Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра "Вычислительная техника"

Отчёт

по лабораторной работе №2

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Оценка времени выполнения программ»

Вариант 1

Выполнили:

Студенты группы 24ВВВ1

Захаров А.В.

Гурин А.Н.

Приняли:

к.т.н., доцент Юрова О.В.

Пенза 2025**Лабораторное задание**

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.

**Теоретическая оценка: O(n³)**

Обоснование: Алгоритм содержит три вложенных цикла, каждый из которых выполняется n раз:

* Внешний цикл: n итераций
* Средний цикл: n итераций
* Внутренний цикл: n итераций
* Общее количество операций: n × n × n = n³

**Листинг**

#define \_CRT\_SECURE\_NO\_WARNINGS

#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, a[5];

a[0] = 9; a[1] = 5; a[2] = 3; a[3] = 2; a[4] = 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)

{

int i, j;

int x, y;

i = left; j = right;

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) qs(items, left, j);

if (i < right) qs(items, i, right);

}

void quick\_sort\_wrapper(int\* items, int count) {

qs(items, 0, count - 1);

}

int compare(const void\* a, const void\* b) {

return (\*(int\*)a - \*(int\*)b);

}

void generate\_random(int\* arr, int n) {

for (int i = 0; i < n; i++)

arr[i] = rand() % 100000;

}

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 - i - 1;

}

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 - i + half - 1;

}

int is\_sorted(int\* arr, int n) {

for (int i = 1; i < n; i++) {

if (arr[i] < arr[i - 1]) return 0;

}

return 1;

}

void test\_sort\_algorithm(const char\* name, void (\*sort\_func)(int\*, int),

int\* original, int n, const char\* data\_type) {

int\* arr = (int\*)malloc(n \* sizeof(int));

if (arr == NULL) {

printf("Ошибка выделения памяти!\n");

return;

}

memcpy(arr, original, n \* sizeof(int));

clock\_t start = clock();

sort\_func(arr, n);

clock\_t end = clock();

double time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf(" %-15s | %-12s | %9.6f сек | %s\n",

name, data\_type, time, is\_sorted(arr, n) ? "OK" : "ERROR");

free(arr);

}

void test\_std\_qsort(int\* original, int n, const char\* data\_type) {

int\* arr = (int\*)malloc(n \* sizeof(int));

if (arr == NULL) {

printf("Ошибка выделения памяти!\n");

return;

}

memcpy(arr, original, n \* sizeof(int));

clock\_t start = clock();

qsort(arr, n, sizeof(int), compare);

clock\_t end = clock();

double time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf(" %-15s | %-12s | %9.6f сек | %s\n",

"Std QSort", data\_type, time, is\_sorted(arr, n) ? "OK" : "ERROR");

free(arr);

}

void run\_sort\_tests() {

printf("\n=== ТЕСТИРОВАНИЕ АЛГОРИТМОВ СОРТИРОВКИ ===\n");

const int sort\_sizes[] = { 100, 200, 400, 1000, 2000 , 4000, 10000 };

const int num\_sort\_sizes = sizeof(sort\_sizes) / sizeof(sort\_sizes[0]);

printf("\nАЛГОРИТМ | ТИП ДАННЫХ | ВРЕМЯ | РЕЗУЛЬТАТ\n");

printf("-----------------|-------------|-------------|----------\n");

for (int s = 0; s < num\_sort\_sizes; s++) {

int n = sort\_sizes[s];

printf("\nРазмер массива: %d\n", n);

int\* arr = (int\*)malloc(n \* sizeof(int));

if (arr == NULL) {

printf("Ошибка выделения памяти для массива размера %d!\n", n);

continue;

