МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. Шухова»
(БГТУ им. В. Г. Шухова)

Кафедра программного обеспечения вычислительной
техники и автоматизированных систем

Лабораторная работа №3
по дисциплине: «Алгоритмы и структуры данных»
по теме: **«Сравнительный анализ методов сортировки (C)»**

Выполнил/а: ст. группы ПВ-231
Чупахина София Александровна
Проверил: Акиньшин Данил Иванович

Белгород, 2024

Цель работы: изучение методов сортировки массивов и приобретение навыков в проведении сравнительного анализа различных методов сортировки.

Задания

1. Изучить временные характеристики алгоритмов.
2. Изучить методы сортировки:
   2.1. включением;
   2.2. выбором;
   2.3.обменом:
      2.3.1. улучшенная обменом 1;
      2.3.2. улучшенная обменом 2;
   2.4.Шелла;
   2.5. Хоара;
   2.6.пирамидальная.
3. Программно реализовать методы сортировки массивов.
4. Разработать и программно реализовать средство для проведения экспериментов по определению временных характеристик алгоритмов сортировки.
5. Провести эксперименты по определению временных характеристик алгоритмов сортировки. Результаты экспериментов представить в виде таблицы 9, клетки которой содержат количество операций сравнения при выполнении алгоритма сортировки массива с заданным количеством элементов. Провести эксперимент для упорядоченных, неупорядоченных и упорядоченных в обратном порядке массивов (для каждого типа массива заполнить отдельную таблицу).
6. Построить график зависимости количества операций сравнения от количества элементов в сортируемом массиве.
7. Определить аналитическое выражение функции зависимости количества операций сравнения от количества элементов в массиве.
8. Определить порядок функций временной сложности алгоритмов сортировки при сортировке упорядоченных, неупорядоченных и упорядоченных в обратном порядке массивов.

Содержание:

Задание 1:

Временная сложность алгоритма — зависимость времени выполнения алгоритма от количества обрабатываемых входных данных. Существует 2 способа оценить временную сложность алгоритма: с помощью аналитического выражения (оно же – функция временной сложности) и порядка функции временной сложности. Аналитическое выражение имеет вид t = t(N), где N – количество, размерность входных данных, а t – результат, выраженный как время в секундах/микросекундах или как, к примеру, количество выполняемых операций. Этот метод позволяет оценить временные характеристики конкретной реализации алгоритма в конкретной вычислительной системе – в разных вычислительных системах, к примеру, одни и те же операции могут занимать разное количество времени, и количество операций в разных реализациях также может разниться. В отличие от аналитического выражения, порядок функции временной сложности (ПФВС) зависит от наиболее быстрорастущего при увеличении N члена аналитического выражения и остается неизменным в разных реализациях и вычислительных системах, позволяя сравнивать алгоритмы на более абстрактном уровне. ПФВС записывается как O(f(n)).

Задание 2:

Кратко опишем заданные методы сортировки:

Задание 2.1

Сортировка вставками относится к базовым сортировкам. Принцип ее работы следующий: циклически перебираются элементы массива от второго до n-го, и для каждого из этих элементов ищется место в левой от него, отсортированной части массива. Значение текущего элемента сохраняется в отдельную переменную и сравнивается с соседним слева: если текущий элемент больше или равен соседнему, то он остается на своей позиции, и алгоритм переходит к следующему: если он меньше, то соседний элемент смещается вправо на 1, в то время как рассматриваемая для текущего элемента позиция на 1 смещается влево, и он сравнивается уже со следующим элементом. Если достигнут край массива, то цикл сравнений и смещений также прекращается.
Как можно заметить, для этой сортировки справедливо следующее: если некоторый элемент уже стоит на его месте, внутренний цикл, отвечающий за поиск его места в последовательности, сразу прекратит выполнение. Соответственно, сортировка уже упорядоченного массива выполнится куда быстрее. Мы вернемся к этому важному свойству при анализе порядка функции временной сложности в задании 8 и при изучении сортировки Шелла в задании 3.4.

Задание 2.2

Следующая базовая сортировка – сортировка выбором. Она имеет следующий принцип работы: по неотсортированной части массива осуществляется пробег с помощью цикла, в этой части выбирается минимальный элемент и

присоединяется к концу отсортированной части, меняясь местами с тем элементом, что раньше стоял после ее конца. Длина неотсортированной части после каждого такого пробега уменьшается на 1. Соответственно, и тут потребуется два вложенных цикла, однако уже без возможности досрочного прекращения работы одного из них, даже в упорядоченном массиве: ведь при выборе минимального элемента из неотсортированной части нельзя сказать, что он минимальный, не дойдя до ее конца.

Задание 2.3

Базовую сортировку обменом также называют пузырьковой сортировкой. В этом способе сортировки в случае, если их порядок нарушен, обмениваются между собой соседние элементы: последний и предпоследний, предпоследний и третий с конца и так далее. В результате такого обмена к моменту, когда последовательность обменов дойдет до начала массива, будет найден и вставлен в начало массива его наименьший элемент, и такой способ его перемещения сравнивают с тем, как всплывает в воде пузырек воздуха – отсюда и название. Проход от последнего к первому элементу выполняется n раз, где n – количество элементов в массиве, и с каждым проходом очередной наименьший для несортированной части элемент находит свое место. Для этого алгоритма количество сравнений также постоянно.

Задание 2.3.1

Попытавшись установить и проверить некоторые условия, при которых можно понять, что можно завершить пузырьковую сортировку в целом либо же прервать внутренний цикл, мы получим улучшенные варианты этой сортировки. Один из них – вариант, при котором для каждого прохода от последнего элемента к первому вводится флаг – совершались ли обмены на этом проходе. Если за весь проход ни разу не был совершен обмен, значит, массив уже отсортирован и выполнение функции можно прервать.

Задание 2.3.2

Другой вариант предполагает, что при каждом проходе будет запоминаться индекс последнего (то есть самого левого, близкого к началу) элемента, участвовавшего в обмене. Тогда при следующем проходе можно будет учесть это и прекратить проход, когда он дойдет до элемента с этим индексом. Ведь если после него обменов не совершалось, значит, часть массива левее этого элемента уже отсортирована.

Задание 2.4

Сортировка Шелла представляет собой модификацию сортировки вставками. Эта модификация заключается в том, что над массивом выполняется множество таких сортировок, но каждая из них при поиске места для элемента в отсортированной части сравнивает не элементы, стоящие рядом друг с другом, а элементы, расположенные на некотором расстоянии, то есть с некоторым шагом. Как уже было упомянуто, сортировка вставками занимает кратно меньше времени в случае, когда массив упорядочен. Получается, что если постепенно приводить

массив к более упорядоченному виду сортировками с большим шагом (занимающими не так много времени по причине того, что массивы, в которые войдут элементы с шагом, будут иметь меньшую длину), то дальнейшие сортировки с меньшим шагом также будут выполняться за меньшее количество времени. В конечном итоге даже финальная сортировка со стандартным, единичным шагом не окажет существенного влияния на время выполнения алгоритма: основная работа по сортировке была выполнена в пробегах до этого, и суммарное время выполнения всех пробегов будет меньшим, чем время стандартного выполнения сортировки вставками.
Существуют разные подходы к выбору шага для этой сортировки. Сам Шелл изначально предлагал шаг в половину длины массива, после каждого пробега уменьшающийся вдвое. Есть подход, где шаг выбирается как максимально возможное (не большее длины массива) число Фибоначчи. Оптимальной считается полученная эмпирически (не имеющая задающей ее формулы) последовательность Марцина Циура (первые числа – 1, 4, 10, 23, 57, 132). Мы остановимся на последовательности Дональда Кута, который вычисляет шаги так: $h_t = 1$; $h_{t-1} = 3h_t + 1$. Общее количество пробегов t в таком случае равно $\lfloor \log_3 \rfloor - 1$, то есть целой части логарифма длины массива по основанию 3, уменьшенной на 1.

