Московский государственный университет
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Отчет по заданию №1

«Методы сортировки»

Вариант 2 / 1 / 2 / 5

Выполнил:
студент 104 группы
Киперь Д. Ю.

Преподаватели:
Цыбров Е. Г.
Кулагин А. В.

Москва
2025

Содержание

Постановка задачи

Задание требовало от меня сделать следующее:

- написать генератор массивов длин 10, 100, 1000, 10000, элементами которых будут числа типа **long long int**;
- сортировать числа в массивах в порядке неубывания;
- написать два алгоритма сортировок: Сортировка простым выбором и Пирамидальная сортировка;
- сделать подсчет количества перестановок и сравнений в каждой из сортировок;
- провести эксперементальное сравнение двух алгоритмов сортировки;

Структура программы и определение функций

Моя программа разделена на два модуля:

- генератор массивов (с случайными элементами типа **long long int**) для сортировки
- сортировщик массивов

В первом модуле, как следует из названия, создаются и записываются в файлы массивы для сортировки. В другом модуле реализовано считывание числе из файлов и запись их в массив, с последующей сортировкой двумя типами сортировок: пирамидальной и простым выбором. Их результаты записываются в файлы в каталог **output/«name of sort»**.

Генерация массивов

Сначала объявим заголовочные файлы и константы.

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

#define INPUT_CATALOG "files/inputs/"
#define OUTPUT_CATALOG "files/outputs/"

#define SMALL_NUM 10
#define LITTLE_NUM 100
#define MIDDLE_NUM 1000
#define BIG_NUM 10000

#define MAX_BUF_SIZE 80

#define BORDER 100000
```

Процесс генерации массивов включает в себя создание массива заданной длины с элементами типа **long long int**.
Так как длина массива может доходить до 10000, то для удобства он будет сохранятся в отдельный файл.

```
FILE* file_open(char* filename, char* mode) {
    FILE* file = fopen(filename, mode);
    if (file == NULL) {
        if (strcmp("w", mode) == 0) {
            printf("Ошибка при открытии файла на запись!!\n");
        }
```

```
        if (strcmp("r", mode) == 0) {
            printf("Ошибка при открытии файла на чтение!!\n");
        }

        return NULL;
    }

    return file;
}
```

Данная функция написана в генераторе чтобы при ошибке создания файла, было выведено сообщение в консоли.

```
void print_list(long long int* arr, int n) {
    for (int i = 0; i < n; i++) {
        printf("%lld ", arr[i]);
    }
}
```

Эта вспомогательная функция выводит в консоль элементы массива (была написана для упрощенной отладки).

```
long long int get_rand_number() {
    return ((long long int)rand() << 32) | rand();
}
```

Эта функция генерирует случайное число, которое будет использовано для заполнения массива.

```
long long int* generate_array(int n) {
    long long int* arr = (long long int*) malloc(sizeof(long long int) * n);

    for (int i = 0; i < n; i++) {
        arr[i] = get_rand_number();

        while (arr[i] >= LLONG_MAX - BORDER || arr[i] <= LLONG_MIN + BORDER) {
            arr[i] = get_rand_number();
        }
    }

    return arr;
}

long long int* generate_sorted_array(int n) {
    long long int* arr = (long long int*) malloc(sizeof(long long int*) * n);

    arr[0] = get_rand_number();
```

```
    for (int i = 1; i < n; i++) {
        int ost = get_rand_number() % 10;

        while (ost == 0) {
            ost = get_rand_number() % 10;
        }

        arr[i] = arr[i - 1] + ost;
    }

    return arr;
}
```

```
long long int* generate_rotated_sorted_array(int n) {
    long long int* arr = (long long int*) malloc(sizeof(long long int*) * n);

    arr[0] = get_rand_number();
    for (int i = 1; i < n; i++) {
        int ost = get_rand_number() % 10;

        while (ost == 0) {
            ost = get_rand_number() % 10;
        }

        arr[i] = arr[i - 1] - ost;
    }

    return arr;
}
```

Эти три функции обеспечивают генерацию трех массивов (в несортированном, сортированном, сортированном в обратном порядке). Память под каждый массив выделяется динамически, и в массив помещаются случайные числа типа **long long int**.