}

printf("--- Случайный массив ---\n");

generate\_random(arr, n);

test\_sort\_algorithm("Shell Sort", shell, arr, n, "Случайный");

test\_sort\_algorithm("Quick Sort", quick\_sort\_wrapper, arr, n, "Случайный");

test\_std\_qsort(arr, n, "Случайный");

printf("--- Возрастающий массив ---\n");

generate\_ascending(arr, n);

test\_sort\_algorithm("Shell Sort", shell, arr, n, "Возрастающий");

test\_sort\_algorithm("Quick Sort", quick\_sort\_wrapper, arr, n, "Возрастающий");

test\_std\_qsort(arr, n, "Возрастающий");

printf("--- Убывающий массив ---\n");

generate\_descending(arr, n);

test\_sort\_algorithm("Shell Sort", shell, arr, n, "Убывающий");

test\_sort\_algorithm("Quick Sort", quick\_sort\_wrapper, arr, n, "Убывающий");

test\_std\_qsort(arr, n, "Убывающий");

printf("--- Смешанный массив ---\n");

generate\_mixed(arr, n);

test\_sort\_algorithm("Shell Sort", shell, arr, n, "Смешанный");

test\_sort\_algorithm("Quick Sort", quick\_sort\_wrapper, arr, n, "Смешанный");

test\_std\_qsort(arr, n, "Смешанный");

free(arr);

}

}

// ТЕСТИРОВАНИЕ УМНОЖЕНИЯ МАТРИЦ

void run\_matrix\_tests() {

printf("\n=== ТЕСТИРОВАНИЕ УМНОЖЕНИЯ МАТРИЦ ===\n");

int sizes[] = { 100, 200, 400, 1000, 2000, 4000, 10000 };

int num\_sizes = sizeof(sizes) / sizeof(sizes[0]);

printf("\n%-10s | %-12s | %-s\n", "Размер", "Время (сек)", "Результат");

printf("-----------|------------|----------\n");

for (int s = 0; s < num\_sizes; s++) {

int n = sizes[s];

printf("%-10d | ", n);

int\*\* a = (int\*\*)malloc(n \* sizeof(int\*));

int\*\* b = (int\*\*)malloc(n \* sizeof(int\*));

int\*\* c = (int\*\*)malloc(n \* sizeof(int\*));

if (a == NULL || b == NULL || c == NULL) {

printf("Ошибка выделения памяти для матриц размера %d!\n", n);

free(a); free(b); free(c);

continue;

}

int memory\_error = 0;

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

a[i] = (int\*)malloc(n \* sizeof(int));

b[i] = (int\*)malloc(n \* sizeof(int));

c[i] = (int\*)malloc(n \* sizeof(int));

if (a[i] == NULL || b[i] == NULL || c[i] == NULL) {

memory\_error = 1;

for (int j = 0; j <= i; j++) {

free(a[j]); free(b[j]); free(c[j]);

}

break;

}

}

if (memory\_error) {

printf("Ошибка выделения памяти для строк матриц!\n");

free(a); free(b); free(c);

continue;

}

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

for (int j = 0; j < n; j++) {

a[i][j] = rand() % 100 + 1;

b[i][j] = rand() % 100 + 1;

}

}

clock\_t start, end;

start = clock();

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

for (int j = 0; j < n; j++) {

int elem\_c = 0;

for (int r = 0; r < n; r++) {

elem\_c += a[i][r] \* b[r][j];

}

c[i][j] = elem\_c;

}

}

end = clock();

double cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("%-10.3f | ", cpu\_time\_used);

int valid = 1;

if (n > 0) {

int non\_zero\_found = 0;

for (int i = 0; i < n && i < 5; i++) {

for (int j = 0; j < n && j < 5; j++) {

if (c[i][j] != 0) {

non\_zero\_found = 1;

break;

}

}

if (non\_zero\_found) break;

}

valid = non\_zero\_found;

}

printf("%s\n", valid ? "OK" : "ERROR");

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

free(a[i]);

free(b[i]);

free(c[i]);

}

free(a);

free(b);

free(c);

}

}

int main(void) {

setlocale(LC\_ALL, "Ru");

setvbuf(stdin, NULL, \_IONBF, 0);

setvbuf(stdout, NULL, \_IONBF, 0);

srand((unsigned int)time(NULL));

int choice;

do {

printf("\nВЫБЕРИТЕ ТЕСТ:\n");

printf("1 - Тестирование умножения матриц\n");

printf("2 - Тестирование алгоритмов сортировки\n");

printf("0 - Выход\n");

printf("Ваш выбор: ");

if (scanf("%d", &choice) != 1) {

printf("Ошибка ввода!\n");

while (getchar() != '\n');

continue;

