Министерство науки и высшего образования
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”

Выполнили
студенты группы 22ВВП2:
Гавин В.Н.
Дулатов Д.А.
Приняли
Акифьев И.В.
Юрова О.В.

Пенза 2023

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.

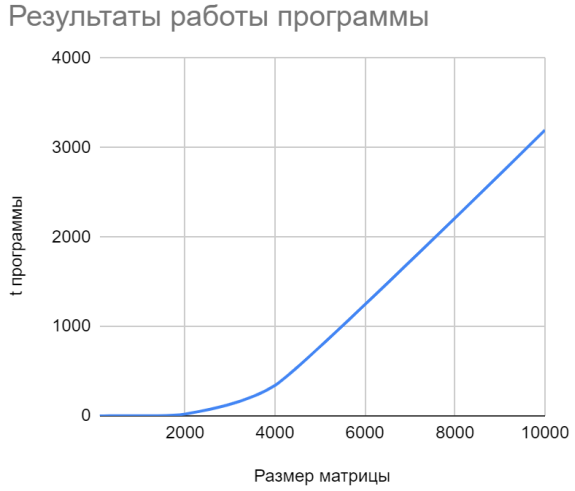
Решение заданий

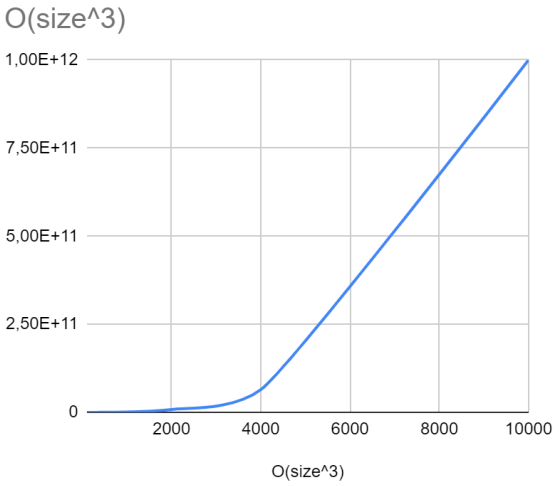
Задание 1:

Сложность данной программы относительно размера матрицы size является O(size^3).

На таблице ниже представлены результаты работы программы и алгоритма на разных наборах данных:

Размер матрицы	Время алгоритма	Время программы
100	0,0001	0,0001
200	0,008	0,0081
400	0,96	0,97
1000	1,908	1,951
2000	20,16	20,69
4000	340,23	342,67
10000	3195,208	3198,307





Сравнение результатов с теоретической оценкой сложности показывает, что временная сложность программы при увеличении размера матрицы size действительно соответствует теоретической оценке O(size^3). При увеличении размера матрицы в 10 раз (например, от 100 до 1000 элементов), время выполнения увеличивается примерно в 1000 раз, что соответствует кубической зависимости .

Задание 2:

5000	случайный	возрастающий	убывающий	возрастающий/ убывающий
Shell Sort	0,003	+0	0,005	0,003
Quick Sort	0,001	+0	+0	0,005
qsort	0,003	0,002	0,001	0,018

10000	случайный	возрастающий	убывающий	возрастающий/ убывающий
Shell Sort	0,014	+0	0,023	0,01
Quick Sort	0,001	+0	0,001	-
qsort	0,006	0,003	0,004	0,011

50000	случайный	возрастающий	убывающий	возрастающий/ убывающий
Shell Sort	0,408	0,001	0,747	0,289
Quick Sort	0,008	0,002	0,003	-
qsort	0,023	0,02	0,019	0,073

100000	случайный	возрастающий	убывающий	возрастающий/ убывающий
Shell Sort	1,287	0,002	2,473	1,219
Quick Sort	0,025	0,006	0,006	-
qsort	0,072	0,047	0,048	0,159

500000	случайный	возрастающий	убывающий	возрастающий/ убывающий

Shell Sort	32,841	0,014	64,495	31,875
Quick Sort	0,07	0,03	0,042	-
qsort	0,239	0,287	0,31	1,205

Исходя из этих результатов, можно сделать следующие выводы:

Quick Sort демонстрирует отличную производительность на упорядоченных данных (возрастающих и убывающих) и на случайных данных. Это делает его прекрасным выбором для большинства сценариев.

