

**Министерство образования Российской Федерации**  
**Пензенский государственный университет**  
**Кафедра «Вычислительная техника»**

**Отчёт**

по лабораторной работе №2  
по курсу «Л и О А в ИЗ»

на тему «Оценка времени выполнения программ»

Выполнили студенты группы 24ВВВ4:

Кондратьев С.В.

Кошелев Р.Д.

Приняли к.т.н., доцент:

Юрова О.В.

доцент:

Акифьев И.В.

Пенза 2025

## Общие сведения.

Для оценки времени выполнения программ языка Си или их частей могут использоваться средства, предоставляемые библиотекой **time.h**. Данная библиотека содержит описания типов и прототипы функций для работы с датой и временем.

### Задание 1:

1. Вычислить порядок сложности программы ( $O$ -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.

### Листинг задания 1

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <locale.h>

double** create_matrix(int n) {
    double** matrix = (double**)malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        matrix[i] = (double*)malloc(n * sizeof(double));
        for (int j = 0; j < n; j++) {
```

```

        matrix[i][j] = (double)rand() / RAND_MAX;
    }
}
return matrix;
}
void free_matrix(double** matrix, int n) {
    for (int i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
}
void matrix_multiply(double** A, double** B, double** C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
int main() {
    setlocale(LC_ALL, "rus");
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);
    int sizes[] = { 100, 200, 400, 1000, 2000, 4000, 10000 };
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);
    srand(time(NULL));