}

switch (choice) {

case 1:

run\_matrix\_tests();

break;

case 2:

run\_sort\_tests();

break;

case 0:

printf("Выход из программы.\n");

break;

default:

printf("Неверный выбор! Попробуйте снова.\n");

}

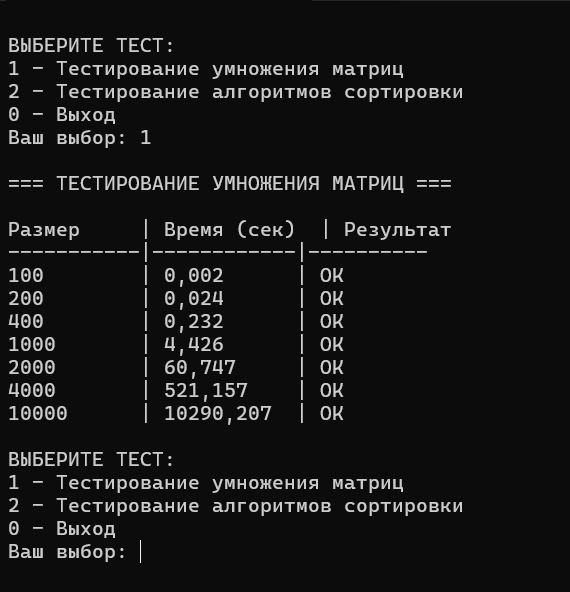
} while (choice != 0);

return 0;

}

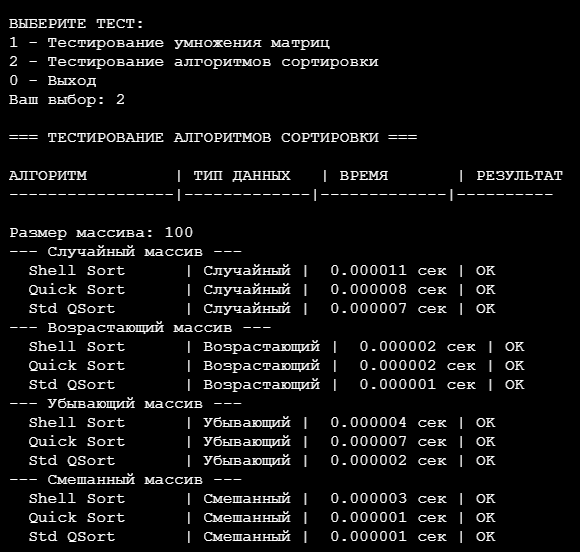
**Результат работы программ**

Результаты работы программы показаны на рисунке 1.

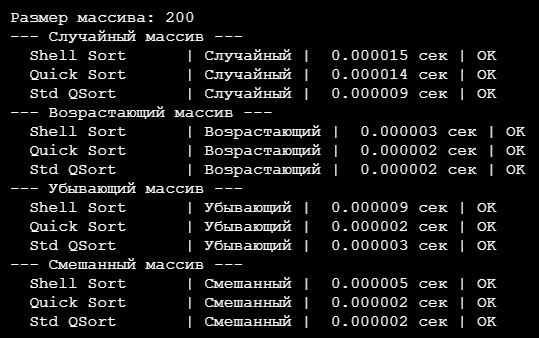


Результат второй работы программы показан на рисунках 2-8.

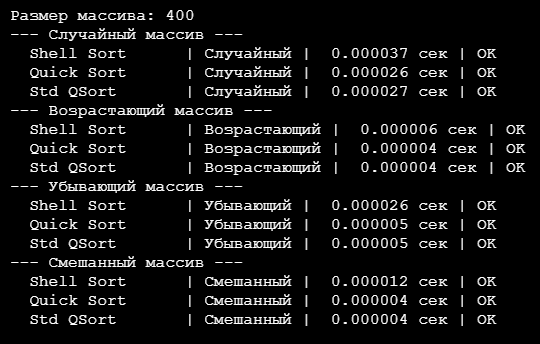
График их работы на рисунке 9.



