

Национальный исследовательский ядерный университет «МИФИ»

Институт интеллектуальных кибернетических систем

Кафедра №12 «Компьютерные системы и технологии»



ОТЧЕТ

О выполнении лабораторной работы №4

«Обработка одномерных массивов»

Студент: Кафанов С.П.

Группа: Б21-515

Преподаватель: Храпов А.С.

Москва — 2021

Задача. Вариант №5

Необходимо спроектировать и реализовать на языке C программу, осуществляющую по запросам пользователя ввод, обработку и вывод последовательности данных, которая представляется в виде массива.

Структура «Автомобиль»:

- марка (строка длиной до 16 символов, которая может включать в себя только буквы, дефис и пробелы);
- ФИО владельца (строка произвольной длины);
- пробег (дробное число, соответствующее величине пробега в тыс. км).

Сортировки

1. Гномья сортировка (Gnome sort).
2. Сортировка вставками с бинарным поиском (Insertion sort with binary search).
3. Поразрядная сортировка (Radix sort).

Использованные типы данных.

Int – для работы с простыми целочисленными данными

Long double – для поля пробег

Size_of – для сортировок: арифметика указателя воид, хранение неопределённо большого числа

Char – для строк, чисел небольшого диапазона и работы с байтами памяти

*** - для работы с динамической памятью и передачи переменных в функцию, просто раоты с указателями

*Void ** - для написания универсальных сортировок

Код.

Сортировки:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "CarStruct.c"
#include "Comporators.h"

void new_swap(void * a, void * b, size_t size) {
    void * buf = malloc(size);
```

```

memcpy(buf, a, size);
memcpy(a, b, size);
memcpy(b, buf, size);
free(buf);
}

void gnome_sort(void * first, size_t number, size_t size, int (*comparator) (const void*, const void*)) {
    void * cars = first;

    for (size_t i = 1; i < number; ++i) {
        for (size_t j = i; j; --j) {
            void * p1 = (void*) ((size_t) cars + size*(j - 1));
            void * p2 = (void*) ((size_t) cars + size*j);
            if (comparator(p1, p2) >= 0) {
                new_swap(p1, p2, size);
            }
        }
    }
}

size_t binarySearch(void * cars, void * el, size_t number, size_t size, int (*comparator) (const void*, const void*)) {
    size_t left = 0, right = number;
    while (left != right) {
        size_t m = (left+right)/2;
        void * p = (void*) ((size_t) cars + size*m);
        if (comparator(el, p) > 0) left = m + 1;
        else if (comparator(el, p) < 0) right = m;
        else return m;
    }
    return left;
}

// Функция для сортировки массива a [] размера 'n'
void insertionSort(void * cars, size_t number, size_t size, int (*comparator) (const void*, const void*)) {
    size_t i, loc;
    for (i = 1; i < number; ++i) {
        void * el = (void*) ((size_t) cars + i*size);
        void * buf = malloc(size);
        memcpy(buf, el, size);
        // найти место, где выбрано, может быть
        loc = binarySearch(cars, el, i, size, comparator);

        // Переместить все элементы после расположения, чтобы создать пространство
        void* donor = (void*) ((size_t) cars + (loc)*size); //OK
        void* recipient = (void*) ((size_t) cars + (loc+1)*size); //OK
        size_t len = i-loc; //OK
        memmove(recipient, donor, len*size);
        memmove(donor, buf, size);
        free(buf);
    }
}

//должна возвращать указатель на индексированный байт с конца и размер элемента, по которому идёт
// если индекс -1 то указатель на начало
void* el(void * first, size_t index, size_t * size_ell, char *** el_ptr, size_t max_len, char field) {
    void * ptr = NULL;
    struct Car* car = (struct Car*)first;
    switch (field) {
        case 0:
            *size_ell = 16;
            if (index == -1)
                ptr = (void*) (car->mark);
            else
                ptr = (void*) ((size_t) (car->mark) + (*size_ell - sizeof(char)*(index+1)));
            break;
        case 2:
            *size_ell = 10; //sizeof(long double)-6;
            if (index == -1)
                ptr = (void*) &(car->mileage);
            else
                ptr = (void*) ((size_t) &(car->mileage) + (sizeof(char)*(index)));
            break;
        case 1: {
            char ** p = &(car->name);
            *el_ptr = p;

            if (index == -1)
                ptr = (void*) (car->name);
            else
                ptr = (void*) ((size_t) (car->name) + sizeof(char)*(max_len - (index+1)));
        }
    }
}

```

