|  |
| --- |
|  |
| 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
2. Генерация списка длиной n со случайными значениями

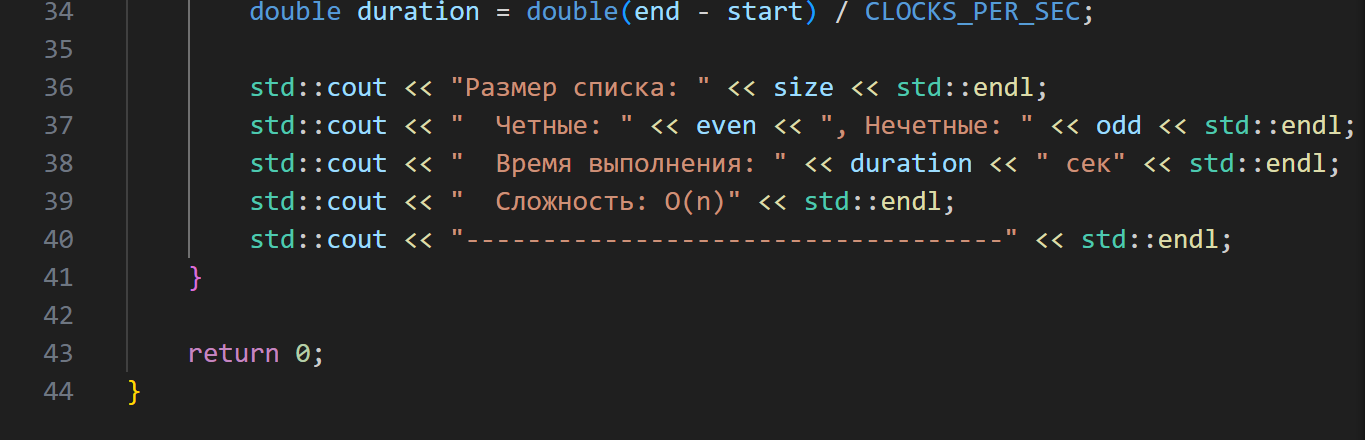
* Длина n изменяется программой и выводится в консоли
* Программа создаёт список длиной n, где каждый элемент – случайное целое число в заданном диапазоне [1; 100000]

1. Подсчет четных элементов (countEven)
   * Функция countEvenOdd принимает массив arr и ссылки на переменные even и odd
   * Алгоритм инициализирует счётчики even и odd нулевыми значениями
   * Проходит по всем элементам массива, проверяя каждый элемент на чётность
   * Если элемент чётный (делится на 2 без остатка), увеличивает счётчик even
   * Если элемент нечётный, увеличивает счётчик odd
   * В конце работы функция заполняет переданные по ссылке переменные результатами
2. Подсчет четных элементов (countOdd)
   * Выполняется в той же функции countEvenOdd одновременно с подсчётом чётных
   * Не требует отдельного прохода по массиву, что оптимизирует производительность
   * Оба подсчёта выполняются за один проход по массиву
3. Тестирование на разных размерах списков:
   * Программа создаёт несколько списков разных размеров (1000, 10000, 100000)
   * Для каждого списка измеряется время работы функции подсчёта
   * Выводится размер массива, количество чётных и нечётных элементов, время выполнения
4. Конец программы

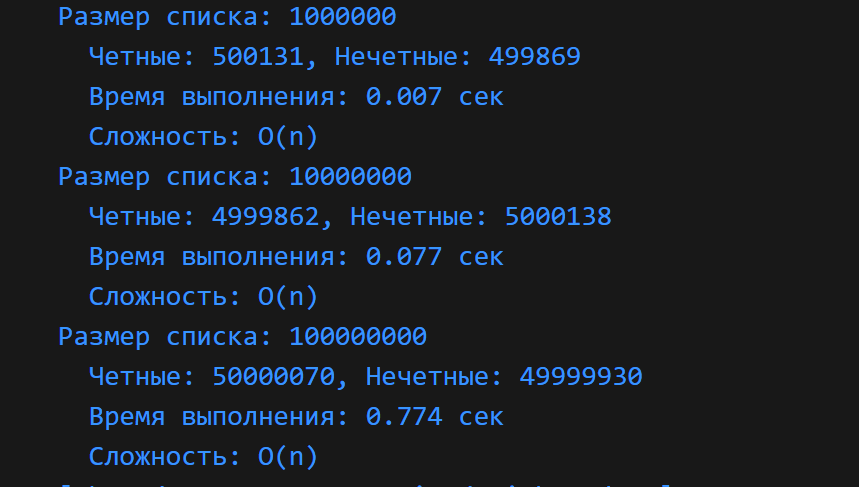
Таблица назначения переменных

|  |  |  |
| --- | --- | --- |
| Переменная | Тип | Назначение |
| sizes | int[3] | Массив размеров списков для тестирования |
| size | int | Размер списка |
| arr | std::vector<int> | Массив (список) случайных целых чисел |
| even | int | Счётчик чётных чисел в массиве |
| odd | int | Счётчик нечётных чисел в массиве |
| start | clock\_t | Время начала выполнения функции подсчёта |
| end | clock\_t | Время окончания выполнения функции подсчёта |
| Duration | double | Время выполнения функции в секундах |

