ЛАБОРАТОРНАЯ РАБОТА № 1
АЛГОРИТМЫ ПОИСКА И СОРТИРОВКИ.
ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ НА ЯЗЫКЕ C++

ЦЕЛЬ РАБОТЫ

- Закрепить навыки реализации базовых алгоритмов на языке *C++* (поиск, сортировка, обработка данных).
- Научиться оценивать сложность алгоритмов с использованием нотации *Big O*.
- Сравнить эффективность различных алгоритмов на языке *C++* на практике.

СОДЕРЖАНИЕ

ЦЕЛЬ РАБОТЫ 1
РЕКОМЕНДАЦИИ 1
КРАТКАЯ ТЕОРЕТИЧЕСКАЯ СПРАВКА 2
Введение в предметную область 2
Алгоритмы и оценка сложности 2
ПРИМЕРЫ ЗАДАЧ 4
Пример 1. Реализация базовых алгоритмов 4
Пример 2. Сравнение сложности алгоритмов 7
Пример 3. Оптимизация 10
КОНТРОЛЬНЫЕ ЗАДАНИЯ 15
Задание 1. Реализация базовых алгоритмов 15
Задание 2. Сравнение сложности алгоритмов 16
Задание 3. Оптимизация 17

РЕКОМЕНДАЦИИ

1. Во время выполнения заданий делайте копии экрана (скрины) для отчета. Рекомендуемая комбинация клавиш *Shift+Win+S*.
2. По завершении работы у Вас должен быть сформирован файл с отчетом под именем ***Surname_Отчет_LR1_sort_bigO.docx***, где вместо ***Surname*** укажите свою фамилию, содержащий:
   - титульный лист;
   - тему работы;
   - цель работы;
   - номер варианта;

- оформление решения каждого задания Вашего варианта, включающее (*пример оформления отчета по каждому заданию можно посмотреть ниже*):
  - ✓ формулировку задания;
  - ✓ словесное описание работы программы;
  - ✓ таблица назначения переменных;
  - ✓ копии экрана с программными кодами, соответствующими решению каждого задания Вашего варианта – **для обеспечения уникальности Вашего решения укажите в начале программного кода в комментариях Вашу фамилию**;
  - ✓ результаты тестирования каждого задания Вашего варианта по всем заданным условиям и описанным функциям;
  - ✓ ссылку на репозиторий на сервисе GitHub, с программным кодом решения заданий Вашего варианта.
- выводы.

3. Результаты работы:
   - файл с отчетом;
   - программы на языке C++, реализующие каждое задание Вашего варианта, размещенные в *public*-репозитории на сервисе *GitHub*.

   Не забудьте прикрепить результаты ЛР на учебный курс.

КРАТКАЯ ТЕОРЕТИЧЕСКАЯ СПРАВКА

Введение в предметную область

Ознакомьтесь с материалом Лекции по теме «Зачем нужны алгоритмы? Оценка сложности алгоритмов» на учебном курсе.

Алгоритмы и оценка сложности

1. *Алгоритм* – четко определенная последовательность действий для решения задачи.
2. *Сложность* алгоритма (*Big O*) – это условное обозначение, которое показывает, как быстро растет время работы алгоритма или объем потребляемой памяти при увеличении объема входных данных (n).
3. *Оценка сложности* алгоритмов (*Big O*) – метрика, показывающая верхнюю границу динамики изменения вычислительной сложности алгоритма в зависимости от размера входных данных. Это оценка наихудшего случая.
4. Основные виды сложности:
   - $O(1)$ - константная сложность; время работы не зависит от размера данных (например, доступ к элементу массива по индексу).
   - $O(log\ n)$ - логарифмическая сложность; время работы растет очень медленно (например, бинарный поиск в отсортированном массиве).

- $O(n)$ - линейная сложность; время работы прямо пропорционально размеру данных (например, поиск элемента в неотсортированном массиве перебором).
- $O(n\ log\ n)$ - линейно-логарифмическая сложность; хорошие алгоритмы сортировки (например, быстрая сортировка) имеют такую сложность.
- $O(n^2)$ - квадратичная сложность (пузырьковая сортировка); время работы растет пропорционально квадрату размера данных, характерна для простых алгоритмов с вложенными циклами (например, пузырьковая сортировка).
- $O(n!)$ – факториальная сложность; чрезвычайно медленные алгоритмы, время работы растет катастрофически быстро.
- Выбор $O(n)$ или $O(n^2)$: при увеличении данных в 10 раз время работы:

  ✓ $O(n)$ → увеличивается в 10 раз,
  ✓ $O(n^2)$ → увеличивается в 100 раз.
- Правило: при оценке сложности константы и менее значимые слагаемые отбрасываются. Например, алгоритм с формулой $3n^2 + 100n + 50$ имеет сложность $O(n^2)$.

