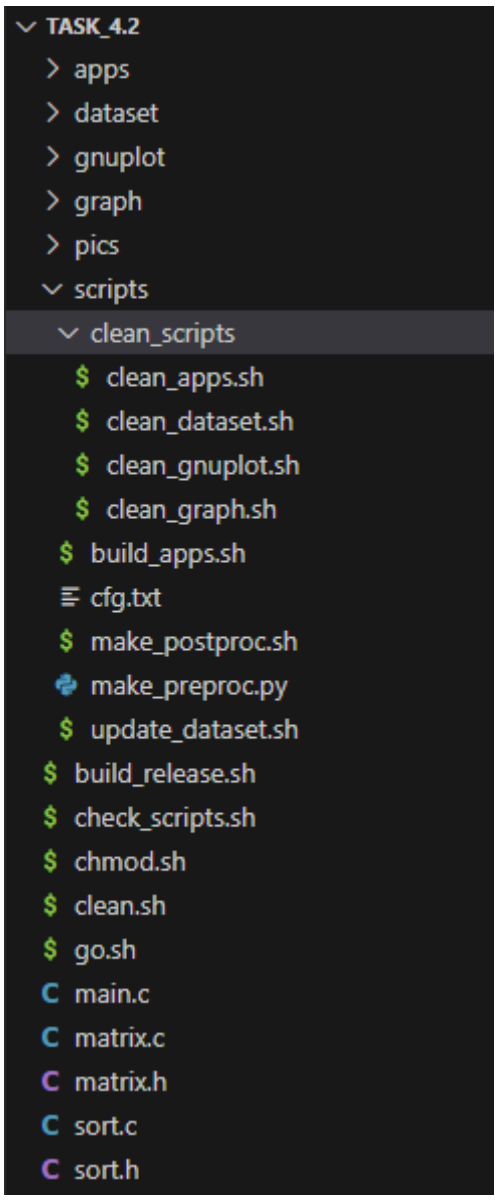
Смирнов Иван ИУ7-22Б - 2023г.

Отчет

Задание №4.2

Задание

В данном задании необходимо проанализировать время выполнения двух реализаций сортировки строк матрицы (по возрастанию суммы элементов строки), а именно: сортировка с кешированием суммы элементов строки, сортировка без кеширования. Программа собирается с несколькими типами оптимизаций (Os, O0, O3). Вид рабочей папки выглядит следующим образом:



(рис.1 – файловая структура задания №2)

Основной алгоритм сортировки – сортировка выбором. Ниже приведены 2 реализации.

1) С кешированием суммы

```
// Реализация 1
// Функция находит минимум в диапазоне от strat_ind до end_ind (с кешированием суммы)
size_t find_min_elem1(int a[][NMAX], size_t start_ind, size_t end_ind)
{
    size_t min_elem = start_ind;
    int min_sum = find_sum(a[start_ind], end_ind);
```

```
    for (size_t j = start_ind; j < end_ind; j++)
    {
        int cur_sum = find_sum(a[j], end_ind);
        if (cur_sum < min_sum)
        {
            min_sum = cur_sum;
            min_elem = j;
        }
    }
    return min_elem;
}
// Функция сортировки выбором
void selection_sort1(int a[][NMAX], size_t a_size)
{
    size_t min_elem = 0;
    for (size_t i = 0; i < a_size; i++)
    {
        min_elem = find_min_elem1(a, i, a_size);
        swap_arrays(a[i], a[min_elem], a_size);
    }
}
// конец
```

2) Без кеширования суммы

```
// Реализация 2
// Функция находит минимум в диапазоне от strat_ind до end_ind (без кеширования суммы)
size_t find_min_elem2(int a[][NMAX], size_t start_ind, size_t end_ind)
{
    size_t min_elem = start_ind;
    for (size_t j = start_ind; j < end_ind; j++)
    {
        if (find_sum(a[j], end_ind) < find_sum(a[min_elem], end_ind))
        {
            min_elem = j;
        }
    }
```

```
    return min_elem;
}
// Функция сортировки выбором
void selection_sort2(int a[][NMAX], size_t a_size)
{
    size_t min_elem = 0;
    for (size_t i = 0; i < a_size; i++)
    {
        min_elem = find_min_elem2(a, i, a_size);
        swap_arrays(a[i], a[min_elem], a_size);
    }
}
// конец
```

В обеих реализациях используются следующие вспомогательные функции:

```
// Вспомогательные функции для обоих реализаций сортировок
// Функция находит сумму элементов массива
int find_sum(const int *a, size_t a_size)
{
    int sum = 0;
    for (size_t i = 0; i < a_size; i++)
        sum += a[i];
    return sum;
}
// Функция меняет местами массивы (по-элементно)
void swap_arrays(int a[], int b[], size_t n)
{
    for (size_t i = 0; i < n; i++)
    {
        int tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }
}
```

Все реализации находятся в файле `sort.c`. Описание каждой из реализаций можно прочитать в заголовочном файле `sort.h`.

