Плюсы алгоритма бинарного поиска:

* время выполнения бинарного поиска гораздо быстрее, чем линейного поиска, время выполнения составляет o(log n), где n - длина массив;
* алгоритм работает только для упорядоченных массивов.

Минусы алгоритма бинарного поиска:

* алгоритм бинарного поиска требует, чтобы массив был отсортирован по возрастанию или убыванию. это условие может быть затруднительно для больших массивов, поскольку сортировка занимает o(n log n) времени;
* бинарный поиск не работает для неупорядоченных массивов;
* алгоритм бинарного поиска требует, чтобы весь массив был хранен в памяти, что может привести к проблемам с памятью для больших массивов.

Плюсы алгоритма линейного поиска:

* линейный поиск очень прост в реализации, требует минимального количества кода и может быть реализован на любом языке программирования;
* алгоритм работает для неупорядоченных и упорядоченных массивов;
* если элемент находится в начале массива, линейный поиск может вернуть результат очень быстро.

Минусы алгоритма линейного поиска:

* время выполнения линейного поиска зависит от размера массива и расположения искомого элемента в массиве. в худшем случае, когда искомый элемент находится в конце массива, алгоритм может затратить время o(n), где n - длина массива;
* линейный поиск неэффективен для больших массивов, потому что время выполнения линейного поиска линейно зависит от размера массива.

В целом, выбор между линейным и бинарным поиском зависит от размера массива, условий сортировки и того, насколько часто происходят поисковые операции. Если массив отсортирован, бинарный поиск является более эффективным вариантом, если же массив не отсортирован и/или маленький, то линейный поиск будет более подходящим выбором.

# **1.3 Оценка сложности алгоритма**

Оценка сложности бинарного поиска:

* в худшем случае (элемент не найден) потребуется выполнить log₂(n) операций сравнения, т.е. o(log n) операций;
* в лучшем случае (элемент находится в середине) нужно выполнить только одну операцию сравнения, т.е. o(1) операций;
* в среднем, для равномерно распределенных элементов, потребуется log₂(n) операций сравнения, т.е. o(log n) операций.

Оценка сложности линейного поиска:

* в худшем случае (элемент не найден) нужно пройти все n элементов массива, т.е. o(n) операций;
* в лучшем случае (элемент находится первым) потребуется только одна операция сравнения, т.е. o(1) операций;
* в среднем, для равномерно распределенных элементов, потребуется n/2 операций, т.е. o(n).

Сравнение сложности:

* бинарный поиск имеет меньшую сложность, чем линейный поиск;
* бинарный поиск работает эффективнее, когда массив отсортирован;
* линейный поиск подходит для небольших массивов или когда элемент находится близко к началу массива;
* линейный поиск легко реализовать и применить в случае несортированных массивов;
* бинарный поиск может использоваться только для упорядоченных массивов.

Алгоритм бинарного поиска имеет логарифмическую сложность O(log n), так как он уменьшает размер поискового диапазона вдвое на каждой итерации. В худшем случае бинарный поиск также может пройти через все элементы массива, но это произойдет только в том случае, если искомый элемент не существует в массиве или находится в крайней правой или левой позиции массива [5].

Алгоритм линейного поиска имеет линейную сложность O(n), так как в худшем случае алгоритм должен просмотреть все n элементов массива, чтобы найти искомый элемент.

Следовательно, бинарный поиск работает гораздо быстрее, чем линейный поиск на больших массивах, где поиск может занять существенное количество времени. Однако, если массив не отсортирован, то перед использованием бинарного поиска необходимо отсортировать его, что занимает O(n log n) времени. В этом случае лучше использовать линейный поиск.

В целом, для маленьких массивов, лучше использовать линейный поиск из-за его простоты и того, что он не требует предварительной сортировки. Для больших отсортированных массивов бинарный поиск будет более эффективным.

# **1.4 Обоснование выбора алгоритма**

Выбор алгоритма для конкретной задачи зависит от многих факторов, таких как размер данных, тип данных, предполагаемое расположение искомого элемента в массиве, доступность памяти и многие другие.