Задание 2.5

Наиболее эффективной сортировкой считается сортировка Хоара – потому она и получила второе название «быстрая». Можно сказать, что она базируется на сортировке обменом – и на том предположении, что обменная сортировка становится гораздо эффективнее, если обмен происходит не между соседними элементами, а между элементами на максимальном расстоянии друг от друга. Достигается это максимальное расстояние за счет использования разделителя. Случайным образом в массиве выбирается элемент, который называется опорным – с ним в дальнейшем будут сравниваться все остальные элементы. По массиву осуществляется проход с двух сторон. Можно задавать индекс с помощью возвращающих псевдослучайные числа функций, можно всегда брать элемент на нулевой позиции. Сначала программа продвигается по элементам от левого края к правому, сравнивая значение каждого с опорным. Если значение одного из элементов оказывается больше опорного, то массив сканируется от правого края к левому, и как только в противоположной части находится элемент, наоборот, меньше опорного, то эти элементы меняются местами. Важно заметить две вещи: 1) индексы элементов, найденных при проходах слева направо и справа налево, после обмена значениями не обнуляются, и проходы продолжаются с того же места; 2) как проход слева направо, так и проход справа налево не заканчиваются при достижении позиции опорного элемента, а продолжаются до тех пор, пока эти два прохода не пересекутся, и индекс «левого» элемента не превысит индекс «правого». После этого массив окажется разделен на две части: в левой все элементы будут меньше опорного, а в правой – больше опорного элемента, причем на границу этих частей будут указывать индексы проходов, находящиеся рядом друг с другом. Далее, обратившись к значениям этих индексов, можно

выяснить, сколько элементов содержит как левая, так и правая часть массива. Если число элементов в какой-то из этих частей больше 1, то к этой части также применяется данный алгоритм. В конечном итоге, когда каждая из частей массива будет разбита на куски по одному элементу относительно разных «опор», массив окажется полностью отсортированным.

Задание 2.6

Пирамидальная сортировка является разновидностью сортировки обменом: по сути, в ней тоже в неотсортированной части ищется элемент (только уже не минимальный, а максимальный), присоединяется к концу отсортированной части (она находится в конце массива), и этот алгоритм повторяется с постепенным уменьшением неотсортированной части на 1. Нюанс в том, что максимальный элемент в пирамидальной сортировке ищется не линейным проходом по неотсортированной части и сравнением каждого нового элемента с текущим минимальным, а с использованием более сложного алгоритма – построения пирамиды. Пирамида – это бинарное дерево, в котором значения элементов-родителей не меньше значений дочерних элементов. Задача сводится к тому, чтобы, интерпретируя исходный массив в форме бинарного дерева, преобразовать его в пирамиду – таким образом в корне дерева окажется максимальный элемент. Он меняется местами с последним, и эта же последовательность действий повторяется для массива с длиной меньше на 1, то есть для массива, в котором не учитывается последний, отсортированный максимальный элемент, и так пока длина рассматриваемого массива не выродится до 1. Формирование пирамиды же происходит с помощью вспомогательной функции: если дерево не является пирамидой, но пирамидами являются левая и правая его ветви, то дерево преобразуется в пирамиду путем: 1) запоминания элемента в корне дерева, 2) рассмотрения дочерних элементов от старшего к младшему, от наибольшего значения к наименьшему, и смещения их значений на место родительского элемента, если дочернее значение больше, 3) таким образом осуществляется проход до некоего конечного элемента, не имеющего дочерних, либо до пары элементов, не больших родительского; тогда бывший корневой элемент располагается на освободившемся месте. Повторяя этот вспомогательный алгоритм, идя от последних элементов дерева, имеющих хотя бы один дочерний элемент, к его корню, можно превратить в пирамиду любое дерево.

Задание 3:

В ходе реализации методов сортировки использовались следующие вспомогательные функции:

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

//Обменивает значениями переменные типа int по адресам a и b
void swapInt(int *a, int *b) {
```

```
    int temp = *a;
    *a = *b;
    *b = temp;
}

//Возвращает максимальное из значений переменных a и b
int max2(int a, int b) {
    return (a > b) ? a : b;
}

//Возвращает логарифм числа argument по основанию base,
округленный вверх, если is_ceil = 1, и округленный вниз в
противном случае
int logarithm(size_t argument, int base, bool is_ceil) {
    int cur_argument = 1;
    int logarithm = 0;
    while (cur_argument < argument) {
        cur_argument *= base;
        logarithm++;
    }
    if (!is_ceil) {
        return (cur_argument == argument) ? logarithm : logarithm
- 1;
    } else {
        return logarithm;
    }
}
```

Перейдем к реализации самих алгоритмов сортировки.

```
//Осуществляет сортировку включением (вставками) массива array
размера size
void insertSort(int *array, int size) {
    for (int cur_ind = 1; cur_ind < size; cur_ind++) {
        int cur_el = array[cur_ind];
        int insert_ind = cur_ind;
        while (array[insert_ind - 1] > cur_el && insert_ind > 0) {
            array[insert_ind] = array[insert_ind - 1];
            insert_ind--;
        }
        array[insert_ind] = cur_el;
    }
}

//Осуществляет сортировку выбором массива array размера size
void choiceSort(int *array, int size) {
```

```
    for (int i = 0; i < size - 1; i++) {
        int cur_min_ind = i;
        for (int j = i + 1; j < size; j++) {
            if (array[j] < array[cur_min_ind]) {
                cur_min_ind = j;
            }
        }
        swapInt(array + i, array + cur_min_ind);
    }
}

//Осуществляет сортировку обменом (пузырьковую сортировку) массива
array размера size
void bubbleSort(int *array, int size) {
    for (int n = 0; n < size - 1; n++) {
        for (int i = size - 1; i > n; i--) {
            if (array[i] < array[i-1]) {
                swapInt(array + i, array + i - 1 );
            }
        }
    }
}

//Осуществляет сортировку обменом (пузырьковую сортировку) с
проверкой на наличие перестановок в текущем внешнем цикле массива
array размера size
void bubbleSort_improve1(int *array, int size) {
    for (int n = 0; n < size - 1; n++) {
        bool are_exchange = false;
        for (int i = size - 1; i > n; i--) {
            if (array[i] < array[i-1]) {
                swapInt(array + i, array + i - 1 );
                are_exchange = true;
            }
        }
        if (!are_exchange) {
            break;
        }
    }
}

//Осуществляет сортировку обменом (пузырьковую сортировку) с
учетом индекса последней перестановки в прошлом внешнем цикле
массива array размера size
void bubbleSort_improve2 (int *array, int size) {
    int exchange_ind = 0;
```

```
    int cur_exchange_ind = size-1;
    for (int n = 0; n < size - 1; n++) {
        for (int i = size - 1; i > exchange_ind; i--) {
            if (array[i] < array[i-1]) {
                swapInt(array + i, array + i - 1);
                cur_exchange_ind = i;
            }
        }
        exchange_ind = cur_exchange_ind;
    }
}

void shellSort (int *array, int size) {
    int amount_of_repeats = logarithm(size, 3, 0) - 1;
    if (amount_of_repeats < 1) {
        amount_of_repeats = 1;
    }
    int h = 1;
    for (int i = 1; i < amount_of_repeats; i++) {
        h = 3*h + 1;
    }
    //printf("%d %d \n",amount_of_repeats, h);
    while (h > 0) {
        for (int cur_ind = h; cur_ind < size; cur_ind++) {
            int cur_el = array[cur_ind];
            int insert_ind = cur_ind;
            while (array[insert_ind - h] > cur_el && insert_ind >
0) {
                array[insert_ind] = array[insert_ind - h];
                insert_ind -= h;
            }
            array[insert_ind] = cur_el;
        }
        h = (h-1) / 3;
    }
}

