Министерство науки и высшего образования РФ
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”

Выполнил студент гр. 22ВВВ3:
Кулахметов С.И.

Приняли:
к.т.н., доцент Юрова О.В.
к.э.н., доцент Акифьев И.В.

Пенза 2023

Цель работы

Ознакомиться с возможностями библиотеки time.h Языка C++. Реализовать алгоритмы, представленные в задании лабораторной работы и замерить время их выполнения.

Лабораторное задание

Задание 1 (Алгоритм умножения матриц)

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.

3. Построить график зависимости времени выполнения программы от размера матриц и сравнить полученный результат с теоретической оценкой.

Задание 2 (Алгоритмы сортировки Шелла и быстрой сортировки)

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 вышеуказанных наборах данных.

Пояснительный текст к прорамме

Программа написана на языке C++, имеет консольный интерфейс и базируется на 6-ти основных функциях: MatrixMultiplication*()* - отвечает за перемножение матриц; RandomValues*()* - отвечает за выполнение алгоритмов сортировок с радномными числами; *IncreasingSequence()* - отвечает за выполнение алгоритмов сортировок с возрастающей числовой последовательностью; *DescendingSequence()* - отвечает за выполнение алгоритмов сортировок с убывающей числовой последовательностью; *IncDesSequence()* - отвечает за выполнение алгоритмов

сортировок с возрастающей и убывающей числовой последовательностью; Qsort() - отвечает за применение быстрой сортировки ко всем указанным выше видам числовых последоваьтельностей.

Результаты работы программы

Задание 1

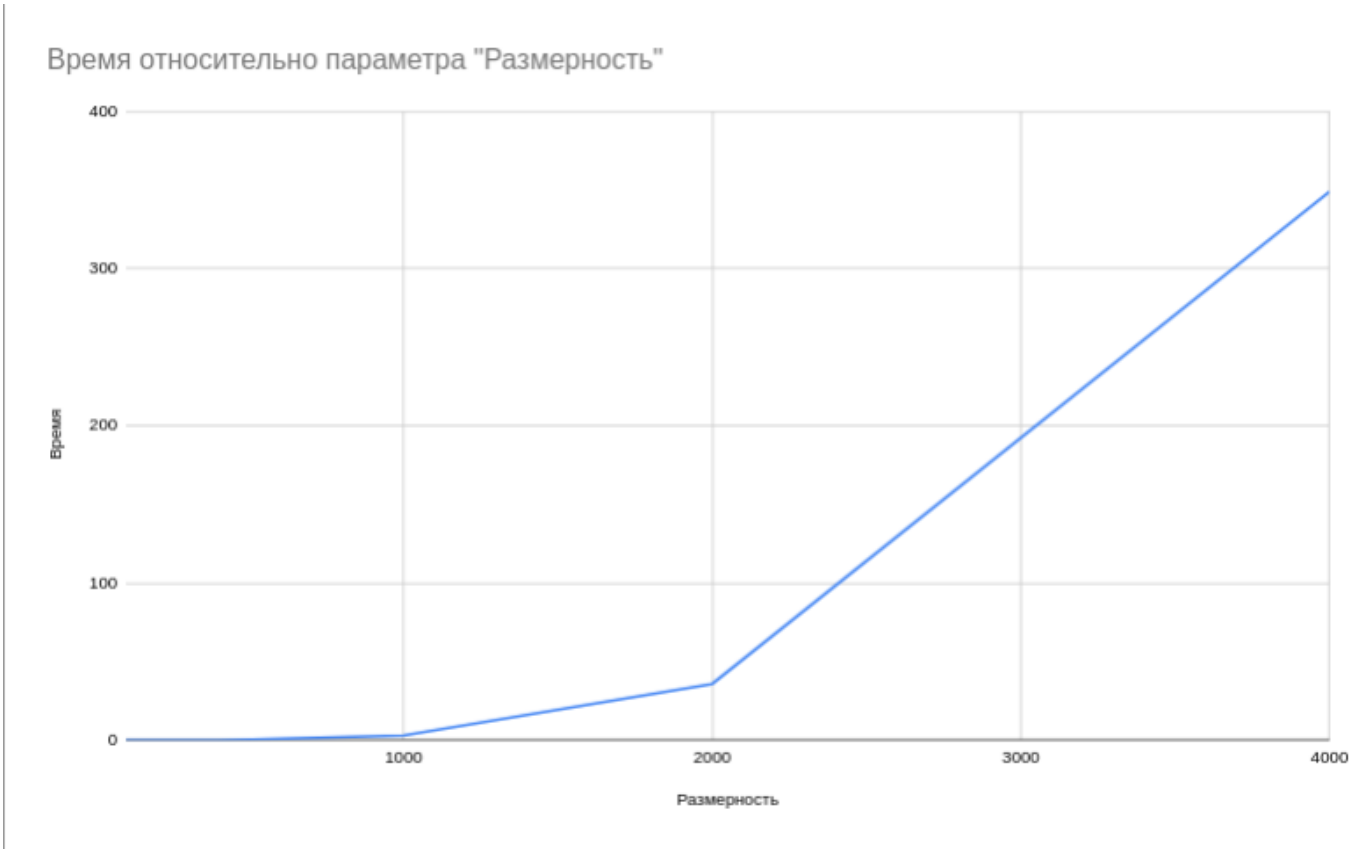
1. Сложность выполнение алгоритма перемножения матриц $O(N^3)$.

```
for (i = 0; i < Y1; i++)
{
     for (j = 0; j < X2; j++)
     {
          sum = 0;
          for(k = 0; k < X1; k++)
          {
              sum = sum + matrix1[i][k] * matrix2[k][j];
          }
          matrixresult[i][j] = sum;
     }
}
```