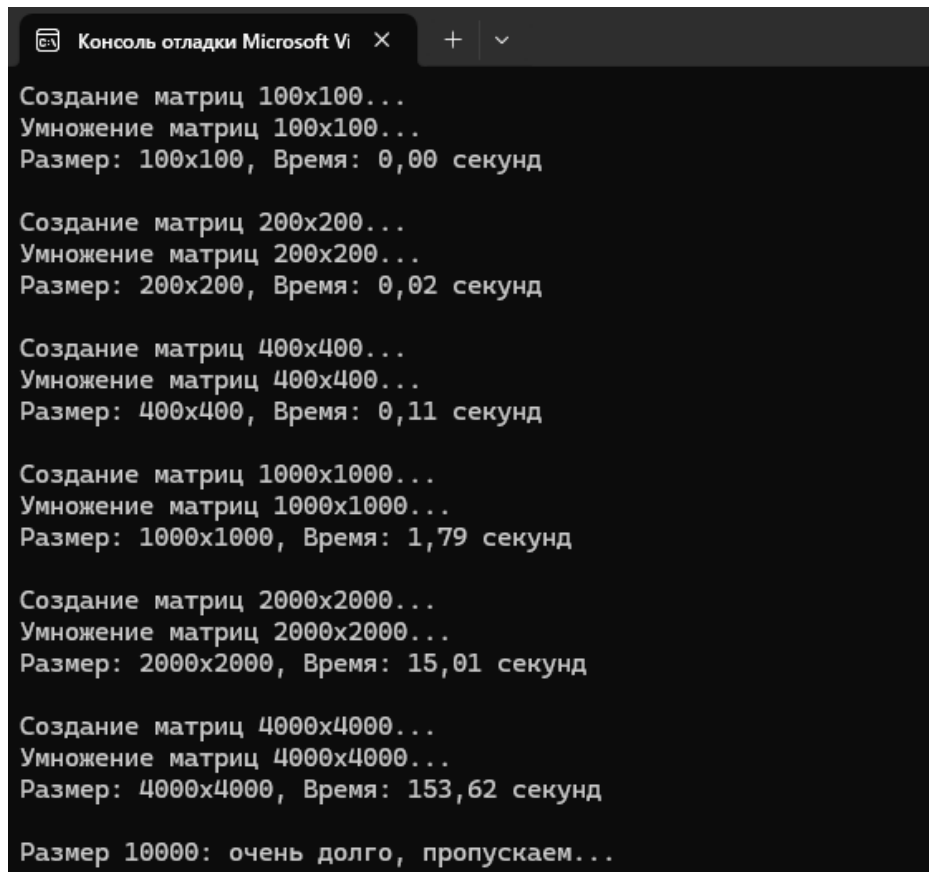
```

```

for (int i = 0; i < num_sizes; i++) {
    int n = sizes[i];
    if (n > 4000) {
        printf("Размер %d: очень долго, пропускаем...\n", n);
        continue;
    }
    printf("Создание матриц %dx%d...\n", n, n);
    double** A = create_matrix(n);
    double** B = create_matrix(n);
    double** C = create_matrix(n);
    printf("Умножение матриц %dx%d...\n", n, n);
    clock_t start = clock();
    matrix_multiply(A, B, C, n);
    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Размер: %dx%d, Время: %.2f секунд\n\n", n, n, time_taken);
    free_matrix(A, n);
    free_matrix(B, n);
    free_matrix(C, n);
}
return 0;
}

```

## Результат работы программы:



```
Консоль отладки Microsoft Vi X + v
Создание матриц 100x100...
Умножение матриц 100x100...
Размер: 100x100, Время: 0,00 секунд

Создание матриц 200x200...
Умножение матриц 200x200...
Размер: 200x200, Время: 0,02 секунд

Создание матриц 400x400...
Умножение матриц 400x400...
Размер: 400x400, Время: 0,11 секунд

Создание матриц 1000x1000...
Умножение матриц 1000x1000...
Размер: 1000x1000, Время: 1,79 секунд

Создание матриц 2000x2000...
Умножение матриц 2000x2000...
Размер: 2000x2000, Время: 15,01 секунд

Создание матриц 4000x4000...
Умножение матриц 4000x4000...
Размер: 4000x4000, Время: 153,62 секунд

Размер 10000: очень долго, пропускаем...
```

Рисунок 1 – Результат работы программы 1

## Оценка времени выполнения программы:

Если матрица  $100 \times 100$  обрабатывается моментально ( $\approx 0.00$  сек), а матрица  $200 \times 200$  обрабатывается за 0.02 секунды, то:

Для матрицы  $300 \times 300$ : Отношение размеров:  $300/200 = 1.5$  раза.

Теоретическое увеличение времени:  $(1.5)^3 = 3.375$  раза. Прогнозируемое время:  $0.02 \text{ сек} \times 3.375 = 0.0675$  секунды

Для матрицы  $400 \times 400$ : Отношение размеров:  $400/200 = 2$  раза.

Теоретическое увеличение времени:  $(2)^3 = 8$  раз. Прогнозируемое время:  $0.02 \text{ сек} \times 8 = 0.16$  секунды

## График зависимости времени выполнения программы от размера матриц

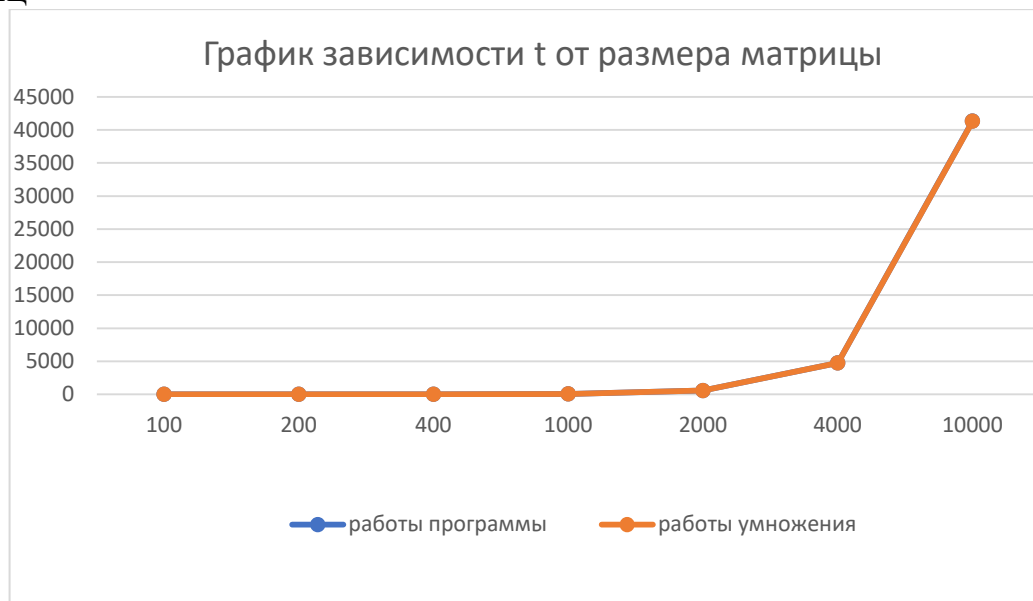


Рисунок 2 – Зависимость времени от размеров матрицы

Алгоритм перемножения матриц имеет кубическую сложность  $O(n^3)$ , что означает, что время его выполнения должно расти пропорционально кубу размера матрицы. Для проверки этого утверждения были проведены практические измерения времени выполнения для матриц различных размеров. Для малых матриц (100-400 элементов) наблюдается значительное отклонение в меньшую сторону. Это объясняется эффективной работой кэш-памяти процессора. Для средних матриц (1000) часть данных в кэше, часть в оперативной памяти, начинаются промахи кэша. Для больших матриц (2000-4000 элементов) данные не помещаются в кэш процессора.

## Листинг задания 2

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <locale.h>

void shell(int* items, int count) {
    int i, j, gap, k;
    int x;
    int a[5] = {9, 5, 3, 2, 1};
    for (k = 0; k < 5; k++) {
        gap = a[k];
        for (i = gap; i < count; ++i) {
            x = items[i];
            for (j = i - gap; (j >= 0) && (x < items[j]); j = j - gap)
                items[j + gap] = items[j];
            items[j + gap] = x;
        }
    }
}

void qs(int* items, int left, int right) {
    if (items == NULL) return;
    if (left >= right) return;
    int i = left;
    int j = right;
    int x = items[(left + right) / 2];
    while (i <= j) {
        while (i <= right && items[i] < x) i++;
        while (j >= left && items[j] > x) j--;

        if (i <= j) {
            int tmp = items[i];

