Смирнов Иван ИУ7-22Б - 2023г.

Отчет

Задание №4

Исследование характеристик программного обеспечения

Целью работы является изучение расположения в памяти локальных переменных и представления структур.

Задание №1

По условию задания был проведен эксперимент замера среднего времени выполнения функции nanosleep для задержки в 1с, 100мс, 50мс, 10мс разными методами (gettimeofday, clock_gettime, clock, __rdtsc). Для этого было создано несколько программ с разными методами замера времени (см. папку /task_4/1):

Вариант 1 – **gettimeofday (main-1.c)**

```c
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

#define OK 0

struct timespec
{
   time_t tv_sec;
   long tv_nsec;
};

int nanosleep(const struct timespec *req, struct timespec *rem);

// Замерительный метод - gettimeofday
int main (void)
{
   struct timespec tw = {0,10*1e+6};
   struct timespec tr;

   struct timeval current_time;
   unsigned long long beg, end;

   gettimeofday(&current_time, NULL);

   beg = current_time.tv_sec * 1000ULL + current_time.tv_usec / 1000ULL;
   nanosleep(&tw, &tr);
   gettimeofday(&current_time, NULL);
   end = current_time.tv_sec * 1000ULL + current_time.tv_usec / 1000ULL;
```

```
   printf("%llu\n", end-beg);

   return OK;
}
```

Вариант 2 – **clock_gettime (main-2.c)**

```
#define _POSIX_C_SOURCE 199309L

#include <time.h>
#include <stdio.h>
#include <unistd.h>

#define OK 0

// Замерительный метод - clock_gettime
int main (void)
{
   struct timespec tw = {0,10*1e+6};
   struct timespec tr;

   struct timespec start, ending;
   unsigned long long beg, end;

   clock_gettime( CLOCK_REALTIME, &start );

   beg = start.tv_sec * 1000ULL + start.tv_nsec / 1000ULL;
   nanosleep(&tw, &tr);
   clock_gettime( CLOCK_REALTIME, &ending );
   end = ending.tv_sec * 1000ULL + ending.tv_nsec / 1000ULL;

   printf("%llu\n", end-beg);

   return OK;
}
```

Вариант 3 – **clock (main-3.c)**

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

#define OK 0

struct timespec
{
   time_t tv_sec;
```

```
    long tv_nsec;
};

int nanosleep(const struct timespec *req, struct timespec *rem);

// Замерительный метод - clock()
int main (void)
{
    struct timespec tw = {0,10*1e+6};
    struct timespec tr;

    double time_spent = 0;

    clock_t begin = clock();
    nanosleep(&tw, &tr);
    clock_t end = clock();

    time_spent += (double)(end - begin) * 1000ULL / CLOCKS_PER_SEC;
    printf("%f\n", time_spent);

    return OK;
}
```

Вариант 4 – **__rdtsc (main-4.c)**

```
#include <time.h>
#include <stdio.h>
#include <unistd.h>
#include <x86gprintrin.h>

#define OK 0

struct timespec
{
    time_t tv_sec;
    long tv_nsec;
};

int nanosleep(const struct timespec *req, struct timespec *rem);

// Замерительный метод - __rdtsc()
int main (void)
{
    struct timespec tw = {0,10*1e+6};
    struct timespec tr;

    unsigned long long t1 = __rdtsc();
    nanosleep(&tw, &tr);
```

```
    unsigned long long t2 = __rdtsc();

    printf("%llu\n", (t2 - t1)/CLOCKS_PER_SEC);

    return OK;
}
```

Так же в рабочей папке присутствуют скрипты для удобства работы с измерениями.

build_release.sh ($key) – скрипт сборки программы с ключом (ключ указывает на номер собираемой программы, например, команда **./build_release.sh 3** соберет программу main-3.c).

```
#!/bin/bash

con="-"
if [ -z "$1" ]
then
    key=""
else
    key="${con}$1"
fi

gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -Wfloat-equal -Wfloat-conversion -
Wvla -c main"$key".c
gcc main"$key".o -o app.exe -lm
```

dataset.sh ($key) – скрипт, который выполняет 20 тестовых измерений времени у программы (номер программы определяется ключом). Результаты измерений помещаются в текстовые файлы вида **./dataset/t_$key_$i.txt**, где i – номер измерения (теста).

