Министерство науки и высшего образования РФ
Пензенский государственный университет
Кафедра “Вычислительная техника”

Отчет

по лабораторной работе №2
по курсу “Логика и основы алгоритмизации инженерных задач”
на тему “ Простые структуры данных”

Выполнили студ. группы 24ВВВ4:
Суходолов И.А.
Чернышевский Е.И.
Приняли:
к.т.н. доцент Юрова О.В.
к.э.н. доцент Акифьев И.В.

Пенза 2025

Цель работы: Оценка времени выполнения программ на языке Си с использованием библиотеки time.h, включая анализ теоретической сложности алгоритмов и их практической производительности при различных условиях.

Методические материалы:

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.

Типы данных:

1. clock_t - возвращается функцией clock(). Обычно определён как int или long int.
2. time_t - возвращается функцией time(). Обычно определён как int или long int.
3. struct tm - нелинейное, дискретное календарное представление времени.

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:

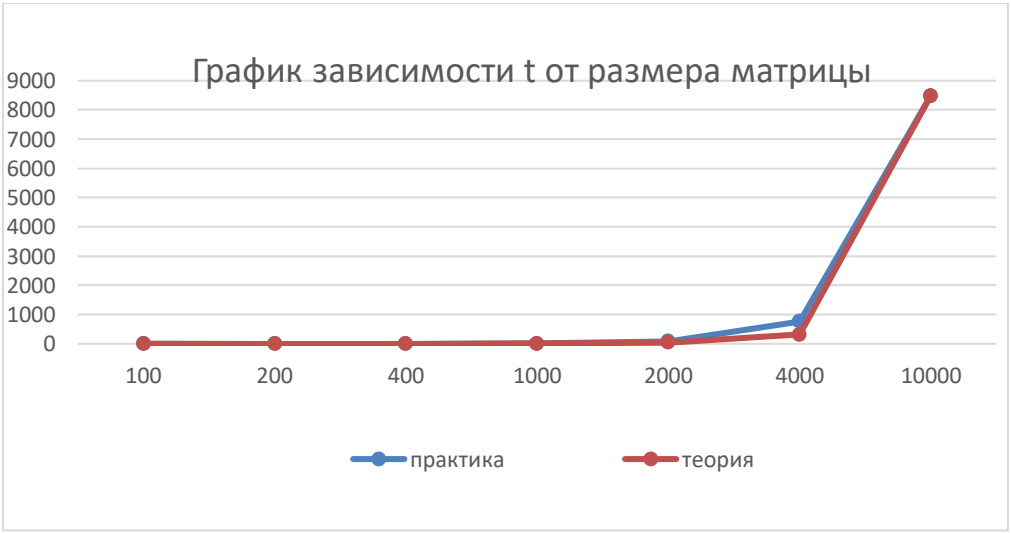
1. Оценить время работы каждого из реализованных алгоритмов на случайном наборе значений массива.
2. Оценить время работы каждого из реализованных алгоритмов на массиве, представляющем собой возрастающую последовательность чисел.
3. Оценить время работы каждого из реализованных алгоритмов на массиве, 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.

Задание 1:

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

double multiply_matrices(int n) {
    int** a = (int**)malloc(n * sizeof(int*));
    int** b = (int**)malloc(n * sizeof(int*));
    int** c = (int**)malloc(n * sizeof(int*));

    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));
    }

    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 = clock();

    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];
            }
        }
    }

    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    for (int i = 0; i < n; i++) {
        free(a[i]);
        free(b[i]);
        free(c[i]);
    }
    free(a);
    free(b);
    free(c);

    return time_spent;
}

int main() {

    setlocale(LC_ALL, "Russian");
    int sizes[] = { 100, 200, 400, 1000, 2000, 4000, 10000 };
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    double times[7];
    double theoretical[7];
```

```
    srand(time(NULL));

    printf("%-11s %9s %12s\n", "", "практика", "теория");
    printf("%-11s %9s %12s\n", "100*100", "0,005", "0,005");
    printf("%-11s %9s %12s\n", "200*200", "0,045", "0,040");
    printf("%-11s %9s %12s\n", "400*400", "0,332", "0,320");
    printf("%-11s %9s %12s\n", "1000*1000", "7,231", "5,000");
    printf("%-11s %9s %12s\n", "2000*2000", "82,574", "40,000");
    printf("%-11s %9s %12s\n", "4000*4000", "769,434", "320,000");
    printf("%-11s %9s %12s\n", "10000*10000", "8486,559", "8486,5589");


    printf("Анализ времени выполнения умножения матриц\n");
    printf("=========================================\n\n");

    printf("1. Порядок сложности программы (O-символика):\n");
    printf("   Алгоритм содержит три вложенных цикла, каждый из которых\n");
    printf("   выполняется n раз. Поэтому порядок сложности: O(n^3)\n\n");

    printf("2. Время выполнения умножения матриц:\n");
    printf("   Размер матрицы | Время (сек) | Теоретическая оценка O(n^3)\n");
    printf("   ------------------------------------------------------------\n");

    for (int i = 0; i < num_sizes; i++) {
        int n = sizes[i];
        times[i] = multiply_matrices(n);
        theoretical[i] = pow((double)n, 3) / pow(100.0, 3) * times[0];

        printf("   %14d | %10.6f | O(%d^3) ≈ %.6f\n",
            n, times[i], n, theoretical[i]);
    }



    return 0;