//Осуществляет сортировку Хоара (быструю сортировку) массива array
размера size
void quickSort(int *array, int size) {
    int reference_ind = 0;
    int left_ind = 0;
    int right_ind = size - 1;
    while (left_ind <= right_ind) {
        while (array[left_ind] < array[reference_ind] && left_ind
< size) {
```

```
            left_ind++;
        }
        while (array[right_ind] > array[reference_ind] &&
right_ind >= 0) {
            right_ind--;
        }
        if (left_ind <= right_ind) {
            swapInt(array + left_ind, array + right_ind);
            left_ind++;
            right_ind--;
        }
    }
    if (right_ind > 0) {
        quickSort(array, right_ind + 1);
    }
    if (left_ind < size - 1) {
        quickSort(array + left_ind, size - left_ind);
    }
}

//Преобразует массив array размера size, который в виде бинарного
дерева не является пирамидой, но имеет ветви, являющиеся
пирамидами, в пирамиду
void makeHeap(int *array, int first_ind, int last_ind) {
    int root_el, cur_son_ind, cur_parent_ind;
    root_el = array[first_ind];
    cur_son_ind = 2*first_ind+1;
    cur_parent_ind = first_ind;
    if ((cur_son_ind<last_ind) &&
(array[cur_son_ind]<array[cur_son_ind+1])) {
        cur_son_ind++;
    }
    while ((cur_son_ind<=last_ind) &&
(root_el<array[cur_son_ind])) {
        swapInt(&array[cur_son_ind], &array[cur_parent_ind]);
        cur_parent_ind = cur_son_ind;
        cur_son_ind = 2*cur_son_ind+1;
        if ((cur_son_ind<last_ind) &&
(array[cur_son_ind]<array[cur_son_ind+1])) {
            cur_son_ind++;
        }
    }
}

//Осуществляет пирамидальную сортировку массива array размера size
void heapSort(int *array, int size) {
```

```
    int first_ind, last_ind;
    first_ind = size/2 ; last_ind = size-1;
    while (first_ind > 0) {
        first_ind = first_ind - 1;
        makeHeap(array, first_ind, last_ind);
    }
    while (last_ind > 0) {
        swapInt(&array[0], &array[last_ind]);
        last_ind--;
        makeHeap(array, first_ind, last_ind);
    }
}
```

Задание 4:

Для проведения экспериментов по вычислению временных характеристик поступим следующим образом. В дополнение к функциям сортировки объявим отдельные функции генерации упорядоченных, случайных и упорядоченных в обратном порядке массивов заданного размера. Затем для каждой функции сортировки в отдельном файле создадим ее копию, которая возвращает значение типа int – количество выполненных сравнений. Счетчик сравнений устанавливается в начале каждой функции и возрастает на один: 1) внутри каждого цикла while или for, причем в цикле for инкрементирование счетчика можно задать вместе с модификацией переменной цикла, 2) после выполнения каждого цикла, потому что проверка условия выполнялась и тогда, когда условие продолжения вернуло значение «ложь», тело цикла не было выполнено и счетчик сравнений не был инкрементирован, 3) перед каждой условной конструкцией. Также, если функция рекурсивна или использует вспомогательный алгоритм, при каждом ее вызове к счетчику прибавляется возвращаемое ей значение – количество совершенных при ее вызове сравнений.
Далее в теле main мы можем создать массивы как функций генерации, так и функций сортировки. Перебирая с помощью циклов for способы генерации в первом цикле, способы сортировки во втором и размеры массивов в третьем, выведем на экран три таблицы, отображающих, какое количество операций сравнений выполняют данные алгоритмы при таких условиях.

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

//Сохраняет по адресу array массив целых чисел размера size со
случайными значениями в диапазоне от size/2 до -size/2
void generateRandomArray(int *array, int size) {
    for (size_t i = 0; i < size; i++) {
        array[i] = rand() % size - size/2;
    }
}
```

```
//Сохраняет по адресу array массив целых чисел размера size со
значениями в диапазоне от -size/2 до size/2, идущими в порядке
возрастания с шагом 1
void generateOrderedArray(int *array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = i - size/2;
    }
}

//Сохраняет по адресу array массив целых чисел размера size со
значениями в диапазоне от size/2 до -size/2, идущими в порядке
убывания с шагом 1
void generateOrderedBackwards(int *array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = size/2 - i;
    }
}

//Обменивает значениями переменные типа int по адресам a и b
void swapInt(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

//Возвращает максимальное из значений переменных a и b
int max2(int a, int b) {
    return (a > b) ? a : b;
}

//Возвращает логарифм числа argument по основанию base,
округленный вверх, если is_ceil = 1, и округленный вниз в
противном случае
int logarithm(size_t argument, int base, bool is_ceil) {
    int cur_argument = 1;
    int logarithm = 0;
    while (cur_argument < argument) {
        cur_argument *= base;
        logarithm++;
    }
    if (!is_ceil) {
        return (cur_argument == argument) ? logarithm : logarithm
- 1;
    } else {
        return logarithm;
```

```
    }
}

int insertSort(int *array, int size) {
    int compares = 0;
    for (int cur_ind = 1; cur_ind < size; cur_ind++, compares++) {
        int cur_el = array[cur_ind];
        int insert_ind = cur_ind;
        while (array[insert_ind - 1] > cur_el && insert_ind > 0) {
            array[insert_ind] = array[insert_ind - 1];
            insert_ind--;
            compares+=2;
        }
        compares+=2;
        array[insert_ind] = cur_el;
    }
    compares++;
    return compares;
}

int choiceSort(int *array, int size) {
    int compares = 0;
    for (int i = 0; i < size - 1; i++, compares++) {
        int cur_min_ind = i;
        for (int j = i + 1; j < size; j++, compares++) {
            compares++;
            if (array[j] < array[cur_min_ind]) {
                cur_min_ind = j;
            }
        }
        compares++;
        swapInt(array + i, array + cur_min_ind);
    }
    compares++;
    return compares;
}

int bubbleSort(int *array, int size) {
    int compares = 0;
    for (int n = 0; n < size - 1; n++, compares++) {
        for (int i = size - 1; i > n; i--, compares++) {
            compares++;
            if (array[i] < array[i-1]) {
                swapInt(array + i, array + i - 1 );
            }
        }
```

```
        compares++;
    }
    compares++;
    return compares;
}

int bubbleSort_improve1(int *array, int size) {
    int compares = 0;
    for (int n = 0; n < size - 1; n++, compares++) {
        bool are_exchange = false;
        for (int i = size - 1; i > n; i--, compares++) {
            compares++;
            if (array[i] < array[i-1]) {
                swapInt(array + i, array + i - 1 );
                are_exchange = true;
            }
        }
        compares++;
        compares++;
        if (!are_exchange) {
            break;
        }
    }
    compares++;
    return compares;
}

int bubbleSort_improve2 (int *array, int size) {
    int compares = 0;
    int exchange_ind = 0;
    int cur_exchange_ind = size-1;
    for (int n = 0; n < size - 1; n++, compares++) {
        for (int i = size - 1; i > exchange_ind; i--, compares++)
{
            compares++;
            if (array[i] < array[i-1]) {
                swapInt(array + i, array + i - 1);
                cur_exchange_ind = i;
            }
        }
        compares++;
        exchange_ind = cur_exchange_ind;
    }
    compares++;
    return compares;
}
```

```
int shellSort (int *array, int size) {
    int compares = 0;
    int amount_of_repeats = logarithm(size, 3, 0) - 1;
    int h = 1;
    for (int i = 1; i < amount_of_repeats; i++, compares++) {
        h = 3*h + 1;
    }
    compares++;
    while (h > 0) {
        for (int cur_ind = h; cur_ind < size; cur_ind++,
compares++) {
            int cur_el = array[cur_ind];
            int insert_ind = cur_ind;
            while (array[insert_ind - h] > cur_el && insert_ind >
0) {
                array[insert_ind] = array[insert_ind - h];
                insert_ind -= h;
                compares+=2;
            }
            compares+=2;
            array[insert_ind] = cur_el;
        }
        compares++;
        h = (h-1) / 3;
        compares++;
    }
    compares++;
    return compares;
}
```

```
int quickSort(int *array, int size) {
    int compares = 0;
    int reference_ind = 0;
    int left_ind = 0;
    int right_ind = size - 1;
    while (left_ind <= right_ind) {
        while (array[left_ind] < array[reference_ind] && left_ind
< size) {
            left_ind++;
            compares+=2;
        }
        compares+=2;
        while (array[right_ind] > array[reference_ind] &&
right_ind >= 0) {
            right_ind--;
```

```
            compares+=2;
        }
        compares+=2;
        compares++;
        if (left_ind <= right_ind) {
            swapInt(array + left_ind, array + right_ind);
            left_ind++;
            right_ind--;
        }
        compares++;
    }
    compares++;
    compares++;
    if (right_ind > 0) {
        compares += quickSort(array, right_ind + 1);
    }
    compares++;
    if (left_ind < size - 1) {
        compares += quickSort(array + left_ind, size - left_ind);
    }
    return compares;
}
```

```
int makeHeap(int *array, int first_ind, int last_ind) {
    int compares = 0;
    int root_el, cur_son_ind, cur_parent_ind;
    root_el = array[first_ind];
    cur_son_ind = 2*first_ind+1;
    cur_parent_ind = first_ind;
    compares+=2;
    if ((cur_son_ind<last_ind) &&
(array[cur_son_ind]<array[cur_son_ind+1])) {
        cur_son_ind++;
    }
    while ((cur_son_ind<=last_ind) &&
(root_el<array[cur_son_ind])) {
        swapInt(&array[cur_son_ind], &array[cur_parent_ind]);
        cur_parent_ind = cur_son_ind;
        cur_son_ind = 2*cur_son_ind+1;
        compares+=2;
        if ((cur_son_ind<last_ind) &&
(array[cur_son_ind]<array[cur_son_ind+1])) {
            cur_son_ind++;
        }
        compares+=2;
    }
```

```
    compares+=2;
    return compares;
}

int heapSort(int *array, int size) {
    int compares = 0;
    int first_ind, last_ind;
    first_ind = size/2 ; last_ind = size-1;
    while (first_ind > 0) {
        first_ind = first_ind - 1;
        compares += makeHeap(array, first_ind, last_ind);
        compares++;
    }
    compares++;
    while (last_ind > 0) {
        swapInt(&array[0], &array[last_ind]);
        last_ind--;
        compares += makeHeap(array, first_ind, last_ind);
        compares++;
    }
    compares++;
    return compares;
}

int main () {
    int (*sorting_method[8]) (int*, int) = {insertSort,
choiceSort, bubbleSort, bubbleSort_improve1,
                                            bubbleSort_improve2,
shellSort, quickSort, heapSort};
    void (*generating_method[8]) (int*, int) =
{generateOrderedArray, generateRandomArray,
generateOrderedBackwards};

    for (int cur_array_type_ind = 0; cur_array_type_ind < 3;
cur_array_type_ind++) {
        for (int cur_size = 5; cur_size <= 45; cur_size += 5) {
            printf("%d\t", cur_size);
        }
        printf("\n");
        for (int cur_method_ind = 0; cur_method_ind < 8;
cur_method_ind++) {
            for (int cur_size = 5; cur_size <= 45; cur_size += 5)
{
                int test_array[cur_size];
                generating_method[cur_array_type_ind](test_array,
cur_size);
```

```
                printf("%d\t",
sorting_method[cur_method_ind](test_array, cur_size));
            }
            printf("\n");
        }
        printf("\n");
    }
}
```