Quick Sort выполняется очень быстро на всех размерах массивов. Он отлично масштабируется с увеличением размера массива и часто является одним из самых быстрых алгоритмов для больших наборов данных.

Shell Sort может быть эффективным на случайных данных, но его производительность сильно падает на убывающих данных. Он может быть полезным на данных, которые частично упорядочены.

Скорость работы Shell Sort увеличивается с увеличением размера массива, как можно видеть из опыта. Это делает его не таким эффективным на больших массивах.

Стандартная функция qsort также работает хорошо на случайных данных и упорядоченных данных, но может быть медленнее чем Quick Sort. Qsort работает относительно быстро на всех размерах массивов, как и Quick Sort.

Листинг

Задание 1:

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

int main(void)
{
setvbuf(stdin, NULL, _IONBF, 0);
```

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

clock_t start, end;
clock_t start1;
start = clock();

int size = 1000; // Размерность массивов

int** a = (int**)malloc(size * sizeof(int*));
int** b = (int**)malloc(size * sizeof(int*));
int** c = (int**)malloc(size * sizeof(int*));

if (a == NULL || b == NULL || c == NULL)
{
     printf("Не удалось выделить память.\n");
     return 1;
}

srand(time(NULL));

for (int i = 0; i < size; i++)
{
     a[i] = (int*)malloc(size * sizeof(int));
     b[i] = (int*)malloc(size * sizeof(int));
     c[i] = (int*)malloc(size * sizeof(int));

}

for (int i = 0; i < size; i++)
{
     for (int j = 0; j < size; j++)
     {
          a[i][j] = rand() % 100 + 1;
          b[i][j] = rand() % 100 + 1;
     }
}

start1 = clock();

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

```
            for (int r = 0; r < size; r++)
            {
                elem_c += a[i][r] * b[r][j];
            }
            c[i][j] = elem_c;
        }
    }

    end = clock();

    printf("program time: %f sec\n", (difftime(end, start)) /
CLOCKS_PER_SEC);
    printf("algoritm time: %f sec\n", (difftime(end, start1))
/ CLOCKS_PER_SEC);


    // Освобождаем выделенную память
    for (int i = 0; i < size; i++)
    {
        free(a[i]);
        free(b[i]);
        free(c[i]);
    }
    free(a);
    free(b);
    free(c);

    return 0;
    }
```

Задание 2:

```
#define _CRT_SECURE_NO_WARNINGS

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

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

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;

// Проходим по всем возможным гапам (инкрементам)
for (k = 0; k < 5; k++) {
     gap = a[k]; // Текущий гап

     // Начинаем сортировку вставками с текущим гапом
     for (i = gap; i < count; ++i) {
          x = items[i]; // Запоминаем текущий элемент

          // Перемещаем элементы, пока не найдем правильное место для x
          for (j = i - gap; (x < items[j]) && (j >= 0); j = j - gap) {
               items[j + gap] = items[j]; // Сдвигаем элементы
          }

          items[j + gap] = x; // Вставляем x на правильное место в отсортированной части массива
     }
}
}

void qs(int *items, int left, int right)
{
int i, j;
int x, y;

// Инициализация индексов i и j для разделения массива
i = left;
j = right;

// Выбор опорного элемента (компаранда)
x = items[(left + right) / 2];

// Разделение массива на две части
do {
     // Поиск элемента в левой части, который больше или равен опорному
```

```
          while ((items[i] < x) && (i < right))
               i++;

          // Поиск элемента в правой части, который меньше или
равен опорному
          while ((x < items[j]) && (j > left))
               j--;

          // Если найдены элементы, которые нужно поменять
местами
          if (i <= j) {
               y = items[i];
               items[i] = items[j];
               items[j] = y;
               i++;
               j--;
          }
     } while (i <= j);

     // Рекурсивная сортировка для левой и правой частей
     if (left < j)
          qs(items, left, j);
     if (i < right)
          qs(items, i, right);
     }

