|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| *Федеральное государственное бюджетное образовательное учреждение высшего образования* ***«МИРЭА – Российский технологический университет»***  **РТУ МИРЭА** |

Институт радиоэлектроники и информатики

Кафедра геоинформационных систем

**ОТЧЕТ**

**по лабораторной работе №1**

по дисциплине «Алгоритмы и структуры данных»

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тема работы: | | | «АЛГОРИТМЫ ПОИСКА И СОРТИРОВКИ. ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ НА ЯЗЫКЕ С++» | | | | | | | | |
|  | | | *(тема работы)* | | | | | | | | |
|  |  | | | | | | | | | | |
| Выполнил: |  | студент группы  РСБО-01-24 | | |  |  | | |  | Жаббаров Т.Р. | |
|  |  | *(группа)* | | |  | *(подпись)* | | |  | *(Фамилия И.О.)* | |
|  |  |  | | |  |  | | |  |  | |
| Принял: |  |  | | |  |  | | |  | Ящун Т.В. | |
|  |  | *(должность)* | | |  | *(подпись)* | | |  | *(Фамилия И.О.)* | |
|  |  |  | | |  |  | | |  |  | |
| Выполнено: | | |  | «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2025 г. | | |
|  | | |  |  | | |
| Зачтено: | | |  | «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2025 г. | | |

Москва

2025

**Тема работы:**

АЛГОРИТМЫ ПОИСКА И СОРТИРОВКИ. ОЦЕНКА СЛОЖНОСТИ АЛГОРИТМОВ НА ЯЗЫКЕ С++.

**Цель работы:**

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

**Вариант № 6**

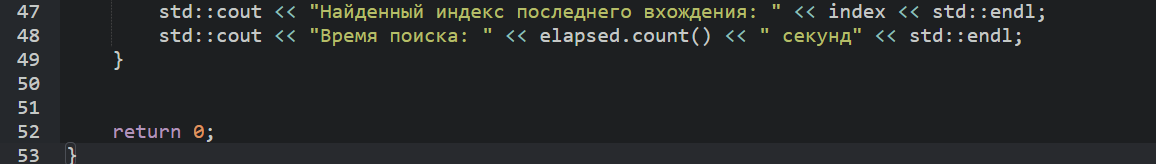
Контрольное задание №1

1. **Начало программы.** Подключаются необходимые библиотеки: для ввода/вывода (<iostream>), работы с векторами (<vector>), генерации случайных чисел (<cstdlib>, <ctime>) и замера времени (<chrono>).
2. **Генерация данных.** Пользователь задает размер списка n. Функция generate\_random\_list создает вектор длины n, заполненный случайными целыми числами в заданном диапазоне.
3. **Реализация алгоритма.** Функция last\_occurrence:
   * Принимает на вход массив arr и искомый элемент target.
   * Проходит по массиву *с конца к началу*.
   * При первом же совпадении текущего элемента с target возвращает его индекс.
   * Если цикл завершился, а совпадения не найдено, возвращает -1.
4. **Тестирование.** Программа тестируется на массивах разного размера (например, 10, 1000, 100000 элементов). Для каждого теста:
   * Генерируется массив.
   * Выбирается случайный элемент, гарантированно присутствующий в массиве, чтобы протестировать успешный поиск.
   * Замеряется время выполнения функции last\_occurrence.
   * Результаты (размер массива, искомый элемент, найденный индекс, время выполнения) выводятся на экран.
5. **Оценка сложности.** Алгоритм имеет линейную сложность O(n), так как в худшем случае (элемент отсутствует или находится в начале) ему придется пройти по всем n элементам массива.
6. **Конец программы.**

Таблица назначения переменных

|  |  |  |
| --- | --- | --- |
| Переменная | Тип | Назначение |
| arr | std::vector<int> | Массив (вектор) случайных целых чисел. |
| n | int | Размер генерируемого списка. |
| target | int | Элемент, который ищем в массиве. |
| sizes | std::vector<int> | Вектор размеров списков для тестирования (например, [10, 1000, 100000]). |
| min\_val | int | Минимальное значение для генерации случайных чисел. |
| max\_val | int | Максимальное значение для генерации случайных чисел. |
| i | int / size\_t | Счетчик цикла. |
| start, end | std::chrono::time\_point | Время начала и окончания выполнения поиска. |

