МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Отчет по лабораторной работе**

**«Сортировки и их производительность»**

**Выполнил:**

Студент группы 3824Б1ФИ1

Катышев Дмитрий Владимирович

**Проверила:**

Ассистент ВВСП ИИТММ

Панова Е.А.

Рецензент:

Нижний Новгород  
2024

# Содержание

#### [Введение………………………………………………………………………………………3](#Введение)

#### [1 Постановка задачи……………………………………………………………4](#_Постановка_задачи)

#### [2 Руководство пользователя…………………………………………………..5](#_Руководство_пользователя)

#### [3 Руководство программиста………………………………………………….6](#Руководство_программиста)

##### [3.1Описание структуры программы………………..………………………………….6](#Описание_структуры_программы)

##### [3.2 Описание структур данных……………………………………….………………..6](#Описание_структур_данных)

##### [3.3 Описание алгоритмов……………...………………………………………………..6](#Описание_алгоритмов)

## [4 Эксперименты…………………………………………………………………9](#Эксперименты)

### [Заключение……………………………………………………………...………17](#Заключение)

### [Список литературы………………...…………………………………………..18](#Список_литературы)

### [Приложение………………...…………………………………………………...19](#Приложение)

# Введение

Стоит начать с того, почему же задача сортировки массива является важной. Это действительно так, потому что в контексте отсортированного массива можно применить бинарные поиск или иные эффективные способы поиска данных. Также отсортированный массив делает более читаемыми данные и позволяет их визуализировать (построить графики, проводить метрики и т.д.). Существует еще множество случаев, когда нужны отсортированные данные, поэтому крайне важно не только отсортировать их, но и сделать это эффективно. В данной работе изучаются различные алгоритмы, выполняющие эту задачу, в том числе и один специфичный, требующий особых условий (он будет представлен позже). По результатам данной работы, можно будет ответить на вопрос, почему же неэффективно пользовать лишь одной сортировкой во всех случаях (заранее можно сказать, что действительно нельзя).

# Постановка задачи

Задача лабораторной работы заключается в разработке программы для сортировки массивов с использованием нескольких алгоритмов. Программа должна включать:

1. Реализацию пузырьковой сортировки, сортировки вставками, выбора, быстрой сортировки, сортировки слиянием и сортировки подсчетом.
2. Оптимизация алгоритма быстрой сортировки для повышения ее производительности.
3. Реализацию гибридного подхода, комбинирующего различные методы сортировки.
4. Проведение тестирования программы на различных наборах данных с анализом времени работы и константного времени.

В рамках работы также проводится анализ производительности алгоритмов сортировки на больших объемах данных (до 100 миллионов элементов) и выводятся графики, иллюстрирующие зависимость времени работы от размера входного массива.

# Руководство пользователя

**Запуск программы**

1. Для использования программы необходимо:

* Запустить скомпилированный исполняемый файл. Программа автоматически генерирует тестовые данные для сортировки.
* Выберите алгоритм сортировки: пузырьковая, вставками, выбором, слиянием, быстрая сортировка или гибридный вариант, а также сортировка подсчетом.
* Выберите количество элементов для сортировки.
* Выберите необходимо ли проверять верность сортировки.
* Программа выведет результаты сортировки и время, затраченное на обработку данных для каждого алгоритма. А в случае выбора проверки верности сортировки будет указано корректно ли была выполнена работа.

1. Программа позволяет задать следующие типы входных данных:

* **Случайные данные**: элементы массива генерируются случайным образом.
* **Прямо отсортированные данные**: массив уже отсортирован по возрастанию.
* **Обратно отсортированные данные**: массив отсортирован по убыванию.

1. Программа выводит на экран:

* Время работы выбранного алгоритма для выбранного типа входных данных.
* Опционально выводит на экран верно ли была выполнена сортировка.

# Руководство программиста

## Описание структуры программы

Программа состоит из нескольких частей:

**Основной файл** (main.cpp): Включает функции для генерации тестовых данных, выбор алгоритма сортировки и вывод результатов.

**Файл реализации сортировок** (sort.cpp): Реализованы все алгоритмы сортировки (пузырьковая, слияние, быстрая сортировка и гибридный алгоритм).

**Библиотека реализованных сортировок** (sort.h): Включает функции для замера времени и вывода результатов.

## Описание структур данных

Программа использует несколько структур данных:

• Массив чисел (тип double или int), который сортируется в зависимости от выбранного алгоритма.