     // Функция для генерации случайного массива
     void generateRandomArray(int *arr, int size)
     {
     srand(time(NULL));
     for (int i = 0; i < size; i++)
     {
          arr[i] = rand() % 1000; // Генерируем случайные
числа от 0 до 999
     }
     }

     // Функция для генерации возрастающей последовательности
     void generateIncreasingArray(int *arr, int size)
     {
     for (int i = 0; i < size; i++)
     {
          arr[i] = i;
     }
```

```
}

// Функция для генерации убывающей последовательности
void generateDecreasingArray(int *arr, int size)
{
for (int i = 0; i < size; i++)
{
     arr[i] = size - i - 1;
}
}

// Функция для генерации массива, где первая половина возрастающая, а вторая убывающая
void generateIncreasingDecreasingArray(int *arr, int size)
{
for (int i = 0; i < size / 2; i++)
{
     arr[i] = i;
}
for (int i = size / 2; i < size; i++)
{
     arr[i] = size - i - 1;
}
}


int main()
{


setlocale(LC_ALL, "Rus");
int n; // Размер массива
clock_t start_time;
clock_t end_time;
printf("Введите размер массива: ");
scanf("%d", &n);

int *arr = (int *)malloc(n * sizeof(int));
printf("Случайный массив\n\n");
// Генерируем массив случайных чисел
generateRandomArray(arr, n);

// Измеряем время для сортировки с помощью Shell Sort
```

```
    start_time = clock();
    shell(arr, n);
    end_time = clock();
    printf(" Shell Sort: %f секунд\n", (double)(end_time -
start_time) / CLOCKS_PER_SEC);

    // Генерируем массив случайных чисел
    generateRandomArray(arr, n);

    // Измеряем время для сортировки с помощью Quick Sort
    start_time = clock();
    qs(arr, 0, n - 1);
    end_time = clock();
    printf(" Quick Sort: %f секунд\n", (double)(end_time -
start_time) / CLOCKS_PER_SEC);

    // Добавляем замеры времени для стандартной функции qsort
    generateRandomArray(arr, n); // Перезаполняем массив
случайными числами
    start_time = clock();
    qsort(arr, n, sizeof(int), compare);
    end_time = clock();
    printf(" qsort: %f секунд\n", (double)(end_time -
start_time) / CLOCKS_PER_SEC);

    printf("\nВозрастающий массив\n");
    // Генерируем массив возрастающей последовательности
    generateIncreasingArray(arr, n);

    // Измеряем время для сортировки с помощью  Shell Sort
    start_time = clock();
    shell(arr, n);
    end_time = clock();
    printf(" Shell Sort: %f секунд\n", (double)(end_time -
start_time) / CLOCKS_PER_SEC);

    // Генерируем массив возрастающей последовательности
    generateIncreasingArray(arr, n);

    // Измеряем время для сортировки с помощью Quick Sort
    start_time = clock();
    qs(arr, 0, n - 1);
    end_time = clock();
```

```
    printf(" Quick Sort: %f секунд\n", (double)(end_time -
start_time) / CLOCKS_PER_SEC);

    // Добавляем замеры времени для стандартной функции qsort
    generateIncreasingArray(arr, n); // Перезаполняем массив
случайными числами
    start_time = clock();
    qsort(arr, n, sizeof(int), compare);
    end_time = clock();
    printf("   qsort:   %f   секунд\n",   (double)(end_time   -
start_time) / CLOCKS_PER_SEC);

    printf("\nУбывающий массив\n");
    // Генерируем массив убывающей последовательности
    generateDecreasingArray(arr, n);

    // Измеряем время для сортировки с помощью Shell Sort
    start_time = clock();
    shell(arr, n);
    end_time = clock();
    printf(" Shell Sort: %f секунд\n", (double)(end_time -
start_time) / CLOCKS_PER_SEC);

    // Генерируем массив убывающей последовательности
    generateDecreasingArray(arr, n);