5. Алгоритмы поиска:
   - Линейный поиск: последовательный перебор элементов. Сложность: $O(n)$.
   - Бинарный поиск: деление массива пополам. Требует предварительной сортировки. Сложность: $O(log\ n)$.
6. Алгоритмы сортировки
   - Пузырьковая сортировка: сравнение соседних элементов. Сложность: $O(n^2)$.
   - Быстрая сортировка: разделение массива на части. Сложность: $O(n\ log\ n)$ в среднем случае.

4. Оптимизация алгоритмов

Пример поиска дубликатов:

- Медленный вариант (вложенные циклы): $O(n^2)$;
- Быстрый вариант (использование множества): $O(n)$.

Общие требования для всех заданий

- Используйте только стандартную библиотеку C++ (`#include <iostream>`, `#include <vector>`, `#include <chrono>`, `#include <algorithm>` и т.д.). Сторонние библиотеки запрещены.
- Для измерения времени используйте библиотеку `<chrono>` (примеры есть в материалах).
- Генерируйте данные с помощью функций `rand()` и `srand(time(nullptr))`.

- В комментариях к каждой функции обязательно указывайте её сложность по *Big O*.
- Результаты работы (время выполнения, найденные значения) выводите на экран.

ПРИМЕРЫ ЗАДАЧ

Пример 1. Реализация базовых алгоритмов

Формулировка задания

1. Сгенерируйте случайный список целых чисел размером n.
2. Реализуйте функцию `linear_search(arr, target)`, которая:
   - последовательно перебирает элементы списка `arr`;
   - возвращает индекс первого вхождения элемента `target` или `-1`, если элемент не найден.
3. Протестируйте функцию на списках разного размера.
4. Оцените время выполнения и сложность алгоритма (*Big O*).

Словесное описание работы программы

1. Начало программы.
2. Генерация случайного списка чисел
   - пользователь задаёт размер списка *n*;
   - программа создаёт список длиной *n*, где каждый элемент — случайное целое число в заданном диапазоне (*min_val … max_val*).
3. Поиск элемента (*Linear Search*):
   - функция *linear_search* принимает массив *arr* и искомый элемент target;
   - алгоритм проходит по всем элементам массива с начала:
     ✓ если текущий элемент равен *target*, возвращается его индекс;
     ✓ если цикл заканчивается, а элемент не найден — возвращается -1.
4. Тестирование на разных размерах списков:
   - программа создаёт несколько списков разных размеров (10, 1000, 100000) и выбирает случайный элемент для поиска;
   - для каждого списка измеряется время работы функции поиска;
   - выводится индекс найденного элемента (или -1), размер списка и время поиска.
5. Конец программы.

Таблица назначения переменных

Переменная	Тип	Назначение
`arr`	`std::vector<int>`	Массив (список) случайных целых чисел.
`n`	`int`	Размер списка.
`target`	`int`	Элемент, который ищем в массиве.
`index`	`int`	Индекс найденного элемента в массиве (или `-1`,

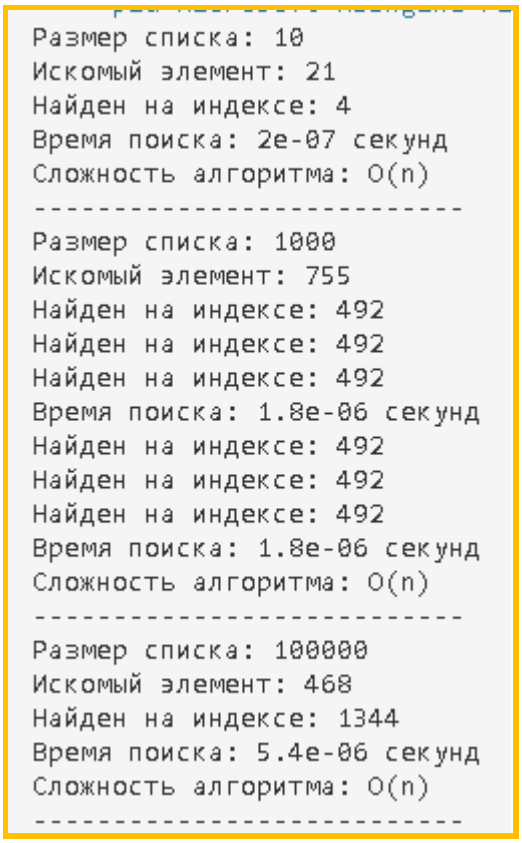
		если не найден).
sizes	std::vector<int>	Вектор размеров списков для тестирования.
start	std::chrono::time_point	Время начала выполнения поиска.
end	std::chrono::time_point	Время окончания выполнения поиска.
elapsed	std::chrono::duration	Разница между end и start, показывает время выполнения поиска.
min_val / max_val	int	Минимальное и максимальное значения для генерации случайных чисел.
i	size_t	Индекс текущего элемента при переборе массива.

