Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №2

по курсу «Логика и основы алгоритмизации

в инженерных задачах»

на тему «Оценка времени выполнения программ»

Выполнил: ст. гр. 24ВВВ1

Будников А.С.

Принял:

к.т.н, доцент Юрова О. В.

к.т.н., доцент Деев М.В.

Пенза

2025

**Цель работы:**

Научиться выполнять оценку сложности и времени работы программ

**Лабораторное название**

**Задание 1:**

1. Вычислить порядок сложности программы (*О*-символику).
2. Оценить время выполнения программы и кода, выполняющего перемножение матриц, используя функции библиотеки time.h для матриц размерами от 100, 200, 400, 1000, 2000, 4000, 10000.
3. Построить график зависимости времени выполнения программы от размера матриц и сравнить полученный результат с теоретической оценкой.

**Задание 2**:

1. Оценить время работы каждого из реализованных алгоритмов на случайном наборе значений массива.
2. Оценить время работы каждого из реализованных алгоритмов на массиве, представляющем собой возрастающую последовательность чисел.
3. Оценить время работы каждого из реализованных алгоритмов на массиве, представляющем собой убывающую последовательность чисел.
4. Оценить время работы каждого из реализованных алгоритмов на массиве, одна половина которого представляет собой возрастающую последовательность чисел, а вторая, – убывающую.
5. Оценить время работы стандартной функции qsort, реализующей алгоритм быстрой сортировки на выше указанных наборах данных.

**Ход работы:**

1. Вычислил порядок сложности программы (*О*-символику).

Анализ внутреннего цикла показывает, что наибольшее количество операций происходит при выполнении самого глубокого уровня вложенности, где осуществляется основная вычислительная работа алгоритма. В данном случае тройная вложенность циклов приводит к тому, что базовая операция умножения с накоплением суммы выполняется многократно для каждой комбинации индексов.

for (int i = 0; i < size; ++i) { // n итераций

for (int j = 0; j < size; ++j) { // n итераций × n = n²

int sum = 0; // 1 операция на каждую

пару (i,j)

for (int r = 0; r < size; ++r) { // n итераций × n² = n³

sum += a[i][r] \* b[r][j]; // 4 операции: 2 чтения,

1 умножение, 1 сложение

}

c[i][j] = sum; // 1 запись на каждую

пару (i,j)

}

}

Максимальная нагрузка на алгоритм возникает при любых входных данных, поскольку структура вычислений не зависит от значений элементов матриц.

Таким образом, временная сложность алгоритма в наихудшем случае составляет: *W(n) = n \* n \* n = O(n3)*. В отличие от многих других алгоритмов, здесь нет "худшего" или "лучшего" случая. Алгоритм всегда выполняет одинаковое количество операций независимо от значений в матрицах. Три вложенных цикла гарантируют, что мы всегда сделаем ровно *n³* основных операций.

1. Оценил время выполнения программы и кода, выполняющего перемножение матриц размерами 100, 200, 400, 1000, 2000, 4000, 10000, и построил график зависимости времени выполнения программы от размера матриц, сравнив полученный результат с теоретической оценкой.