2. Время перемножения матриц размерностями: 100, 200, 400, 1000, 2000, 4000 (см. Приложение Б).

Размерность матрици	Время перемножения (сек.)
100	0,010016
200	0,024738
400	0,154357
1000	3,273310
2000	35,99690
4000	348,7280

3. Построен график, показывающий зависимость времени перемножения от размера матрицы.



Задание 2

Оценка времени работы алгоритмов (см. Приложение Б).

1000	Числовые последовательности			
Сортировки	Случайная	Возрастающая	Убывающая	Смешанная
Шелла	0,000310	0,000250	0,000205	0,000075
QS	0,000152	0,000205	0,000158	0,000146
qsort	0,000562	0,000200	0,000156	0,000040

10.000	Числовые последовательности			
Сортировки	Случайная	Возрастающая	Убывающая	Смешанная
Шелла	0,012614	0,000780	0,000896	0,000350
QS	0,002997	0,000469	0,000563	0,000536
qsort	0,007174	0,000298	0,000354	0,000186

Вывод

В ходе выполнения данной лабораторной работы были получены навыки реализации алгоритма перемножения матриц, сортировки Шелла и быстрой сортировки. А также навыки замера времени выполнения программы.

Ссылка на *GitHub* репозиторий с лабораторной работой

https://github.com/KulakhmetovS/Lab2

Приложение А

Листинг программы

```
#include <iostream>
#include <stdlib.h>
#include <string>
#include <iomanip>
#include <time.h>

using namespace std;


// Функции для умножения матриц и сортировок
void MatrixMultiplication();
void AlgTime();

// Функции, отвечающие за сортировку данных
void RandomValues();    // Рандомная последовательность
void IncreasingSequence();  // Возрастающаяя последовательность
void DescendingSequence();  // Убывающая последовательность
void     IncDesSequence();           //     Возрастающе-убывающая
последовательность
void Qsort();   // Быстрая сортировка всех последовательностей

// Функции, отвечающие за последовательность данных
int PartialSorting(int* , int);
int PartialSort(int *, int);
int ShellShort(int*, int);  // Сортировка Шелла
void QS(int *, int, int);   // Быстрая сортировка
int CombPartSort(int* , int);

int compare(const void*, const void*);



int main()
{
    int action = 0;

    cout << "Press \"Alt+F4\" to quit" << endl;

    while(1)
    {
        cout << "#  Choose the task: ";
        cin >> action;
           if ((action < 1) || (action > 2)) {cout << "Invalid
operation" << endl;}
            else if(action == 1) MatrixMultiplication();      //
Умножение матриц
        else if(action == 2) AlgTime(); // Сортировки
    }

    return 0;
}
```

```
void MatrixMultiplication()
{
    int X1 = 0, Y1 = 0, X2 = 0, Y2 = 0, i, j, k, sum;
    int **matrix1, **matrix2, **matrixresult;
    clock_t start_time, end_time;
    double cpu_time_used;

    cout << "<- Matrix multiplication ->" << endl;
    cout << "1'st matrix" << endl << "Enter Y-positions: ";
    cin >> Y1;
    cout << "Enter X-positions: ";
    cin >> X1;
    cout << "2'nd matrix" << endl << "Enter Y-positions: ";
    cin >> Y2;
    cout << "Enter X-positions: ";
    cin >> X2;

    // 1'st matrix initialisation
    matrix1 = new int*[Y1];
    for(i = 0; i < Y1; i++)
    matrix1[i] = new int[X1];

    // filling the 1'st matrix
    for(i = 0; i < Y1; i++)
    {
        for(j = 0; j < X1; j++)
        {
        matrix1[i][j] = rand() % (100 + 100 + 1) - 100;
        }
        //cout << endl;
    }

    // 2'nd matrix initialisation
    matrix2 = new int*[Y2];
    for(i = 0; i < Y2; i++)
    matrix2[i] = new int[X2];

    // filling the 2'nd matrix
    for(i = 0; i < Y2; i++)
    {
        for(j = 0; j < X2; j++)
        {
        matrix2[i][j] = rand() % (100 + 100 + 1) - 100;
        }
        //cout << endl;
    }

    // sum matrix initialisation
    matrixresult = new int*[Y1];
    for(i = 0; i < Y1; i++)
    matrixresult[i] = new int[X2];

    // matrix multiplication
    if (X1 == Y2)
    {
    start_time = clock();
```

```
            for (i = 0; i < Y1; i++)
            {
                 for (j = 0; j < X2; j++)
                 {
                     sum = 0;
                     for(k = 0; k < X1; k++)
                     {
                         sum = sum + matrix1[i][k] * matrix2[k][j];
                     }
                     matrixresult[i][j] = sum;
                 }
            }
        end_time = clock();
        }

             else cout << "Error: it's not possible to perform
multiplication" << endl;

        // clearing memory
        for(i = 0; i < Y1; i++)
        {
            delete[] matrix1[i];
        }
        delete[] matrix1;

        for(i = 0; i < Y2; i++)
        {
            delete[] matrix2[i];
        }
        delete[] matrix2;

        for(i = 0; i < Y1; i++)
        {
            delete[] matrixresult[i];
        }
        delete[] matrixresult;

             cpu_time_used = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
        cout << "Time: " << cpu_time_used << endl;
    }

    void AlgTime()
    {
    int action = 0;
    while(1)
        {