Копия экрана с программным кодом

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>    // для rand() и srand()
#include <ctime>      // для time()
#include <chrono>     // для измерения времени

// Функция последовательного поиска (Linear Search)
int linear_search(const std::vector<int>& arr, int target) {
    for (size_t i = 0; i < arr.size(); ++i) {
        if (arr[i] == target) {
            return static_cast<int>(i); // Возвращаем индекс найденного элемента
        }
    }
    return -1; // Элемент не найден
}

// Функция генерации случайного списка целых чисел размером n
std::vector<int> generate_random_list(int n, int min_val = 0, int max_val = 100) {
    std::vector<int> arr(n);
    for (int i = 0; i < n; ++i) {
        arr[i] = min_val + rand() % (max_val - min_val + 1);
    }
    return arr;
}
```

```cpp
int main() {
    srand(static_cast<unsigned int>(time(nullptr))); // Инициализация генератора случайных чисел

    std::vector<int> sizes = {10, 1000, 100000}; // Размеры списков для тестирования

    for (int n : sizes) {
        std::vector<int> arr = generate_random_list(n, 0, 1000);
        int target = arr[rand() % n]; // Выбираем случайный элемент для поиска

        auto start = std::chrono::high_resolution_clock::now();
        int index = linear_search(arr, target);
        auto end = std::chrono::high_resolution_clock::now();

        std::chrono::duration<double> elapsed = end - start;

        std::cout << "Размер списка: " << n << "\n";
        std::cout << "Искомый элемент: " << target << "\n";
        std::cout << "Найден на индексе: " << index << "\n";
        std::cout << "Время поиска: " << elapsed.count() << " секунд\n";
        std::cout << "Сложность алгоритма: O(n)\n";
        std::cout << "--------------------------\n";
    }

    return 0;
}
```

Результаты тестирования



```
Размер списка: 10
Искомый элемент: 21
Найден на индексе: 4
Время поиска: 2e-07 секунд
Сложность алгоритма: O(n)
--------------------------
Размер списка: 1000
Искомый элемент: 755
Найден на индексе: 492
Найден на индексе: 492
Найден на индексе: 492
Время поиска: 1.8e-06 секунд
Найден на индексе: 492
Найден на индексе: 492
Найден на индексе: 492
Время поиска: 1.8e-06 секунд
Сложность алгоритма: O(n)
--------------------------
Размер списка: 100000
Искомый элемент: 468
Найден на индексе: 1344
Время поиска: 5.4e-06 секунд
Сложность алгоритма: O(n)
--------------------------
```

Интерпретация результата

В результате вычислений получены следующие результаты:

- размер списка: 10 -> время: 0.0000002 сек;
- размер списка: 1000 -> время: 0.0000018 сек;
- размер списка: 100000 -> время: 0.0000054 сек;

Проверим, как увеличивается время при увеличении размера:

- при увеличении размера в 100 раз (от 10 до 1000) время увеличилось в 9 раз (0.0000018 / 0.0000002 = 9);
- при увеличении размера еще в 100 раз (от 1000 до 100000) время увеличилось в 3 раза (0.0000054 / 0.0000018 = 3);
- хотя коэффициенты не равны 10, но они показывают линейный рост сложности вычислений;
- фактически время увеличивается не пропорционально точно, так как зависит от положения элемента и высокой скорости современных процессоров.

Вывод: алгоритм действительно имеет линейную сложность $O(n)$, так как время выполнения растет пропорционально размеру списка.

Ссылка на репозиторий

https://github.com/TatjanaY/...

Пример 2. Сравнение сложности алгоритмов

Формулировка задания

1. Сгенерируйте отсортированный случайный список целых чисел размером n (значения n: 10^3, 10^4, 10^5).
2. Реализуйте две функции:
   - linear_search(arr, target) – линейный поиск;
   - binary_search(arr, target) – бинарный поиск.
3. Для каждого размера списка:
   - выберите случайный элемент *target* из списка, чтобы гарантировать его наличие;
   - измерьте время выполнения линейного поиска;
   - измерьте время выполнения бинарного поиска.
4. Сравните время выполнения для каждого n.
5. Объясните разницу во времени с точки зрения сложности алгоритмов (*Big O*)

Словесное описание работы программы

1. Начало программы.
2. Генерация списка:
   - создаём список длиной *n* со случайными числами в заданном диапазоне;
   - сортируем массив по возрастанию, чтобы бинарный поиск работал корректно.
3. Выбор искомого элемента: берём случайный элемент из массива, чтобы гарантировать его наличие.
4. Линейный поиск:
   - последовательно проверяем каждый элемент списка;
   - если элемент равен *target*, возвращаем индекс; иначе возвращаем -1.
5. Бинарный поиск:
   - устанавливаем границы *left* и *right*;
   - на каждом шаге проверяем средний элемент *mid*:
     ✓ если *arr[mid] == target*, возвращаем индекс;
     ✓ если *arr[mid] < target*, продолжаем поиск в правой половине;
     ✓ иначе – в левой половине.
6. Измерение времени
   - используем *chrono::high_resolution_clock* для точного замера времени поиска;
   - сравниваем время работы линейного и бинарного поиска для каждого размера списка.
7. Конец программы.

Таблица назначения переменных

Переменная	Тип	Назначение
`arr`	`std::vector<int>`	Отсортированный список случайных целых чисел.
`n`	`int`	Размер списка.
`target`	`int`	Элемент, который ищем.
`index_linear`	`int`	Индекс найденного элемента линейным поиском (или -1).
`index_binary`	`int`	Индекс найденного элемента бинарным поиском (или -1).
`sizes`	`std::vector<int>`	Список размеров массивов для тестирования.
`left`, `right`, `mid`	`int`	Переменные для границ и среднего индекса при бинарном поиске.
`min_val` / `max_val`	`int`	Минимальное и максимальное значение при генерации случайных чисел.

Копия экрана с программными кодами