Также была реализована функция генерации матрицы случайных чисел (`init`). При этом заполнение матрицы случайными числами было организовано таким образом, чтобы при одинаковой размерности матрицы (например, для 15x15) для разных уровней оптимизации матрица будет заполняться одинаковыми случайными числами. Описание функции находится в заголовочном файле `matrix.h`; сама функция в `matrix.c`.

В данной задаче мы будем рассматривать только квадратную матрицу, так как проводить тестовые измерения будет проще именно с квадратной матрицей. Так же с помощью квадратной матрицы можно получить более наглядный график, то есть сравнить время выполнения программы для матрицы `50x50` с матрицей `60x60` будет проще, чем время для матрицы `50x80` с матрицей `60x70`. Проще ориентироваться по изменению одной составляющей, чем по двум.

Программу необходимо запустить с одним аргументом, который отвечает за размерность матрицы (в программе это переменная `size`). Такие переменные как максимальная размерность матрицы (`NMAX`) и тип сортировки (`SORT_TYPE`) определяются на этапе сборки. Программа выводит на экран время работы выбранной сортировки в микросекундах (`1e-6c`).

Чтобы проанализировать время для матриц разного размера, разных типов оптимизации и разных реализаций сортировок, была написана целая анализирующая система. Необходимо запустить скрипт ***`go.sh`***.

```
#!/bin/bash

./scripts/build_apps.sh
./scripts/update_dataset.sh
```

```
./scripts/clean_scripts/clean_graph.sh
python3 ./scripts/make_preproc.py
./scripts/make_postproc.sh
```

Проанализируем работу скрипта. **(1)** Сначала из папки `scripts` запускается скрипт ***./scripts/build_apps.sh***, который собирает все возможные версии программы (с типами оптимизации и двумя типами сортировок, то есть собирается в сумме 3*2=6 различных исполняемых файлов).

Вся информация о сборке программы хранятся в файле ***./scripts/cfg.txt***

cfg.txt:

Os O0 O3

1 2

100

20

1 2 5 10 15 20 25 35 40 50 60 65 75 85 90 100

В первой строке написаны типы оптимизации (Os, O0, O3). Во второй строке расположены разные реализации сортировки (описаны ранее). В третьей строке написано максимальное количество строк/столбцов в матрице, по-другому, - размерность матрицы (по условию задачи оно равно 100). В четвертой строке написано количество проводимых измерений (тестов), то есть сколько раз запустится каждый из 6 исполняемых файлов. В пятой строке написаны все тестируемые размерности матрицы.
Далее все 6 исполняемых файлов собираются с помощью скрипта ***build_release.sh:***

```
#!/bin/bash


gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla
-"${1}" -DSORT_TYPE="${2}" -DNMAX="${3}" -c main.c
gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla
-DNMAX="${3}" -c matrix.c
```

```
gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -Wvla
-DNMAX="${3}" -c sort.c

gcc -o ./apps/app_"$1"_s"$2"_n"$3".exe main.o matrix.o sort.o -lm
```

Все исполняемые файлы помещаются в папку **`apps`**

(2) Далее запускается скрипт ***`./scripts/update_data.sh`***, который запускает `tests` раз (в данном случае 20) на разных размерностях матрицы каждый из исполняемых файлов. Результаты (время в мкс) помещаются в текстовые файлы в папку `dataset`. Однако в данном эксперименте, для достижения более точного результата, было проведено около 7000 тестов для каждой из 6 программ.

(3) Далее запускается скрипт ***`./scripts/make_preproc.py`***, который на основе данных из **`dataset`** формирует новые данные, необходимые для дальнейшего анализа (среднее, минимум и максимум, медианное значение, верхний и нижний квартили). Все полученные данные скрипт сохраняет в текстовых файлах в папку ***graph.*** Перед запуском скрипта удаляется старый “graph” (если существовал) с помощью вспомогательного скрипта **clean_graph.sh.** (Данные из graph были помещены в архив из-за ограничения в 1МБ в мудле).