            cout << "Choose the task: ";
            cin >> action;
               if ((action < 0) || (action > 5)) {cout << "Invalid
operation" << endl;}
            else if(action == 0) main();
            else if(action == 1) RandomValues();
            else if(action == 2) IncreasingSequence();
            else if(action == 3) DescendingSequence();
            else if(action == 4) IncDesSequence();
```

```
            else if(action == 5) Qsort();
        }
    }

    void RandomValues()
    {
        clock_t start_time, end_time, start_time1, end_time1;
        double cpu_time_used, cpu_time_used1;
        int length = 0, i;

        cout << "<- Sort random values ->" << endl;
        cout << "Enter array length: ";
        cin >> length;

        int *array = new int[length];

        for(i = 0; i < length; i++)
        {
        array[i] = rand() % 100;
        }

        // Shell short
        start_time = clock();
        ShellShort(array, length);
        end_time = clock();
            cpu_time_used  =  ((double)(end_time  -  start_time))  /
CLOCKS_PER_SEC;
        cout << "Time of Shell short: " << cpu_time_used << endl;

        // quick sort
        start_time1 = clock();
        QS(array, 0, length);
        end_time1 = clock();
           cpu_time_used1  =  ((double)(end_time1  -  start_time1))  /
CLOCKS_PER_SEC;
        cout << "Time of Quick short: " << cpu_time_used1 << endl;


        delete[] array;
    }

    void IncreasingSequence()
    {
        clock_t start_time, end_time, start_time1, end_time1;
        double cpu_time_used, cpu_time_used1;
        int length = 0, i;

        cout << "<- Sort increasing values ->" << endl;
        cout << "Enter array length: ";
        cin >> length;

        int *array = new int[length];

        for(i = 0; i < length; i++)
        array[i] = rand() % 100;
```

```
        // partial sort
        PartialSorting(array, length);

        // Shell short
        start_time = clock();
        ShellShort(array, length);
        end_time = clock();
            cpu_time_used  =  ((double)(end_time  -  start_time))  /
CLOCKS_PER_SEC;
        cout << "Time of Shell short: " << cpu_time_used << endl;

        // quick sort
        start_time1 = clock();
        QS(array, 0, length);
        end_time1 = clock();
           cpu_time_used1  =  ((double)(end_time1  -  start_time1))  /
CLOCKS_PER_SEC;
        cout << "Time of Quick short: " << cpu_time_used1 << endl;


        delete[] array;
    }

    void DescendingSequence()
    {
        clock_t start_time, end_time, start_time1, end_time1;
        double cpu_time_used, cpu_time_used1;
        int length = 0, i;

        cout << "<- Sort descending values ->" << endl;
        cout << "Enter array length: ";
        cin >> length;

        int *array = new int[length];

        for(i = 0; i < length; i++)
        array[i] = rand() % 100;

        // partial sort
        PartialSort(array, length);

        // Shell short
        start_time = clock();
        ShellShort(array, length);
        end_time = clock();
            cpu_time_used  =  ((double)(end_time  -  start_time))  /
CLOCKS_PER_SEC;
        cout << "Time of Shell short: " << cpu_time_used << endl;

        // quick sort
        start_time1 = clock();
        QS(array, 0, length);
        end_time1 = clock();
           cpu_time_used1  =  ((double)(end_time1  -  start_time1))  /
CLOCKS_PER_SEC;
        cout << "Time of Quick short: " << cpu_time_used1 << endl;
```

```
        delete[] array;
    }

    void IncDesSequence()
    {
        clock_t start_time, end_time, start_time1, end_time1;
        double cpu_time_used, cpu_time_used1;
        int length = 0, i;

          cout << "<- Sort Increasing and descending values ->" <<
endl;
        cout << "Enter array length: ";
        cin >> length;

        int *array = new int[length];

        for(i = 0; i < length; i++)
        array[i] = rand() % 100;

        // partial sort
        CombPartSort(array, length);

        // Shell short
        start_time = clock();
        ShellShort(array, length);
        end_time = clock();
            cpu_time_used  =  ((double)(end_time  -  start_time))  /
CLOCKS_PER_SEC;
        cout << "Time of Shell short: " << cpu_time_used << endl;

        // quick sort
        start_time1 = clock();
        QS(array, 0, length);
        end_time1 = clock();
           cpu_time_used1  =  ((double)(end_time1  -  start_time1))  /
CLOCKS_PER_SEC;
        cout << "Time of Quick short: " << cpu_time_used1 << endl;


        delete[] array;
    }

    int compare(const void *x1, const void *x2)
    {
        return (*(int*)x1 - *(int*)x2);
    }

    void Qsort()
    {
             clock_t  start_time,  end_time,  start_time1,  end_time1,
start_time2, end_time2, start_time3, end_time3;
              double  cpu_time_used,  cpu_time_used1,  cpu_time_used2,