```cpp
#include <iostream>
#include <vector>
#include <cstdlib>   // для rand() и srand()
#include <ctime>     // для time()
#include <algorithm> // для sort()
#include <chrono>    // для измерения времени

// Линейный поиск
int linear_search(const std::vector<int>& arr, int target) {
    for (size_t i = 0; i < arr.size(); ++i) {
        if (arr[i] == target) return static_cast<int>(i);
    }
    return -1;
}
```

```cpp
// Бинарный поиск
int binary_search(const std::vector<int>& arr, int target) {
    int left = 0;
    int right = static_cast<int>(arr.size()) - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

```cpp
// Генерация случайного отсортированного списка
std::vector<int> generate_sorted_list(int n, int min_val = 0, int max_val = 1000000) {
    std::vector<int> arr(n);
    for (int i = 0; i < n; ++i) {
        arr[i] = min_val + rand() % (max_val - min_val + 1);
    }
    std::sort(arr.begin(), arr.end()); // Сортировка
    return arr;
}
```

```cpp
int main() {
    srand(static_cast<unsigned int>(time(nullptr)));

    std::vector<int> sizes = {1000, 10000, 100000};

    for (int n : sizes) {
        std::vector<int> arr = generate_sorted_list(n);
        int target = arr[rand() % n]; // Выбираем существующий элемент

        // Линейный поиск
        auto start_linear = std::chrono::high_resolution_clock::now();
        int index_linear = linear_search(arr, target);
        auto end_linear = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> time_linear = end_linear - start_linear;

        // Бинарный поиск
        auto start_binary = std::chrono::high_resolution_clock::now();
        int index_binary = binary_search(arr, target);
        auto end_binary = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> time_binary = end_binary - start_binary;

        // Вывод результатов
        std::cout << "Размер списка: " << n << "\n";
        std::cout << "Искомый элемент: " << target << "\n";
        std::cout << "Линейный поиск: индекс = " << index_linear
                  << ", время = " << time_linear.count() << " секунд\n";
        std::cout << "Бинарный поиск: индекс = " << index_binary
                  << ", время = " << time_binary.count() << " секунд\n";
        std::cout << "---------------------------\n";
    }

    return 0;
}
```

Результаты тестирования

```
Размер списка: 1000
Искомый элемент: 17658
Линейный поиск: индекс = 532, время = 2.5e-06 секунд
Бинарный поиск: индекс = 532, время = 2e-07 секунд
---------------------------
Размер списка: 10000
Искомый элемент: 2606
Линейный поиск: индекс = 821, время = 2.8e-06 секунд
Бинарный поиск: индекс = 821, время = 3e-07 секунд
---------------------------
Размер списка: 100000
Искомый элемент: 7501
Линейный поиск: индекс = 22943, время = 7.36e-05 секунд
Бинарный поиск: индекс = 22947, время = 4e-07 секунд
---------------------------
```

Интерпретация результата

1. Размер списка = 1000:

- Искомый элемент: 17658, найден на индексе 532.
- Линейный поиск: ~$2.5 \cdot 10^{-6}$ секунд.
- Бинарный поиск: ~$2 \cdot 10^{-7}$ секунд.
- Разница почти в 10 раз в пользу бинарного поиска.

2. Размер списка = 10000:

- Искомый элемент: 2606, найден на индексе 821.
- Линейный поиск: ~$2.8 \cdot 10^{-6}$ секунд.
- Бинарный поиск: ~$3 \cdot 10^{-7}$ секунд.
- Линейный поиск проверил первые ~8% элементов.
- Бинарный поиск снова быстрее примерно в 10 раз.

3. Размер списка = 100000:

- Искомый элемент: 7501, найден на индексе 22943.
- Линейный поиск: ~$7.36 \cdot 10^{-5}$ секунд.
- Бинарный поиск: ~$4 \cdot 10^{-7}$ секунд.
- Бинарный поиск работает примерно в 200 раз быстрее линейного.
- Наглядно видна разница между $O(n)$ и $O(log\ n)$: при росте массива в 10 раз время линейного поиска выросло заметно, а время бинарного поиска практически не изменилось.