```
void create_file(
    char* filename,
    char* filname_sorted,
    char* filename_rotated_sorted,
    int n
) {
    long long int* arr = generate_array(n);
    long long int* sorted_arr = generate_sorted_array(n);
    long long int* rotated_sorted_arr = generate_rotated_sorted_array(n);

    FILE* file = file_open(filename, "w");
    FILE* file_sorted = file_open(filname_sorted, "w");
    FILE* file_rotated_sorted = file_open(filename_rotated_sorted, "w");

    for (int i = 0; i < n; i++) {
        fprintf(file, "%lld\n", arr[i]);
    }

    for (int i = 0; i < n; i++) {
        fprintf(file_sorted, "%lld\n", sorted_arr[i]);
    }

    for (int i = 0; i < n; i++) {
        fprintf(file_rotated_sorted, "%lld\n", rotated_sorted_arr[i]);
    }
```

```
    fclose(file);
    fclose(file_sorted);
    fclose(file_rotated_sorted);

    free(arr);
    free(sorted_arr);
    free(rotated_sorted_arr);
}
```

Эта функция принимает имя файлов (несортированного, отсортированного и сортированного в обратном порядке) и размер каждого из массивов. После создаются файлы, и происходит запись данных из массивов в файлы. В самом конце чистим память.

```
void generate_data() {
    int num_params[] = { SMALL_NUM, LITTLE_NUM, MIDDLE_NUM, BIG_NUM };
    int len = sizeof(num_params) / sizeof(num_params[0]);

    for (int i = 0; i < len; i++) {
        char filename[MAX_BUF_SIZE];
        char filename_sorted[MAX_BUF_SIZE];
        char filename_rotated_sorted[MAX_BUF_SIZE];

        sprintf(
            filename,
            "%sunsorted%d.txt",
            INPUT_CATALOG,
            num_params[i]
        );
        sprintf(
            filename_sorted,
            "%ssorted%d.txt",
            INPUT_CATALOG,
            num_params[i]
        );
        sprintf(
            filename_rotated_sorted,
            "%srotated_sorted%d.txt",
            INPUT_CATALOG,
            num_params[i]
        );

        create_file(
            filename,
            filename_sorted,
            filename_rotated_sorted,
            num_params[i]
```

```
            );
        }
    }
```

Одна из основных функций в генераторе, ибо является корнем создания файлов. Она отвечает за создание имен файлов, и вызов функции, которая создает файл с таким именем и числом параметров.

```
int main(void) {
    srand(time(NULL));

    generate_data();

    return 0;
}
```

Эта функция является входной точкой всего генератора. В ней мы подключаем возможность генерировать случайные числа, а затем вызываем функцию, которая генерирует данные и записывает их в файлы.

Сортировки и вспомогательные функции

Моей задачей было реализовать сортировку простым выбором и пирамидальную сортировку. Выведем заголовочные файлы и константы, которые я использовал:

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

#define INPUT_CATALOG "files/inputs/"
#define OUTPUT_CATALOG "files/outputs/"

#define SMALL_NUM 10
#define LITTLE_NUM 100
#define MIDDLE_NUM 1000
#define BIG_NUM 10000

#define UNSORTED "unsorted"
#define SORTED "sorted"
#define SORTED_ROTATED "rotated_sorted"

#define MAX_BUF_SIZE 1000
#define MEAN_BUF_SIZE 200
```

Сортировка простым выбором

Суть сортировки простым выбором заключается в том, что сначала мы проходимся по всему массиву, находим в нем максимальный элемент и меняем его с элементом, стоящим в конце массива. Далее сдвигаемся с конца массива на 1 влево и проделываем то же самое, только максимальный эл-т ищется до $n-2$ индекса (включительно), а так же меняется местами с эл-ом с индексом $n-2$. Реализация такого алгоритма приложена ниже.

```
void selection_sort(long long int* arr, int n) {
    int j = n - 1;
    int max_i = 0;

    int compare = 0, transp = 0;

    for (int i = 0; i < n; i++) {
        for (int q = 0; q <= j; q++) {
            compare++;

            if (arr[q] > arr[max_i]) {
                max_i = q;
            }
        }
```

```
            swap(&arr[max_i], &arr[j]);
            transp++;

            j--;
            max_i = 0;
        }

        printf(
            "Selection sort, размер массива - %d\n\tКоличество сравнений - %d,
            количество перестаовок - %d\n",
            n,
            compare,
            transp
        );
    }
```