```

        *size_ell = strlen(car->name)+1;
        break;
    }
    default:
        break;
}
return ptr;
}

void radix_sort_uni(void * cars, size_t number, size_t size, char field) {
    int sc = sizeof(char);
    char ** buf = NULL;
    //находим максимальную длину той памяти, которую будем сравнивать
    size_t max_len = 0;
    for (size_t index = 0; index < number; ++index) {
        size_t len = 0;
        void* car = (void*) ((size_t)cars + index*size);
        void * p = el(car, -1, &len, &buf, 0, field);
        if (max_len < len) max_len = len;
    }

    // унификация длинн
    for (size_t i = 0; i < number; ++i) {
        size_t len = 0;
        void * car = (void*) ((size_t)cars + i*size);
        char **p = NULL;
        void * byte = el(car, -1, &len, &p, max_len, field);
        if (len < max_len) {

            // нужен указатель на указатель на имя
            // чтобы положить туда указатель на новую память
            *p = realloc(*p, (1 + max_len) * sizeof(char));

            int l = strlen(*p);
            for (int k = 1; k <= max_len; ++k) (*p)[k] = '\0';
        }
    }

    // основной цикл:
    // - рассортировать по значению текущего байта
    // - поменять массив
    void * copy_arr = malloc(size*number);
    for (size_t index = 0; index < max_len; ++index) {
        //корзинки для массивов указателей на указатели на элементы: ->|=>
        void *** box = malloc(256*sizeof(void**));
        for (size_t i = 0; i < 256; ++i) {
            box[i] = malloc(sizeof(void**));
            box[i][0] = NULL;
            /**((void**)((size_t)box+i*sizeof(void**))) = malloc(sizeof(void**));
            /**((void**)((size_t)box+i*sizeof(void**))) = NULL;
        }
        memcpy(copy_arr, cars, size*number);

        // разбрасываем указатели на элементы по корзинам
        for (size_t i = 0; i < number; ++i) {
            size_t len = 0;
            void * car = (void*) ((size_t)copy_arr + i*size);
            void * byte = el(car, -1, &len, &buf, max_len, field);

            byte = el(car, index, &len, &buf, max_len, field);
            unsigned char n = *((unsigned char*)byte);

            // смотрим сколько уже есть указателей
            // и добавляем к ним ещё один
            size_t k = 0;
            while (box[n][k] != NULL) k++;
            box[n] = realloc(box[n], sizeof(void *) * (k + 2));
            box[n][k + 1] = NULL;
            box[n][k] = car;
        }

        int a = 0;
        // переставляем индексы в массиве в соответствии с корзинами
        for (size_t i = 0; i < 256; ++i) {
            int k = 0;
            while (box[i][k] != NULL) { // вероятно ошибка при выделении памяти
                memcpy((void*) ((size_t)cars+a*size), box[i][k], size); // ошибка индекса i
                k++;
                a++;
            }
        }
    }
}

```

```

        for (size_t i = 0; i < 256; ++i) free(box[i]);
        free(box);
    }
    free(copy_arr);
    //освобождаем нулевую память
    for (size_t i = 0; i < number; ++i) {
        size_t len = 0;
        void * car = (void*) ((size_t)cars + i*size);
        void * byte = el(car, -1, &len, &buf, max_len, field);

        if (len < max_len) {
            char **p = NULL;
            void *buf1 = el(car, -1, &len, &p, 0, field);

            // нужен указатель на указатель на имя
            // чтобы положить туда указатель на новую память
            *p = realloc(*p, (1 + strlen((char*)buf1)) * sizeof(char));
        }
    }
}

void sort(struct Car ** all_cars, int number_of_cars, char index_of_sort) {
    printf("With which characteristic do you wanna sort array?\n");
    printf("1) Mark\n");
    printf("2) Name\n");
    printf("3) Mileage\n");
    int chose = -1;
    while (scanf("%d", &chose) <= 0) { getchar(); printf("Error!\n"); } getchar();
    if (chose > 3 || chose < 1) {
        printf("Wrong number. Try again.\n");
        sleep(3);
        return;
    }
    chose--;
    int (*comp[3])(const void*, const void*) = {comp_mark, comp_name, comp_mileage};

    switch(index_of_sort) {
        case 0: gnome_sort(*all_cars, number_of_cars, sizeof(struct Car), comp[chose]); break;
        case 1: insertionSort(*all_cars, number_of_cars, sizeof(struct Car), comp[chose]); break;
        case 2: radix_sort_uni(*all_cars, number_of_cars, sizeof(struct Car), (char) chose); break;
        default: return;
    }
}

```

Меню:

```

#include <stdio.h>
#include <stdlib.h>
#include "MenuClass.c"
#include <string.h>
#include <unistd.h>

/*
struct Folder{
    char * name;
    struct Folder ** folders;
    struct Folder * previous;
    char ** functions;
};

struct Menu{
    struct Folder * root;
    struct Folder ** all_folders;
    int nof;
};*/

struct Menu init_menu() {
    struct Menu menu;
    menu.root = (struct Folder*) malloc(sizeof(struct Folder));

    menu.root->folders = (struct Folder**) malloc(sizeof(struct Folder));
    menu.root->folders[0] = NULL;

    menu.root->previous = NULL;
    //menu.name->name = (char*) malloc(sizeof(char)*5);

    menu.root->name = (char*) malloc(sizeof(char) * strlen("root")+1);
    memcpy(menu.root->name, "root", strlen("root")+1); //"root";

    menu.root->functions = (char**) malloc(sizeof(char*)*2);
    //menu.root->functions[0] = (char*) malloc(sizeof(char)*5);
    menu.root->functions[0] = "EXIT";
}

```

```

    menu.root->functions[1] = NULL;

    menu.all_folders = (struct Folder**) malloc(sizeof(struct Folder*));
    menu.all_folders[0] = NULL;

    menu.nof = 0;

    return menu;
}

int number_of_folders(struct Folder folder) {
    int i = 0;
    while (folder.folders[i]) i++;
    return i;
}

int number_of_functions(struct Folder folder) {
    int i = 0;
    while (folder.functions[i]) i++;
    return i;
}

void add_folder(char * name, char * host_name, struct Menu * menu) {
    struct Folder * f = menu->root;
    if (menu->nof == 0){
        f->folders[0] = (struct Folder*) malloc(sizeof(struct Folder));

        f->folders[0]->previous = f;

        f = f->folders[0];

        f->name = (char*) malloc(sizeof(char) * strlen(name)+1);
        memcpy(f->name, name, strlen(name)+1);

        f->folders = (struct Folder**) malloc(sizeof(struct Folder*));
        f->folders[0] = NULL;

        f->functions = (char**) malloc(sizeof(char*));
        f->functions[0] = NULL;

        int nof = menu->nof;
        menu->all_folders = (struct Folder **) realloc(menu->all_folders, sizeof(struct Folder*) * (nof + 2));
        menu->all_folders[nof] = f;
        menu->all_folders[nof+1] = NULL;
        menu->nof++;
    }
    else {
        int nof = menu->nof;
        for (int i = 0; i < nof; ++i) {
            if (!strcmp(menu->all_folders[i]->name, host_name)) {
                f = menu->all_folders[i];
                int nof1 = number_of_folders(*f);

                f->folders = (struct Folder**) realloc(f->folders, (nof1+2) * sizeof(struct Folder*));

                f->folders[nof1+1] = NULL;
                f->folders[nof1] = (struct Folder*) malloc(sizeof(struct Folder));
                f->folders[nof1]->previous = f;

                f = f->folders[nof1];

                f->name = (char*) malloc(sizeof(char) * strlen(name)+1);
                memcpy(f->name, name, strlen(name)+1);

                f->folders = (struct Folder**) malloc(sizeof(struct Folder*));
                f->folders[0] = NULL;

                f->functions = (char**) malloc(sizeof(char*));
                f->functions[0] = NULL;

                menu->all_folders = (struct Folder **) realloc(menu->all_folders, sizeof(struct Folder*) * (nof +
2));

                menu->all_folders[nof] = f;
                menu->all_folders[nof+1] = NULL;
                menu->nof++;
                break;
            }
        }
        if (!strcmp(host_name, menu->root->name)) {
            f = menu->root;

```