Данные из `graph` выглядят примерно так:

```
. . .
108 137 105 105 106 107 712 40
139 137 105 105 106 107 712 40
712 137 105 105 106 107 712 40
134 267 134 219 233 316 503 50
191 267 134 219 233 316 503 50
. . .
```

1-ый столбец – время из dataset

2-ой столбец – полученное среднее

3-ий столбец – минимум

4-ый столбец – нижний квартиль

5-ый столбец – медианное значение

6-ой столбец – верхний квартиль

7-ой столбец – максимум

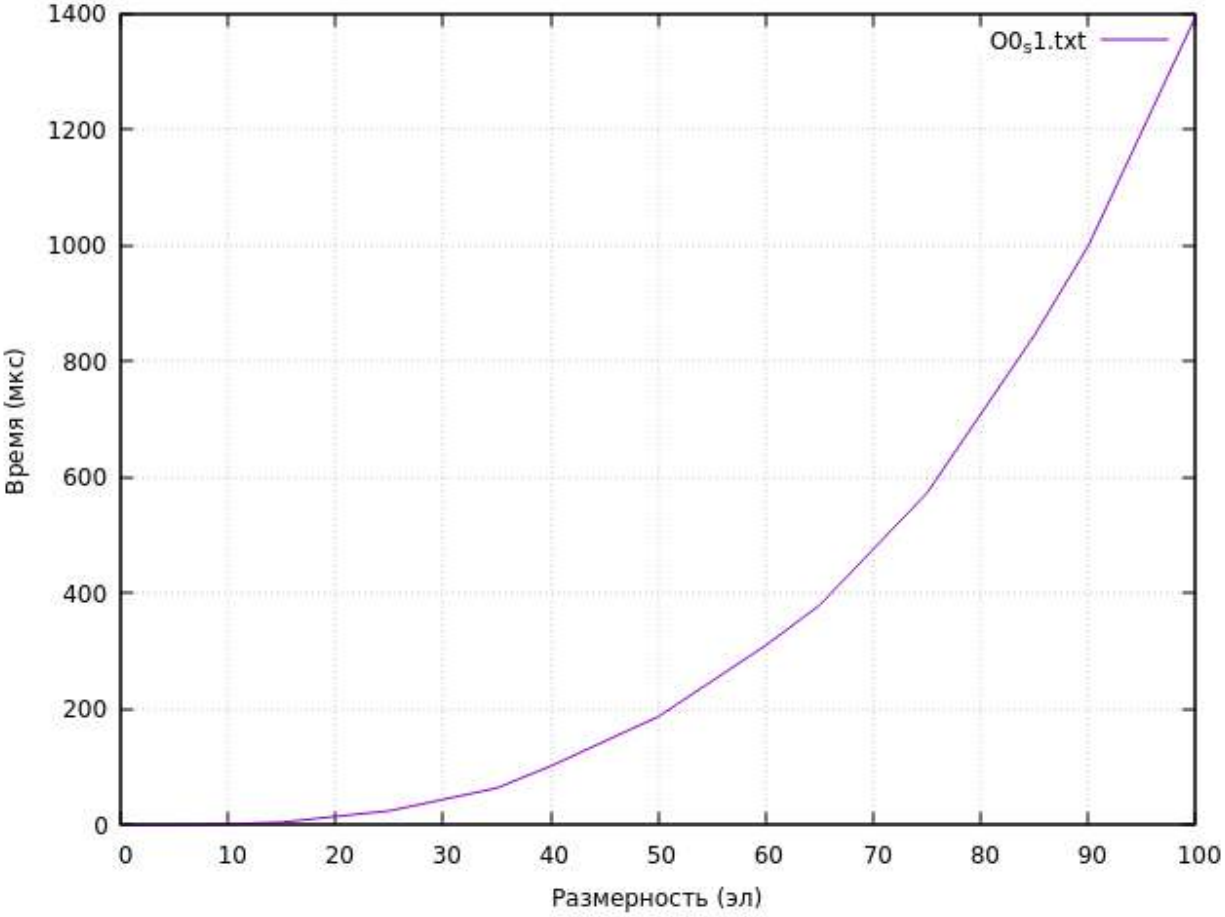
8-ой столбец – размер массива

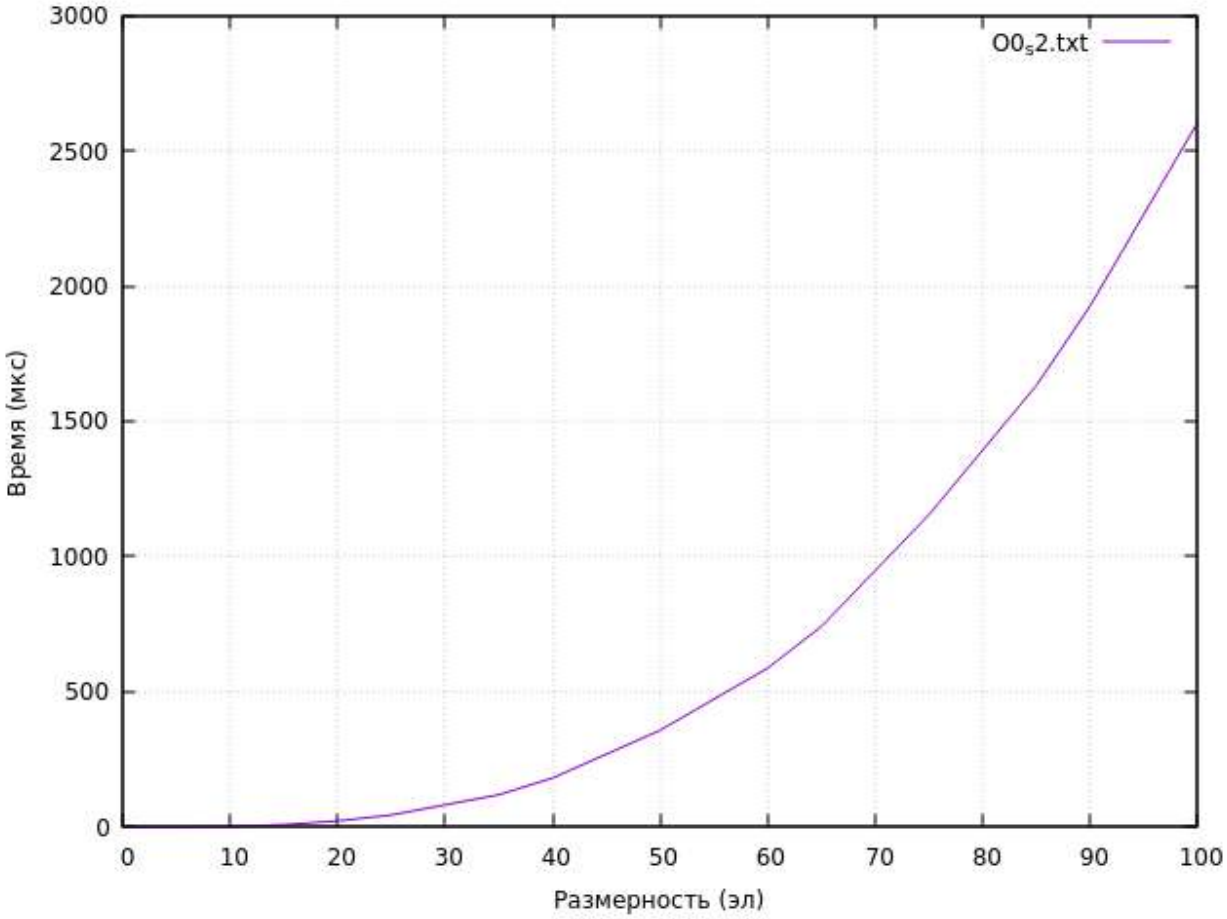
На основе этих данных можно построить графики зависимости времени сортировки от размерности матрицы. По заданию необходимо построить обычные кусочно-линейные графики зависимости времени в любых единицах измерения (использует мкс) от размерности матрицы. Для этого используем 2-ой и 8-ой столбцы соответсвенно.