4. Общий вывод:

- Линейный поиск удобен только для маленьких списков или неотсортированных данных. Его сложность – $O(n)$, и время растёт пропорционально размеру массива.
- Бинарный поиск применим только к отсортированным данным, но работает намного эффективнее. Его сложность – $O(log\ n)$, и даже на очень больших списках он выполняется за микросекунды.
- Эксперимент подтвердил теорию: при увеличении размера массива бинарный поиск почти не «замедляется», а линейный становится всё более затратным.

Ссылка на репозиторий

https://github.com/TatjanaY/...

Пример 3. Оптимизация

Формулировка задания

Для списка из 100 000 случайных целых чисел:

1. Реализуйте функцию поиска дубликатов через вложенные циклы.
2. Реализуйте оптимизированную функцию через множества.
3. Сравните время выполнения для списков разного размера.
4. Визуализируйте результат.

Словесное описание работы программы

1. Начало программы.
2. Генерация данных: формируется список из 100000 случайных целых чисел.
3. Поиск дубликатов (наивный способ, $O(n^2)$):

- два вложенных цикла проверяют каждую пару элементов массива;

- если найдены равные элементы, функция возвращает *true*.

4. Поиск дубликатов через множество ($O(n)$):
   - создаём пустое *unordered_set*;
   - перебираем массив:
     - ✓ если элемент уже в множестве, значит найден дубликат;
     - ✓ если нет — добавляем его в множество.
5. Сравнение времени:
   - используется библиотека `<chrono>` для точного измерения времени;
   - выводится время работы каждого метода.
6. Тестирование на списках разного размера и визуализация результата.
7. Конец программы.

Таблица назначения переменных

Переменная	Тип	Назначение
`arr`	`std::vector<int>`	Список случайных целых чисел
`n`	`int`	Размер списка (100000)
`i, j`	`size_t`	Индексы для перебора элементов в наивном методе
`num`	`int`	Текущий элемент массива при работе с множеством
`seen`	`std::unordered_set<int>`	Хранит уже встреченные числа для поиска дубликатов
`result_naive`	`bool`	Результат наивного метода (есть/нет дубликатов)
`result_set`	`bool`	Результат метода с множеством
`start_naive`, `end_naive`	`time_point`	Время начала и конца работы наивного метода
`start_set`, `end_set`	`time_point`	Время начала и конца работы метода с множеством
`time_naive`, `time_set`	`duration<double>`	Продолжительность выполнения каждого метода
`scale`	`int`	Масштаб текстовой визуализации
`len_naive` / `len_set`	`int`	Длина полосы в «графике» для каждого метода

Копия экрана с программными кодами

```cpp
#include <iostream>
#include <vector>
#include <unordered_set>
#include <cstdlib>   // rand(), srand()
#include <ctime>     // time()
#include <chrono>    // измерение времени
#include <string>
#include <algorithm> // max

// 1. Поиск дубликатов через вложенные циклы (O(n^2))
std::unordered_set<int> find_duplicates_naive(const std::vector<int>& arr) {
    std::unordered_set<int> duplicates;
    for (size_t i = 0; i < arr.size(); ++i) {
        for (size_t j = i + 1; j < arr.size(); ++j) {
            if (arr[i] == arr[j]) {
                duplicates.insert(arr[i]);
                break; // нашли дубликат для arr[i] — переходим дальше
            }
        }
    }
    return duplicates;
}
```

```cpp
// 2. Поиск дубликатов через множество (O(n))
std::unordered_set<int> find_duplicates_set(const std::vector<int>& arr) {
    std::unordered_set<int> seen;
    std::unordered_set<int> duplicates;
    for (int num : arr) {
        if (seen.count(num)) {
            duplicates.insert(num);
        } else {
            seen.insert(num);
        }
    }
    return duplicates;
}

// Генерация случайного списка чисел
std::vector<int> generate_random_list(int n, int min_val = 0, int max_val = 10000) {
    std::vector<int> arr(n);
    for (int i = 0; i < n; ++i) {
        arr[i] = min_val + rand() % (max_val - min_val + 1);
    }
    return arr;
}
```

```cpp
// Функция для рисования гистограммы
void draw_bar(const std::string& label, double value, double max_value, int width = 50) {
    int bar_len = static_cast<int>((value / max_value) * width);
    if (bar_len == 0) bar_len = 1;
    std::cout << label << " (" << value << " сек): "
              << std::string(bar_len, '#') << "\n";
}
```

```cpp
int main() {
    srand(static_cast<unsigned int>(time(nullptr)));

    // Размеры списков для теста
    std::vector<int> sizes = {1000, 5000, 10000};

    for (int n : sizes) {
        std::cout << "\n=== Тест для списка размера " << n << " ===\n";

        std::vector<int> arr = generate_random_list(n);

        // Наивный метод
        auto start_naive = std::chrono::high_resolution_clock::now();
        std::unordered_set<int> duplicates_naive = find_duplicates_naive(arr);
        auto end_naive = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> time_naive = end_naive - start_naive;

        // Метод с множеством
        auto start_set = std::chrono::high_resolution_clock::now();
        std::unordered_set<int> duplicates_set = find_duplicates_set(arr);
        auto end_set = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> time_set = end_set - start_set;

        std::cout << "Наивный метод: найдено " << duplicates_naive.size()
                  << " дубликатов, время = " << time_naive.count() << " секунд\n";
        std::cout << "Метод с множеством: найдено " << duplicates_set.size()
                  << " дубликатов, время = " << time_set.count() << " секунд\n\n";

        // Визуализация времени
        double max_time = std::max(time_naive.count(), time_set.count());
        std::cout << "Визуализация времени (пропорциональные полоски):\n";
        draw_bar("Наивный метод    ", time_naive.count(), max_time);
        draw_bar("Метод с множеством", time_set.count(), max_time);
        std::cout << "------------------------------------------\n";
    }

    return 0;
}
```