```
#!/bin/bash

if [ -z "$1" ]
then
    echo "please, enter programm key!"
else
    ./build_release.sh "$1"
    for (( i=1; i <= 20; i++ ))
    do
        echo "test-'$i'"
        rm -f ./dataset/t_"$1"_"$i".txt
        ./app.exe >> ./dataset/t_"$1"_"$i".txt
    done
    echo "dataset done!"
fi
```

get_avg.sh ($key) **–** скрипт, который на основе данных, полученных из **dataset.sh**, считает среднее арифметическое измерений и записывает его в файл **./avg/t_$key.txt**.

```
#!/bin/bash

if [ -f "./dataset/t_'$1'_1.txt" ]
then
    summ=0
    n=0
    for (( i=1; i <= 20; i++ ))
    do
        while read -r n; do
            summ=$(( summ + n ))
        done < "./dataset/t_'$1'_'$i'.txt"
    done
    avg=$(( summ / 20 ))
    rm -f ./avg/t_"$1".txt
    echo "$avg" >> ./avg/t_"$1".txt
else
    echo "please, enter programm key!"
fi
```

Вспомогательные скрипты **check_scripts.sh** (проверка shellcheck всех скриптов), **chmod.sh** (выдача права на изменение для всех скриптов) и **clean.sh** (очистка временных файлов, а также текстовых файлов из каталогов ./dataset и ./avg) были взяты из лабораторных работ курса “Программирование на языке Си”. Данные 3 скрипта также будут использованы при решении “Задания 2”.

Ниже приведена таблица измерений (время измеряется в мс):

Реальное время	gettimeofday	clock_gettime	clock	__rdtsc
1000	1000	14060	0.8	2501
100	102	7377	0.05	255
50	53	1035	0.07	126
10	12	13625	0.02	31

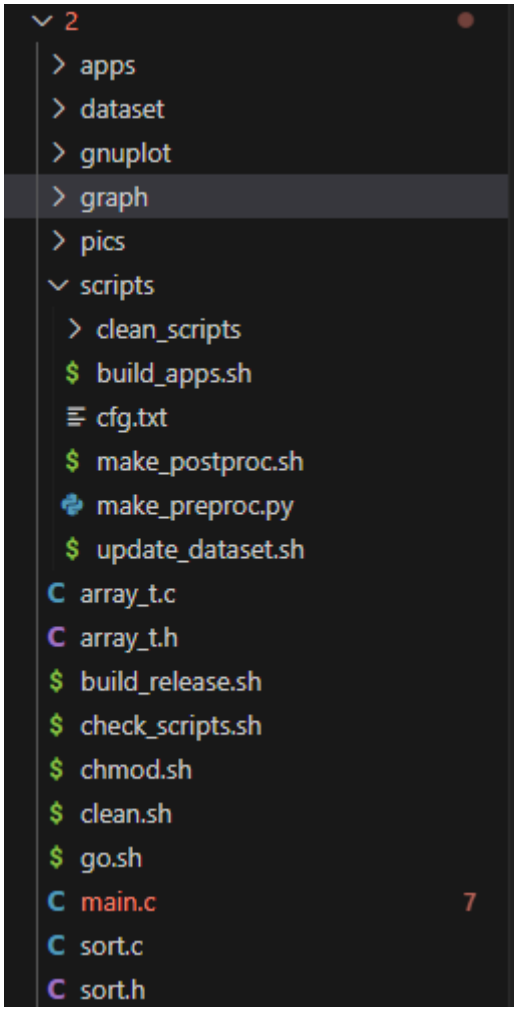
Как видно из таблицы наиболее точный метод – **gettimeofday.** Именно этот метод был использован при решении задания №2.

Метод __rdtsc имеет погрешность результата почти в 1.5 раза во всех рассматриваемых случаях. То есть присутствует закономерность, однако смещение в 1.5 раза - достаточно большое.

Методы clock_gettime и clock (из данных таблицы) являются самыми нестабильными. На разных машинах (Ubuntu 22/ Windows 11) функции выдают разные результаты. Clock() сообщает, сколько процессорного времени используется; процессорное время в свою очередь зависит от количества активных потоков (которое в разные состояния работы компьютера может быть разным). Из-за этого пользователь получает не тот результат, который хотел бы увидеть.

Задание №2

В данном задании необходимо проанализировать время выполнения трех разных реализаций сортировки выбором (selection sort) с разными типами оптимизации (O0 и O2) на случайно-отсортированном массиве, а также на отсортированном в прямом порядке.



(рис.1 – файловая структура задания №2)

Разные реализации сортировки основаны на разном обращении к элементам массива. Далее приведен код (на языке Си) каждой из реализаций.

1) Использование операции индексации a[i]