    // Измеряем время для сортировки с помощью Quick Sort
    start_time = clock();
    qs(arr, 0, n - 1);
    end_time = clock();
    printf(" Quick Sort: %f секунд\n", (double)(end_time -
start_time) / CLOCKS_PER_SEC);

    // Добавляем замеры времени для стандартной функции qsort
    generateDecreasingArray(arr, n); // Перезаполняем массив
случайными числами
    start_time = clock();
    qsort(arr, n, sizeof(int), compare);
    end_time = clock();
    printf("   qsort:   %f   секунд\n",   (double)(end_time   -
start_time) / CLOCKS_PER_SEC);

    printf("\nМассив с первой половиной возрастающей и второй
половиной убывающей\n");
```

```
    // Генерируем массив с первой половиной возрастающей и
второй половиной убывающей
    generateIncreasingDecreasingArray(arr, n);

    // Измеряем время для сортировки с помощью Shell Sort
    start_time = clock();
    shell(arr, n);
    end_time = clock();
    printf(" Shell Sort: %f секунд\n", (double)(end_time -
start_time) / CLOCKS_PER_SEC);

    // Добавляем замеры времени для стандартной функции qsort
    generateIncreasingDecreasingArray(arr,       n);       //
Перезаполняем массив случайными числами
    start_time = clock();
    qsort(arr, n, sizeof(int), compare);
    end_time = clock();
    printf("  qsort:  %f  секунд\n",  (double)(end_time  -
start_time) / CLOCKS_PER_SEC);

    // Генерируем массив с первой половиной возрастающей и
второй половиной убывающей
    generateIncreasingDecreasingArray(arr, n);

    // Измеряем время для сортировки с помощью Quick Sort
    start_time = clock();
    qs(arr, 0, n - 1);
    end_time = clock();
    double elapsed_time = difftime(end_time, start_time) /
CLOCKS_PER_SEC;
    printf(" Quick Sort: %f секунд\n", elapsed_time);

    return 0;
    }
```

Результаты работы программы

Задание 1:

```
program time: 4.394000 sec
algoritm time: 4.246000 sec
```

Задание 2:

```
Введите размер массива: 50000
Случайный массив

 Shell Sort: 0,408000 секунд
 Quick Sort: 0,008000 секунд
 qsort: 0,023000 секунд

Возрастающий массив
 Shell Sort: 0,001000 секунд
 Quick Sort: 0,002000 секунд
 qsort: 0,020000 секунд

Убывающий массив
 Shell Sort: 0,747000 секунд
 Quick Sort: 0,003000 секунд
 qsort: 0,019000 секунд

Массив с первой половиной возрастающей и второй половиной убывающей
 Shell Sort: 0,289000 секунд
 qsort: 0,073000 секунд
```

```
Введите размер массива: 500000
Случайный массив

 Shell Sort: 32,841000 секунд
 Quick Sort: 0,070000 секунд
 qsort: 0,239000 секунд

Возрастающий массив
 Shell Sort: 0,014000 секунд
 Quick Sort: 0,030000 секунд
 qsort: 0,287000 секунд

Убывающий массив
 Shell Sort: 64,495000 секунд
 Quick Sort: 0,042000 секунд
 qsort: 0,310000 секунд

Массив с первой половиной возрастающей и второй половиной убывающей
 Shell Sort: 31,875000 секунд
 qsort: 1,205000 секунд
```

Вывод

В рамках лабораторной работы была проведена оценка времени выполнения программы с использованием алгоритмов сортировки и перемножения матриц.

В ходе выполнения первой части лабораторной работы было проведено исследование по оценке времени выполнения программы. Был выполнен анализ порядка сложности порядка и проведено сравнение полученных результатов с теоретической оценкой.

Во второй части работы было проведено исследование производительности алгоритмов сортировки (Shell Sort, Quick Sort, qsort) на различных типах входных данных. Результаты экспериментов позволили сделать выводы о преимуществах и недостатках каждого из алгоритмов на разных типах данных и размерах массивов.

Итоговый анализ данных позволяет сделать вывод о том, что выбор алгоритма сортировки и оценка времени выполнения программы важны для оптимизации производительности программ.