Задание 5

Теперь для получения таблиц с количествами операций сравнения, вызванных различными сортировками на упорядоченных, неупорядоченных и упорядоченных в обратном порядке массивах с длинами от 5 до 45 с шагом 5, достаточно запустить программу и перенести результаты из консоли в таблицу.

5	10	15	20	25	30	35	40	45
13	28	43	58	73	88	103	118	133
29	109	239	419	649	929	1259	1639	2069
29	109	239	419	649	929	1259	1639	2069
11	21	31	41	51	61	71	81	91
17	37	57	77	97	117	137	157	177
16	31	46	61	76	172	202	232	262
56	171	336	551	816	1131	1496	1911	2376
56	156	259	391	518	650	793	949	1084

5	10	15	20	25	30	35	40	45
19	80	153	258	361	484	721	868	1121
29	109	239	419	649	929	1259	1639	2069
29	109	239	419	649	929	1259	1639	2069
33	117	231	432	555	957	1152	1672	1792
29	87	203	365	511	891	1207	1771	1895
32	73	156	265	394	376	564	628	770
47	139	234	366	507	627	689	871	1108
48	136	235	371	446	578	733	865	992

5	10	15	20	25	30	35	40	45
33	118	253	438	673	958	1293	1678	2113
29	109	239	419	649	929	1259	1639	2069
29	109	239	419	649	929	1259	1639	2069
33	118	253	438	673	958	1293	1678	2113
29	109	239	419	649	929	1259	1639	2069
36	121	256	441	676	462	570	744	930
64	185	364	585	864	1185	1564	1985	2464
48	120	207	319	406	534	645	757	892

Количество сравнений, выполненных в ходе исполнения сортировочных алгоритмов на упорядоченных массивах различной длины:

	Количество элементов в массиве								
Сортировка	5	10	15	20	25	30	35	40	45
Включением	13	28	43	58	73	88	103	118	133
Выбором	29	109	239	419	649	929	1259	1639	2069
Обменом	29	109	239	419	649	929	1259	1639	2069
Обменом 1	11	21	31	41	51	61	71	81	91
Обменом 2	17	37	57	77	97	117	137	157	177
Шелла	16	31	46	61	76	172	202	232	262
Хоара	56	171	336	551	816	1131	1496	1911	2376
Пирамидальная	56	156	259	391	518	650	793	949	1084

Количество сравнений, выполненных в ходе исполнения сортировочных алгоритмов на неупорядоченных массивах различной длины:

	Количество элементов в массиве								
Сортировка	5	10	15	20	25	30	35	40	45
Включением	19	80	153	258	361	484	721	868	1121
Выбором	29	109	239	419	649	929	1259	1639	2069
Обменом	29	109	239	419	649	929	1259	1639	2069
Обменом 1	33	117	231	432	555	957	1152	1672	1792
Обменом 2	29	87	203	365	511	891	1207	1771	1895
Шелла	32	73	156	265	394	376	564	628	694
Хоара	47	139	234	366	507	627	689	871	1108
Пирамидальная	48	136	235	371	446	578	733	865	992

Количество сравнений, выполненных в ходе исполнения сортировочных алгоритмов на упорядоченных в обратном порядке массивах различной длины:

	Количество элементов в массиве								
Сортировка	5	10	15	20	25	30	35	40	45
Включением	33	118	253	438	673	958	1293	1678	2113
Выбором	29	109	239	419	649	929	1259	1639	2069
Обменом	29	109	239	419	649	929	1259	1639	2069
Обменом 1	33	118	253	438	673	958	1293	1678	2113
Обменом 2	29	109	239	419	649	929	1259	1639	2069
Шелла	36	121	256	441	676	462	570	744	882
Хоара	64	185	364	585	864	1185	1564	1985	2464
Пирамидальная	48	120	207	319	406	534	645	757	892

Задание 6

По значениям таблиц без труда можно составить графики зависимости количества сравнений от длины массива для определенной сортировки.

Примечание: следует учесть, что графики сортировки выбором и обменом всегда оказываются наложены друг на друга, поскольку для этих сортировок выполняется одинаковое количество сравнений.