```c
// Реализация 1
// Функция находит минимум в диапазоне от strat_ind до end_ind
size_t find_min_elem1(const int a[], 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 (a[j] < a[min_elem])
        {
            min_elem = j;
        }
    }
    return min_elem;
}
// Функция сортировки выбором
void selection_sort1(int *a, size_t a_size)
{
    size_t min_elem = 0;
    int tmp_elem = 0;
    for (size_t i = 0; i < a_size; i++)
    {
        min_elem = find_min_elem1(a, i, a_size);
        tmp_elem = a[i];
        a[i] = a[min_elem];
        a[min_elem] = tmp_elem;
    }
}
```

2) Использование замены операции индексации на выражение *(a + i)

```c
// Реализация 2
size_t find_min_elem2(const int a[], 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 (*(a + j) < *(a + min_elem))
        {
            min_elem = j;
        }
    }
    return min_elem;
}

void selection_sort2(int *a, size_t a_size)
{
```

```
    size_t min_elem = 0;
    int tmp_elem = 0;
    for (size_t i = 0; i < a_size; i++)
    {
        min_elem = find_min_elem2(a, i, a_size);
        tmp_elem = *(a + i);
        *(a + i) = *(a + min_elem);
        *(a + min_elem) = tmp_elem;
    }
}
```

3) Использование указателей для работы с массивом

```
// Реализация 3
void selection_sort3(int *pb, int *pe)
{
    int min_elem = 0;
    int tmp_elem = *pb;
    for (int i = 0; i < (pe - pb); i++)
    {
        min_elem = i;
        for (int j = i; j < (pe - pb); j++)
        {
            //printf("l:%dr:%d\n", *(pb + j), min_elem);
            if (*(pb + j) < *(pb + min_elem))
            {
                min_elem = j;
            }
        }
        tmp_elem = *(*(&(pb)) + i);
        *(pb + i) = *(pb + min_elem);
        *(pb + min_elem) = tmp_elem;
    }
}
```

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

Также были реализованы функции генерации списка случайно-отсортированных чисел (*init*), а также уже отсортированного (*init sorted*). Описание функций находится в заголовочном файле *array_t.h*; сами функции в *array_t.c*.

```
void init(int *a, int size)
{
    srand(time(NULL));
    for (int i = 0; i < size; i++)
        a[i] = (rand() % (size * 3));
}
```

```
void init_sorted(int *a, int size)
{
    for (int i = 0; i < size; i++)
        a[i] = i;
}
```

Листинг основной программы (***main.c***):

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

#include <time.h>
#include <sys/time.h>

#include "array_t.h"
#include "sort.h"

#define OK 0
#define ERR_IO 1

#ifndef NMAX
#error NMAX IS NOT DEFINED
#endif
#ifndef SORT_TYPE
#error TYPE IS NOT DEFINED
#endif
#ifndef ARR_TYPE
#error ARR_TYPE IS NOT DEFINED
#endif

typedef int array_t[NMAX];

// Структуру переопределяю, так как для стандарта c99 это сделать необходимо
(иначе возникнет ошибка компиляции)
struct timespec
{
   time_t tv_sec;
   long tv_nsec;
};

int nanosleep(const struct timespec *req, struct timespec *rem);

// Замерительный метод - gettimeofday
int main(int argc, char **argv)
{
    if (argc != 2)
        return ERR_IO;
    array_t a;
```

```
    int sort_type = SORT_TYPE;
    int size = atoi(argv[1]);
    int array_type = ARR_TYPE;

    switch (array_type)
    {
        case 1:
            init(a, size);
            break;
        case 2:
            init_sorted(a, size);
            break;
    }

    struct timeval current_time;
    unsigned long long beg, end;

    // Действия
    switch(sort_type)
    {
        case 1:
            gettimeofday(&current_time, NULL);
            beg = current_time.tv_sec * 1000ULL + current_time.tv_usec / 1000ULL;
            selection_sort1(a, size);
            gettimeofday(&current_time, NULL);
            end = current_time.tv_sec * 1000ULL + current_time.tv_usec / 1000ULL;
            break;
        case 2:
            gettimeofday(&current_time, NULL);
            beg = current_time.tv_sec * 1000ULL + current_time.tv_usec / 1000ULL;
            selection_sort2(a, size);
            gettimeofday(&current_time, NULL);
            end = current_time.tv_sec * 1000ULL + current_time.tv_usec / 1000ULL;
            break;
        case 3:
            gettimeofday(&current_time, NULL);
            beg = current_time.tv_sec * 1000ULL + current_time.tv_usec / 1000ULL;
            selection_sort3(a, (a + size));
            gettimeofday(&current_time, NULL);
            end = current_time.tv_sec * 1000ULL + current_time.tv_usec / 1000ULL;
            break;
        default:
            return ERR_IO;
    }

    printf("%llu\n", (end - beg));

    return OK;
}
```