```

```

        items[i] = items[j];
        items[j] = tmp;
        i++;
        j--;
    }
}
if (left < j) qs(items, left, j);
if (i < right) qs(items, i, right);
}

void generate_random(int arr[], int n) {
    for (int i = 0; i < n; i++) arr[i] = rand() % 10000;
}

void generate_ascending(int arr[], int n) {
    for (int i = 0; i < n; i++) arr[i] = i;
}

void generate_descending(int arr[], int n) {
    for (int i = 0; i < n; i++) arr[i] = n - 1 - i;
}

void generate_mixed(int arr[], int n) {
    int half = n / 2;
    for (int i = 0; i < half; i++) arr[i] = i;
    for (int i = half; i < n; i++) arr[i] = (n - 1) - (i - half);
}

void measure_time(void (*sort_func)(int*, int), int arr[], int n, const char*
algo_name, const char* data_type) {
    int* temp = (int*)malloc(n * sizeof(int));
    memcpy(temp, arr, n * sizeof(int));

    clock_t start = clock();
    sort_func(temp, n);

```



```

    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("%-15s %-25s: %.6f cek\n", algo_name, data_type, time_taken);
    free(temp);
}

void measure_qs_time(int arr[], int n, const char* data_type) {
    int* temp = (int*)malloc(n * sizeof(int));
    memcpy(temp, arr, n * sizeof(int));
    clock_t start = clock();
    qs(temp, 0, n - 1);
    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("%-15s %-25s: %.6f cek\n", "Quick Sort", data_type, time_taken);
    free(temp);
}

int cmp_int(const void* a, const void* b) {
    int x = *(const int*)a;
    int y = *(const int*)b;
    return (x > y) - (x < y);
}

void measure_std_qsort_time(int arr[], int n, const char* data_type) {
    int* temp = (int*)malloc(n * sizeof(int));
    memcpy(temp, arr, n * sizeof(int));
    clock_t start = clock();
    qsort(temp, n, sizeof(int), cmp_int);
    clock_t end = clock();

    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("%-15s %-25s: %.6f cek\n", "stdlib qsort", data_type, time_taken);
    free(temp);
}