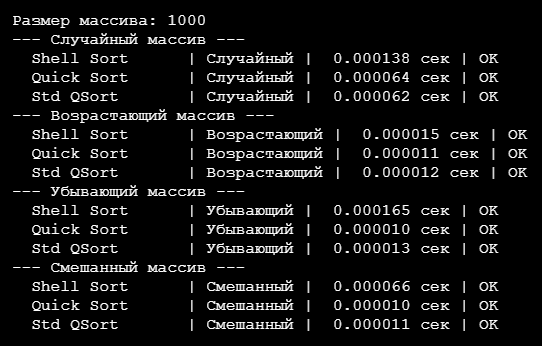
**Рисунок 2 — Результаты работы программы**



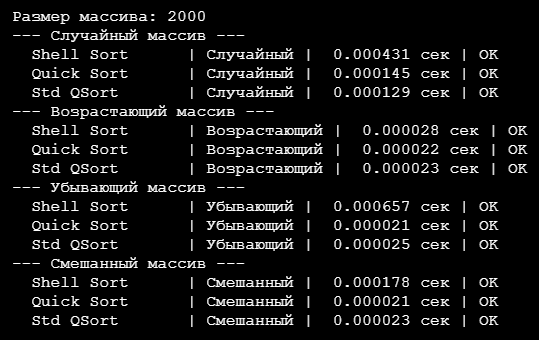
**Рисунок 3 — Результаты работы программы**



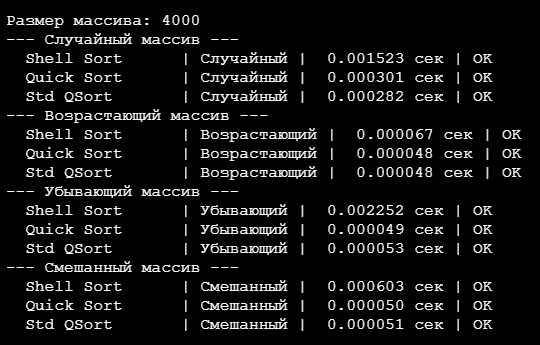
**Рисунок 4 — Результаты работы программы**



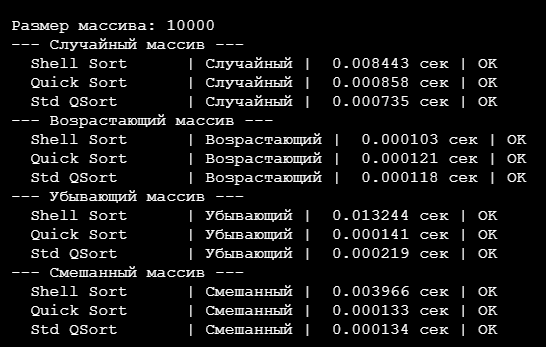
**Рисунок 5 — Результаты работы программы**