cpu_time_used3;
        int length = 0, i;
```

```
        cout << "<- Include Quick sort ->" << endl;
        cout << "Enter array length: ";
        cin >> length;

        int *array = new int[length];
        int *mas = new int[length];
        int *mas1 = new int[length];
        int *mas2 = new int[length];

        for(i = 0; i < length; i++)
        {
        array[i] = rand() % 100;
        mas[i] = array[i];
        mas1[i] = array[i];
        mas2[i] = array[i];
        }

        // Quick short
        start_time = clock();
        qsort(array, length, sizeof(int), compare);
        end_time = clock();
            cpu_time_used = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
           cout << "Time of quich sorting Random sequence: " <<
cpu_time_used << endl;

        // partial sort
        PartialSorting(mas, length);

        // Quick short
        start_time1 = clock();
        qsort(mas, length, sizeof(int), compare);
        end_time1 = clock();
           cpu_time_used1 = ((double)(end_time1 - start_time1)) /
CLOCKS_PER_SEC;
          cout << "Time of quich sorting Increacing sequence: " <<
cpu_time_used1 << endl;

        // partial sort
        PartialSort(mas1, length);

        // quick sort
        start_time2 = clock();
        qsort(mas1, length, sizeof(int), compare);
        end_time2 = clock();
           cpu_time_used2 = ((double)(end_time2 - start_time2)) /
CLOCKS_PER_SEC;
          cout << "Time of quick shorting Descending sequence: " <<
cpu_time_used2 << endl;

        // partial sort
        CombPartSort(mas2, length);
        // quick sort
        start_time3 = clock();
        qsort(mas2, length, sizeof(int), compare);
```

```
        end_time3 = clock();
           cpu_time_used3 = ((double)(end_time3 - start_time3)) /
CLOCKS_PER_SEC;
          cout << "Time of quick shorting Increacing and Descending
sequence: " << cpu_time_used3 << endl;

        delete[] array;
        delete[] mas;
        delete[] mas1;
        delete[] mas2;
    }

    int PartialSorting(int *array, int array_size)
    {
        int tmp = 0, j = 0;

        while(j <= array_size / 3)
        {
            for(int i = 0; i < array_size; i++)
                {
                    if(array[i] > array[i + 1])
                    {
                        tmp = array[i];
                        array[i] = array[i + 1];
                        array[i + 1] = tmp;
                    }
                }
            j++;
        }

        return *array;
    }

    int PartialSort(int *array, int array_size)
    {
        int tmp = 0, j = 0;

        while(j <= array_size / 3)
        {
            for(int i = 0; i < array_size; i++)
                {
                    if(array[i] < array[i + 1])
                    {
                        tmp = array[i];
                        array[i] = array[i + 1];
                        array[i + 1] = tmp;
                    }
                }
            j++;
        }

        return *array;
    }

    int CombPartSort(int *array, int array_size)
    {
```

```
    int tmp = 0, j = 0;
    int left = array_size / 2;

    while(j <= array_size)
    {
        for(int i = 0; i < left; i++)
            {
                if(array[i] > array[i + 1])
                {
                    tmp = array[i];
                    array[i] = array[i + 1];
                    array[i + 1] = tmp;
                }
            }
        j++;
    }

    j = 0;
    while(j <= array_size)
    {
        for(int i = left; i < array_size; i++)
            {
                if(array[i + 1] > array[i])
                {
                    tmp = array[i];
                    array[i] = array[i + 1];
                    array[i + 1] = tmp;
                }
            }
        j++;
    }

    return *array;
}

int ShellShort(int *array, int array_size)
{
    int i, d = array_size / 2;
    int *mas = new int[array_size];

    for(i = 0; i < array_size; i++)
    mas[i] = array[i];

    while(d >= 1)
    {
        for(i = d; i < array_size; i++)
        {
            int j = i;
            while((j >= d) && (mas[j - d] > mas[j]))
            {
                int t = mas[j];
                mas[j] = mas[j - d];
                mas[j - d] = t;
                j = j - d;
            }
        }
```

```
        d = d / 2;
    }

    delete[] mas;

    return 0;
}

void QS(int *array, int left, int right)
{
    int i = left, j = right;
    int x = array[(left + right) / 2], y;

    do
    {
        while((array[i] < x) && (i < right)) i++;
        while((x < array[j]) && (j > left)) j--;

        if(i <= j)
        {
            y = array[i];
            array[i] = array[j];
            array[j] = y;
            i++;
            j--;

        }
    } while(i <= j);

    if(left < j) QS(array, left, j);
    if(i < right) QS(array, i, right);
}
```