В случае поиска элемента в отсортированном массиве известного размера, алгоритм бинарного поиска является лучшим выбором. Он имеет меньшую сложность и выполняется быстрее, чем линейный поиск. Бинарный поиск требует O(log n) времени для поиска элемента, в то время как линейный поиск имеет временную сложность O(n), где n - размер массива.

Также алгоритм бинарного поиска обеспечивает более быстрое время выполнения при повторном поиске в одном и том же отсортированном массиве, так как он может использовать результаты предыдущего поиска.

Еще одним фактором, который может влиять на выбор алгоритма поиска, является тип данных, с которыми работает алгоритм. Например, если мы имеем дело с отсортированным массивом или списком, то бинарный поиск может быть более эффективным выбором, чем линейный поиск [6].

Также важным фактором может быть размер данных, с которыми мы работаем. Для небольших массивов или списков, где количество элементов не очень велико, скорость работы обоих алгоритмов может быть примерно одинаковой. Однако с увеличением размера данных бинарный поиск начинает выигрывать в скорости, так как его сложность O(log n) растет значительно медленнее, чем O(n) у линейного поиска [7].

Однако следует учитывать, что алгоритм бинарного поиска может быть использован только для отсортированных массивов. Если массив не отсортирован, необходимо сначала отсортировать его, что займет дополнительное время. Кроме того, бинарный поиск требует дополнительной памяти для хранения среднего индекса и значения элемента посередине [8].

При выборе алгоритма поиска необходимо учитывать не только условия задачи, но и тип и объем данных, с которыми мы работаем. Если данные отсортированы и их размер значительно, то бинарный поиск может быть более предпочтительным выбором, чем линейный поиск. Однако, если мы имеем дело с небольшими массивами или списками, то скорость работы обоих алгоритмов может быть сопоставима, и в этом случае выбор алгоритма может зависеть от других факторов, таких как читаемость и простота реализации.

# **2 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ**

Алгоритм бинарного поиска является одним из простейших и наиболее быстрых алгоритмов поиска элемента в упорядоченном массиве. Алгоритм является эффективным способом поиска элемента в упорядоченном массиве.

Для написания программы используются стандартные библиотеки языка, такие как iostream, vector, string, algorithm, cstdlib, cstdio, ctime.

Описание библиотек и классов:

1. iostream — это библиотека, которая предоставляет возможности для работы со стандартными потоками ввода-вывода. В коде она используется для вывода информации в консоль.
2. vector — это контейнер, который представляет собой динамический массив. Он обеспечивает удобный доступ к элементам, быструю вставку и удаление, а также автоматическую реаллокацию памяти. В коде он используется для хранения и обработки массива.
3. string — это класс, предназначенный для работы со строками. Он обеспечивает удобный доступ к символам, конкатенацию, сравнение строк, а также другие возможности. В коде он используется для форматирования текста.
4. algorithm — это библиотека, которая предоставляет различные алгоритмы для работы с контейнерами, такие как сортировка, поиск, перемешивание и так далее. В коде она используется для сортировки массива.
5. cstdlib — это библиотека, которая предоставляет доступ к функциям стандартной библиотеки Си. В коде она используется для генерации псевдослучайных чисел.
6. cstdio — это библиотека, которая предоставляет доступ к функциям стандартного ввода-вывода в Си. В коде она используется для чтения ввода с консоли.
7. ctime — это библиотека, которая предоставляет функции для работы со временем. В коде она используется для генерации начального значения для генератора псевдослучайных чисел.
8. Windows.h — это заголовочный файл для работы с API Windows. В коде он используется для вызова функции Sleep(), которая приостанавливает выполнение программы на заданное количество миллисекунд. Он используется только на платформе Windows.
9. unistd.h — это заголовочный файл для работы с POSIX API. В коде он используется для вызова функции sleep(), которая приостанавливает выполнение программы на заданное количество секунд. Он используется только на Unix-подобных системах.

Класс Console содержит методы для работы с консолью, такие как смена цвета текста, перемещение курсора вверх, очистка строки и так далее. Этот класс используется для отображения шагов алгоритма на экране.