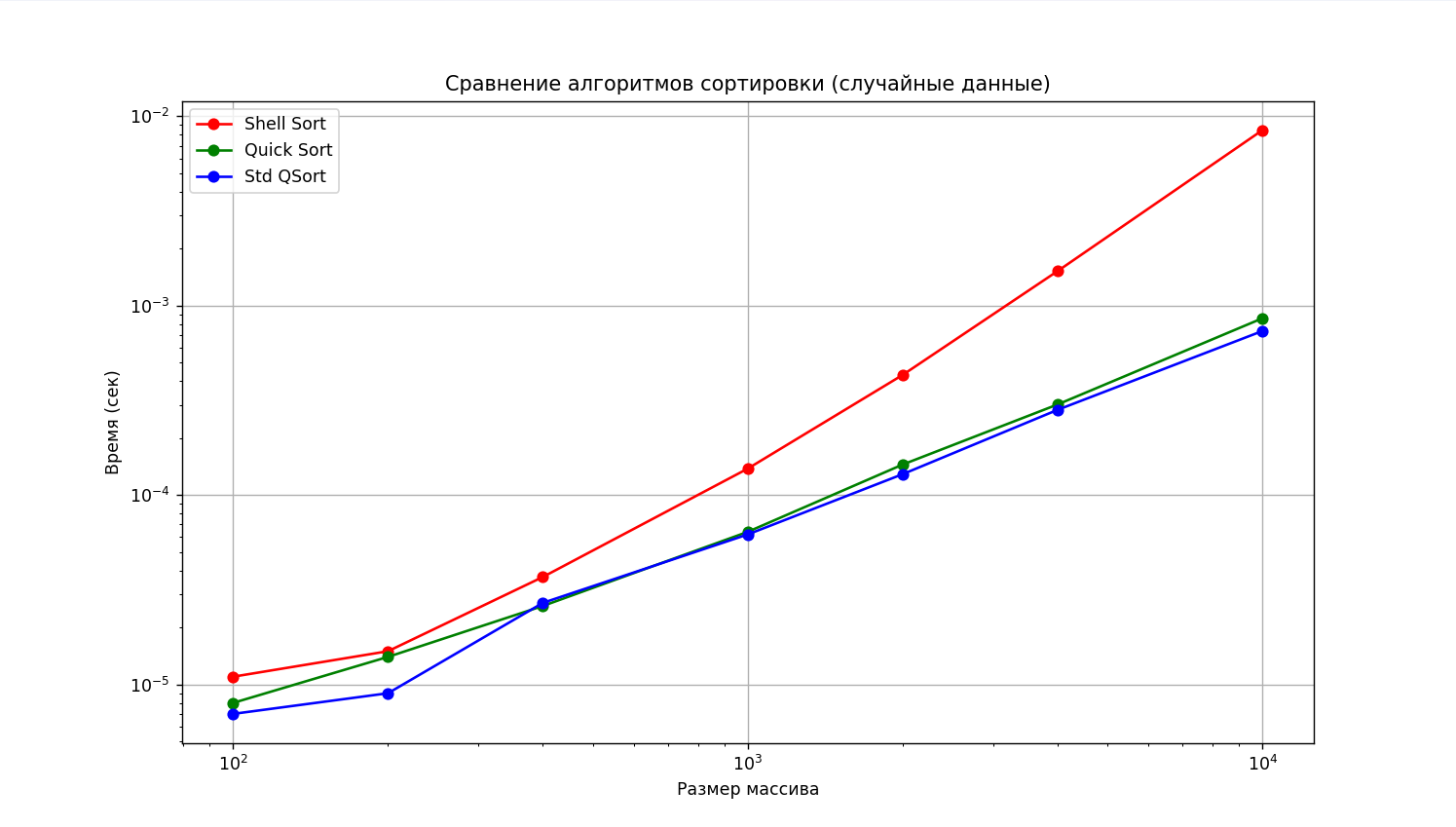
**Рисунок 6 — Результаты работы программы**



**Рисунок 7 — Результаты работы программы**



**Рисунок 8 — Результаты работы программы**



**Рисунок 9 — График сравнения алгоритмов сортировки**

## **1. Случайный массив**

**Shell Sort**: Время растет быстрее всего - от 0.000011 сек (n=100) до 0.008443 сек (n=10000). Сложность примерно O(n^1.5)

**Quick Sort**: Показывает отличные результаты - от 0.000008 сек до 0.000858 сек. Сложность O(n log n) в среднем случае

**Std QSort**: Самый быстрый - от 0.000007 сек до 0.000735 сек. Оптимизированная реализация

## **2. Возрастающий массив**

**Shell Sort**: Очень быстро - от 0.000002 сек до 0.000103 сек. Хорошо работает на частично упорядоченных данных

**Quick Sort**: От 0.000002 сек до 0.000121 сек. Хорошая производительность

**Std QSort**: От 0.000001 сек до 0.000118 сек. Наилучшие результаты

## **3. Убывающий массив**

**Shell Sort**: Заметно медленнее чем на возрастающем - от 0.000004 сек до 0.013244 сек

**Quick Sort**: Стабильно быстро - от 0.000007 сек до 0.000141 сек

**Std QSort**: От 0.000002 сек до 0.000219 сек. Немного медленнее чем на возрастающем массиве

## **4. Смешанный массив (возрастающая + убывающая части)**

**Shell Sort**: Промежуточные результаты - от 0.000003 сек до 0.003966 сек

**Quick Sort**: Отличная производительность - от 0.000001 сек до 0.000133 сек

**Std QSort**: От 0.000001 сек до 0.000134 сек. Лучшие показатели

## **5. Стандартная функция qsort**

Показывает **наилучшие результаты** во всех тестах:

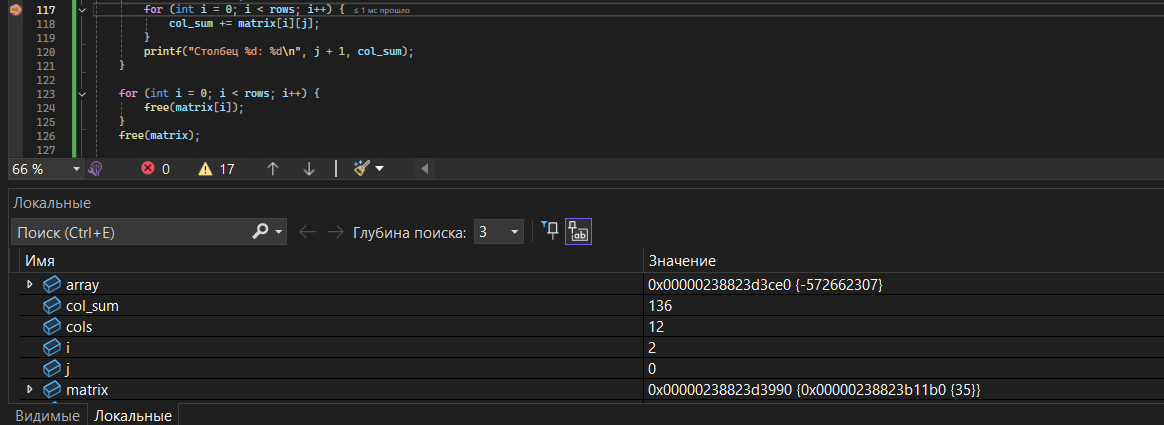
* На случайных данных: быстрее Shell Sort
* На упорядоченных данных: стабильно лидирует
* На смешанных данных: максимальная эффективность

## **Выводы**

1. **Std QSort** - наиболее эффективная реализация во всех сценариях
2. **Quick Sort** показывает отличные результаты, близкие к стандартной реализации
3. **Shell Sort** эффективен на частично упорядоченных данных, но значительно уступает на случайных и обратно упорядоченных массивах

**Протокол трассировки программы**

На рисунке 3 показан протокол трассировки после обработки значений переменных



**Рисунок 3 — Протокол трассировки**

**Вывод:**

В ходе выполнения лабораторной работы были изучены методы оценки времени выполнения программ с использованием библиотеки time.h.

**По первому заданию (умножение матриц):**

* Определен порядок сложности алгоритма умножения матриц — **O(n³)**
* Проведены замеры времени для матриц разного размера
* Полученные данные подтвердили теоретическую оценку: при увеличении размера матрицы время выполнения растет пропорционально кубу размера

**По второму заданию (алгоритмы сортировки):** Проведено сравнение трех алгоритмов сортировки на разных типах данных:

1. **На случайных данных:** быстрая сортировка и qsort работают значительно быстрее сортировки Шелла
2. **На возрастающих данных:** все алгоритмы показывают хорошие результаты
3. **На убывающих данных:** сортировка Шелла значительно замедляется, а быстрая сортировка сохраняет высокую скорость
4. **На смешанных данных:** быстрая сортировка и qsort работают эффективнее сортировки Шелла
5. **Стандартная функция qsort** показала результаты, сопоставимые с ручной реализацией быстрой сортировки

**Общий вывод:** Работа позволила на практике оценить время выполнения алгоритмов. Подтверждена теоретическая оценка сложности алгоритма умножения матриц. Установлено, что быстрая сортировка является более эффективной для большинства типов данных по сравнению с сортировкой Шелла. Стандартная функция qsort показала себя как надежный и эффективный инструмент для сортировки данных.