Пирамидальная сортировка

Структура данных (бинарная) пирамида представлет собой объект -массив, который можно рассматривать как почти полное бинарное дерево. Каждый узел этого дерева соответствует эл-ту массива. Дерево полностью на всех уровнях за исключением возможно наинизшего, который заполняется слева направо. Корнем дерева является A[1] (где 1 – порядковый номер в массиве а не индекс). Для заданного индекса (нумерация с 1) можно легко вычислить:

- его родителя **A[i / 2]**
- его левого ребенка **A[2 * i]**
- его правого ребенка **A[2 * i + 1]**

На большинстве компьютеров (особенно в наше время) операция $2i$ выполняется с помощью одной команды, побитового сдвига влево на 1 бит, аналогично операция $2i+1$ также выполняется очень быстро, операция получения родителя выполняется сдвигом i на бит вправо.
В варианте моей пирамидальной сортировки выполняется свойтство (свойство неубывающих пирамид)

$$A[Parent(i)] <= A[i]$$

Так как в задании было сказано, что отсортировать нужно в неубывающем порядке. Таким образом наименьший элемент пирамиды находится в ее корне.

```
void heapify(long long int* arr, int n, int i, int* compare, int* transp) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n) {
        (*compare)++;

        if (arr[left] > arr[largest]) {
            largest = left;
        }
    }

    if (right < n) {
        (*compare)++;

        if (arr[right] > arr[largest]) {
            largest = right;
        }
    }
```

```
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        (*transp)++;

        heapify(arr, n, largest, compare, transp);
    }
}
```

На каждом шаге определяется больший эл-нт из $arr[i]$, $arr[left]$, $arr[right]$ и его индекс записывается в переменную largest. Если индекс largest не равен индексу i, то происходит перестановка эл-ов, таким образов для узла i и его дочерних узлов выполняется св-во невозрастания пирамиды, но так как теперь $arr[i]$ оказывется на месте $arr[largest]$, то необходимо рекурсивно вызвать функцию heapify для узла $arr[largest]$.
Заметим, что размер каждого поддерева не превышает величину $2n/3$, причем наихудший случай осуществляется, когда последний уровень заполнен наполовину. Таким образом, время выполнения функции описывается рекурентным соотношением[1]:

$$T(n) = T(2n/3) + O(1)$$

А решение данного рекурентного соотношения имеет вид:

$$T(n) = O(\lg n)$$

Далее идет сама пирамидальная сортировка.

```
void heap_sort(long long int* arr, int n) {
    int compare = 0, transp = 0;

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i, &compare, &transp);
    }

    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);
        transp++;

        heapify(arr, i, 0, &compare, &transp);
    }

    printf(
        "Heap sort, размер массива - %d\n\tКоличество сравнений - %d,
            количество перестановок - %d\n",
        n,
        compare,
        transp
    );
}
```

Мы знаем, что каждый вызов heapify стоит нам $O(\lg n)$ и таких вызово будет n штук, т.е мы можем оценить время работы алгоритма как $O(n \lg n)$. Это верхняя граница, будучи корректной не является асимптотически точной.