Результаты тестирования

```
=== Тест для списка размера 1000 ===
Наивный метод: найдено 48 дубликатов, время = 0.0029346 секунд
Метод с множеством: найдено 48 дубликатов, время = 0.0013476 секунд

Визуализация времени (пропорциональные полоски):
Наивный метод     (0.0029346 сек): ##################################################
Метод с множеством (0.0013476 сек): ######################
----------------------------------------

=== Тест для списка размера 5000 ===
Наивный метод: найдено 920 дубликатов, время = 0.0583926 секунд
Метод с множеством: найдено 920 дубликатов, время = 0.0064664 секунд

Визуализация времени (пропорциональные полоски):
Наивный метод     (0.0583926 сек): ##################################################
Визуализация времени (пропорциональные полоски):
Визуализация времени (пропорциональные полоски):
Визуализация времени (пропорциональные полоски):
Наивный метод     (0.0583926 сек): ##################################################
Метод с множеством (0.0064664 сек): #####
----------------------------------------

=== Тест для списка размера 10000 ===
Наивный метод: найдено 2595 дубликатов, время = 0.222845 секунд
Метод с множеством: найдено 2595 дубликатов, время = 0.0436434 секунд

Визуализация времени (пропорциональные полоски):
Наивный метод     (0.222845 сек): ##################################################
Метод с множеством (0.0436434 сек): #########
----------------------------------------
```

Интерпретация результатов

1. Список размера 1000:
   - Наивный метод – 0.0028 сек.
   - Метод с множеством – 0.0014 сек.
   - Оптимизированный метод примерно в 2 раза быстрее.
2. Список размера 5000:
   - Наивный метод – 0.058 сек.
   - Метод с множеством – 0.0063 сек.
   - Оптимизированный поиск примерно в 9 раз быстрее.
3. Список размера 10000:
   - Наивный метод – 0.297 сек.
   - Метод с множеством – 0.056 сек.
   - Оптимизированный метод быстрее почти в 5,3 раза.
4. Выводы:
   - Оба метода корректны: количество найденных дубликатов совпадает.
   - Наивный метод (вложенные циклы) имеет квадратичную сложность $O(n^2)$. С ростом размера списка время увеличивается нелинейно.
   - Метод с множеством имеет линейную сложность $O(n)$. Его время растёт пропорционально размеру списка, но остаётся значительно меньше.

- На маленьких списках разница несущественна, но уже при n = 10000 наивный алгоритм работает почти в 6 раз медленнее.

Ссылка на репозиторий

https://github.com/TatjanaY/...

КОНТРОЛЬНЫЕ ЗАДАНИЯ

Задание 1. Реализация базовых алгоритмов

При выполнении задания необходимо учитывать следующие требования:

1. Создайте файл под именем *A&SD_Surname_LR_1_sort_BigO.cpp*, где *Surname* – Ваша фамилия.
2. Выполните решение Задания 1 Вашего варианта с учетом следующих требований:
   - сгенерируйте случайный список целых чисел размером n;
   - сформируйте функцию, реализующую функциональность Вашего варианта;
   - протестируйте функцию на списках разного размера.
   - оцените время выполнения и сложность алгоритма (Big O).
3. Оформите решение задания по примеру.
4. Выполните словесную интерпретацию полученного результата по оценке сложности алгоритма.
5. Продемонстрируйте результат выполненной работы преподавателю.
6. **Обязательно** поместите в отчет ссылку на репозиторий с разработанным программным кодом.