```

        f->folders = (struct Folder**) realloc(f->folders, (nof+2) * sizeof(struct Folder));

        f->folders[nof+1] = NULL;
        f->folders[nof] = (struct Folder*) malloc(sizeof(struct Folder));
        f->folders[nof]->previous = f;

        f = f->folders[nof];

        f->name = (char*) malloc(sizeof(char) * strlen(name)+1);
        memcpy(f->name, name, strlen(name)+1);

        f->folders = (struct Folder**) malloc(sizeof(struct Folder));
        f->folders[0] = NULL;

        f->functions = (char**) malloc(sizeof(char*));
        f->functions[0] = NULL;

        menu->all_folders = (struct Folder **) realloc(menu->all_folders, sizeof(struct Folder*) * (nof +
2));

        menu->all_folders[nof] = f;
        menu->all_folders[nof+1] = NULL;
        menu->nof++;
    }
}

void add_function(char * name, char * host_name, struct Menu menu) {
    int nof = menu.nof;
    for (int i = 0; i < nof; ++i) {

        if (!strcmp(menu.all_folders[i]->name, host_name)) {
            struct Folder * f = menu.all_folders[i];
            int nof1 = number_of_functions(*f);
            f->functions = (char**) realloc(f->functions, sizeof(char*) * (nof1+2));
            f->functions[nof1+1] = NULL;

            f->functions[nof1] = (char*) malloc(sizeof(char) * (strlen(name) + 1));
            memcpy(f->functions[nof1], name, strlen(name));
            f->functions[nof1][strlen(name)] = '\0';

            break;
        }
    }
}

void delete_all_folders(struct Menu * menu) {
    for (int i = 0; i < menu->nof; ++i) {
        struct Folder * f = menu->all_folders[i];
        free(f->folders);
        free(f->name);

        for (int i = 0, l = number_of_functions(*f); i < l; ++i) {
            free(f->functions[i]);
        }
        free(f->functions);
        free(f);
    }
}

void clear_root(struct Menu * menu) {
    free(menu->root->name);
    free(menu->root->folders);
    //я делал указатель на статическую память
    //free(* (menu->root->functions));
    free(menu->root->functions);
    free(menu->root);
    free(menu->all_folders);
}

struct Menu create_menu() {
    struct Menu menu = init_menu();
    char enter_data[] = "Entering of Data.";
    char get_out_data[] = "Getting out of Data.";
    char work_with_data[] = "Work with Data.";
    char data_time[] = "Timing.";
    char root[] = "root";
    char sort[] = "Sort all cars.";

    char read_from_terminal[] = "Read data from the terminal window.";
    char read_from_file[] = "Read data from a file.";
    char random_generation[] = "Generate random data.";

```

```

char output_to_terminal[] = "Print cars to terminal.";
char save_data_to_file[] = "Save data to file.";
char insert[] = "Insert a new car.";
char insert_in_sorted_array[] = "Insert a new car in the sorted array.";
char erase[] = "Delete several elements.";
char timing[] = "Timing.";
char gnome_sort[] = "Gnome Sort.";
char insert_binary_sort[] = "Insert Sort.";
char radix_sort[] = "Radix Sort.";

add_folder(enter_data, root, &menu);
add_folder(get_out_data, root, &menu);
add_folder(work_with_data, root, &menu);
add_folder(data_time, root, &menu);
add_folder(sort, root, &menu);

add_function(read_from_terminal, enter_data, menu);
add_function(read_from_file, enter_data, menu);
add_function(random_generation, enter_data, menu);

add_function(output_to_terminal, get_out_data, menu);
add_function(save_data_to_file, get_out_data, menu);

add_function(insert, work_with_data, menu);
add_function(insert_in_sorted_array, work_with_data, menu);
add_function(erase, work_with_data, menu);

add_function(gnome_sort, sort, menu);
add_function(insert_binary_sort, sort, menu);
add_function(radix_sort, sort, menu);

add_function(timing, data_time, menu);

return menu;
}

void print_menu(struct Folder * f) {
    for (int i = 0; i < strlen(f->name)+10; ++i) printf("="); printf("\n");
    printf("||   %s   ||\n", f->name);
    for (int i = 0; i < strlen(f->name)+10; ++i) printf("="); printf("\n");

    printf("0-> [< go back >]\n");

    int i = 0;
    while (f->folders[i++])
        printf("%d-> [%s]\n", i, f->folders[i-1]->name);

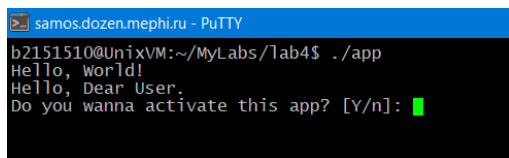
    int j = 0;
    while (f->functions[j++])
        printf("%d-> [< %s >]\n", i+j-1, f->functions[j-1]);
}

```

Тесты.

Отсортировать массив из 100 сгенерированных машин по именам владельцев, используя Gnome Sort.

Скриншоты.