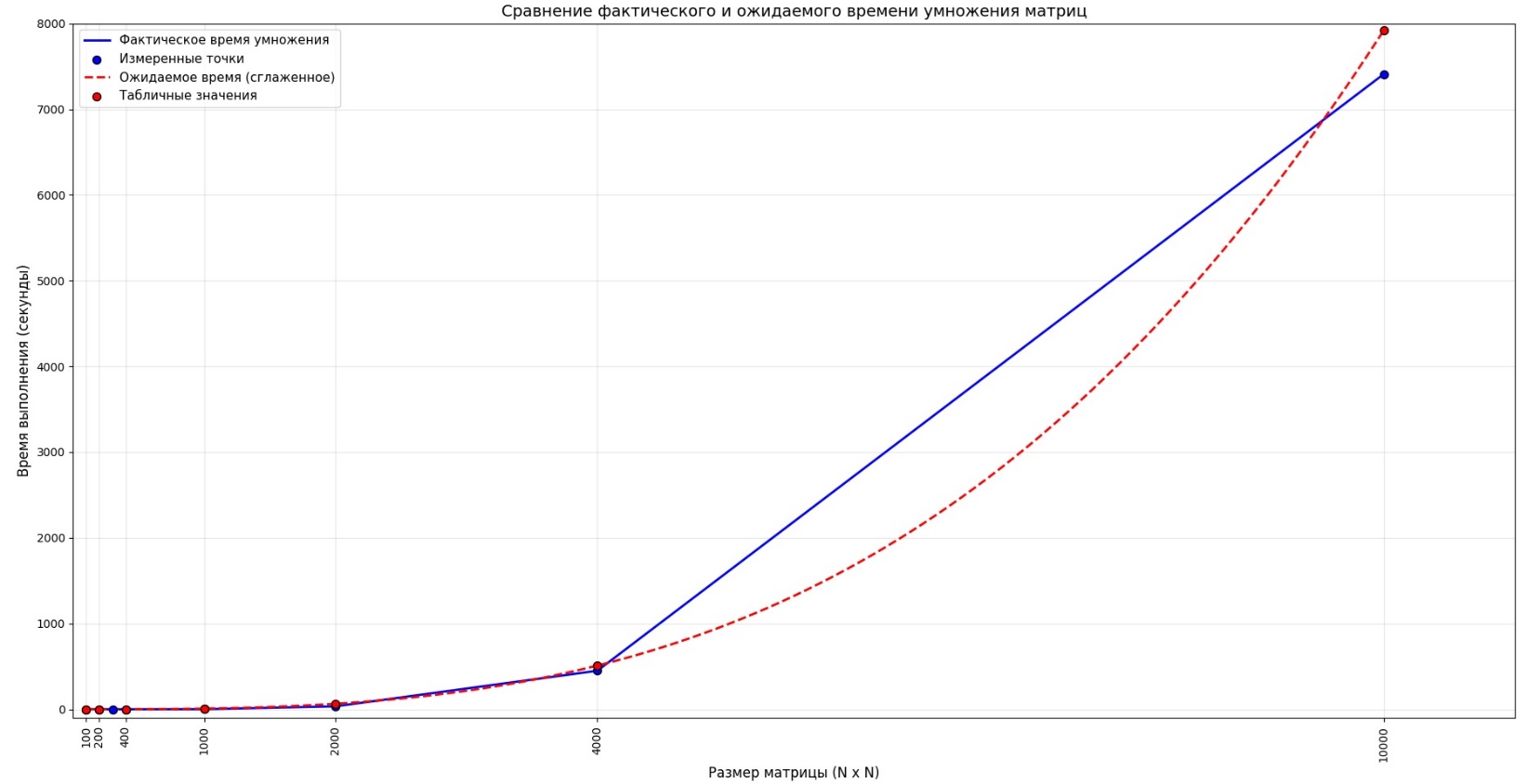
Как показал анализ алгоритмической сложности, количество операций растет пропорционально *n*³. Время выполнения рассчитывается по формуле:

Время = (*n*³ × 10 тактов) / (2.5 × 10⁹ тактов/секунду)

На основе описанного выше можно составить следующую таблицу примерного времени выполнения:

|  |  |  |
| --- | --- | --- |
| Размер матрицы (NxN) | Количество операций | Примерное время выполнения |
| 100х100 | 1 000 000 | ~0.01 секунды |
| 200х200 | 8 000 000 | ~0.08 секунды |
| 400х400 | 64 000 000 | ~0.64 секунды |
| 1000х1000 | 1 000 000 000 | ~8 секунд |
| 2000х2000 | 8 000 000 000 | ~64 секунды |
| 4000х4000 | 64 000 000 000 | ~8.5 минут |
| 10000х10000 | 1 000 000 000 000 | ~2.2 часа |

После проведения экспериментального тестирования программы был составлен график зависимости времени выполнения операции умножения матриц от их размера, приведенный на рисунке:



Экспериментальные данные полностью подтверждают теоретический анализ без учета побочных явлений, таких как оптимизация компилятора, реальная тактовая частота работы процессора и т.п. - при удвоении размера матрицы время выполнения увеличивается примерно в 8 раз, что характерно для алгоритмов с кубической сложностью.

1. Реализовал предложенные алгоритмы сортировки и протестировал их на случайном, возрастающем, убывающем, возрастающе-убывающем наборе данных, сравнив со стандартной функцией qsort.