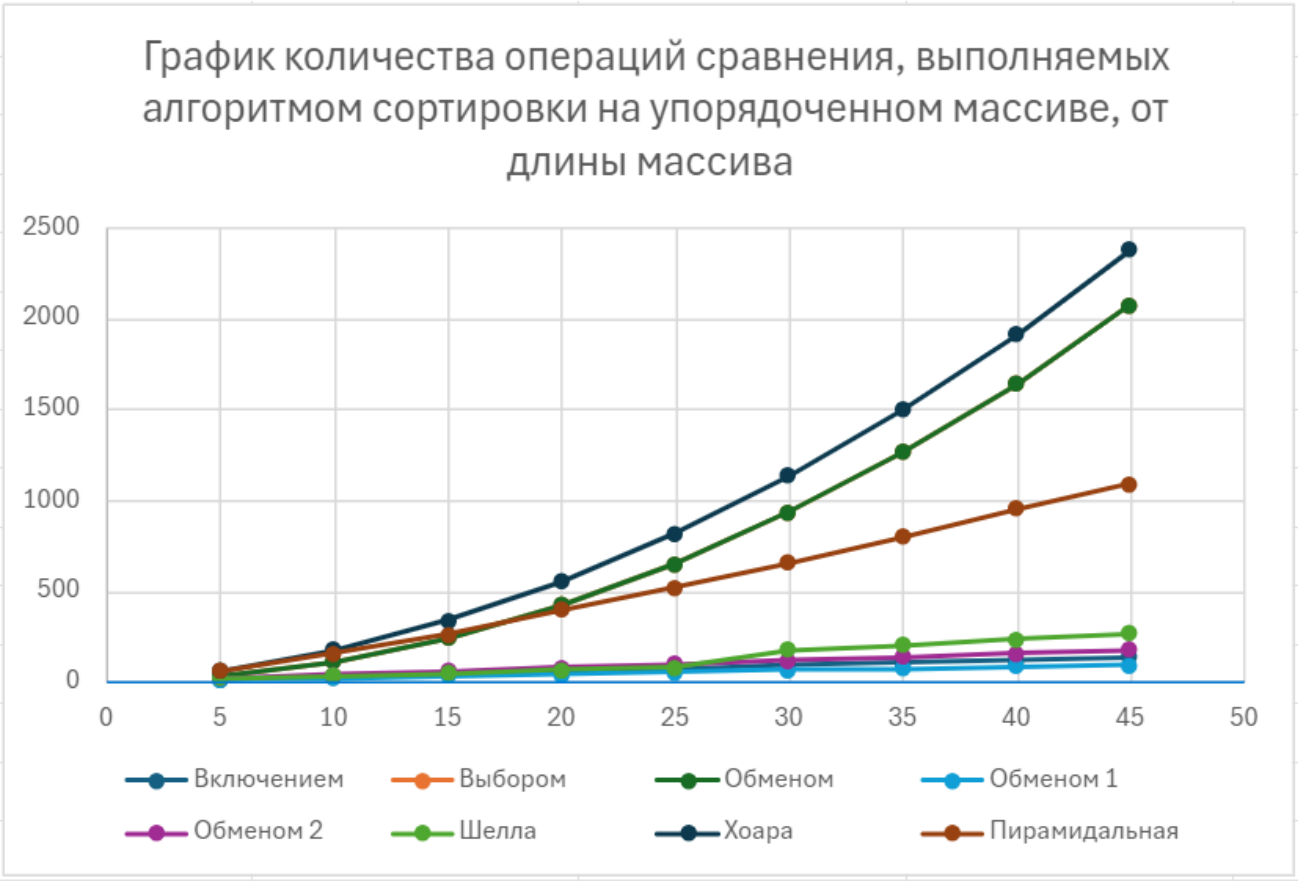


График количества операций сравнения, выполняемых алгоритмом сортировки на упорядоченном массиве, от длины массива
2500
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
40
45
50
Включением
Выбором
Обменом
Обменом 1
Обменом 2
Шелла
Хоара
Пирамидальная