• Указатель на массив (тип double или int), который позволяет удобно «перемещаться» по массиву.

• Структура clock\_t используется для измерения времени

## Описание алгоритмов

Bubble sort — это популярный, но неэффективный алгоритм сортировки. Он работает путем многократного изменения местами соседних элементов, которые находятся не на своем месте.Выполняется некоторое количество проходов по массиву — начиная от начала массива, перебираются пары соседних элементов массива. Если 1-й элемент пары больше 2-го, элементы переставляются (выполняется обмен). Пары элементов массива перебираются (проходы по массиву повторяются) либо (n - 1)(n−1) (ь раз, либо до тех пор, пока на очередном проходе не обнаружится, что более не требуется выполнять перестановки (обмены) (массив отсортирован). При каждом проходе алгоритма по внутреннему циклу очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива (как бы «всплывает» до нужной позиции, как пузырёк в воде — откуда и название алгоритма)

Selection sort — один из простейших алгоритмов сортировки работает так, сначала в массиве находится наименьший элемент, и он меняется местами с первым элементом (возможно, и с собой, если первый элемент как раз и есть наименыший). Потом находится следующий наименьший элемент и меняется местами со вторым элементом, Подобное продолжается до упорядочивания всего массива. Этот метод называется сортировкой выбором, так как при его работе многократно выбирается наименыший из оставшихся элемент. Как видно из реализации, внутренний цикл сортировки выбором просто выполняет сравнения текущего элемента с найденным к этому моменту наименьшим элементом — проще не придумаешь. Перемещение элементов выполняется за пределами внутреннего цикла: каждый обмен помещает один элемент в его окончательную позицию, и, значит, количество обменов равно N. Поэтому время выполнения в основном обусловливается количеством сравнений.

Insertion sort — алгоритм, который часто применяют игроки в бридж для упорядочения своих карт, состоит в следующем: они просматривают карты поочередно и вставляют каждую из них на свое мест среди уже просмотренных (сохраняя упорядоченность). В компьютерной реализации потребуется место для вставки текущего элемента — для этого перед вставкой необходимо сдвинуть большие элементы на одну позицию вправо, а затем вставить текущий элемент на освободившееся место. элементы массива слева от него становятся упорядоченными, и при достижении правого конца упорядоченным становится весь массив. Как и в сортировке выбором, элементы слева от текущего индекса упорядочены в процессе сортировки, но они не (обязательно) находятся в своих окончательных позициях, так как могут быть позже сдвинуты, чтобы освободить место для найденного меньшего элемента. Но после завершения пробега индекса массив оказывается полностью упорядоченным. В отличие от сортировки выбором, время выполнения сортировки вставками зависит от начального упорядочения входных данных. Например, если элементы большого массива уже упорядочены (или почти упорядочены), т сортировка вставками выполняется значительно быстрее, чем в случае первоначального случайного и обратного упорядочения.

Merge sort — алгоритм сортировки, который массив в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Quick sort — алгоритм сортировки является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы (таким образом улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов).

Общая идея алгоритма состоит в следующем:

Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива. От выбора опорного элемента не зависит корректность алгоритма, но в отдельных случаях может сильно зависеть его эффективность (см. ниже).

Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующих друг за другом: «элементы меньшие опорного», «равные» и «большие».

Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

На практике массив обычно делят не на три, а на две части: например, «меньшие опорного» и «равные и большие»; такой подход в общем случае эффективнее, так как упрощает алгоритм разделения.

Hybrid quick sort — гибридная сортировка, использующая в своей основе быструю сортировку и сортировку вставками. В случае когда подмассив исходного массива меньше определенного порога (эмпирическим путем на моей системе было выявлено лучшим значение 128 элементов и меньше) то он сортируется с помощью сортировки вставками, иначе — быстрой сортировкой. Так как быстрая сортировка основана на рекурсивной обработке массива, то гарантированно окажется большое количество подмассивов, которые эффективнее отсортировать сортировкой вставками. Результаты данной оптимизированной сортировки представлены ниже в разделе [4 Эксперименты](#_4_Эксперименты).