Приложение Б
Время перемножения матриц

```
<- Matrix multiplication ->
1'st matrix
Enter Y-positions: 100
Enter X-positions: 100
2'nd matrix
Enter Y-positions: 100
Enter X-positions: 100
Time: 0,010016
```

```
<- Matrix multiplication ->
1'st matrix
Enter Y-positions: 200
Enter X-positions: 200
2'nd matrix
Enter Y-positions: 200
Enter X-positions: 200
Time: 0,024738
```

```
<- Matrix multiplication ->
1'st matrix
Enter Y-positions: 400
Enter X-positions: 400
2'nd matrix
Enter Y-positions: 400
Enter X-positions: 400
Time: 0,154357
```

```
<- Matrix multiplication ->
1'st matrix
Enter Y-positions: 1000
Enter X-positions: 1000
2'nd matrix
Enter Y-positions: 1000
Enter X-positions: 1000
Time: 3,27331
```

```
<- Matrix multiplication ->
1'st matrix
Enter Y-positions: 2000
Enter X-positions: 2000
2'nd matrix
Enter Y-positions: 2000
Enter X-positions: 2000
Time: 35,9969
```

```
<- Matrix multiplication ->
1'st matrix
Enter Y-positions: 4000
Enter X-positions: 4000
2'nd matrix
Enter Y-positions: 4000
Enter X-positions: 4000
Time: 348,728
```