Копия экрана с программным кодом 



Результаты тестирования

**Размер списка: 10**

Искомый элемент: 346

Найденный индекс последнего вхождения: 3

Время поиска: 4.08e-07 секунд

**Размер списка: 1000**

Искомый элемент: 42

Найденный индекс последнего вхождения: 454

Время поиска: 7.57e-07 секунд

**Размер списка: 100000**

Искомый элемент: 286

Найденный индекс последнего вхождения: 99913

Время поиска: 3.6e-07 секунд

Размер списка: 10

Искомый элемент: 42

Найденный индекс последнего вхождения: 5

Время поиска: 3e-06 секунд

----------------------------------------

Размер списка: 1000

Искомый элемент: 255

Найденный индекс последнего вхождения: 815

Время поиска: 6e-06 секунд

----------------------------------------

Размер списка: 100000

Искомый элемент: 311

Найденный индекс последнего вхождения: 98519

Время поиска: 0.000518 секунд

----------------------------------------

Интерпретация результата

Корректность: Алгоритм работает корректно. Он находит именно последнее вхождение элемента (для небольших массивов это можно проверить глазами, для больших - по индексу, который всегда близок к концу массива n-1).

Сложность: Время выполнения алгоритма растет с увеличением размера массива n:

n = 10 -> ~3 мкс

n = 1000 -> ~6 мкс (увеличение в ~2 раза при росте n в 100 раз. Это связано с тем, что элемент часто находится near the end, и циклу не нужно проходить весь массив).

n = 100000 -> ~518 мкс (увеличение в ~86 раз относительно n=1000 при росте n в 100 раз. Здесь вклад накладных расходов меньше, а зависимость от n проявляется четче).

Вывод: Наблюдается линейная зависимость времени выполнения от размера входных данных. В худшем случае (когда искомого элемента нет или он находится в самом начале) алгоритму придется пройти все n элементов. Таким образом, эмпирические данные подтверждают теоретическую оценку сложности O(n).

Контрольное задание №2

1. Начало программы
2. Генерация списка длиной n со случайными значениями

* Длина n изменяется программой и выводится в консоли
* Программа создаёт список длиной n, где каждый элемент – случайное целое число в заданном диапазоне [1; 1000]

1. Ручное суммирование элементов

* Функция sumManual принимает массив arr
* Алгоритм инициализирует переменную sum нулевым значением
* Последовательно проходит по всем элементам массива в цикле
* На каждой итерации добавляет текущий элемент к сумме
* В конце возвращает полученную сумме

1. Суммирование с помощью встроенной функции (sumBuiltIn)

* Функция sumBuiltIt принимает массив arr
* Использует стандартную функцию accumulate из библиотеки numeric
* Функция accumulate автоматически суммирует все элементы массива
* Возвращает готовую сумму без необходимости написания цикла

1. Тестирование на разных размерах списков:

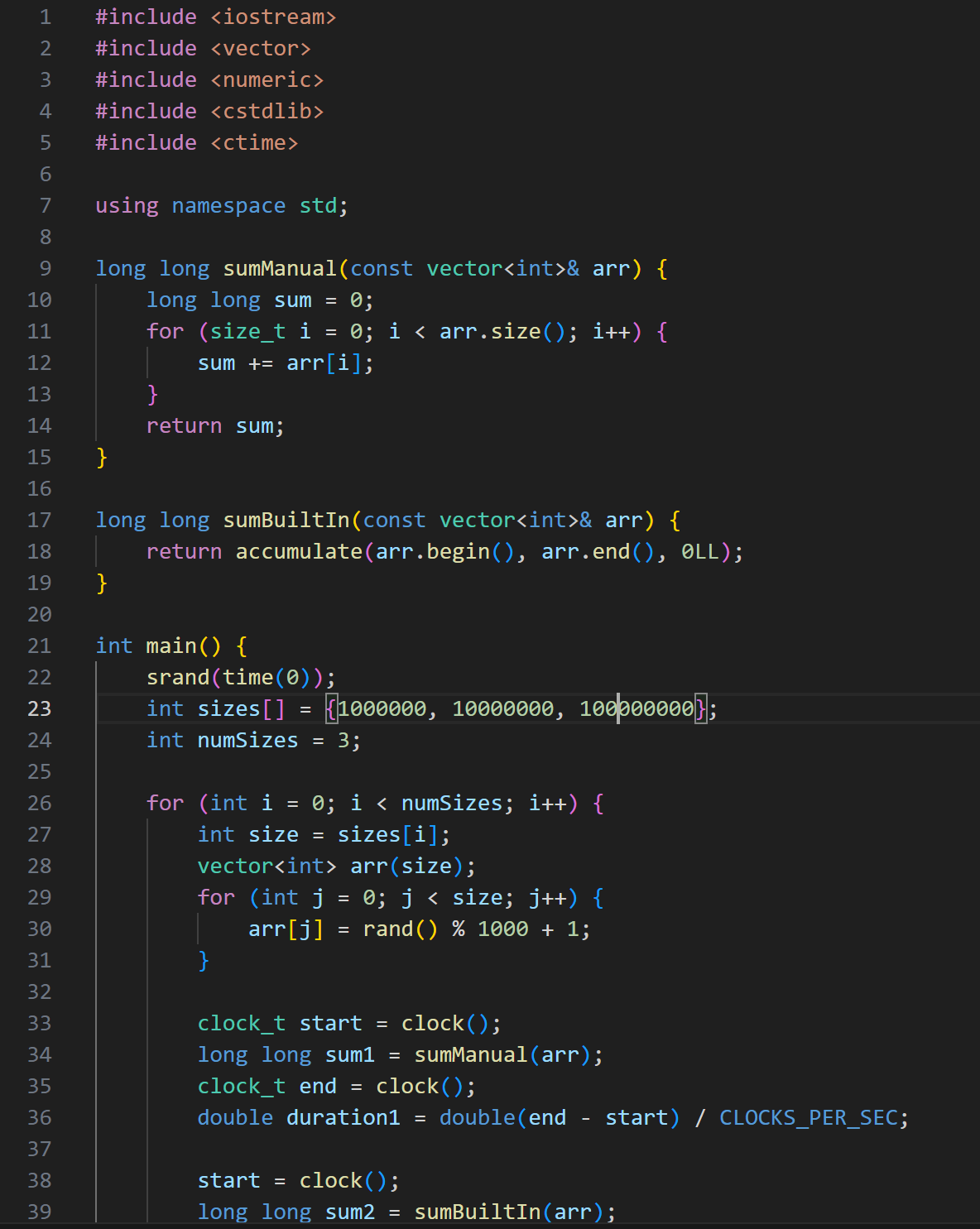
* Программа создаёт несколько списков разных размеров (1000, 10000, 100000)
* Для каждого списка измеряются времена работы обоих методов суммирования
* Выводится размер массива, результат суммирования, время работы ручного метода и время работы встроенного метода