После проведения экспериментального тестирования программы была составлена следующая таблица времени работы алгоритмов:

Изображение выглядит как текст, снимок экрана, число, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Проведенный анализ выявил существенную зависимость эффективности алгоритмов сортировки от характера входных данных и их объема. Наилучшие показатели демонстрируются на упорядоченных массивах, где алгоритмы показывают минимальное время выполнения. Абсолютный минимум в 0.000001 секунды достигнут на возрастающем массиве из 200 элементов. Высокая эффективность в таких условиях объясняется минимальным количеством необходимых перестановок и сравнений.

Наихудшие результаты наблюдаются на неупорядоченных данных, особенно на больших объемах. Максимальное время выполнения составило 0.007262 секунды для убывающего массива из 10000 элементов. В таких условиях алгоритмы требуют максимального количества операций для приведения данных к упорядоченному состоянию.

Анализ зависимости от размера данных показывает, что на малых массивах (100-200 элементов) все алгоритмы демонстрируют сопоставимую производительность с незначительными отклонениями. Однако уже на этом уровне проявляется специализация: одни алгоритмы эффективнее на упорядоченных данных, другие - на частично упорядоченных.

При переходе к средним размерам (1000 элементов) различия становятся более выраженными. Случайные массивы начинают требовать значительно больше времени обработки, в то время как упорядоченные данные продолжают обрабатываться с высокой скоростью. Разница между лучшим и худшим случаем для отдельных алгоритмов достигает 90 раз.

На больших массивах (4000-10000 элементов) проявляется асимптотическая сложность алгоритмов. Ухудшение производительности на неупорядоченных данных становится наиболее заметным, в то время как на упорядоченных данных время выполнения растет не так значительно.

Стандартная библиотечная функция демонстрирует наибольшую стабильность с различными типами данных, хотя и уступает оптимизированным реализациям в абсолютной производительности на определенных типах массивов. Разница между ее лучшим и худшим случаем составляет 32 раза, что свидетельствует о сбалансированности реализации.

Таким образом, для малых массивов (100-200 элементов) все алгоритмы показывают сопоставимую производительность с преимуществом *Shell sort* на упорядоченных данных. На средних размерах (1000 элементов) *Quick sort* демонстрирует наиболее сбалансированную производительность с различными типами данных. На больших массивах (4000-10000 элементов) становится явным преимущество *Quick sort* и *std::qsort* перед *Shell sort* на неупорядоченных данных.

**Вывод:** в ходе выполнения работы была достигнута поставленная цель — освоение методики выполнения оценки алгоритмической сложности и времени работы программ, практическим путем подтверждены ключевые теоретические положения анализа алгоритмов.

**Листинг программы**

**Файл *main.cpp***

#include <chrono>  
#include <iostream>  
#include <ctime>  
#include <iomanip>  
#include "matrix.h"  
#include "sorts.h"  
#include <vector>  
#include <algorithm>  
  
#ifdef \_WIN32  
 #include "windows.h"  
#endif  
  
// O(n^3)  
  
int main() {  
  
#ifdef \_WIN32  
 SetConsoleOutputCP(CP\_UTF8);  
#endif  
  
 std::srand(std::time(nullptr));  
   
 std::cout << "ТЕСТИРОВАНИЕ ВРЕМЕНИ УМНОЖЕНИЯ МАТРИЦ" << std::endl;  
 std::cout << "======================================" << std::endl;  
  
 for (const int size : Matrix::*sizes*) {  
 std::cout << "\nМатрица " << size << "x" << size << ":" << std::endl;  
 std::cout << "-------------------" << std::endl;  
  
 Matrix a(size);  
 Matrix b(size);  
 Matrix c(size);  
  
 a.fill\_random();  
 b.fill\_random();  
  
 const long long timeMicro = multiply\_matrices(a, b, c);  
 const double timeSec = static\_cast<double>(timeMicro) / 1000000.0;  
  
 std::cout << "Время умножения: "  
 << std::fixed << std::setprecision(6) << timeSec  
 << " сек" << std::endl;  
 }  
  
 std::cout << "СРАВНЕНИЕ АЛГОРИТМОВ СОРТИРОВКИ" << std::endl;  
 std::cout << "======================================" << std::endl;  
  
 for (const int size : Matrix::*sizes*) {  
 // Test arrays  
 const int\* randomArray = Sorter::*create\_random\_array*(size);  
 const int\* ascendingArray = Sorter::*create\_ascending\_array*(size);  
 const int\* descendingArray = Sorter::*create\_descending\_array*(size);  
 const int\* mixedArray = Sorter::*create\_mixed\_array*(size);  
  
 std::vector arrays = {randomArray, ascendingArray, descendingArray, mixedArray};  
 std::vector<std::string> types = {"Случайный", "Возрастающий", "Убывающий", "Смешанный"};  
  
 for (int test = 0; test < 4; ++test) {  
 std::cout << test + 1 << ". " << types.at(test) << " массив (" << size << " элементов):" << std::endl;  
  