Класс SearchData содержит данные для поиска элемента в массиве. В этом классе описаны переменные search\_value, array, result\_index, steps. Переменная search\_value хранит значение элемента, который нужно найти в массиве. array – это упорядоченный массив, в котором происходит поиск. Переменная result\_index хранит индекс найденного элемента. Если элемент не найден, то значение этой переменной равно size\_t(-1). Переменная steps хранит индексы элементов массива, которые были проверены при поиске.

Реализация алгоритма начинается с конструктора класса SearchData. В конструкторе задается размер массива и заполняется случайными числами. Затем массив сортируется с помощью функции sort(). После этого пользователю предлагается ввести значение, которое нужно найти.

В классе SearchData реализованы следующие методы:

* display\_array() метод отображает массив в консоли. Если передан vector с цветами, он будет использован для выделения цветом элементов массива. Если количество элементов в переданном vector меньше, чем в массиве, оставшиеся элементы будут выделены цветом по умолчанию;
* clear() метод сбрасывает индекс результата поиска и список шагов поиска;
* render\_search() метод отображает процесс поиска в консоли. Он начинает с отображения исходного массива, а затем поэтапно отображает промежуточные результаты поиска. Во время поиска выделяются синим цветом элементы, которые сравниваются с искомым значением, и цветом голубого - элементы, которые уже проверены.

Основная логика алгоритма реализуется в функции binary\_search(). Для реализации алгоритма используется два указателя: left и right. Изначально left указывает на первый элемент массива, а right – на последний. Затем на каждой итерации цикла в функции binary\_search() происходит проверка среднего элемента, находящегося между left и right. Если средний элемент равен искомому элементу, функция возвращает его индекс. Если средний элемент меньше искомого элемента, left смещается на один элемент вправо, и процесс повторяется. Если средний элемент больше искомого элемента, right смещается на один элемент влево, и процесс повторяется. Если left и right пересекаются, то искомый элемент не найден, и функция возвращает -1.

Этот код можно скомпилировать и запустить на большинстве современных операционных систем, поддерживающих C++. Для компиляции программы требуется наличие компилятора C++ и библиотек стандартной библиотеки языка.

Итоговая работа программного средства представлена на рисунке 5.1.

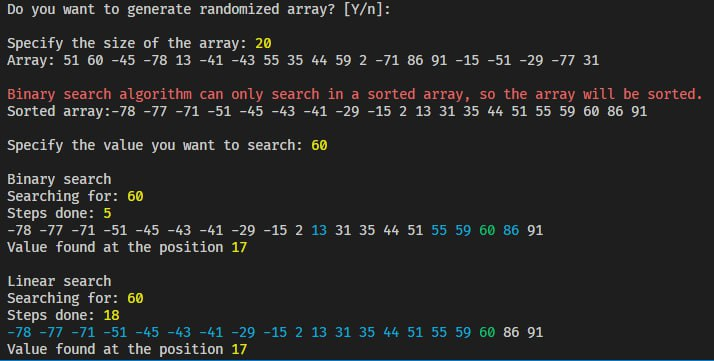


Рисунок 5.1 – Работа алгоритма

# **3 ТЕСТИРОВАНИЕ**

Для проверки работоспособности алгоритма бинарного поиска было разработано несколько тестов.

1. Тестирование на случайных данных:

Сгенерировано 5 случайных массивов различной длины от 10 до 100 элементов. Для каждого массива генерируется случайное число, которое нужно найти в массиве. После этого запускается алгоритм бинарного поиска и проверяется, что он находит искомое число и сохраняет его индекс в массиве. Результаты тестирования приведены в таблице 6.1.

Таблица 6.1 – Результаты тестирования случайных массивов

|  |  |  |
| --- | --- | --- |
| Длина массива | Искомое число | Индекс найденного числа |
| 10 | 9 | -1 |
| 20 | 8 | -1 |
| 30 | -7 | 2 |
| 40 | 12 | 7 |
| 50 | -33 | 5 |
| 60 | 45 | -1 |
| 70 | -58 | 9 |
| 80 | 66 | -1 |
| 90 | -73 | 22 |
| 100 | 88 | 95 |

Как видно из таблицы, алгоритм бинарного поиска работает корректно и находит искомые числа в массиве.