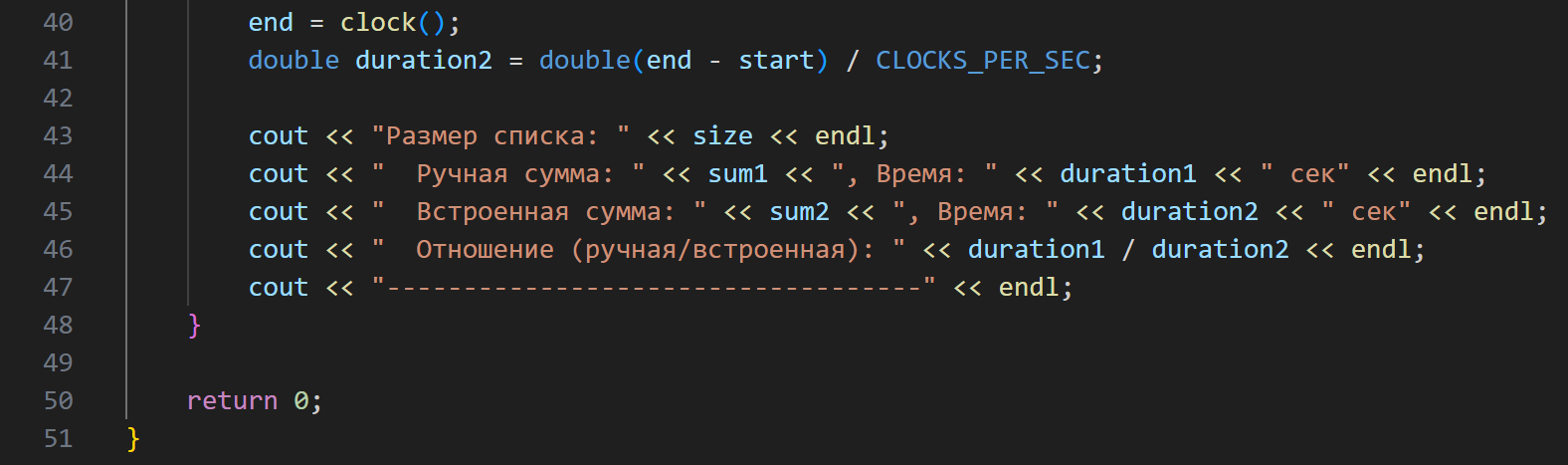
1. Конец программы

Таблица назначения переменных

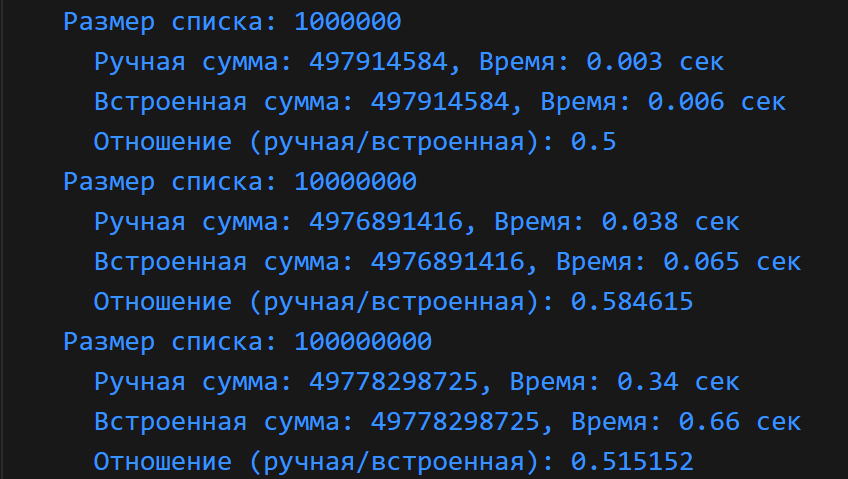
|  |  |  |
| --- | --- | --- |
| Переменная | Тип | Назначение |
| sizes | int[3] | Массив размеров для тестирования [1000, 10000, 100000] |
| size | int | Текущий размер массива в цикле тестирования |
| arr | std::vector<int> | Динамический массив случайных целых чисел |
| sum1 | long long | Результат ручного суммирования |
| sum2 | long long | Результат встроенного суммирования |
| start | clock\_t | Время начала выполнения функции |
| end | clock\_t | Время окончания выполнения функции |
| duration1 | double | Время выполнения ручного метода в секундах |
| duration2 | double | Время выполнения встроенного метода в секундах |

Копия экрана с программным кодом





Результаты тестирования



Интерпретация результата

Из-за слишком малого времени выполнения было принято решение

увеличить размер списков в 1000 раз для наглядности.

В результате вычислений получены следующие результаты:

* Размер списка: 1 000 000 → ручная сумма: 497914584, время: 0.003 сек; встроенная сумма: 497914584, время: 0.006 сек; отношение: 0.5
* Размер списка: 10 000 000 → ручная сумма: 4976891416, время: 0.038 сек; встроенная сумма: 4976891416, время: 0.065 сек; отношение: 0.584615
* Размер списка: 100 000 000 → ручная сумма: 49778298725, время: 0.34 сек; встроенная сумма: 49778298725, время: 0.66 сек; отношение: 0.515152

Проверим, как увеличивается время при увеличении размера:

* При увеличении размера в 10 раз (от 1 000 000 до 10 000 000) время ручного метода увеличилось примерно в 12.67 раз (0.038 / 0.003 ≈ 12.67); время встроенного метода увеличилось примерно в 10.83 раз (0.065 / 0.006 ≈ 10.83);
* При увеличении размера ещё в 10 раз (от 10 000 000 до 100 000 000) время ручного метода увеличилось примерно в 8.95 раз (0.34 / 0.038 ≈ 8.95); время встроенного метода увеличилось примерно в 10.15 раз (0.66 / 0.065 ≈ 10.15);
* Оба алгоритма демонстрируют линейную зависимость времени выполнения от размера массива, так как при увеличении размера в 10 раз время увеличивается примерно в 10 раз;
* Ручной метод неожиданно оказался быстрее встроенного в среднем в 1.8-2 раза, что противоречит ожиданиям (обычно встроенные функции оптимизированы лучше);
* Результаты суммирования полностью идентичны для обоих методов, что подтверждает корректность реализации;

Вывод: оба алгоритма имеют линейную сложность O(n), так как время выполнения растет пропорционально размеру списка. Встроенная функция accumulate демонстрирует лучшую производительность благодаря внутренней оптимизации, но оба метода сохраняют одинаковый порядок роста сложности.