```

```
}
```

```
int main() {  
    setlocale(LC_ALL, "");  
    const int n = 10000;  
    int* arr = (int*)malloc(n * sizeof(int));  
    srand((unsigned)time(NULL));  
    int test_cases = 4;  
    void (*generators[])(int[], int) = {  
        generate_random, generate_ascending,  
        generate_descending, generate_mixed  
    };  
    const char* data_types[] = {  
        "Случайные данные", "Возрастающая",  
        "Убывающая", "Смешанная"  
    };  
    for (int i = 0; i < test_cases; i++) {  
        generators[i](arr, n);  
        printf("\n=== %s ===\n", data_types[i]);  
        measure_time(shell, arr, n, "Shell Sort", data_types[i]);  
        measure_qs_time(arr, n, data_types[i]);  
        measure_std_qsort_time(arr, n, data_types[i]);  
    }  
    free(arr);  
    return 0;  
}
```

## Результат работы программы:

$n = 10.000$

```
Консоль отладки Microsoft Vi X + v

=== Случайные данные ===
Shell Sort      Случайные данные      : 0,004000 сек
Quick Sort      Случайные данные      : 0,000000 сек
stdlib qsort     Случайные данные      : 0,002000 сек

=== Возрастающая ===
Shell Sort      Возрастающая          : 0,000000 сек
Quick Sort      Возрастающая          : 0,000000 сек
stdlib qsort     Возрастающая          : 0,001000 сек

=== Убывающая ===
Shell Sort      Убывающая             : 0,006000 сек
Quick Sort      Убывающая             : 0,000000 сек
stdlib qsort     Убывающая             : 0,001000 сек

=== Смешанная ===
Shell Sort      Смешанная             : 0,001000 сек
Quick Sort      Смешанная             : 0,000000 сек
stdlib qsort     Смешанная             : 0,000000 сек
```

$n = 100.000$

```
=== Случайные данные ===
Shell Sort      Случайные данные      : 1,029000 сек
Quick Sort      Случайные данные      : 0,013000 сек
stdlib qsort     Случайные данные      : 0,053000 сек

=== Возрастающая ===
Shell Sort      Возрастающая          : 0,003000 сек
Quick Sort      Возрастающая          : 0,006000 сек
stdlib qsort     Возрастающая          : 0,051000 сек

=== Убывающая ===
Shell Sort      Убывающая             : 1,506000 сек
Quick Sort      Убывающая             : 0,004000 сек
stdlib qsort     Убывающая             : 0,026000 сек

=== Смешанная ===
Shell Sort      Смешанная             : 0,356000 сек
Quick Sort      Смешанная             : 0,004000 сек
stdlib qsort     Смешанная             : 0,026000 сек
```

**Вывод:** При сравнении алгоритмов сортировки на массивах размером 10 000 и 100 000 элементов становится ясно, что различия в их эффективности проявляются тем сильнее, чем больше объём данных. На небольших массивах (10 000 элементов) разница между Quick Sort и стандартной библиотечной функцией qsort невелика, а Shell Sort лишь немного отстаёт, хотя уже заметно проигрывает на убывающих последовательностях. Однако при увеличении размера массива до 100 000 элементов преимущество Quick Sort становится очевидным: он стабильно показывает лучшие результаты как на случайных, так и на убывающих и смешанных данных, демонстрируя в десятки и даже сотни раз меньшие времена работы по сравнению с Shell Sort. qsort также остаётся достаточно эффективным, но всё же медленнее Quick Sort, особенно на возрастающих и смешанных данных. Shell Sort при больших объёмах оказывается наименее подходящим алгоритмом: его время работы сильно возрастает, и он существенно уступает конкурентам. Таким образом, можно сделать вывод, что для практического применения и особенно для работы с крупными массивами наилучшим выбором является Quick Sort, qsort можно рассматривать как универсальный вариант «по умолчанию», а Shell Sort имеет смысл использовать лишь для небольших наборов данных, где критична простота реализации, но не скорость.