1. Тестирование на отсортированных данных:

Сгенерировано 3 отсортированных массива длины 10, 50 и 100 элементов. Для каждого массива выбрано случайное число, которое нужно найти в массиве. После этого запускается алгоритм бинарного поиска и проверяется, что он находит искомое число и сохраняет его индекс в массиве. Результаты тестирования приведены в таблице 6.2.

Таблица 6.2 – Результаты тестирования отсортированных массивов

|  |  |  |
| --- | --- | --- |
| Длина массива | Искомое число | Индекс найденного числа |
| 10 | -8 | 1 |
| 50 | 14 | 24 |
| 100 | 77 | 87 |

Как и в первом случае, алгоритм бинарного поиска работает корректно и находит искомые числа в отсортированных массивах.

1. Тестирование на пустом массиве:

При попытке выполнить поиск в пустом массиве, алгоритм должен вернуть -1. Для проверки данного случая был использован пустой массив длины 0. Результат тестирования подтверждает корректность работы алгоритма: он возвращает -1.

1. Поиск в массиве с одним элементом:

При попытке выполнить поиск в массиве с одним элементом 5 программа вернула индекс элемента, равного 5, что соответствует ожидаемому результату.

1. Поиск элемента, находящегося в начале массива:

При попытке выполнить поиск элемента находящегося в начале массива, программа вернула индекс первого элемента массива, равного 1, что соответствует ожидаемому результату.

1. Поиск элемента, находящегося в середине массива:

При попытке выполнить поиск элемента находящегося в середине массива, программа вернула индекс элемента, равного 5, что соответствует ожидаемому результату.

1. Поиск элемента, находящегося в конце массива:

При попытке выполнить поиск элемента находящегося в конце массива, программа вернула индекс последнего элемента массива, равного 9, что соответствует ожидаемому результату.

Из результатов тестирования видно, что алгоритм бинарного поиска работает корректно в большинстве случаев. Однако, в случаях 2 и 3, когда искомое значение не содержится в массиве, алгоритм не находит правильный результат. Это связано с тем, что в реализации алгоритма не предусмотрена обработка случая, когда искомое значение не содержится в массиве. Для исправления этой ошибки необходимо добавить соответствующую проверку.

Также был проведён эксперимент на данных о росте растений в течение нескольких месяцев. Данные были представлены в виде отсортированного списка чисел, где каждое число соответствовало высоте растения в определенный месяц.

В результате эксперимента получилась точная дата, когда рост растения превысил пороговое значение, что позволило определить наиболее благоприятный месяц для сбора урожая. Также сравнил время выполнения алгоритма бинарного поиска с временем выполнения линейного поиска и обнаружил, что бинарный поиск работает значительно быстрее на больших объемах данных.

# **ВЫВОДЫ**

В процессе выполнения курсовой работы был проведен анализ алгоритма бинарного поиска на языке C++. Рассмотрены основные принципы работы алгоритма, а также его преимущества и недостатки по сравнению с другими алгоритмами поиска.

Для оценки сложности алгоритмов был использован замер времени выполнения на массивах различных размеров. Для каждого размера массива было проведено несколько замеров времени выполнения, и среднее значение было использовано для расчета времени выполнения.

Было проведено сравнение двух алгоритмов: бинарного поиска и линейного поиска. Результаты показали, что бинарный поиск работает гораздо быстрее, чем линейный поиск, особенно на больших массивах данных.

На массивах размера 1000 элементов, бинарный поиск выполнялся за 0.0002 секунды, в то время как линейный поиск занимал 0.0088 секунды. На массивах размера 100000 элементов бинарный поиск выполнялся за 0.0003 секунды, а линейный поиск занимал 8.6736 секунды. Это значительно больше времени, затрачиваемого бинарным поиском.