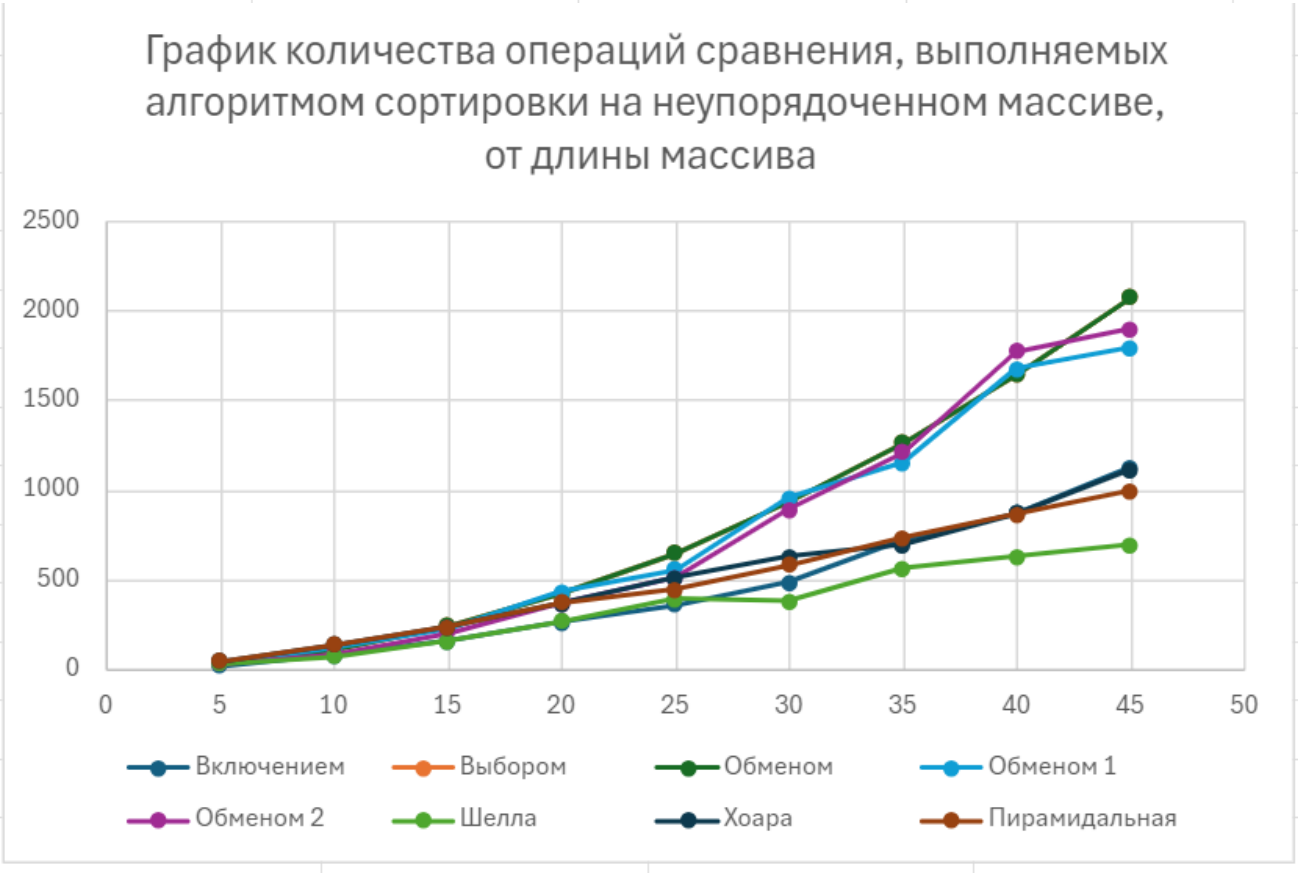
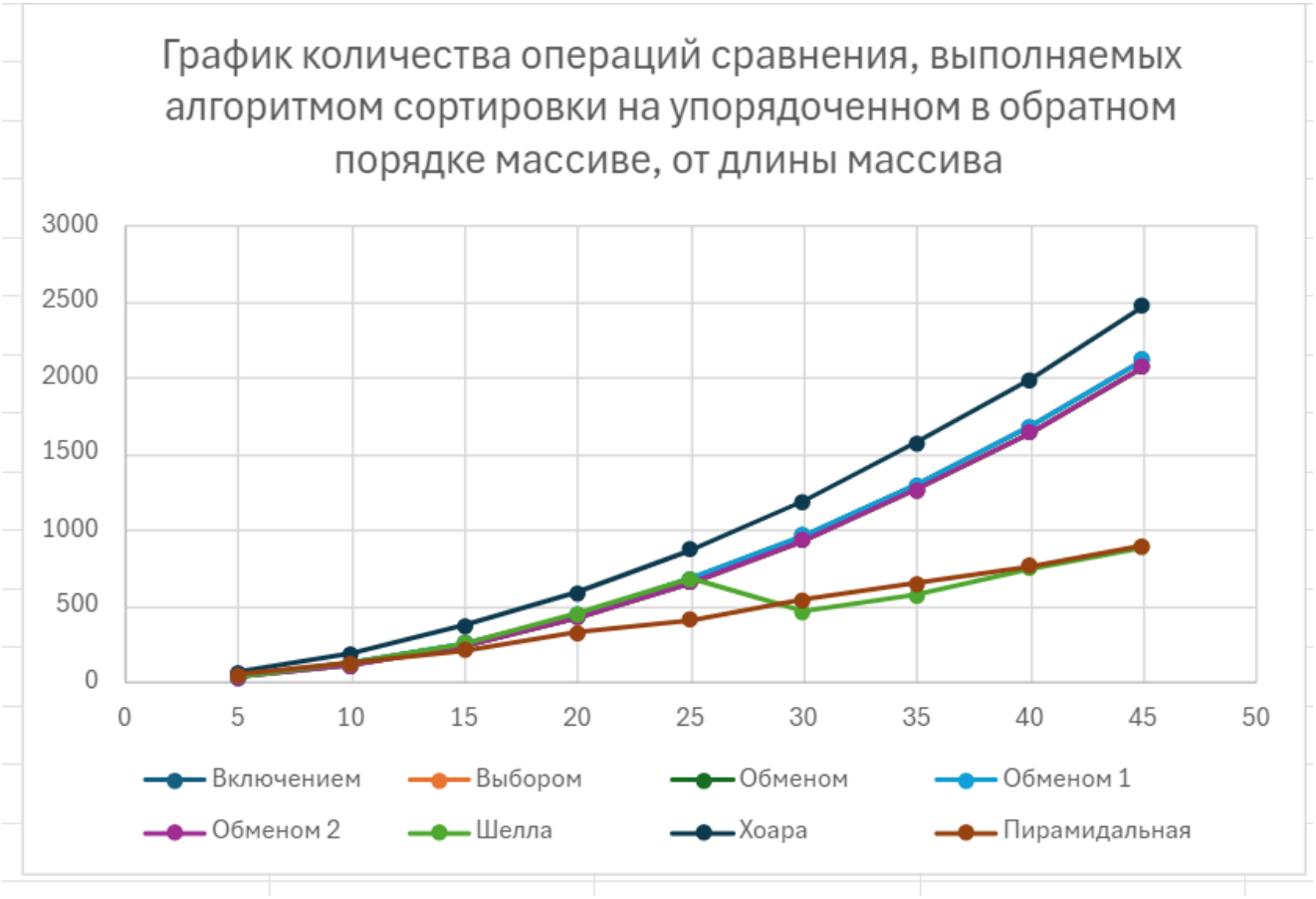


График количества операций сравнения, выполняемых алгоритмом сортировки на неупорядоченном массиве, от длины массива
2500
2000
1500
1000
500
0
0
5
10
15
20
25
30
35
40
45
50
Включением
Выбором
Обменом
Обменом 1
Обменом 2
Шелла
Хоара
Пирамидальная



Задание 7

Проанализируем каждую из модифицированных функций сортировки и узнаем, сколько сравнений и при каких условиях она выполняет. Длину массива далее будем обозначать как n.

Для сортировки вставками выполняется n – 1 операций сравнения в теле внешнего цикла, перебирающего слева направо неотсортированные элементы, и 1 сравнение при выходе из него. Для каждой итерации внешнего цикла выполняется и внутренний цикл, перебирающий отсортированные элементы массива. Он завершается либо когда текущий отсортированный элемент меньше или равен тому, которому ищется место, либо когда перебор доходит до первого элемента. Но выполнится ли первое условие, зависит от значений элементов, которые нельзя предусмотреть и от которых нельзя отобразить зависимость в аналитической функции. **Ограничимся худшим и лучшим вариантом**. В худшем варианте при рассмотрении элемента с индексом 1 внутренний цикл выполнится 1 раз, при рассмотрении элемента с индексом 2 – 2 раза, ..., при рассмотрении элемента с индексом n – 1 – n – 1 раз. Итого сумма итераций циклов равна (1 + n – 1) / 2 * (n – 1) = n(n – 1)/2. На выполнение каждой итерации цикла приходится 2 операции сравнения, плюс по 2 операции приходится на завершение каждого из циклов. Сложив все эти значения, получим t = n + 2*n(n – 1)/2 + 2*(n – 1) = n + n(n – 1) + 2(n – 1) = n + n^2 – n + 2n – 2 = n^2 + 2n – 2. В лучшем же варианте внутренний цикл не будет выполняться ни разу, и в каждой итерации внешнего будет проходить лишь 1 проверка условия внутреннего цикла, которая

сразу даст ложь (но займет 2 операции сравнения, поскольку условие состоит из двух частей). Получим формулу $t = n + 2(n – 1) = n + 2n – 2 = 3n – 2$.

Для сортировки выбором также выполняется n – 1 операций сравнения в теле внешнего цикла, отвечающего за хранение первого неотсортированного элемента, и 1 сравнение при выходе из него. Для каждой итерации внешнего цикла выполняется внутренний цикл, перебирающий остальные неотсортированные элементы массива в поисках минимального. Лучший и худший случай здесь не будут различаться, поскольку сравнения элемента с начале неотсортированной части с остальными элементами этой части проводятся до тех пор, пока не будет достигнут конец неотсортированной области, и досрочный выход из цикла не предусмотрен. При сравнении с элементом с индексом 0 внутренний цикл выполнится n – 1 раз, при рассмотрении элемента с индексом 1 – n – 2 раза, ..., при рассмотрении элемента с индексом n – 1 – 1 раз. Итого сумма итераций циклов равно $(n – 1 + 1) / 2 * (n – 1) = n(n – 1)/2$. На выполнение каждой итерации цикла приходится 2 операции сравнения (перебор элементов и проверка, не вышел ли индекс рассматриваемого элемента за длину массива, и сравнение элемента со стоящим в начале неотсортированной части), плюс по 1 операции приходится на завершение каждого из циклов. Сложив все эти значения, получим $t = n + 2*n(n – 1)/2 + (n – 1) = n + n(n – 1) + (n – 1) = n + n^2 – n + n – 1 = n^2 + n – 1$.