Radix sort — сортировка, которая выполняется поразрядно. В данной работе она выполнена с помощью выравнивания по наименее значимой цифре, то есть по правой стороне (LSD-сортировка от английского «Less significant digit»). Основная идея в том чтобы отсортировать массив от наименее значимой цифре к наиболее значимой. Это сортировка в моей реализации наиболее особенная. Это единственная сортировка работающая за линейное время и принимающая в качестве аргумента переменные типа int

# 4 Эксперименты

В ходе данной работы было проведено множество экспериментов. Следует подробнее объяснить методологию тестирования. Для каждой сортировки и для разного числа элементов проводилось порядка 7-10 запусков и выбиралось минимальное время работы. Это было сделано для исключения эффекта «горячих кешей» и получения наиболее достоверных результатов. Также стоит сказать о системе, на которой проводились тесты: процессор AMD Ryzen 7 7700X оперативная память – 32 ГБ.

Также важно пояснить структуру формирования массивов. Случайный массив формируется с помощью встроенной функции rand () языка Си. Прямой (отсортированный) массив путем добавления к псевдослучайному выбранному элементу постоянной константы (каждый следующий элемент больше предыдущего на единицу). Обратный (полностью не отсортированный) массив путем из константы INT\_MAX вычитания псевдослучайного элемента, затем каждый следующий элемент получается уменьшением предыдущего на 1. Теперь, когда была пояснена методология можно перейти непосредственно к тестам.

На рисунке ниже приведен графиком результатов тестирования пузырьковой сортировки:

Несколько аномальным может показать нулевое время работы для прямого массива, данный момент требует пояснения, так как в моей работе использовалась модификация пузырьковой сортировки, в которой проверяется была ли совершена хотя бы одна перестановка, в случае если нет, то программа останавливает работу. И это действительно так, ведь если в течение одного прохода массива не «всплыл», то значит, что все элементы на своих местах и массив уже отсортирован, что позволяет не проводить лишних операций. Исходя из графика можно увидеть очертания параболы, что позволяет предполагать о том, что теоретическая квадратичная сложность подтверждается на практике. Единственно аномальным нахожу тот факт, что в теоретически худшем случае (то есть на обратном массиве) сортировка отработала быстрее чем в среднем случае, возможно это связано с эффективной работой branch prediction unit.

Рис. 1

Следующей тестировалась сортировка выбором. Ничего аномального в результатах не наблюдается, единственный момент, который стоит отметить, что данная сортировка быстрее обработала прямой массив, конечно, данный вопрос требует отдельного детального изучения, но у меня есть единственная реалистично возможная причина этого явления — Свою роль сыграл branch prediction unit (модуль предсказания переходов) за счет чего и снизились временные затраты.

Аналогично, ниже представлен график, исходя из которого, можно предполагать что теоретическая квадратичная сложность подтверждается.

Рис.2

Теперь перейдем к тестированию сортировки вставками. Абсолютно логичным и совпадающим с теоретическими представлениями результаты, в худшем случае программа отработала самое продолжительное время. Исходя из графика можно увидеть очертания параболы, что позволяет предполагать о том, что теоретическая квадратичная сложность подтверждается на практике.

Рис. 3

Некоторым итогом тестирования по трем сортировкам является график представленный ниже, в нем вычислено константное время работы всех трех сортировок выше. Из данного графика можно выявить, что сортировка пузырьком наиболее неэффективная, а сортировка вставками самая быстрая из квадратичных (под квадратичной понимается сортировка со сложность O(n2)).

Рис. 4

Теперь перейдем к более быстрым алгоритмам с теоретической сложностью O(n\*logn). Первым таким алгоритмом будет сортировка слиянием. График зависимости времени работы от числа элементов которой представлен ниже. Данный график действительно очень похож на линию, что позволяет утверждать что теоретическая сложность подтверждается практическим исследованием.

Рис. 5

Следующей представлена быстрая сортировка. Лишь последние два теста показали результаты больше напоминающие квадратичную сложность. У меня есть несколько возможных причин такого результата: из-за такого большого числа элементов сильно возросла глубина рекурсивных вызовов, что вызвало дополнительные временные затраты, еще это может быть связано с выбором опорного элемента, в моей реализации он выбирается методом трех медиан (среднего значения первого, центрального и последнего элементов) и данные псевдослучайные массивы будут худшим случаем в моем способе реализации быстрой сортировки. Возможно одна из этих причин или обе в совокупности и приводят к данному аномальному результату. Если исключить из обзора последние два теста, то теоретическая сложность O(n\*logn) подтверждается.