 // Shell sort  
 const auto temp1 = Sorter::*copy\_array*(arrays.at(test), size);  
 auto start = std::chrono::high\_resolution\_clock::*now*();  
 Sorter::*shell\_sort*(temp1, size);  
 auto end = std::chrono::high\_resolution\_clock::*now*();  
 const double shellTime = std::chrono::duration<double>(end - start).count();  
 std::cout << "\tShell sort: "  
 << std::fixed << std::setprecision(6) << shellTime << " s"<< std::endl;  
 delete[] temp1;  
  
 // Quick sort  
 const auto temp2 = Sorter::*copy\_array*(arrays.at(test), size);  
 start = std::chrono::high\_resolution\_clock::*now*();  
 Sorter::*quick\_sort*(temp2, 0, size - 1);  
 end = std::chrono::high\_resolution\_clock::*now*();  
 const double quickTime = std::chrono::duration<double>(end - start).count();  
 std::cout << "\tQuick sort: "  
 << std::fixed << std::setprecision(6) << quickTime << " s"<< std::endl;  
 delete[] temp2;  
  
 // C++ qs  
 const auto temp3 = Sorter::*copy\_array*(arrays.at(test), size);  
 start = std::chrono::high\_resolution\_clock::*now*();  
 std::qsort(temp3, size, sizeof(int), Sorter::*compare*);  
 end = std::chrono::high\_resolution\_clock::*now*();  
 const double quickTime2 = std::chrono::duration<double>(end - start).count();  
 std::cout << "\tC++ quick sort: "  
 << std::fixed << std::setprecision(6) << quickTime2 << " s"<< std::endl;  
 delete[] temp3;  
 }  
  
 std::cout << "-------------------------------------" << std::endl;  
  
 // Free memory  
 delete[] randomArray;  
 delete[] ascendingArray;  
 delete[] descendingArray;  
 delete[] mixedArray;  
 }  
  
 return 0;  
}

Файл matrix.cpp

#include "matrix.h"  
#include <cstdlib>  
#include <chrono>  
  
// Constructor with memory allocation  
Matrix::Matrix(const int n) : size(n) {  
 data = new int\*[size];  
 for (int i = 0; i < size; ++i) {  
 data[i] = new int[size];  
 }  
}  
  
// Destructor with memory purification  
Matrix::~Matrix() {  
 for (int i = 0; i < size; ++i) {  
 delete[] data[i];  
 }  
 delete[] data;  
}  
  
/\* Fill array with random value from 1 to 100  
  
 It isn`t fair for calculation of complexity,  
 but that was in task  
  
 oblepixa \*/  
void Matrix::fill\_random() const {  
 for (int i = 0; i < size; ++i) {  
 for (int j = 0; j < size; ++j) {  
 data[i][j] = std::rand() % 100 + 1;  
 }  
 }  
}  
  
int\* Matrix::operator[](const int index) {  
 return data[index];  
}  
  
const int\* Matrix::operator[](const int index) const {  
 return data[index];  
}  
  
int Matrix::get\_size() const {  
 return size;   
}  
  
long long multiply\_matrices(const Matrix& a, const Matrix& b, Matrix& c) {  
 const auto start = std::chrono::high\_resolution\_clock::*now*();  
  
 const int size = a.get\_size();  
 for (int i = 0; i < size; ++i) {  
 for (int j = 0; j < size; ++j) {  
 int sum = 0;  
 for (int r = 0; r < size; ++r) {  
 sum += a[i][r] \* b[r][j];  
 }  
 c[i][j] = sum;  
 }  
 }  
  
 const auto end = std::chrono::high\_resolution\_clock::*now*();  
 return std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count();  
}