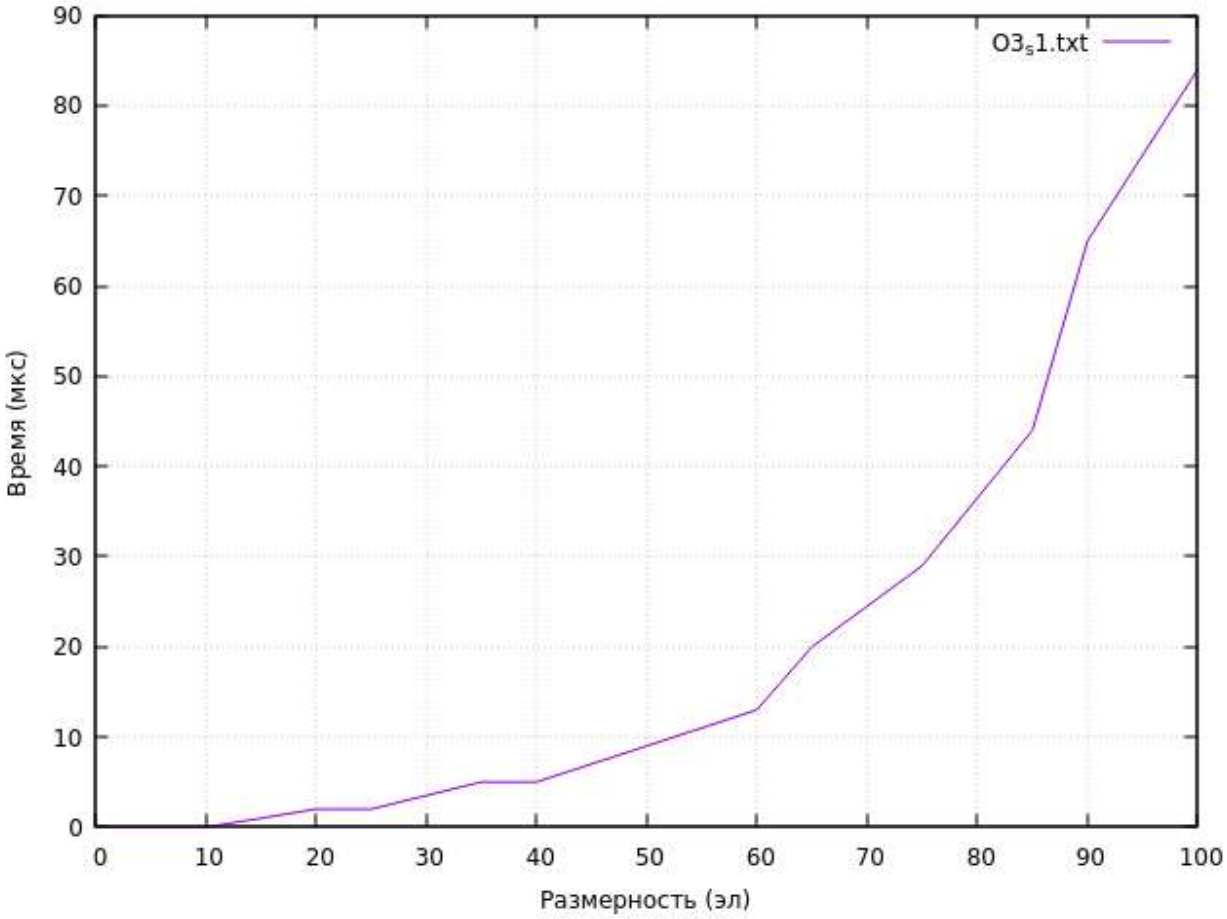
(4) Запускается скрипт ***./scripts/make_outproc.sh***, который строит графики и сохраняет их в образцах (файлах `.gpi`) в папке `gnuplot`.