```

samos.dozen.mephi.ru - PuTTY
b2151510@UnixVM:~/MyLabs/lab4$ ./app
Hello, world!
Hello, Dear User.
Do you wanna activate this app? [Y/n]: █

```


samos.dozen.mephi.ru - PuTTY

```
=====
||  root  ||
=====
0-> [< go back >]
1-> [Entering of Data.]
2-> [Getting out of Data.]
3-> [Work with Data.]
4-> [Timing.]
5-> [Sort all cars.]
6-> [< EXIT >]
```

samos.dozen.mephi.ru - PuTTY

```
=====
||  Entering of Data.  ||
=====
0-> [< go back >]
1-> [< Read data from the terminal window. >]
2-> [< Read data from a file. >]
3-> [< Generate random data. >]
3
Generate random data.
Input the number of new random cars.
Warning!!! All your previous cars will be deleted.
-> 100
```

samos.dozen.mephi.ru - PuTTY

```
=====
||  Sort all cars.  ||
=====
0-> [< go back >]
1-> [< Gnome Sort. >]
2-> [< Insert Sort. >]
3-> [< Radix Sort. >]
1
Gnome Sort.
With which characteristic do you wanna sort array?
1) Mark
2) Name
3) Mileage
2
```

```
samos.dozen.mephi.ru - PuTTY
The 89 car:
Mark: 89
Name: uPyYdhsCyhtzJfBmwetJfIhwtIRkDkgj
Mileage: 1149042.000000
The 90 car:
Mark: 90
Name: uKtHeTrmWvECBpYcYrDwytZhghepGpyvl
Mileage: 7830184.000000
The 91 car:
Mark: 91
Name: vGn1XUj2JkZQzWdLykk1jXGwbxcowya
Mileage: 6353670.000000
The 92 car:
Mark: 92
Name: vVondapJscZhucwefJwHRICTN1buUpwZPrafHmUayCwfZJw
Mileage: 1518398.000000
The 93 car:
Mark: 93
Name: vWVDeedSSAUA
Mileage: 195694.000000
The 94 car:
Mark: 94
Name: nGhWCDZyGdHgdxPcRATpVlpqyNdlqCkqptGgXtew
Mileage: 661166.000000
The 95 car:
Mark: 95
Name: xPmupqKsImQJ7ByEzTSeiuknDsqfIkkmwz
Mileage: 2493173.000000
The 96 car:
Mark: 96
Name: xWgTwtqLLwKZ
Mileage: 5607621.000000
The 97 car:
Mark: 97
Name: xUkhuZc1kgkUwU
Mileage: 8284011.000000
The 98 car:
Mark: 98
Name: xXbHmW/xDQsRTQmCOKu
Mileage: 6842406.000000
The 99 car:
Mark: 99
Name: ZYWhpTmWzJezZ1uzzBspvhznkFHEyYORhWwJFXjZTXKQJ
Mileage: 2551601.000000
Enter any key to continue.
```

samos.dozen.mephi.ru - PuTTY