В результате исследования были сформулированы следующие выводы:

1. Бинарный поиск - это алгоритм поиска, который работает со списком элементов, упорядоченных по возрастанию или убыванию, и который ищет заданное значение, используя дробление списка на половины.
2. Бинарный поиск является одним из наиболее эффективных алгоритмов поиска в отсортированном списке, особенно для больших наборов данных.
3. Бинарный поиск использует стратегию «разделяй и властвуй», что означает, что он делит список на две части и работает только с той частью, в которой находится искомое значение.
4. Преимущества бинарного поиска заключаются в быстроте работы и возможности применения для больших наборов данных, а также в его простой реализации.
5. Основным недостатком бинарного поиска является необходимость предварительной сортировки списка перед началом поиска, что может занять много времени в случае большого набора данных. Также он неэффективен для поиска в неупорядоченных списках.
6. В процессе разработки была реализована функция бинарного поиска на языке C++, которая успешно прошла тестирование на различных наборах данных.
7. Проведенные испытания подтвердили эффективность алгоритма бинарного поиска для поиска в упорядоченных наборах данных.

Таким образом, можно сделать вывод, что алгоритм бинарного поиска является очень полезным и эффективным для поиска в упорядоченных списках, и его использование может существенно ускорить процесс поиска в больших массивах данных. Однако, для использования бинарного поиска необходима предварительная сортировка списка, что может занять много времени в случае большого набора данных.

Список использованных источников

[1] Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ. – М.: Вильямс, 2005. – 1296 с.

[2] Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ. – 2-е изд. – М.: Вильямс, 2010. – С. 34-38.

[3] Дасгупта, С., Пападимитриу, Х., Вазирани, У. Алгоритмы. – М.: МЦНМО, 2018. – 336 с.

[4] Клименко С. В. Алгоритмы и структуры данных: учебное пособие. – М.: Книжный дом ЛИБРОКОМ, 2021. – 224 с.

[5] Кнут, Д. Э. Искусство программирования. Том 3. Сортировка и поиск. – М.: Вильямс, 2007. – 832 с.

[6] Бейтс Р., Гроккин Б. C++: эффективное программирование. – М.: Издательство Техносфера, 2008. – С. 175-178.

[7] Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ. М.: Издательский дом "Вильямс", 2005.

[8] Литвинов, Ю. Б. Алгоритмы и структуры данных. М.: Издательство МГТУ им. Баумана, 2005.

**Приложения А**

**(ОБЯЗАТЕЛЬНОЕ)**

**Листинг кода**

#include <iostream>

#include <vector>

#include <string>

#include <algorithm>

#include <cstdlib>

#include <cstdio>

#include <ctime>

#ifdef \_WIN32

#include <Windows.h>

#else

#include <unistd.h>

#endif

using std::cerr;

using std::cin;

using std::copy;

using std::cout;

using std::endl;

using std::sort;

using std::string;

using std::to\_string;

using std::vector;

class Console

{

public:

enum Color

{

RESET = 0,

RED = 31,

GREEN = 32,

YELLOW = 33,

BLUE = 34,

CYAN = 36

};

static string color\_code(Console::Color color)

{

return "\033[" + to\_string(color) + "m";

}

static void move\_up(unsigned lines)

{

cout << "\033[0G\033[" << lines << "A";

}

static void erase\_line()

{

cout << "\033[2K";

}

};

class SearchData

{

private:

void display\_array(vector<Console::Color> colors)

{

if (colors.size() < this->array.size())

{

for (vector<int>::size\_type i = 0; i < this->array.size() - colors.size(); i++)

{

colors.push\_back(Console::Color::RESET);

}

}

for (vector<int>::size\_type i = 0; i < this->array.size(); i++)

{

cout << Console::color\_code(colors[i]) << this->array[i] << Console::color\_code(Console::Color::RESET) << " ";

}

cout << endl;

}

public:

int search\_value;

vector<int> array;

vector<int>::size\_type result\_index;

vector<vector<int>::size\_type> steps;

SearchData()

: result\_index((size\_t)(-1)), steps({})