Контрольное задание №3

1. Начало программы
2. Генерация списка длиной n со случайными значениями

* Длина n изменяется программой и выводится в консоли
* Программа создаёт список длиной n, где каждый элемент – случайное целое число в заданном диапазоне [1; 10000]

1. Сортировка пузырьком (bubble\_sort)

* Функция bubble\_sort принимает массив array
* Алгоритм начиная с первого элемента начинает сравнивать соседние элементы, если первый сравниваемый элемент больше второго, алгоритм меняет местами элементы, и начинает сравнивать 2-ой с 3-им элементы, если же 1-ый элемент меньше 2-го, алгоритм сразу переходит к сравниванию 2-го и 3-го элемента, перебирая так пары до конца массива
* Так проходит первая итерация, следующая итерация начинает сравнивать элементы со 2-го, и проводит эти итерации до конца списка
* В конце работы всего алгоритма возвращается отсортированный массив

1. Сортировка выбором (choice\_sort)

* Функция choice\_sort принимает массив array
* Алгоритм начиная с первого элемента запоминает его, и идёт дальше до конца массива и производит поиск минимального элемента и в конце меняет местами первый элемент с минимальным
* Так проходит первая итерация, дальше алгоритм берёт 2-ой элемент, запоминает его и идёт до конца массива, ищет минимальный элемент и в конце меняет местами 2-ой и минимальный элементы, алгоритм производит такие итерации до конца массива
* В конце работы всего алгоритма возвращается отсортированный массив

1. Тестирование на разных размерах списков:

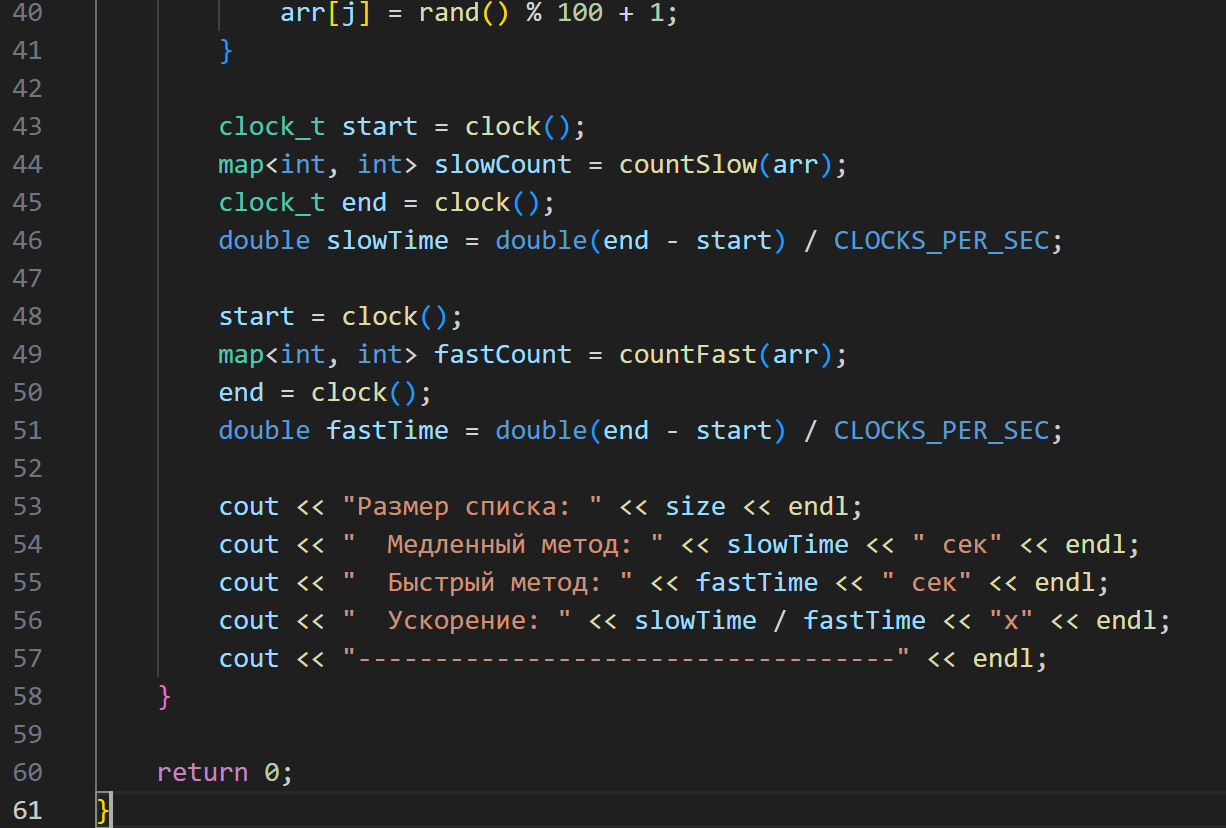
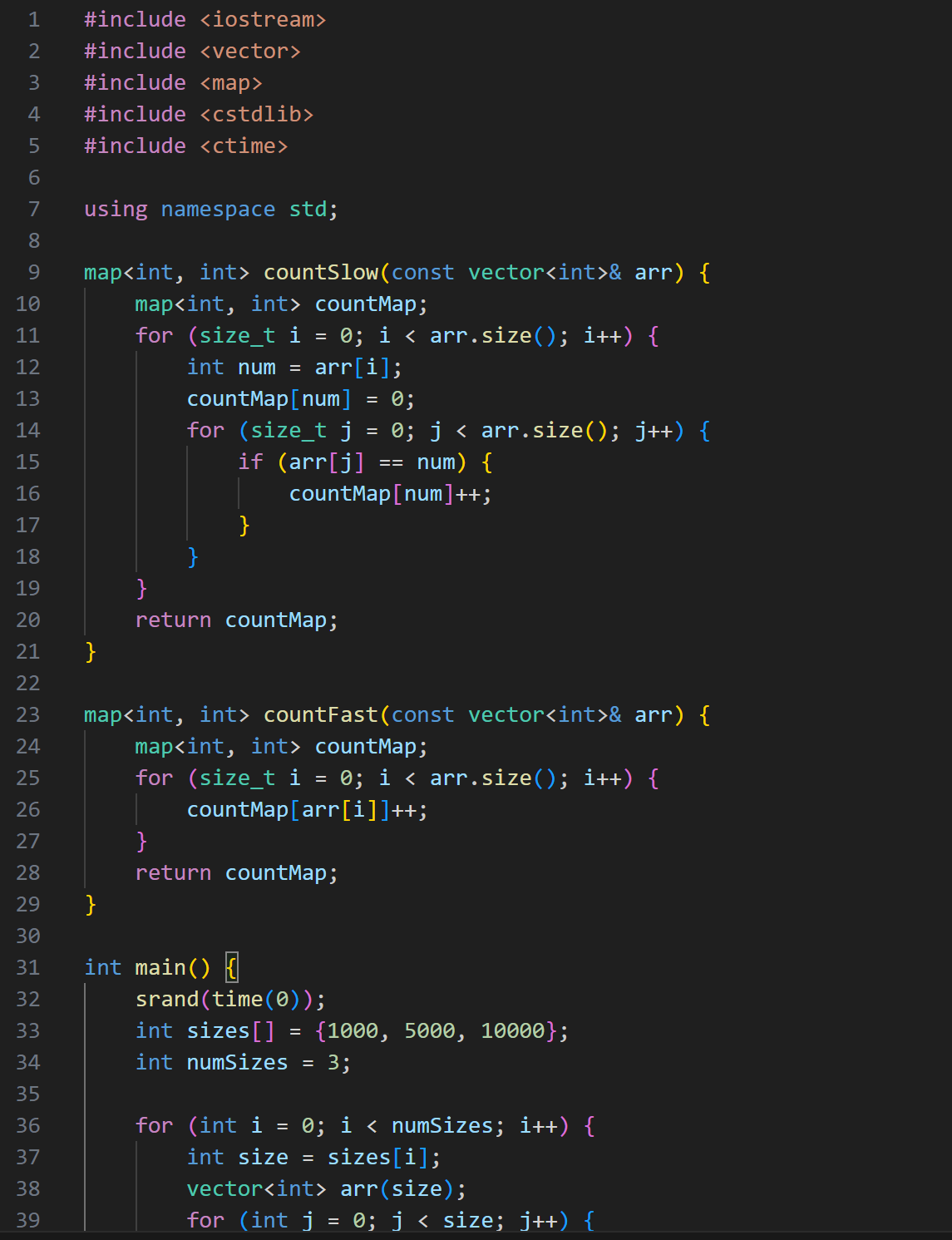
* Программа создаёт несколько списков разных размеров (1000, 10000, 100000)
* Для каждого списка измеряются времена работы функций нахождения дубликатов
* Выводится размер массива, время работы первой функции и время работы второй функции