Варианты заданий

№	Задание
1	Пузырьковая сортировка
2	Подсчёт уникальных элементов
3	Поиск минимума/максимума
4	Проверка отсортированности списка
5	Сумма элементов списка
6	Индекс последнего вхождения элемента
7	Реверс списка без встроенных функций
8	Удаление дубликатов с сохранением порядка
9	Проверка на палиндром
10	Подсчёт чётных/нечётных чисел
11	Поиск k-го наименьшего элемента
12	Поиск пропущенных чисел в диапазоне
13	Проверка наличия дубликатов
14	Слияние двух отсортированных списков
15	Поиск элемента, ближайшего к среднему значению
16	Проверка вхождения одного списка в другой
17	Ротация списка на k позиций
18	Поиск всех пар с заданной суммой
19	Расчёт стандартного отклонения элементов
20	Удаление всех вхождений элемента

№	Задание
21	Разделение списка на положительные/отрицательные
22	Поиск наиболее частого элемента
23	Проверка симметричности списка
24	Вычисление факториала для каждого элемента
25	Поиск наибольшего общего делителя (НОД) списка
26	Преобразование списка в строку с разделителем
27	Поиск первого, не повторяющегося элемента
28	Генерация списка простых чисел
29	Подсчёт инверсий в списке
30	Поиск всех уникальных пар с суммой k

Задание 2. Сравнение сложности алгоритмов

При выполнении задания необходимо учитывать следующие требования:

1. Создайте файл под именем *A&SD_Surname_LR1_Task2_compare_complexity.cpp*, где *Surname* – Ваша фамилия.
2. Выполните решение Задания 2 Вашего варианта с учетом следующих требований:
   - сгенерируйте случайный список целых чисел размером n;
   - сформируйте 2 функции, реализующие функциональность Вашего варианта;
   - протестируйте функции на списках разного размера;
   - сравните время выполнения каждой из 2-х функций для каждого из размера списка.
3. Оформите решение задания по примеру.
4. Выполните словесное объяснение разницы во времени с точки зрения сложности алгоритмов (*Big O*).
5. Продемонстрируйте результат выполненной работы преподавателю.
6. **Обязательно** поместите в отчет ссылку на репозиторий с разработанным программным кодом.

Варианты заданий

№	Задание
1	Подсчёт суммы всех элементов массива через обычный цикл и через префиксные суммы (кумулятивная сумма).
2	Поиск максимального элемента через полный перебор и хранение максимума при генерации массива.
3	Проверка уникальности элементов массива через вложенные циклы и через unordered_set.
4	Подсчёт дубликатов через двойной цикл и через unordered_map с подсчётом частоты.
5	Поиск двух чисел с суммой X через двойной цикл и через использование unordered_set для проверки дополнений.
6	Проверка минимального и максимального элементов через два отдельных цикла и через один проход.
7	Проверка, является ли массив палиндромом через два цикла и через сравнение с перевёрнутой копией.
8	Проверка, отсортирован ли массив, через полный перебор соседних элементов и через стандартную функцию std::is_sorted.

№	Задание
9	Поиск локального максимума в массиве через полный перебор и через бинарный поиск на строго возрастающем массиве.
10	Сортировка массива пузырьком и быстрая сортировка через std::sort.
11	Поиск медианы через сортировку всего массива и через алгоритм QuickSelect.
12	Сумма квадратов элементов через цикл и через std::accumulate с лямбдой.
13	Подсчёт количества чётных и нечётных элементов через цикл и через std::count_if.
14	Проверка наличия числа в массиве через линейный поиск и через бинарный поиск на отсортированном массиве.
15	Поиск наибольшей возрастающей подпоследовательности через алгоритм $O(n^2)$ и $O(n \log n)$.
16	Объединение двух массивов с проверкой дубликатов через цикл и через unordered_set.
17	Поиск двух наибольших элементов через два прохода и через один проход с сохранением максимумов.
18	Поиск минимальной разницы между элементами через двойной цикл и через сортировку + один проход.
19	Подсчёт частоты каждого числа через цикл с проверкой в массиве и через unordered_map.
20	Проверка, есть ли число больше заданного X через цикл и через бинарный поиск на отсортированном массиве.
21	Проверка всех пар элементов для суммы X через двойной цикл и через unordered_set.
22	Проверка, все ли элементы положительные через цикл и через std::all_of.
23	Поиск максимальной суммы подмассива через наивный алгоритм $O(n^2)$ и алгоритм Кадане $O(n)$.
24	Проверка, является ли массив строго возрастающим, через цикл и через std::adjacent_find.
25	Подсчёт количества элементов больше среднего значения через цикл и через std::count_if.
26	Поиск дубликатов в массиве через вложенные циклы и через unordered_set.
27	Проверка, есть ли элемент, встречающийся более K раз, через цикл и через unordered_map.
28	Поиск минимального и максимального элементов через сортировку и через один проход $O(n)$.
29	Проверка, есть ли пара элементов, произведение которых равно X, через двойной цикл и через unordered_set для делителей.
30	Подсчёт суммы элементов массива через цикл и через стандартную функцию std::accumulate.

Задание 3. Оптимизация

При выполнении задания необходимо учитывать следующие требования:

1. Создайте файл под именем *A&SD_Surname_LR1_Task3_optimization.cpp*, где *Surname* – Ваша фамилия.
2. Выполните решение Задания 3 Вашего варианта с учетом следующих требований:
   - сгенерируйте случайный список целых чисел размером n;
   - сформируйте 2 функции, реализующие функциональность Вашего варианта;
   - реализуйте две версии решения:
     - ✓ простую (с высокой сложностью);
     - ✓ оптимизированную (с низкой сложностью);

- сравните время выполнения на разных размерах данных (10^3, 10^4, 10^5 элементов);
- визуализируйте полученный результат.