Программу необходимо запустить с одним аргументом, который отвечает за размер массива (в программе это переменная size). Такие переменные как максимальный размер статического массива (NMAX), тип сортировки (SORT_TYPE), тип массива (ARR_TYPE) определяются на этапе сборки. Программа выводит на экран время работы выбранной сортировки в миллисекундах (0.001 с.).

Чтобы проанализировать время для разного размера массива, 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 ***build_apps.sh***, который собирает все возможные версии программы (с двумя типами компиляции, двумя типами списков и тремя типами сортировок, то есть собирается в сумме 2*2*3=12 различных файлов).

build_apps.sh

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

cfg.txt:

```
O0 O2

20

1 2 3

10000

1 2

1 10 50 100 250 500 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000
```

В первой строке написаны типы оптимизации (O0 O2). Во второй строке написано количество проводимых измерений (тестов), то есть сколько раз запустится каждый из 12 исполняемых файлов. В третьей строке расположены разные реализации сортировки (описаны ранее). В четвертой строке написано максимальное количество элементов в массиве (по условию задачи оно рано 10000). В пятой строке написаны все возможные типы

массива (1 – случайный; 2 - отсортированный). В шестой строке написаны все тестируемые размеры массива.
Далее все 12 исполняемых файлов собираются с помощью скрипта ***build_release.sh***

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 размерах массива каждый из исполняемых файлов. Результаты (время в мс) помещаются в текстовые файлы в папку dataset. При повторном запуске скрипта новые измерения так же поместятся в те же самые текстовые файлы из dataset.

(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 выглядят примерно так:

104 84 50 64 94 100 122 8000

104 84 50 64 94 100 122 8000

108 84 50 64 94 100 122 8000

122 84 50 64 94 100 122 8000

71 98 71 82 99 111 140 9000

74 98 71 82 99 111 140 9000

75 98 71 82 99 111 140 9000

78 98 71 82 99 111 140 9000

79 98 71 82 99 111 140 9000

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 количества элементов в массиве.

Так же в отдельном файле (extra backup) для каждого случая массива помещена табличная информация (см. таблицы далее).

(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).

Главные графики:

(Обычный кусочно-линейный график зависимости времени выполнения в любых единицах измерения времени от числа элементов массива для всех 6 вариантов программы, обрабатывающей наилучший случай.)

(Обычный кусочно-линейный график зависимости времени выполнения в любых единицах измерения времени от числа элементов массива для всех 6 вариантов программы, обрабатывающей массив общего вида.)

(Кусочно-линейный график (6 штук) с ошибкой (среднее, максимум, минимум) для всех вариантов обработки массива при уровне оптимизации O2.)

(График с усами (среднее, максимум, минимум; нижний, средний и верхний квартили) для варианта обработки «через квадратные скобки» при уровне оптимизации O2.)

Графики расположены в папке **pics (**из-за ограничения в 1мб в мудле не могу прикрепить графики в векторном формате**)**

Как видно из графиков (1) и (2) время сортировки выбором для заранее отсортированного массива и для случайного примерно одинаково, так как сам алгоритм подразумевает нахождение следующего минимума после предыдущего (то есть в отсортированном массиве алгоритм все равно будет

искать минимум, так как он изначально не знает (и не может проверить!), что минимум стоит в начале списка).

Из графиков (3) и (4) видно, что чем больше элементов в массиве, тем больше значение ошибки (а значит более нестабильный замер времени, по крайней мере в миллисекундах).

Таблицы (для 1000 измерений):

O0_s1_a1:

Размер массива (n)	Относительная стандартная ошибка среднего (T)
1000	6.356918
2000	1.867374
3000	1.299581
4000	0.996917
5000	0.743025
6000	0.635772
7000	0.440397
8000	0.388772
9000	0.361069
10000	-

O0_s1_a2:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)
1000	4.205673
2000	2.247989
3000	1.505566
4000	1.061014
5000	0.890633
6000	0.798520
7000	0.544568
8000	0.521791
9000	0.441052
10000	-

O0_s2_a1:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)
1000	4.650982
2000	2.432842

3000	1.260343
4000	1.100790
5000	0.876972
6000	0.912548
7000	0.736206
8000	0.533468
9000	0.654083
10000	-