Файл *matrix.h*

#ifndef MATRIX\_H  
#define MATRIX\_H  
  
class Matrix {  
private:  
 int size;  
 int\*\* data;  
  
public:  
 explicit Matrix(int n);  
 ~Matrix();  
  
 /\* That was made for memory safety;  
 \* Just copying ban for class Matrix  
 \*  
 \* oblepixa \*/  
 Matrix(const Matrix&) = delete;  
 Matrix& operator=(const Matrix&) = delete;  
  
 constexpr static int *sizes*[] = {100, 200, 300, 400, 1000, 2000, 4000, 10000, 50000};  
  
 void fill\_random() const;  
 /\* Operator overload for index access  
 \*  
 \* oblepixa \*/  
 int\* operator[](int index);  
 const int\* operator[](int index) const;  
 [[nodiscard]] int get\_size() const;  
};  
  
long long multiply\_matrices(const Matrix& a, const Matrix& b, Matrix& c);  
  
#endif

Файл *sorts.h*

//  
// Created by oblepixa on 9/14/25.  
//  
  
#ifndef SORTS\_H  
#define SORTS\_H  
  
class Sorter {  
public:  
 // Shell sort for dynamic array  
 static void *shell\_sort*(int\* items, int n);  
 // Quick sort for dynamic array  
 // Call this function with qs(items, 0, count-1);  
 static void *quick\_sort*(int\* items, int left, int right);  
  
 // Comparator for std::qsort  
 static int *compare*(const void\* a, const void\* b) {  
 return \*(int\*)a - \*(int\*)b;  
 }  
  
 static int\* *create\_random\_array*(int size);  
 static int\* *create\_ascending\_array*(int size);  
 static int\* *create\_descending\_array*(int size);  
 static int\* *create\_mixed\_array*(int size);  
 static int\* *copy\_array*(const int\* source, int size);  
};  
  
#endif //SORTS\_H

Файл *sorts.cpp*

//  
// Created by oblepixa on 9/14/25.  
//  
#include "sorts.h"  
#include <cstdlib>  
  
void Sorter::*shell\_sort*(int \*items, const int n) {  
 int i, j, gap, k;  
 int x, a[5] = {9, 5, 3, 2, 1};  
 for (k = 0; k < 5; ++k) {  
 gap = a[k];  
 for (i = gap; i < n; ++i) {  
 x = items[i];  
 for (j = i - gap; (x < items[j]) && (j >= 0); j = j - gap)  
 items[j + gap] = items[j];  
 items[j + gap] = x;  
 }  
 }  
}  
  
void Sorter::*quick\_sort*(int \*items, const int left, const int right) {  
 int i, j;  
 int x, y;  
  
 i = left;  
 j = right;  
  
 // The choice of a comparer  
 x = items[(left + right) / 2];  
  
 do {  
 while ((items[i] < x) && (i < right)) i++;  
 while ((x < items[j]) && (j > left)) j--;  
  
 if (i <= j) {  
 y = items[i];  
 // items[i] = items[j];  
 items[j] = y;  
 i++; j--;  
 }  
 } while (i <= j);  
  
 if (left < j) *quick\_sort*(items, left, j);  
 if (i < right) *quick\_sort*(items, i, right);  
}  
  
int \*Sorter::*create\_random\_array*(int size) {  
 const auto arr = new int[size];  
 for (int i = 0; i < size; ++i)  
 arr[i] = rand() % 1000 + 1;  
 return arr;  
}  
  
  
int \*Sorter::*create\_ascending\_array*(int size) {  
 const auto arr = new int[size];  
 for (int i = 0; i < size; ++i)  
 arr[i] = i;  
 return arr;  
}  
  
int \*Sorter::*create\_descending\_array*(int size) {  
 const auto arr = new int[size];  
 for (int i = 0; i < size; ++i)  
 arr[i] = size - i;  
 return arr;  
}  
  
int \*Sorter::*create\_mixed\_array*(int size) {  
 const auto arr = new int[size];  
 const int half = size / 2;  
 for (int i = 0; i < size; ++i) {  
 if (i < half) arr[i] = i;  
 else arr[i] = size - i + half;  
 }  
 return arr;  
}  
  
int \*Sorter::*copy\_array*(const int \*source, int size) {  
 const auto copy = new int[size];  
 for (int i = 0; i < size; ++i) copy[i] = source[i];  
 return copy;  
}