Так же в файле sorts.c (файл где находятся сортировки) находятся следующие вспомогательные функции:

```
// взятие данных из массивов ввода
long long int* crate_array(int n) {
    long long int* res = (long long int*) malloc(sizeof(long long int) * n);

    return res;
}

// перестановка значений двух переменных
void swap(long long int* a, long long int* b) {
    long long int temp = *a;
    *a = *b;
    *b = temp;
}

// вывод массива
void print_list(long long int* arr, int n) {
    for (int i = 0; i < n; i++) {
        printf("%lld ", arr[i]);
    }

    printf("\n");
}
```

Как можно видеть, они просто создают массив, переставляют значения двух переменных и выводят массив.

```
long long int* get_data(char* filename, int n) {
    long long int* arr = crate_array(n);

    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        printf("Ошибка при чтении файла входных данных для сортировки\n");

        return NULL;
    }

    for (int i = 0; i < n; i++) {
        fscanf(file, "%lld", &arr[i]);
    }
```

```
    fclose(file);

    return arr;
}
```

Данная функция нужна для получения данных из входных файлов.

```
void test(char* filename, int n) {
    long long int* arr = get_data(filename, n);

    for (int i = 1; i < n; i++) {
        if (arr[i] < arr[i - 1]) {
            printf("\tОшибка в выходном массиве с количеством эл-ов %d и \n\tпу

            return;
        }
    }

    printf(
        "\tВыходной массив с количством эл-ов %d и
        \n\tпутем %s правильно отсортирован\n",
        n,
        filename
    );

    free(arr);
}

// mark - показатель с каким именем сохранять
void generate_output(long long int* arr, char* mark, char* sort_type, int n) {
    char filename[MAX_BUF_SIZE];

    sprintf(
        filename,
        "%s%s%s_were_%s%d.txt",
        OUTPUT_CATALOG,
        sort_type,
        (strcmp(sort_type, "heap_sort/") == 0 ? "hs" : "sls"),
        mark,
        n
    );

    FILE* output_file = fopen(filename, "w");
    if (output_file == NULL) {
        printf("Ошибка при создании файла для вывода данных\n");
```

```
            return;
        }

        for (int i = 0; i < n; i++) {
            fprintf(output_file, "%lld\n", arr[i]);
        }

        fclose(output_file);

        test(filename, n);
    }
```

Последняя функция нужна для записи уже отсортированного массива в файлы в определенных каталогах, в зависимости от типа сортировки и имени для сохранения.
Функция **test** как раз является **тестирующей функцией**, которая проверяет, чтобы все числа в массиве были отсортированный в невозрастающем порядке

Результаты экспериментов

После написания программы и ее запуска на входных тестах, я получил следующие результаты:

n	**Параметр**	**Номер сгенерированного массива**			**Среднее**
		1	2	3	**значение**
10	Сравнения	41	35	40	39
	Перемещения	30	21	26	26
100	Сравнения	1081	944	1027	1018
	Перемещения	640	516	581	579
1000	Сравнения	17583	15965	16798	16782
	Перемещения	9708	8316	9066	9030
10000	Сравнения	244460	226682	235286	235476
	Перемещения	131956	116696	124144	123266

Таблица 1: Результаты работы Пирамидальной сортировки

n	**Параметр**	**Номер сгенерированного массива**			**Среднее**
		1	2	3	**значение**
10	Сравнения	55	55	55	55
	Перемещения	10	10	10	10
100	Сравнения	5050	5050	5050	5050
	Перемещения	100	100	100	100
1000	Сравнения	500500	500500	500500	500500
	Перемещения	1000	1000	1000	1000
10000	Сравнения	50005000	50005000	50005000	50005000
	Перемещения	10000	10000	10000	10000

Таблица 2: Результаты работы Сортировки простым выбором

Отладка программы

Отладка программы происходила на каждом этапе разработки программы:

- Написание генератора случайных массивов.
- Написание функций для считывания данных из файла.
- Написание сортиров и проверка их корректности

Отладка генератора

Когда писался генератор массивов и возникали ошибки, то она имела две причины

- Ошибка при создании файла, куда класть созданный массив
- Ошибка при работе с самим массивом

Первая категория ошибок была решена путем создания функции-обертки, которая при создании файла проверяла, был он создан или нет, и в случае отрицательного ответа она выводила в консоль сообщение об ошибке, после чего становилось понятно, в чем конкретно в программе была ошибка, а не просто констатация завершения программы с каким-то аварийным кодом.

Ошибки при работе с массивом решались путем дебагинга программы, и внимательной работой с индексами.