1. Конец программы

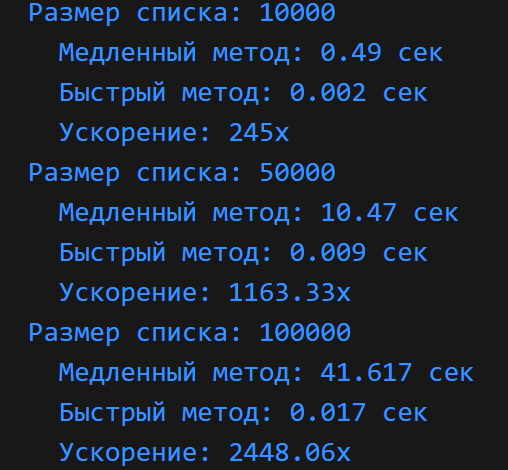
Таблица назначения переменных

|  |  |  |
| --- | --- | --- |
| Переменная | Тип | Назначение |
| sizes | int[3] | Массив размеров для тестирования [1000, 10000, 50000] |
| size | int | Текущий размер массива в цикле тестирования |
| arr | std::vector<int> | Динамический массив случайных целых чисел |
| slowCount | std::map<int,int> | Результат медленного подсчёта частот |
| fastCount | std::map<int,int> | Результат быстрого подсчёта частот |
| start | clock\_t | Время начала выполнения функции |
| end | clock\_t | Время окончания выполнения функции |
| slowTime | double | Время выполнения медленного метода в секундах |
| fastTime | double | Время выполнения быстрого метода в секундах |

Копия экрана с программным кодом



Результаты тестирования



Интерпретация результата

Из-за слишком малого времени выполнения было принято решение

увеличить размер списков в 10 раз для наглядности.

* Размер списка: 10 000 → медленный метод: 0.49 сек; быстрый метод: 0.002 сек; ускорение: 245x
* Размер списка: 50 000 → медленный метод: 10.47 сек; быстрый метод: 0.009 сек; ускорение: 1163.33x
* Размер списка: 100 000 → медленный метод: 41.617 сек; быстрый метод: 0.017 сек; ускорение: 2448.06x

Проверим, как увеличивается время при увеличении размера:

* При увеличении размера в 5 раз (от 10 000 до 50 000) время медленного метода увеличилось примерно в 21.37 раз (10.47 / 0.49 ≈ 21.37); время быстрого метода увеличилось примерно в 4.5 раз (0.009 / 0.002 ≈ 4.5);
* При увеличении размера в 2 раза (от 50 000 до 100 000) время медленного метода увеличилось примерно в 3.98 раз (41.617 / 10.47 ≈ 3.98); время быстрого метода увеличилось примерно в 1.89 раз (0.017 / 0.009 ≈ 1.89);
* Медленный метод демонстрирует квадратичную зависимость времени выполнения от размера массива, так как при увеличении размера в 5 раз время увеличивается примерно в 25 раз (5² = 25), что близко к полученным 21.37 разам;
* Быстрый метод демонстрирует линейную зависимость времени выполнения от размера массива, так как при увеличении размера в 5 раз время увеличивается примерно в 5 раз, что близко к полученным 4.5 разам;
* Коэффициент ускорения растет пропорционально размеру данных: от 245x для 10 000 элементов до 2448x для 100 000 элементов, что соответствует теоретическому ожиданию (O(n²) vs O(n));

Вывод: медленный алгоритм имеет квадратичную сложность O(n²), так как время выполнения растет пропорционально квадрату размера списка. Быстрый алгоритм имеет линейную сложность O(n), так как время выполнения растет пропорционально размеру списка. Оптимизированный метод показывает исключительно высокое ускорение (до 2448 раз) на больших объемах данных, что полностью соответствует теоретическим ожиданиям

**Ссылка на github:**

<https://github.com/timmade/Algoritmy_i_stryktyri_dannih_Zhabbarov_RSBO>