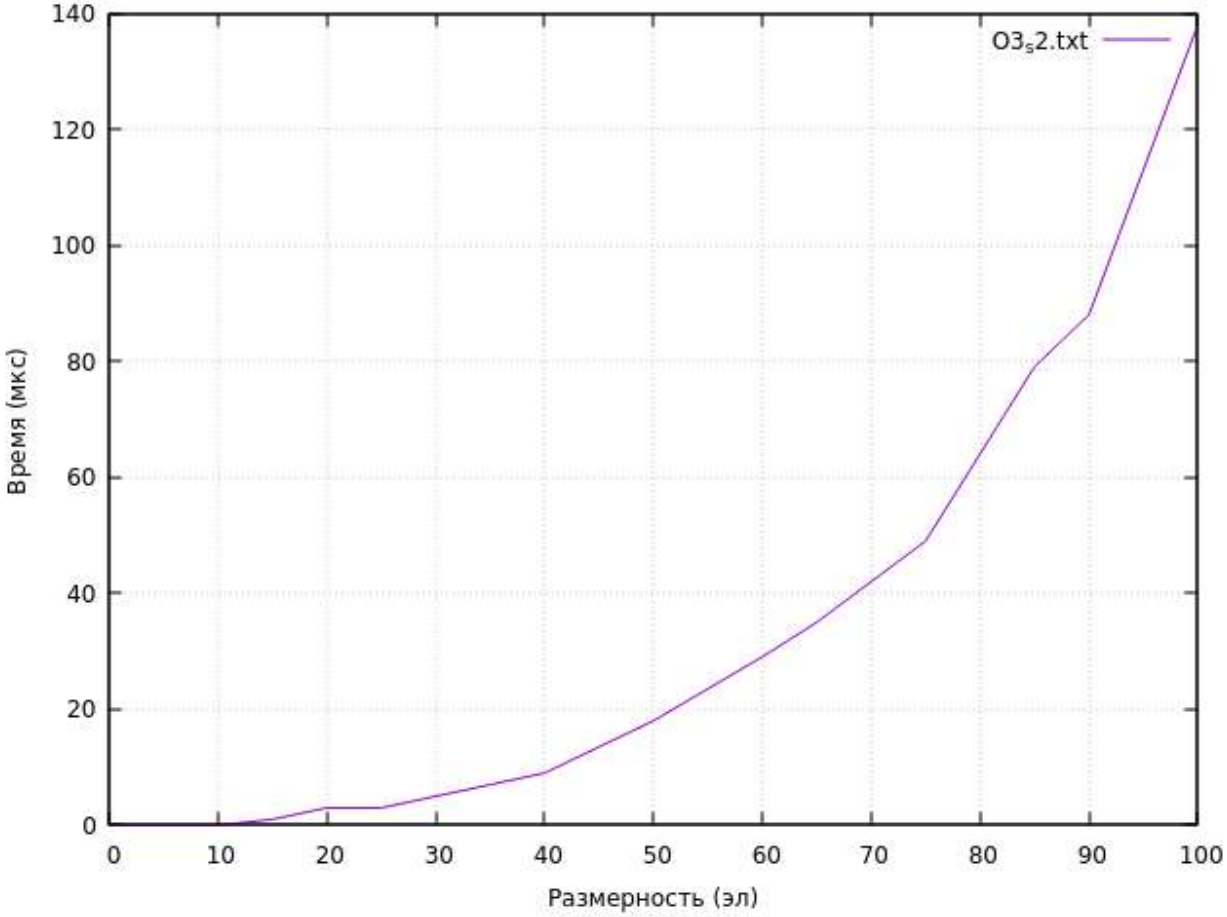
В конце скрипта написана команда, с помощью которой можно увидеть отображения графика в программе `gnuplot (`*`gnuplot $PATH$NAME".gpi" -persist`*`)`.
Графики строятся следующим образом: по оси x – количество элементов (столбец 8) в массиве, по оси y – среднее замеренное время для данного количества элементов (столбец 2).