Отладка сортировок

Отладка сортировок состояла из

- отлаживания функции считывания данных из файла (делалось это через дебагер)
- отлаживания факциий сортировок

На последнем остановимся поподробней. Так как функции сортировок работают с массивами, то их отлаживание происходило путем дебагинга программы (когда индекс выходил ха пределы), а также созданием функции test (была описана выше), которая проверяет результирующий массив на правильность (проверяет чтобы числа в нем шли с неубывающем порядке)

Асимптотика алгоритмов

Асимптотика пирамидальной сортировки

Время работы heapify при ее вызове для работы с узлом, который находится на высоте h, равно $O(h)$, так что общая стоимость процедуры Build Max Heap

```
void heap_sort(long long int* arr, int n) {
    int compare = 0, transp = 0;

    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i, &compare, &transp);
    }
    .....
}
```

ограничена сверху значением

$$\sum_{h=0}^{[\lg n]} \left\lceil \frac{n}{2(h+1)} \right\rceil O(h) = O\left(n \sum_{h=0}^{[\lg n]} \frac{h}{2^h}\right)$$

Последняя сумма вычисляется путем подстановки $x = \frac{1}{2}$ в формулу, что дает

$$\sum_{h=0}^{\inf} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Таким образом, время работы процедуры Build Max Heap имеет верхнюю границу.

$$O\left(n \sum_{h=0}^{[\lg n]} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\inf} \frac{h}{2^h}\right) = O(n)$$

Далее становится очевидно, при выполении следующего кода.

```
for (int i = n - 1; i > 0; i--) {
    swap(&arr[0], &arr[i]);
    transp++;

    heapify(arr, i, 0, &compare, &transp);
}
```

Мы будем n раз вызывать процедуру heapify, из-за этого общая сложность сортировки будет равна $O(n \lg n)$. [1]

Асимптотика сортировки простым выбором

Так как сама сортировка имеет небольшой размер, то ее можно повторно вставить сюда, для большей наглядности.

```
void selection_sort(long long int* arr, int n) {
    int j = n - 1;
    int max_i = 0;

    int compare = 0, transp = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int q = 0; q <= j; q++) {
            compare++;

            if (arr[q] > arr[max_i]) {
                max_i = q;
            }
        }

        swap(&arr[max_i], &arr[j]);
        transp++;

        j--;
        max_i = 0;
    }

    printf(
        "Selection sort, размер массива - %d\n\t
        Количество сравнений - %d, количество перестаовок - %d\n",
        n,
        compare,
        transp
    );
}
```

По циклам видно, что время работы программы зависит от числа эл-ов N, числа сравнений A и числа перестановочных максимумов B. Нетрудно видеть, что независимо от значений исходных ключей.

$$A = \binom{n}{k} = C_n^k = \frac{1}{2}N(N-1)$$

Следовательно, переменной является только величина B. Несмотря на всю безыскусность простого выбора, не так то легко выполнить точный анализ величины B.

$$B = (\min 0, ave(N+1)H_N - 2N, \max[N^2/4]).$$

В этом случае становится особенно интересным максимальное значение. Стандартное отклонение величины B имеет порядок $N^{3/4}$.
Такми образом, среднее время работы программы равно

$$2.5N^2 + 3(N+1)H_N + 3.5N - 11$$

машинных циклов, т.е данная программа работает лишь немногим медленнее прагораммы, реализующей метод простых вставок. [2]

Анализ допущенных ошибок

В ходе разработки программы возникали ошибки работы с памятью (забывал чистить память), которые были устронены. Так же возникали описки в коде, из-за которых в массивах могли генерироваться одинаковые числа, а не разные. Еще одной ошибкой был неправильный знак сравнения в пирамидальной сортировке, из-за которого сортировка работала неправильно.

Еще одной ошибкой, наверное самой большой, являлось столь позднее начало работы над проектом.

Список литературы

[1] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. Третье издание. — М.:«Вильямс», 2013.

[2] Кнут Д. Искусство программирования для ЭВМ. Том 3. — М.: «Мир», 1978.