Для сортировки обменом также выполняется n – 1 операций сравнения в теле внешнего цикла, отвечающего за индекс текущего прохода и, соответственно, количество отсортированных элементов, 1 сравнение при выходе из него. Для каждой итерации внешнего цикла выполняется внутренний цикл, перебирающий пары соседних элементов и сравнивающий их для того, чтобы узнать, есть ли необходимость в обмене. Лучший и худший случай здесь не будут различаться, поскольку сравнения соседних пар проводятся до тех пор, пока не будет достигнут конец неотсортированной области, и досрочный выход из цикла не предусмотрен. При первом пробеге, пробеге с индексом 0, внутренний цикл выполнится n – 1 раз, при рассмотрении элемента с индексом 1 – n – 2 раза (поскольку после каждого прохода минимальный элемент оказывается в начале массива, и больше нет необходимости проводить сравнения в этой области), ..., при рассмотрении элемента с индексом n – 1 – 1 раз. Итого сумма итераций циклов равно $(n – 1 + 1) / 2 * (n – 1) = n(n – 1)/2$. На выполнение каждой итерации цикла приходится 2 операции сравнения (перебор соседних пар и проверка, не вышел ли индекс левого ее элемента за левую границу массива, и сравнение элементов пары), плюс по 1 операции приходится на завершение каждого из циклов. Сложив все эти значения, получим $t = n + 2*n(n – 1)/2 + (n – 1) = n + n(n – 1) + (n – 1) = n + n^2 – n + n – 1 = n^2 + n – 1$.

В первом улучшении сортировки обменом, помимо прочего, в каждой итерации внешнего цикла выполняется проверка флага, контролирующего, проводились ли обмены соседних элементов в этом пробеге. В худшем случае, когда массив отсортирован в обратном порядке и обмены проводятся при каждом пробеге, эти операции только добавят по 1 операции сравнения в каждую итерацию внешнего

цикла. Учтя это, откорректируем стандартную формулу: $t = n + 2*n(n - 1)/2 + (n - 1) + (n - 1) = n + n(n - 1) + 2*(n - 1) = n + n^2 - n + 2n - 2 = n^2 + 2n - 2$. В лучшем же случае после первой итерации внешнего цикла будет понятно, что не был совершен ни один обмен и можно досрочно завершить работу программы. Тогда будет совершено только одно сравнение для внешнего цикла, в о внутреннем же цикле будет совершено $2*(n - 1)$ сравнений ($n - 1$ сравнений пар и $n - 1$ сравнений переменной цикла с границей диапазона перебора), плюс одно сравнение для выхода из внутреннего цикла и одно сравнение флага. Итого получим $t = 1 + 2*(n - 1) + 1 + 1 = 3 + 2*(n - 1) = 3 + 2n - 2 = 2n+1$.

Во втором улучшении сортировки обменом, в каждой итерации внешнего цикла находится индекс последнего элемента, над которым был совершен обмен, и дальнейшие итерации перебирают пары соседних элементов только до этой границы. В худшем случае, когда массив отсортирован в обратном порядке и обмены проводятся до заданной границы, индекс последнего элемента, на котором был совершен обмен, будет совпадать с индексом границы отсортированной части, и мы получим все ту же аналитическую формулу, что и для стандартной сортировки обменом: $t = n + (1+n)(n - 1) = n + n - 1 + n^2 - n = n^2 + n - 1$. В лучшем же случае после первой итерации внешнего цикла будет понятно, что не был совершен ни один обмен и можно установить границу в правый конец массива; после этого будет совершаться лишь одна проверка, чтобы сразу выходить из внутреннего цикла в каждой итерации внешнего. Итого будет совершено $(n - 1)$ сравнений для внешнего цикла и 1 выхода из него, в внутреннем же цикле будет совершено $2*(n - 1)$ сравнений ($n - 1$ сравнений пар и $n - 1$ сравнений переменной цикла с границей диапазона перебора) плюс одно сравнение для выхода, далее по одному сравнению для мгновенного выхода из внутреннего цикла. Итого получим $t = n + 2*(n - 1) + 1 + n - 2 = 2n + 2n - 3 = 4n - 3$.

Переходим к поиску аналитических формул для усовершенствованных сортировок. Сортировка Шелла – по сути, многократное применение сортировки вставками, но не ко всему массиву, а к «подмассивам», которым принадлежат элементы исходного массива с некоторым шагом. Однако размер шага задается рекуррентной функцией, поэтому выразить аналитическую функцию затруднительно. Предположим, мы рассматриваем лучший случай, когда массив уже упорядочен. Тогда для каждой отдельной сортировки вставками подмассива размером k количество операций сравнения будет равно $3n - 2$. Но как найти количество подмассивов и k для каждого из них? Для способа выбора размера шагов, предложенного Дональдом Кнутом, количество подмассивов будет равно $\log_3 n - 1$. Но для того, чтобы найти длину первого подмассива, нужно длину изначального массива нацело поделить на величину шага, которая находится только рекуррентно, как $h_t = 1$, $h_{t-1} = 3h_t+1$. Соответственно, $h_{t-2} = 3h_{t-1} +1 = 3(3h_t + 1) + 1 = 9h_t + 3 + 1$; $h_{t-3} = 3h_{t-2} +1 = 3(9h_t + 3 + 1) + 1 = 27h_t + 9 + 3 + 1$. Учитывая, что $h_t = 1$, можно сделать вывод, что для массива длиной n, первый шаг равен $\sum_{i=0}^{\log_3 n-2} 3^i$. В цикле, перебирающем от большего к меньшему значения

шага, есть два вложенных цикла: из них первый исполняется $n - \sum_{i=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^i$ раз и требует для этого $n - \sum_{i=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^i + 1$ сравнений, а второй не будет для упорядоченных подмассивов выполняться ни разу, вызывая только 2 сравнения для мгновенного выхода. То есть для каждого выполнения внешнего цикла потребуется $n - \sum_{i=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^i + 1 + 2 \times (n - \sum_{i=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^i) =$ $3n - 3 \times \sum_{i=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^i + 1$ сравнений. Сам внешний цикла будет выполняться $\mathrm{lfloor}\log_3 n\mathrm{rfloor} - 1$ раз, влияя при этом на величину шага. Итоговая формула аналитической функции для сортировки Шелла, будет иметь вид: $\sum_{j=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^j n\mathrm{rfloor} - 2(3n - 3 \times \sum_{i=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^i + 1) + \mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2 + \mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 1 =$ $\sum_{j=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^j n\mathrm{rfloor} - 2(3n - 3 \times \sum_{i=0}^{\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2} 3^i + 1) + 2 \times \mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 3$, так как необходимо также прибавить сравнения, происходящие в ходе нахождения h_1 в цикле (их будет $\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 2$) и в ходе циклического уменьшения размера шага ($\mathrm{lfloor}\{\log_3 n\mathrm{rfloor} - 1$).

Для худшего случая не будем даже пробовать ее искать: если массив упорядочен в обратном порядке, то только первый подмассив будет упорядочен в обратном порядке, а остальные будут пересекаться с первым подмассивом после сортировки и будут уже упорядочены хотя бы частично. Поэтому опираться на формулу количества сравнений при сортировке вставками в худшем случае мы не сможем, соответственно, не сможем получить аналитическую формулу числа сравнений для сортировки Шелла в худшем случае.