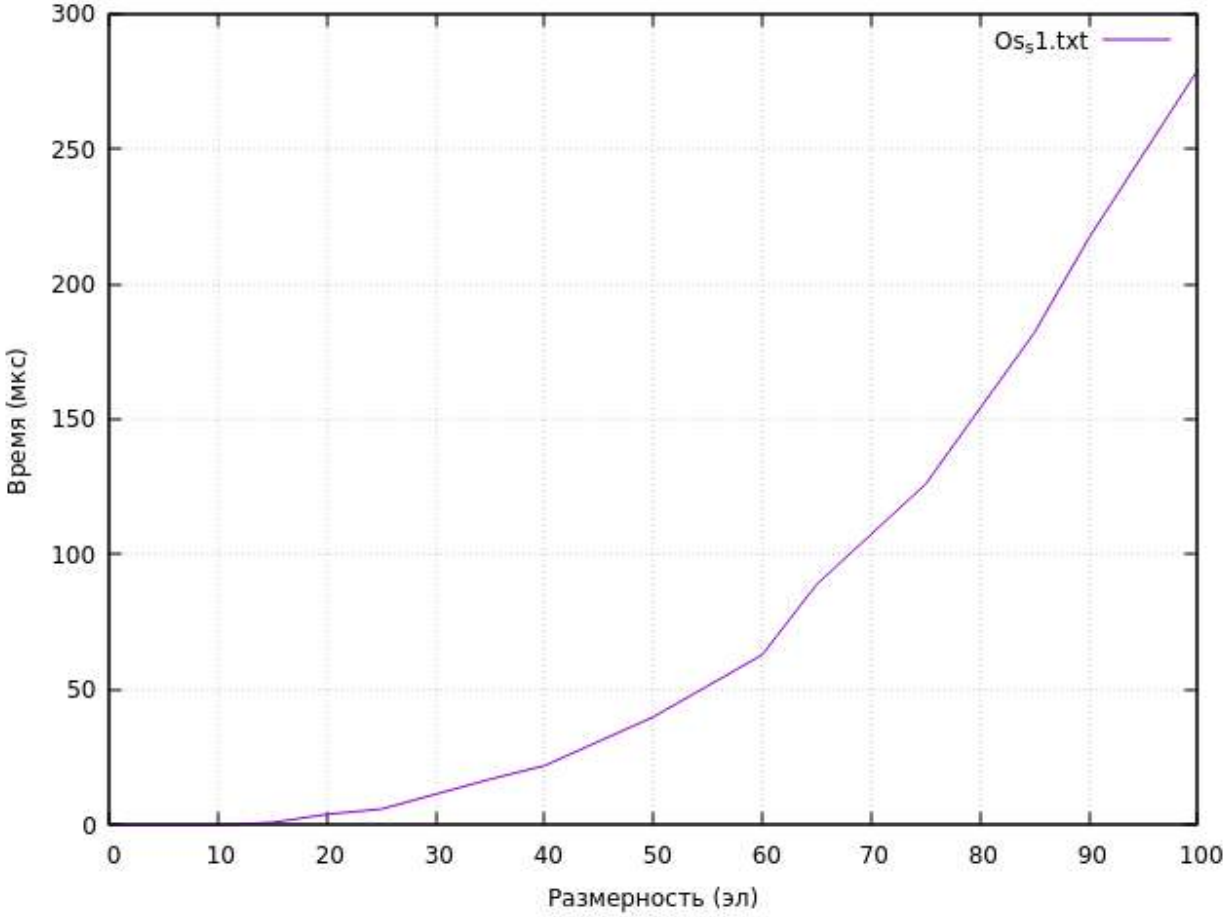
Графики расположены в папке `pics.`

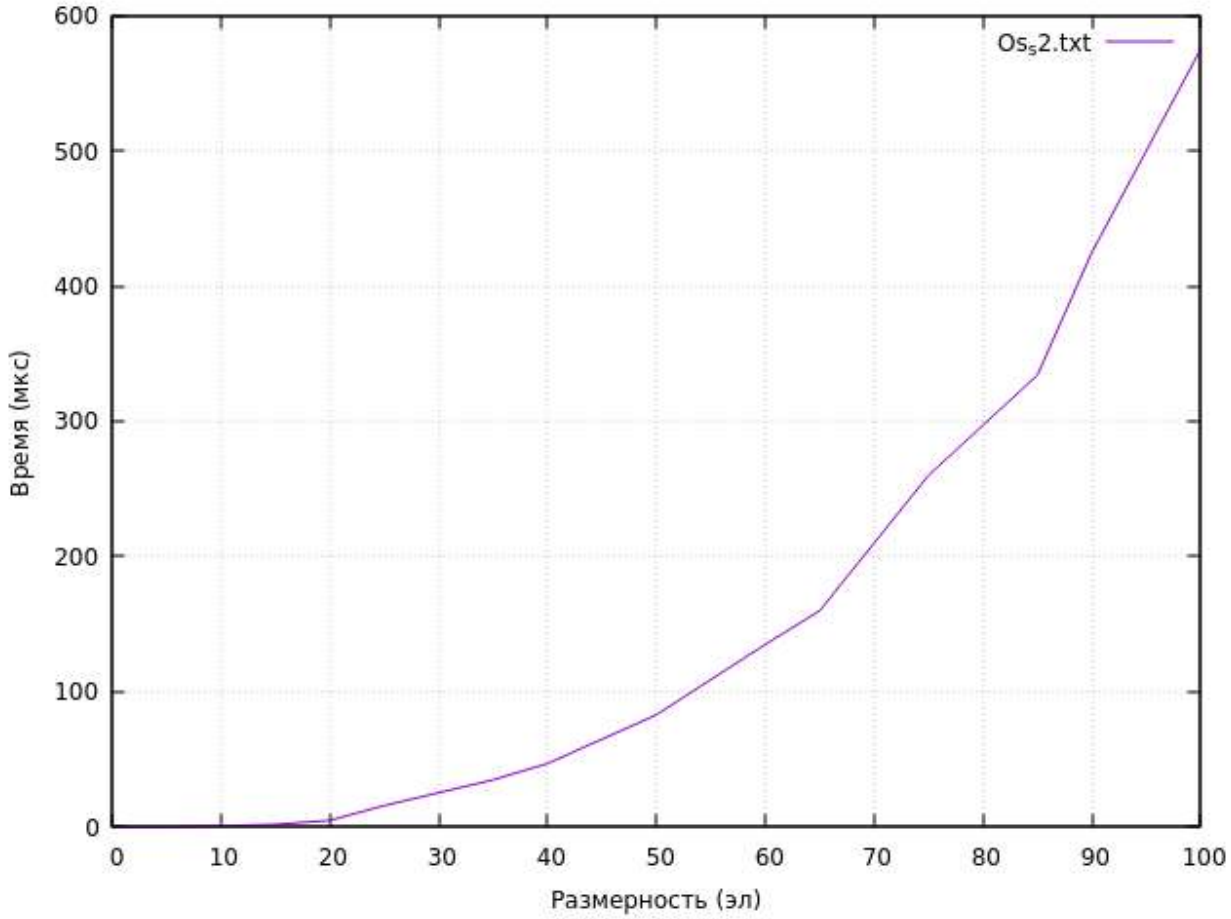

O0s1.txt
Время (мкс)
Размерность (эл)
0
200
400
600
800
1000
1200
1400
0
10
20
30
40
50
60
70
80
90
100


O0s2.txt
Время (мкс)
Размерность (эл)
0
500
1000
1500
2000
2500
3000
0
10
20
30
40
50
60
70
80
90
100


O3_s1.txt
Время (мкс)
Размерность (эл)
0
10
20
30
40
50
60
70
80
90
100


O3_s2.txt
Время (мкс)
Размерность (эл)
0
20
40
60
80
100
120
140
0
10
20
30
40
50
60
70
80
90
100


Os_s1.txt
Время (мкс)
Размерность (эл)
0
50
100
150
200
250
300
0
10
20
30
40
50
60
70
80
90
100



Как видно из графиков время сортировки с кешированием суммы занимает во всех случаях оптимизации меньше времени, чем без кеширования, так как в первой реализации на каждой итерации внутреннего цикла функция суммы вызывается только один раз, а не два, как во второй реализации. И чем больше уровень оптимизации, тем менее ощутима разница во времени. Так при уровне оптимизации `O0` разница во времени находится в пределах 1000мкс, для `Os`, который использует