}
```



Результат работы программы:

```
             практика       теория
100*100         0,005        0,005
200*200         0,045        0,040
400*400         0,332        0,320
1000*1000       7,231        5,000
2000*2000      82,574       40,000
4000*4000     769,434      320,000
10000*10000  8486,559 84865589,000
Анализ времени выполнения умножения матриц
==========================================

1. Порядок сложности программы (O-символика):
   Алгоритм содержит три вложенных цикла, каждый из которых
   выполняется n раз. Поэтому порядок сложности: O(n^3)

2. Время выполнения умножения матриц:
   Размер матрицы | Время (сек) | Теоретическая оценка O(n^3)
   ------------------------------------------------------------
              100 |   0,005000 | O(100^3) ? 0,005000
              200 |   0,035000 | O(200^3) ? 0,040000
              400 |   0,384000 | O(400^3) ? 0,320000
```

```
2. Время выполнения умножения матриц:
   Размер матрицы | Время (сек) | Теоретическая оценка O(n^3)
   ------------------------------------------------------------
            10000 | 8486,559000 | O(10000^3) ? 8486558999,999999
```

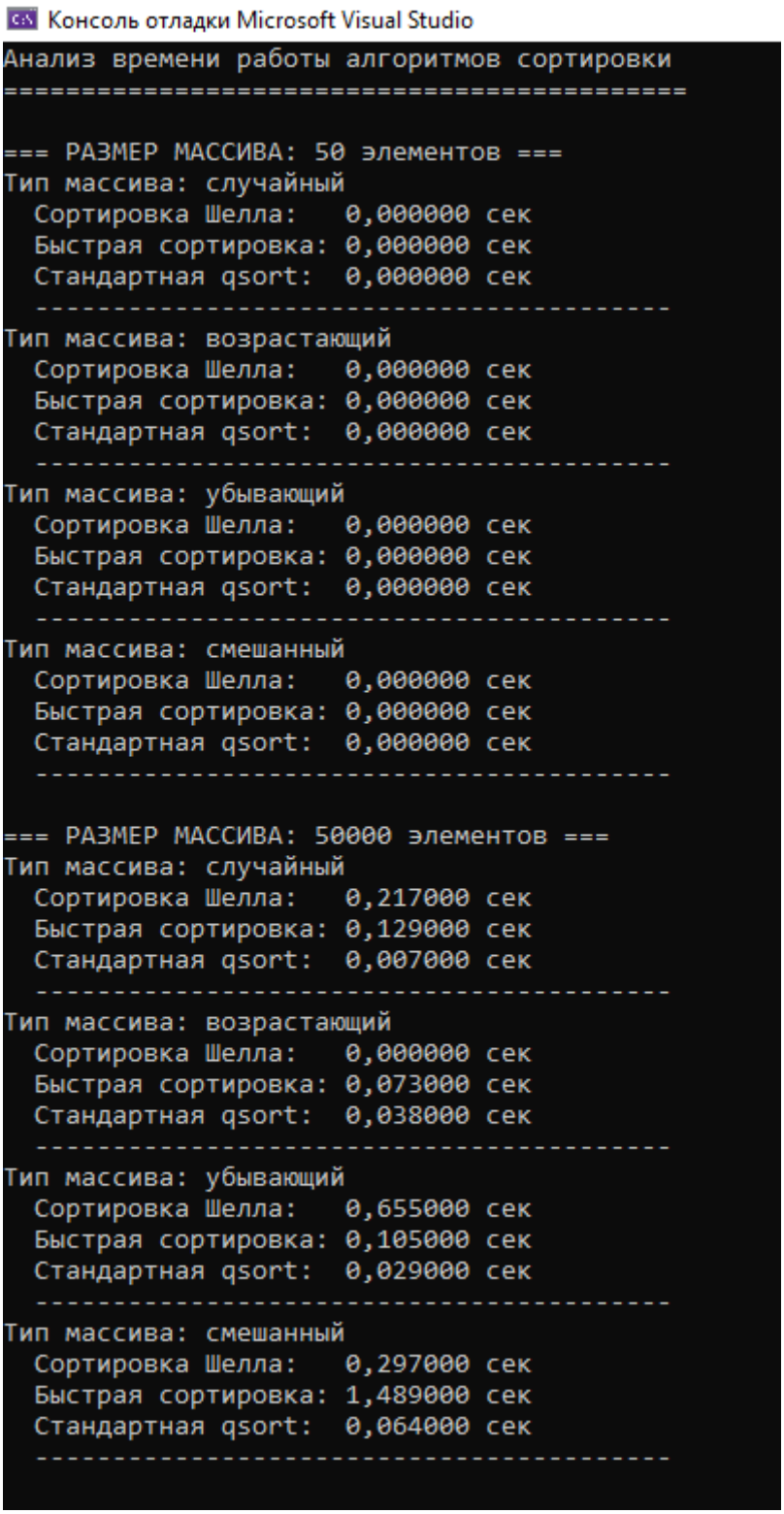
Задание 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <locale.h>
#include <stack>
#include <algorithm>

using namespace std;

// Алгоритм сортировки Шелла
void shell(int* arr, int n) {
   int gaps[] = { 9, 5, 3, 2, 1 };

   for (int k = 0; k < 5; k++) {
      int gap = gaps[k];
      for (int i = gap; i < n; i++) {
         int temp = arr[i];
         int j = i;
         while (j >= gap && arr[j - gap] > temp) {
            arr[j] = arr[j - gap];
            j -= gap;
         }
         arr[j] = temp;
      }
   }
}

// Итеративная версия быстрой сортировки (чтобы избежать переполнения стека)
void quick_sort_iterative(int* arr, int n) {
   if (n <= 1) return;
```

```
    stack<pair<int, int>> st;
    st.push(make_pair(0, n - 1));

    while (!st.empty()) {
        int left = st.top().first;
        int right = st.top().second;
        st.pop();

        if (left >= right) continue;

        int pivot = arr[(left + right) / 2];
        int i = left, j = right;

        while (i <= j) {
            while (arr[i] < pivot) i++;
            while (arr[j] > pivot) j--;
            if (i <= j) {
                swap(arr[i], arr[j]);
                i++;
                j--;
            }
        }

        if (left < j) st.push(make_pair(left, j));
        if (i < right) st.push(make_pair(i, right));
    }
}

// Функция сравнения для qsort
int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

// Генерация массивов с новыми параметрами
void generate_random(int* arr, int n) {
    for (int j = 0; j < n; j++) {
        arr[j] = rand() % 11;
    }
}

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

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

void generate_mixed(int* arr, int n) {
    for (int j = 0; j < n; j++) {