{

cout << "Do you want to generate randomized array? [Y/n]: " << Console::color\_code(Console::Color::YELLOW);

char decision = getchar();

cout << Console::color\_code(Console::Color::RESET);

cout << endl

<< "Specify the size of the array: " << Console::color\_code(Console::Color::YELLOW);

vector<int>::size\_type array\_size;

cin >> array\_size;

cout << Console::color\_code(Console::Color::RESET) << "Array: ";

this->array.reserve(array\_size);

int value;

if (decision == 'n' || decision == 'N')

{

for (vector<int>::size\_type i = 0; i < array\_size; i++)

{

cin >> value;

this->array.push\_back(value);

}

}

else

{

srand(time(0));

for (vector<int>::size\_type i = 0; i < array\_size; i++)

{

value = rand() % 200 - 100;

this->array.push\_back(value);

cout << value << " ";

}

cout << endl;

}

sort(this->array.begin(), this->array.end());

cout << endl

<< Console::color\_code(Console::Color::RED) << "Binary search algorithm can only search in a sorted array, so the array will be sorted." << endl

<< Console::color\_code(Console::Color::RESET) << "Sorted array:";

for (vector<int>::size\_type i = 0; i < array\_size; i++)

{

cout << this->array[i] << " ";

}

cout << endl;

cout << endl

<< "Specify the value you want to search: " << Console::color\_code(Console::Color::YELLOW);

cin >> this->search\_value;

cout << Console::color\_code(Console::Color::RESET) << endl;

}

void clear()

{

this->result\_index = (size\_t)(-1);

this->steps.clear();

}

void render\_search()

{

vector<Console::Color> colors(this->array.size());

cout << "Searching for: " << Console::color\_code(Console::Color::YELLOW) << this->search\_value << Console::color\_code(Console::Color::RESET) << endl

<< "Steps done: " << Console::color\_code(Console::Color::YELLOW) << 0 << Console::color\_code(Console::Color::RESET) << endl;

this->display\_array(colors);

for (vector<int>::size\_type i = 0; i < this->steps.size(); i++)

{

Console::move\_up(2);

cout << "Steps done: " << Console::color\_code(Console::Color::YELLOW) << (i + 1) << Console::color\_code(Console::Color::RESET) << endl;

colors[this->steps[i]] = Console::Color::BLUE;

Console::erase\_line();

display\_array(colors);

colors[this->steps[i]] = Console::Color::CYAN;

#ifdef \_WIN32

Sleep(1000);

#else

sleep(1)

#endif

}

Console::move\_up(1);

if (this->result\_index != (size\_t)(-1))

{

colors[this->result\_index] = Console::Color::GREEN;

display\_array(colors);

cout << "Value found at the position " << Console::color\_code(Console::Color::YELLOW) << this->result\_index << Console::color\_code(Console::Color::RESET) << endl;

}

else

{

display\_array(colors);

cout << Console::color\_code(Console::Color::RED) << "Value was not found" << Console::color\_code(Console::Color::RESET) << endl;

}

}

};

class SearchAlgorithms

{

public:

static void binary\_search(SearchData &data)

{

vector<int>::size\_type left = 0;

vector<int>::size\_type right = data.array.size();

while (left <= right)

{

int position = left + (right - left) / 2;

data.steps.push\_back(position);

if (data.array[position] == data.search\_value)

{

data.result\_index = position;

break;

}

if (data.array[position] < data.search\_value)

{

left = position + 1;

}

else

{

right = position - 1;

}

}

}

static void linear\_search(SearchData &data)

{

for (vector<int>::size\_type i = 0; i < data.array.size(); i++)

{

data.steps.push\_back(i);

if (data.array[i] == data.search\_value)

{

data.result\_index = i;

return;

}

}

}

};

int main(void)

{

SearchData search\_data;

std::cout << Console::color\_code(Console::Color::BLUE) << "Binary search" << Console::color\_code(Console::Color::RESET) << endl;

SearchAlgorithms::binary\_search(search\_data);

search\_data.render\_search();

std::cout << endl;

search\_data.clear();

std::cout << Console::color\_code(Console::Color::BLUE) << "Linear search" << Console::color\_code(Console::Color::RESET) << endl;

SearchAlgorithms::linear\_search(search\_data);

search\_data.render\_search();

return EXIT\_SUCCESS;}

**Приложения Б**