O0_s2_a2:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)
1000	5.102552
2000	1.620811
3000	1.645172
4000	0.956129
5000	0.711474
6000	0.609977
7000	0.564788
8000	0.398500
9000	0.393613
10000	-

O0_s3_a1:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)
1000	3.510317
2000	2.690095
3000	1.361306
4000	1.056013
5000	0.865636
6000	0.869591
7000	0.691187
8000	0.574875
9000	0.453260
10000	-

O0_s3_a2:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)
1000	3.905798
2000	1.606263
3000	1.529228
4000	1.076646

5000	0.954835
6000	0.710043
7000	0.768957
8000	0.483630
9000	0.417889
10000	-

2)

O2_s1_a1:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)	$\frac{ln\ t_{i+1} - ln\ t_i}{ln\ n_{i+1} - ln\ n_i}$
1000	3.541545	-1,208432493
2000	1.532563	-0,427672047
3000	1.288573	-1,362128606
4000	0.870816	-1,244803275
5000	0.659618	-0,178514464
6000	0.638495	-1,856794022
7000	0.479569	1,043341049
8000	0.55126	-4,137255408
9000	0.33863	-1,208432493
10000	-	-

O2_s1_a2:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)	$\frac{ln\ t_{i+1} - ln\ t_i}{ln\ n_{i+1} - ln\ n_i}$
1000	2.910063	-0,458064463
2000	2.118416	-1,285551901
3000	1.257873	-0,790553915
4000	1.001996	-2,732344496
5000	0.544596	1,909139293
6000	0.771334	-0,88550356
7000	0.672916	-4,180255749
8000	0.385070	-2,593010326
9000	0.283727	-0,458064463
10000	-	-

O2_s2_a1:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)	$\frac{ln\ t_{i+1} - ln\ t_i}{ln\ n_{i+1} - ln\ n_i}$
1000	2.428147	-0,277980759
2000	2.002601	-0,852556451
3000	1.417316	-1,565748666
4000	0.903325	-1,609021703
5000	0.630832	-0,984115433
6000	0.527218	-0,913403116
7000	0.457974	-2,617738442
8000	0.322874	2,060114133
9000	0.411541	-0,277980759
10000	-	-

O2_s2_a2:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)	$\frac{ln\ t_{i+1} - ln\ t_i}{ln\ n_{i+1} - ln\ n_i}$
1000	2.051782	-0,616901775
2000	1.337905	-1,150802348
3000	0.839033	0,028185809
4000	0.845864	-1,650433545
5000	0.585271	-0,826985669
6000	0.503356	-2,542827553
7000	0.340127	1,941470246
8000	0.440789	13,05703874
9000	2.051782	-0,616901775
10000	-	-

O2_s3_a1:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)	$\frac{ln\ t_{i+1} - ln\ t_i}{ln\ n_{i+1} - ln\ n_i}$
1000	2.458871	-0,854731017
2000	1.359678	-0,488287936
3000	1.115457	-0,920868393
4000	0.855856	-0,787576435
5000	0.717921	-0,318873176
6000	0.677373	-0,023928962
7000	0.674879	-0,523543064
8000	0.629310	-1,388049767
9000	0.534395	-0,854731017
10000	-	-

O2_s3_a2:

Размер массива (n)	Относительная стандартная ошибка среднего (T, мс)	$\frac{ln\ t_{i+1} - ln\ t_i}{ln\ n_{i+1} - ln\ n_i}$
1000	4.010746	-0,661792094
2000	2.535163	-1,282694618
3000	1.507074	-1,587505249
4000	0.954539	-0,153049306
5000	0.922490	-0,501037665
6000	0.841955	-2,067289462
7000	0.612196	-2,587232649
8000	0.433363	1,728430802
9000	0.531209	-0,661792094
10000	-	-

Расчет формулы из 3-его столбца производился при помощи excel.

Ответы на вопросы:
1) Быстрее всех работает __rdtsc(), так как он сразу считает тики процессора и в отличии от clock() – без задержки.

2) Если в ходе эксперимента в датасете обнаружены несколько одинаковых измерений, то их нельзя заменить на одно, потому что в следствии нельзя определить, какое реальное значение должно принимать время. По распределению Гаусса, чем больше проведено измерений, тем больше вероятность того, что измеренное значение ближе к реальному.

3) В ходе эксперимента замеряется только время целевого алгоритма, то есть сортировки, так как инициализация массива так же занимает время у процессора. На разных платформах также разные операции могут занимать разное время.