параметры `O2`, разница во времени находится уже в меньших пределах, а именно: 300мкс! А для `O3`, несмотря на выхлопы на графиках `O3_s1; O3_s2,` все равно разница минимальна и находится в пределах 50мкс.

Если же сравнивать графики по уровню оптимизации, то самым стабильным из всех будет уровень `O3`, то есть максимальный по оптимизации из данных. Так как эксперимент проводится на маленькой программе с маленьким

кэшем, то заметить увеличение производительности от оптимизации `Os` практически невозможно.

Таблица для `O3`:

Размер N	t_n^1	t_n^2	$\frac{\ln(t_{i+1}^1) - \ln(t_i^1)}{\ln(n_{i+1}) - \ln(n_i)}$	$\frac{\ln(t_{i+1}^2) - \ln(t_i^2)}{\ln(n_{i+1}) - \ln(n_i)}$
35	3,728048	2,734055	-18,5242	-2,42672
40	0,314215	1,977319	4,943143	1,328044
50	0,946819	2,659362	-2,91913	1,082933
60	0,556066	3,239854	20,69281	-9,56085
65	2,913749	1,507197	1,739412	0,120505
75	3,737257	1,533413	-3,08204	-0,67011
85	2,541093	1,410046	-14,4273	13,72541
90	1,113997	3,08992	0,817363	-5,80053
100	1,214184	1,676986	-	-

(проверить данные можно в таблице O3.xlsx)