Оценка времени работы алгоритмов для обработки 1000 элементов

```
<- Sort random values ->
Enter array length: 1000
Time of Shell short: 0,00031
Time of Quick short: 0,000152
```

```
<- Sort increasing values ->
Enter array length: 1000
Time of Shell short: 0,00025
Time of Quick short: 0,000205
```

```
<- Sort descending values ->
Enter array length: 1000
Time of Shell short: 0,000205
Time of Quick short: 0,000158
```

```
<- Sort Increasing and descending values ->
Enter array length: 1000
Time of Shell short: 7,5e-05
Time of Quick short: 0,000146
```

```
<- Include Quick sort ->
Enter array length: 1000
Time of quich sorting Random sequence: 0,000562
Time of quich sorting Increacing sequence: 0,0002
Time of quick shorting Descending sequence: 0,000156
Time of quick shorting Increacing and Descending sequence: 4e-05
```

Оценка времени работы алгоритмов для обработки 10000 элементов

```
<- Sort random values ->
Enter array length: 10000
Time of Shell short: 0,012614
Time of Quick short: 0,002997
```

```
<- Sort increasing values ->
Enter array length: 10000
Time of Shell short: 0,00078
Time of Quick short: 0,000469
```

```
<- Sort descending values ->
Enter array length: 10000
Time of Shell short: 0.000896
Time of Quick short: 0.000563
```

```
<- Sort Increasing and descending values ->
Enter array length: 10000
Time of Shell short: 0.00035
Time of Quick short: 0.000536
```

```
<- Include Quick sort ->
Enter array length: 10000
Time of quich sorting Random sequence: 0.007174
Time of quich sorting Increacing sequence: 0.000298
Time of quick shorting Descending sequence: 0.000354
Time of quick shorting Increacing and Descending sequence: 0.000186
```