Копия экрана с программным кодом 



Результаты тестирования



Интерпретация результата

Из-за слишком малого времени выполнения было принято решении

увеличить размер списков в 100 раз для наглядности.

* Размер списка: 1 000 000 → время выполнения: 0.007 сек; Четные: 500131, Нечетные: 499869
* Размер списка: 10 000 000 → время выполнения: 0.077 сек; Четные: 4999862, Нечетные: 5000138
* Размер списка: 100 000 000 → время выполнения: 0.774 сек; Четные: 50000070, Нечетные: 49999930

Проверим, как увеличивается время при увеличении размера:

* При увеличении размера в 10 раз (от 1 000 000 до 10 000 000) время увеличилось примерно в 11.0 раз (0.077 / 0.007 ≈ 11.0);
* При увеличении размера ещё в 10 раз (от 10 000 000 до 100 000 000) время увеличилось примерно в 10.05 раз (0.774 / 0.077 ≈ 10.05);
* Коэффициенты увеличения времени очень близки к теоретическому значению 10, что подтверждает линейный характер роста;
* Количество чётных и нечётных чисел распределено практически равномерно (около 50%/50%), что соответствует ожиданиям для случайных данных.

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

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:

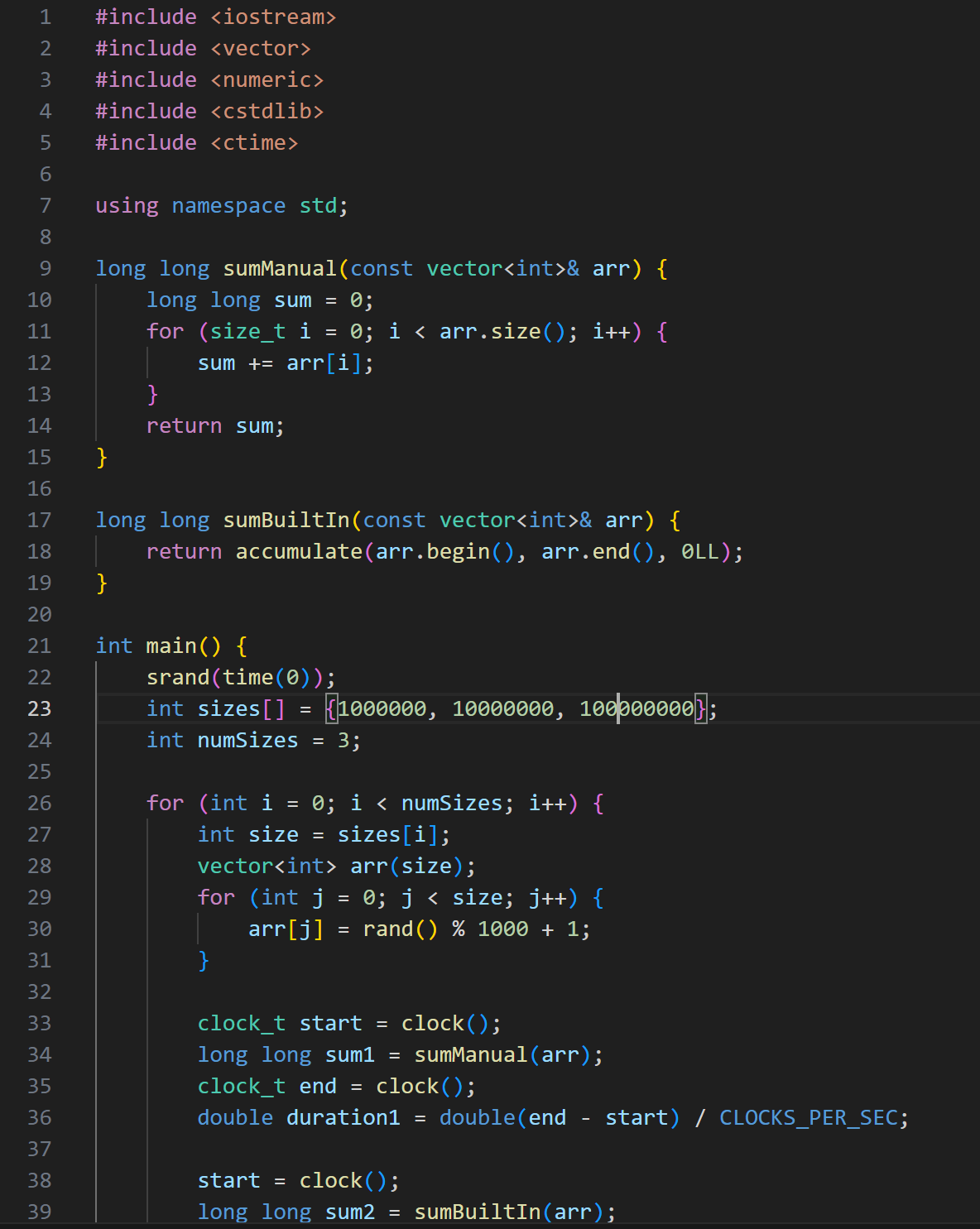
* Программа создаёт несколько списков разных размеров (1000, 10000, 100000)
* Для каждого списка измеряются времена работы обоих методов суммирования
* Выводится размер массива, результат суммирования, время работы ручного метода и время работы встроенного метода