```
Goodbye!
Enter something to finish your work.
```

Таймирование.

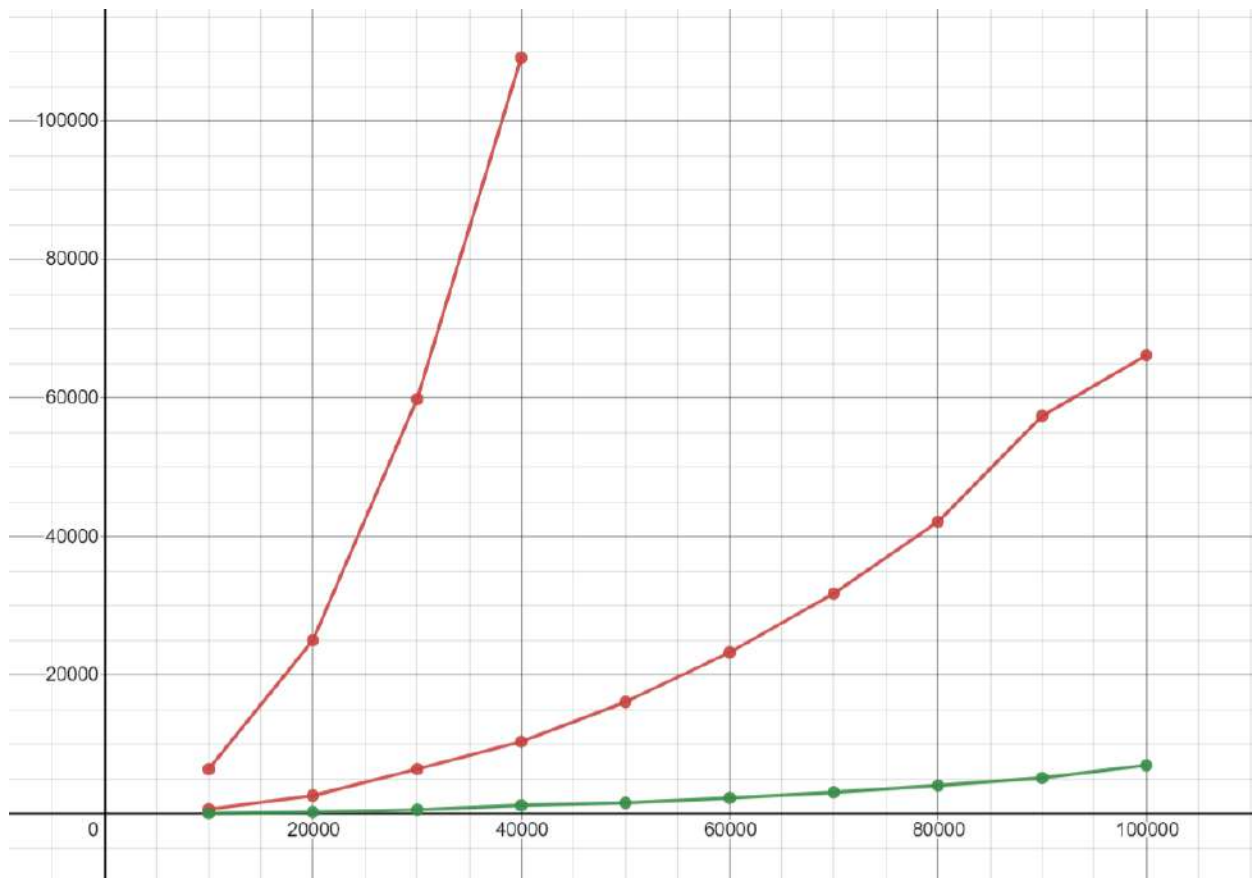
Таймирование производилось в «тиках» для пущей наглядности.

	Gnome Sort	Insert Sort with binary search	Radix sort (LSD)
Время на поле «пробег»			
10000 cars	6090	67	652
20000 cars	23306	256	2640
30000 cars	52737	578	6462
40000 cars	97561	1019	10415
50000 cars	150734	1596	16149
60000 cars	>150000	2337	23307
70000 cars		3504	31783
80000 cars		4272	42148
90000 cars		5697	57430
100000 cars		7212	66186

Асимптотики сортировок в соответствующем таблице порядке:

- $O(n^2)$
- $O(n \cdot \log(n))$
- $O(k \cdot n)$ – где k – это разрядность сортируемой памяти. В моём конкретном случае – количество байт, отводимых под запись каждого сортируемого элемента.

Графики.



Блок схема функций обработки.

Сравнительный анализ.

По результатам проведённого таймирования лучшие результаты показала сортировка с бинарными вставками.

При этом лучшая из представленных по асимптотике поразрядная сортировка оказалась лишь на 2-ом месте показав результаты примерно в 10 раз худшие.

Вероятнее всего, это произошло по следующим причинам:

- Количество данных относительно не велико.
- По своей задумке и идее Radix – специализированная сортировка, поскольку работает напрямую с байтами. Но из-за

условий лабораторной работы пришлось сделать её универсальной для любого набора данных, что привело к значительному падению как эффективности в целом, так и скорости в частности.

Выводы:

В ходе данной лабораторной работы:

- Были закреплены навыки работы с динамической памятью и указателями.
- Я научился работать с указателями типа `void*` и типом данных `size_t`.
- Научился азам работы с Makefile-ом.
- Написал универсальную структуру меню: «аля» каталог файлов с функциями в них, которую можно будет использовать в дальнейшем.
- Написаны 3 универсальные сортировки разной асимптотики.
- Получен опыт работы с массивами структур и структурами в принципе.