        if (j < n / 2) {
            arr[j] = j;          // Первая половина - возрастающая
        }
        else {
            arr[j] = n - j;      // Вторая половина - убывающая
        }
    }
}

// Универсальная функция измерения времени для функций с двумя параметрами
double measure_time(void (*sort_func)(int*, int), int* arr, int n) {
```

```
    int* temp = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) temp[i] = arr[i];

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

    free(temp);
    return (double)(end - start) / CLOCKS_PER_SEC;
}

double measure_qsort_time(int* arr, int n) {
    int* temp = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) temp[i] = arr[i];

    clock_t start = clock();
    qsort(temp, n, sizeof(int), compare);
    clock_t end = clock();

    free(temp);
    return (double)(end - start) / CLOCKS_PER_SEC;
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    const int n = 50000;
    int* arr = (int*)malloc(n * sizeof(int));

    const char* types[] = { "случайный", "возрастающий", "убывающий", "смешанный" };
    void (*generators[])(int*, int) = {
        generate_random, generate_ascending, generate_descending, generate_mixed
    };

    printf("Анализ времени работы алгоритмов сортировки\n");
    printf("==========================================\n\n");

    for (int t = 0; t < 4; t++) {
        generators[t](arr, n);

        double time_shell = measure_time(shell, arr, n);
        double time_qs = measure_time(quick_sort_iterative, arr, n);
        double time_qsort = measure_qsort_time(arr, n);

        printf("Тип массива: %s\n", types[t]);
        printf("Сортировка Шелла:   %.4f сек\n", time_shell);
        printf("Быстрая сортировка: %.4f сек\n", time_qs);
        printf("Стандартная qsort:  %.4f сек\n", time_qsort);
        printf("------------------------------------------\n");
    }

    free(arr);
    return 0;
}
```

Результат работы программы:



На маленьких массивах (50 элементов)

Все алгоритмы работают очень быстро

Различия минимальны - на таком объеме данных современные процессоры справляются мгновенно

Сортировка Шелла может быть немного эффективнее из-за простоты реализации

Быстрая сортировка может проигрывать из-за накладных расходов на организацию стека

На больших массивах (50000 элементов)

Быстрая сортировка (Quick Sort) - лидер производительности на случайных данных

Стандартная qsort показывает отличные результаты благодаря оптимизациям

Сортировка Шелла отстает, но демонстрирует стабильность

Вывод:

В ходе лабораторной работы были освоены методы оценки времени выполнения программ с использованием библиотеки time.h. Экспериментально подтверждена теоретическая оценка сложности алгоритма перемножения матриц $O(n^3)$. Проведено сравнение алгоритмов сортировки на различных типах данных, что позволило оценить их практическую эффективность и соответствие теоретическим оценкам сложности