Для сортировки Хоара это также будет сложной задачей. Количество сравнений в сортировке Хоара не зависит от расположения элементов в нем, от того, упорядочен ли массив или нет – оно зависит от того, как был выбран опорный элемент на каждой итерации. Если опорный элемент выбирается случайно, аналитическую формулу составить не получится. Но если опорный элемент, как в нашей реализации, всегда первый, а в упорядоченном массиве первый элемент всегда наименьший, мы можем найти для наилучшего случая закономерности, позволяющие вычислить аналитическую функцию. Обычно при первом проходе по целому массиву, индекс элемента левой стороны будет возрастать на 1 в каждой итерации первого внутреннего цикла, и каждая будет требовать 2 сравнения. Но в нашем случае это произойдет сразу же, и весь цикл перебора левого потребует 2 операции сравнения. Когда найдется элемент, больший или равный опорному, запустится второй цикл. Он будет уменьшать при каждой итерации индекс правого элемента на 1, используя 2 сравнения. В нашем случае проход закончится при достижении края массива, за n – 1 итераций, и на это будет затрачено 2*n операций сравнения (с учетом проверки на выход). Затем проводится 1 сравнение в теле условной конструкции, еще 2 мы прибавим за 1

исполнение внешнего цикла и выход из него. Итого потребуется 2 + 2*n + 1 + 2 = 2n + 5 сравнения на массив. Если всегда выбирать первый элемент как опорный, то массив окажется разбит на 2 части: одну, в которую входит только опорный элемент, и вторую, с длиной n – 1. Для нее, после выполнения еще двух сравнений на то, одноэлементны ли полученные массивы, функция запустится снова, и там количество сравнений будет 2*(n – 1) + 5, для следующего – 2*(n – 2) + 5 и так далее. Так будет продолжаться, пока длина второго подмассива не достигнет 2. Итого получим, добавив к каждому слагаемому по 2 как сравнения на одноэлементность, (2*n + 7 + 2*2 + 7)/2 * (n – 1) = (2*n+ 7 + 4 + 7)/2 * (n – 1) = (2n+18)/2 * (n – 1) = (n+9)*(n – 1). Не будем рассматривать худший случай, поскольку там придется анализировать то, как массив меняется после каждой перестановки.

Наконец, рассмотрим пирамидальную сортировку. Функция makeHeap создает пирамиду из дерева, две ветви которого уже являются пирамидами, перемещая верхний элемент в низ (на места старших дочерних элементов), пока находится дочерний элемент, больший предыдущего корневого. В случае, когда массив уже упорядочен, первый элемент наоборот, будет наименьшим, и его нужно будет продвигать про максимуму вниз. Для этого будет выполнено 2 сравнения при первом нахождении дочернего элемента, и по 4 на каждую итерацию цикла (2 на проверку условия цикла, 2 на выбор большего дочернего элемента), плюс еще 2 на выход из цикла. Для массива, упорядоченного по возрастанию, корневой элемент нужно будет продвигать вниз до тех пор, пока не будет достигнут элемент без дочерних. Нам предстоит вызывать эту функцию для разных уровней, и в зависимости от уровня корневой элемент, если дерево имеет «незавершенный» уровень, будет уходить либо на завершенный, либо на незавершенный уровень, и для этих случаев будет различаться число итераций цикла. Поэтому точно вычислить значение аналитической функции не выйдет.

Задание 8

Примечание: так как количество циклов в каждом алгоритме, их вложенность и условия досрочного прерывания мы подробно анализировали в задании 7, не будем заострять на этом внимание ниже.

Для сортировки вставками порядок функции временной сложности в лучшем случае, то есть когда массив полностью упорядочена, составляет O(n), в худшем случае, когда массив упорядочен в обратном порядке, ПФВС сортировки вставками составляет $O(n^2)$. Этот вывод можно сделать как минимум из аналитических функций количества сравнений от длины: для лучшего случая t = 3n – 2, и наиболее быстрорастущим членом без учета коэффициента будет n; для худшего случая $t = n^2 + 2n – 2$, и наиболее быстрорастущим членом без учета коэффициента будет являться n^2. Уточнить ПФВС относительно «худшего случая» для промежуточных случаев, когда массив не упорядочен ни в прямом, ни в обратном порядке, возможным не представляется. Понятно, что количество операций сравнения и перестановки в среднем будет меньше, потому что, возможно, какие-то элементы окажутся на своих местах. Но в целом порядок

функции временной сложности сортировки вставками для неупорядоченных массивов также считается равным $O(n^2)$.

Для сортировки выбором порядок функции временной сложности одинаков как для упорядоченного, так и для случайного и упорядоченного в обратном порядке массива, поскольку этот алгоритм не содержит никаких условий, позволяющих провести досрочный выход из него. Так что его ПФВС во всех трех указанных случаях равен $O(n^2)$. Это подтверждается аналитической функцией количества сравнений от длины: для сортировки выбором $t = n^2 + n - 1$, и наиболее быстрорастущий член здесь равен n^2.

Точно так же и для сортировки обменом порядок функции временной сложности одинаков для массива с любым изначальным порядком элементов, поскольку не содержит в себе средств досрочного прекращения проверок. Так что и его ПФВС во всех трех случаях равен $O(n^2)$. Для этого варианта также $t = n^2 + n - 1$, и наиболее быстрорастущий член также n^2.

Однако для первого варианта улучшенной сортировки обменом, в котором проверяется наличии перестановок в текущей итерации, уже имеет разные сценарии выполнения и, соответственно, разные ПФВС для «лучшего» и «худшего» сценария. Если массив уже упорядочен, то после первого же прохода по массиву выполнение алгоритма прекратится, и ПФВС будет равен $O(n)$. В этом случае $t = 2n+1$, порядок ПФВС совпадает с самым быстрорастущим членом (без учета коэффициента). В иных же случаях (как худшем, так и промежуточных между лучшим и худшим) ПФВС этой сортировки будет оставаться таким же, как для сортировки обменом в целом, то есть $O(n^2)$. Тогда $t = n^2 + 2n - 2$, и порядок ПФВС совпадает с самым быстрорастущим членом.

Похожая ситуация будет наблюдаться для второго варианта улучшенной сортировки обменом, в котором запоминается индекс последнего переставленного элемента, чтобы в следующих проходах по массиву прекращать пробег, когда он достигнет этой границы. В лучшем случае, если массив уже упорядочен, после первого же прохода этот индекс будет равен индексу последнего элемента, и остальные проходы будут завершаться, не начавшись, и ПФВС будет равен $O(n)$. В этом случае $t = 4n - 3$, порядок ПФВС совпадает с самым быстрорастущим членом (без учета коэффициента). В иных случаях ПФВС этой сортировки будет оставаться таким же, как для сортировки обменом в целом, то есть $O(n^2)$. Тогда $t = n^2 + n - 1$, и порядок ПФВС совпадает с самым быстрорастущим членом.

Рассчитать порядок функции временной сложности для алгоритма сортировки Шелла по аналитическому выражению проблематично, как и рассчитать само аналитическое выражение. К тому же, ПФВС сортировки Шелла не зависит напрямую от упорядоченности или неупорядоченности массива. Если массив полностью упорядочен, то сортировка Шелла совершит больше сравнений и займет больше времени, чем сортировка вставками, поскольку должна совершить несколько проходов по подмассивам; а в случае, когда массив упорядочен в обратном порядке, количество сравнений и занятое время будет меньше, чем

должна быть сумма сравнений сортировок вставками всех подмассивов для худшего случая: подмассивы будут пересекаться, каждый следующий будет упорядоченней, чем предыдущий. Иначе говоря, ПФВС для сортировки Шелла можно вычислить только как среднее и весьма примерное значение, без разделения на лучший и худший случай, и при этом ПФВС зависит от способа вычисления шага, с которым выполняется сортировка. Для рекурсивного алгоритма Дональда Кнута порядок функции временной сложности аппроксимируется как $O(n\log_2 n)$.

Для сортировки Хоара порядок функции временной сложности зависит не от упорядоченности или неупорядоченности массива, а от того, как выбирается опорный элемент. В лучшем случае, когда опорный элемент каждый раз выбирается так, что делит массив пополам, ПФВС сортировки Хоара равен O(n $\log_2$n). В худшем случае, когда опорный элемент каждый раз выбирается так, что одна из «половин» массива имеет длину 1, ПФВС сортировки Хоара равен O(n^2). В промежуточных случаях ПФВС находится между этими двумя величинами.

Для пирамидальной сортировки порядок функции временной сложности равен O(n $\log_2$n), где множитель n отвечает за проход по ветвям, а множитель $\log_2$n – за построение дерева.

Вывод:

В ходе лабораторной работы изучили методы сортировки и проанализировали их временные характеристики, подтвердив анализ практическими расчетами.