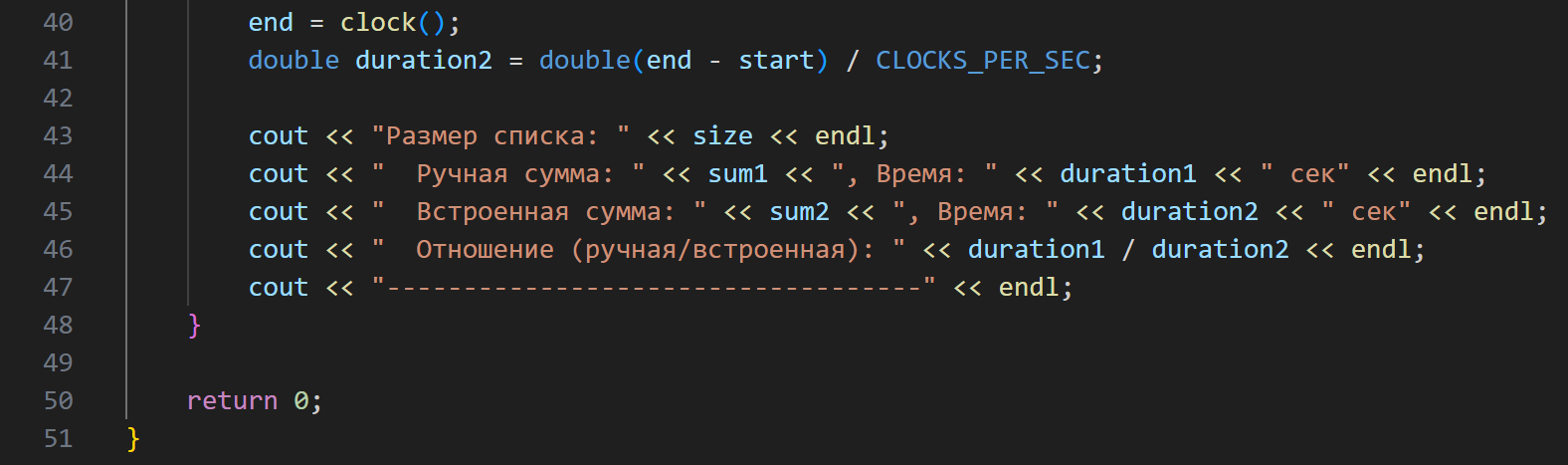
1. Конец программы

Таблица назначения переменных

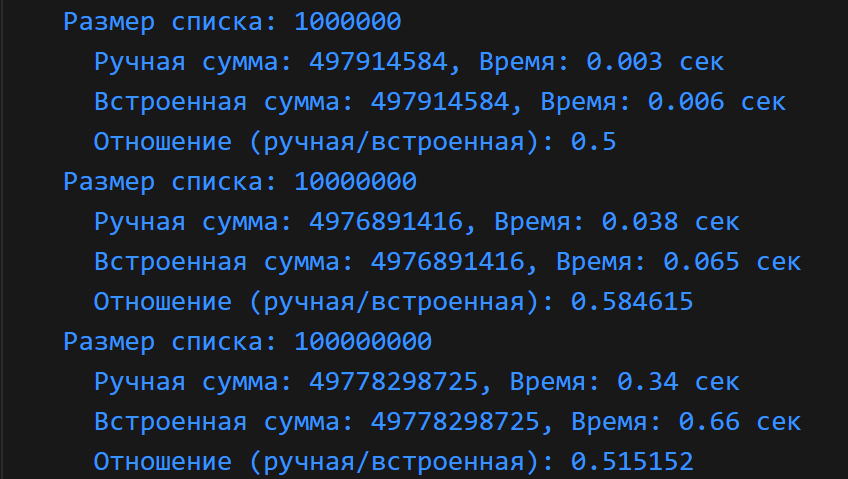
|  |  |  |
| --- | --- | --- |
| Переменная | Тип | Назначение |
| sizes | int[3] | Массив размеров для тестирования [1000, 10000, 100000] |
| size | int | Текущий размер массива в цикле тестирования |
| arr | std::vector<int> | Динамический массив случайных целых чисел |
| sum1 | long long | Результат 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:

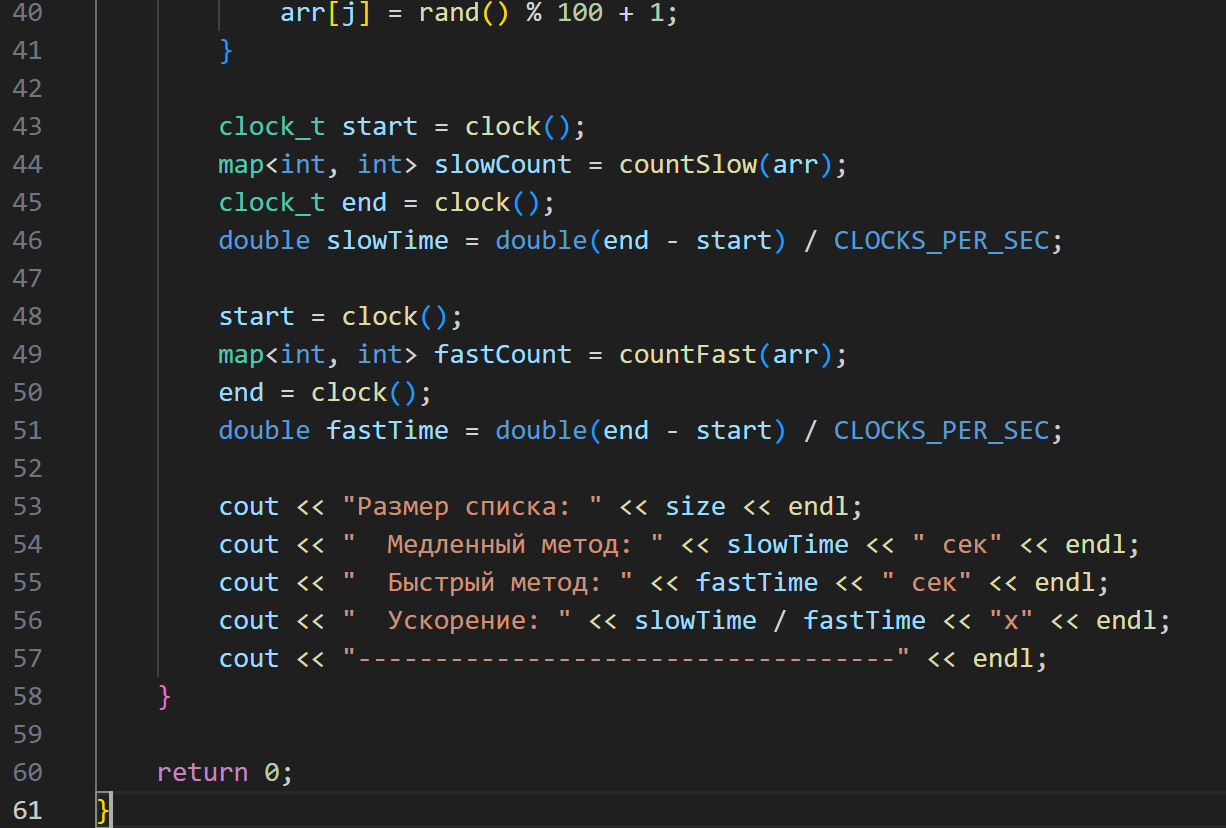
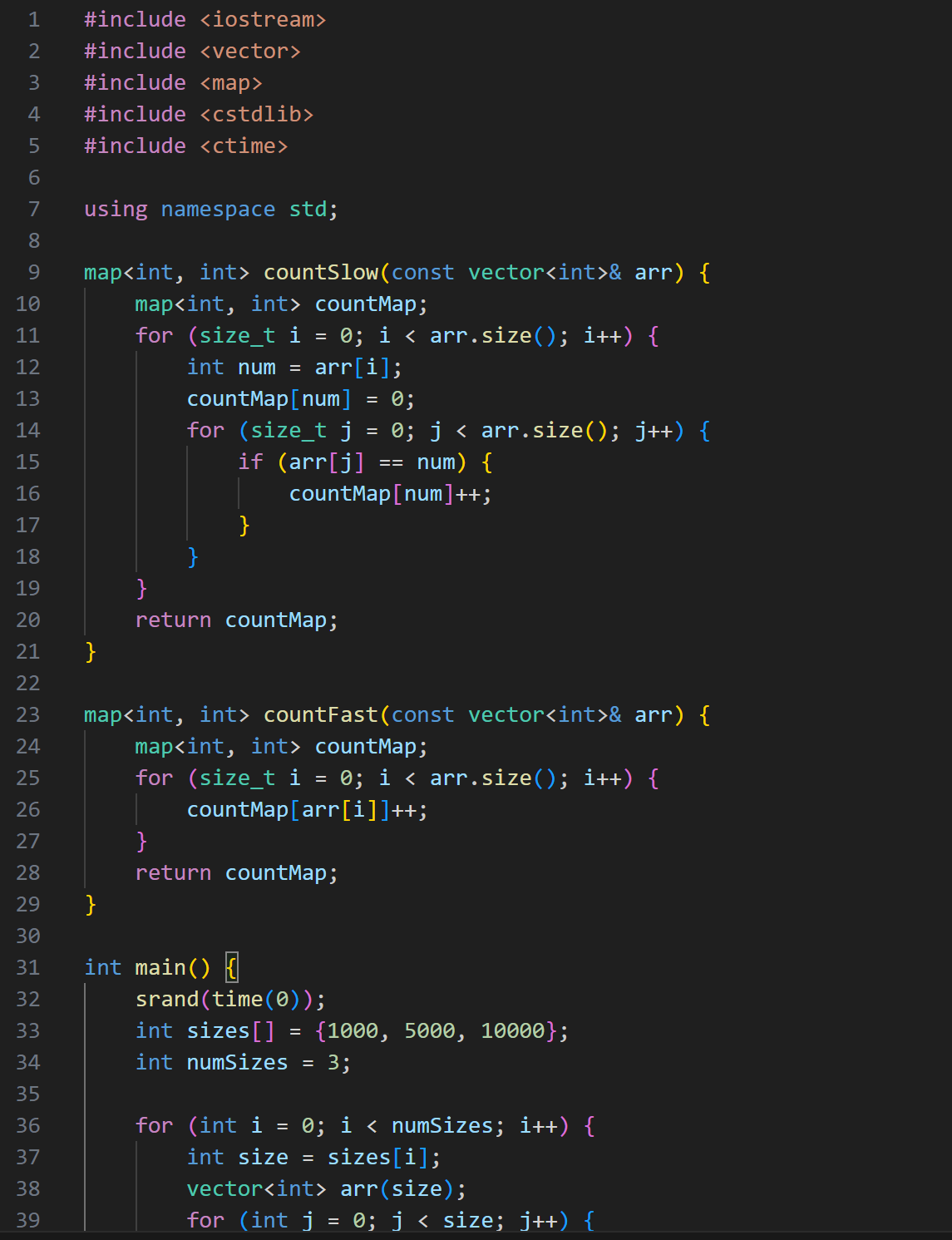
* Программа создаёт несколько списков разных размеров (1000, 10000, 100000)
* Для каждого списка измеряются времена работы функций нахождения дубликатов
* Выводится размер массива, время работы первой функции и время работы второй функции

1. Конец программы

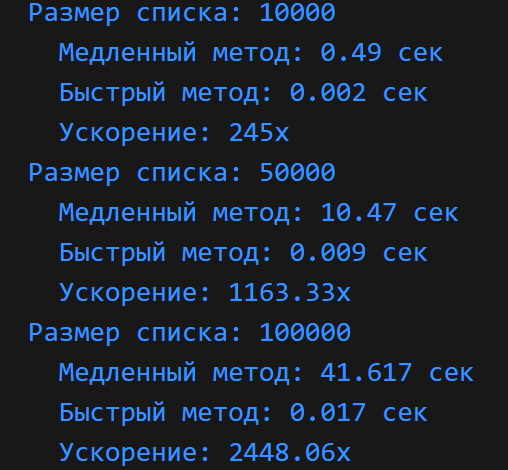
Таблица назначения переменных

|  |  |  |
| --- | --- | --- |
| 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>