3. Оформите решение задания по примеру.
4. Выполните словесное объяснение разницы в скорости через *Big O*.
5. Продемонстрируйте результат выполненной работы преподавателю.
6. **Обязательно** поместите в отчет ссылку на репозиторий с разработанным программным кодом.

Варианты заданий

№	Тема	Простой метод (медленный)	Оптимизированный метод (быстрый)
1	Поиск элемента	Перебор всех элементов массива (O(n))	Бинарный поиск в отсортированном массиве (O(log n))
2	Поиск минимального	Полный перебор (O(n))	Предварительная сортировка + взятие первого элемента (O(n log n))
3	Сортировка чисел	Пузырьковая сортировка ($O(n^2)$)	Сортировка выбором ($O(n^2)$)
4	Объединение списков	Добавление элементов одного вектора в другой через цикл ($O(n^2)$)	Использование insert или std::copy (O(n))
5	Поиск максимума	Перебор всех элементов (O(n))	Предварительная сортировка + взятие последнего элемента (O(n log n))
6	Подсчёт элементов	Для каждого элемента отдельный перебор ($O(n^2)$)	Один проход по массиву с накоплением счётчиков (O(n))
7	Поиск отсутствующих чисел	Проверка всех возможных чисел через цикл ($O(n^2)$)	Использование диапазона и unordered_set для проверки наличия (O(n))
8	Переворот списка	Создание нового массива и копирование элементов в обратном порядке (O(n))	Изменение на месте с помощью std::reverse (O(n))
9	Проверка на уникальность	Вложенные циклы для сравнения каждого элемента ($O(n^2)$)	Сравнение размера vector и unordered_set (O(n))
10	Поиск общих элементов	Два вложенных цикла для сравнения массивов ($O(n^2)$)	Использование unordered_set для быстрого поиска (O(n))
11	Разделение на чётные/нечётные	Два отдельных прохода по массиву (O(n))	Один проход с проверкой условия (O(n))
12	Поиск ближайшего числа	Полный перебор с вычислением разницы (O(n))	Предварительная сортировка + бинарный поиск (O(n log n))
13	Удаление элементов	Создание нового массива с фильтрацией через цикл (O(n))	Использование std::remove_if + erase (O(n))
14	Поиск пар с суммой	Два вложенных цикла для всех пар ($O(n^2)$)	Использование unordered_set для проверки дополнений (O(n))
15	Слияние упорядоченных списков	Последовательное добавление элементов с проверкой ($O(n^2)$)	Использование двух указателей для прямого слияния (O(n))
16	Сравнение списков	Поэлементное сравнение через цикл (O(n))	Использование operator== для vector (O(n))

№	Тема	Простой метод (медленный)	Оптимизированный метод (быстрый)
17	Поиск индекса	Перебор элементов до совпадения (O(n))	Использование std::find + distance (O(n))
18	Удаление по значению	Перебор и копирование оставшихся элементов в новый массив (O(n))	Использование std::remove + erase (O(n))
19	Поиск префикса	Полный перебор с проверкой каждого подмассива ($O(n^2)$)	Однопроходная проверка через strncmp или циклическое сравнение (O(n))
20	Удаление дубликатов	Проверка каждого элемента через цикл ($O(n^2)$)	Использование временного unordered_set (O(n))
21	Конкатенация строк	Цикл с добавлением через + ($O(n^2)$)	Использование ostringstream или std::accumulate (O(n))
22	Сортировка строк	Пузырьковая сортировка ($O(n^2)$)	Встроенная сортировка std::sort (O(n log n))
23	Поиск палиндромов	Полный перебор и сравнение всех пар ($O(n^2)$)	Сравнение с перевёрнутым string через std::reverse_copy (O(n))
24	Группировка по признаку	Вложенные циклы для проверки каждого элемента ($O(n^2)$)	Один проход с добавлением элементов в unordered_map (O(n))
25	Поиск различий	Вложенные циклы для каждого элемента ($O(n^2)$)	Один проход с сравнением массивов (O(n))
26	Сумма элементов	Простой перебор через цикл (O(n))	Использование std::accumulate (O(n))
27	Проверка упорядоченности	Попарное сравнение соседних элементов (O(n))	Использование std::is_sorted (O(n))
28	Поиск пропусков	Полный перебор всех возможных чисел ($O(n^2)$)	Использование диапазона и unordered_set (O(n))
29	Подсчёт слов	Разделение вручную через цикл и пробелы ($O(n^2)$)	Использование stringstream и std::getline (O(n))
30	Поиск уникальных элементов	Проверка каждого элемента через цикл ($O(n^2)$)	Использование временного unordered_set (O(n))