Рис. 6

Перейдем к гибридной сортировке. Исходя из тестов, задуманная оптимизация, можно сказать не дала ожидаемого результата, минимальное ускорение работу (порядок десятых секунд) можно заметить лишь на последних тестах с самым большим числом элементов. В остальном результаты работы аналогичны результатам стандартной быстрой сортировки. Возможно отсутствие какого-то значимого ускорения связано конкретно с особенностями моего процессора.

Рис. 7

Теперь можно подвезти некий итог по логарифмическим сортировкам (логарифмическими понимаются сортировки с теоретической сложностью O(n\*logn)). Ниже представлен график константного времени этих сортировок.

Рис. 8

Перейдем к единственной линейной сортировке — radix sort. График для случайного массива абсолютно линеен, что соответствует ожиданиям. Время работы также линейно для прямого и обратного массивов, но несколько больше чем для случайного массива, что, наверное, связано с особенностью методики создания прямого и обратного массивов, из-за чего в них содержится больше значащих разрядов что увеличивает время работы.

Рис. 9

Заключительным результатом раздела экспериментов можно назвать нахождение константного времени работы поразрядной сортировки, он представлен ниже.

Рис. 10

Далее тестировалась сортировка Шелла. Исходя из полученных результатов можно заключить, что теоретическая сложность сортировки подтверждается, она действительно напоминает почти прямую с некоторым дополнительным коэффициентом (который равен logn исходя из теоретической сложности). Также стоит отметить довольно быструю работу данной сортировки с прямым и обратным массива, что, несомненно, можно отнести к ее плюсам.

Рис. 11

Далее представлен график константного времени работы данной сортировки.

Рис. 11

Исходя из анализа графиков константного времени всех сортировок можно прийти к выводу, что самая большая константа у поразрядной сортировки, а самая малая у сортировки вставками.

# Заключение

Таким образом в ходе работы удалось достигнуть поставленных задач, а именно:

1. Реализовать пузырьковую сортировку, сортировку вставками, выбором, быструю сортировку, сортировку слиянием и сортировку подсчетом.
2. Реализовать гибридный подход, комбинирующий различные методы сортировки.
3. Провести тестирование программы на различных наборах данных с анализом времени работы и константного времени.

К сожалению, вторую поставленную цель достигнуть не удалось и гибридная быстрая сортировка не дала ожидаемого прироста в производительности.

Можно сделать еще один вывод из данной работы – практически каждая сортировка имеет свою сферу применения и является важной. То, какую сортировку стоит применять решать, конечно же каждому программисту самостоятельно, но осознанный выбор должен быть основан на исходных данных (тип данных, их размер, необходимая скорость и т.д).

# Список литературы

1. Кормен Т. и др. Алгоритмы. Построение и анализ:[пер. с англ.]. – Издательский дом Вильямс, 2009.
2. «Фундаментальные алгоритмы на C. Части 1 - 5. Анализ. Структуры данных. Сортировка. Поиск. Алгоритмы на графах» (2003), Роберт Седжвик – Издательство   
   ДиаСофтЮП
3. <https://ru.wikipedia.org/wiki/>

# Приложение

Прикрепляю исходный код с реализацией всех сортировок ниже:

main.cpp

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "sort.h"

int main() {

srand(1382);

double time\_taken = 0.0;

int choose1, choose2,choose3;

clock\_t start = 0;

clock\_t finish = 0;

printf("Choose sort: \n 1. Bubble sort \n 2.Selection sort \n 3. Insert sort \n ");

printf("4. Merge sort \n 5. Quick sort \n 6. Smart quick sort \n 7. Radix sort \n");

scanf\_s("%d", &choose1);

printf("How many random elements you want sort? ");

scanf\_s("%d", &choose2);

int\* arr\_int = NULL;

double\* arr = NULL;

printf("Do you want test accuracy of sorting? 1 - yes no - else ");

scanf\_s("%d", &choose3);

if (choose1 !=7) arr = make\_random\_arr(choose2);

else arr\_int = make\_random\_arr\_int(choose2);

switch (choose1)

{

case 1:

start = clock();

bubbleSort(arr, choose2);

finish = clock();

break;

case 2:

start = clock();

selectSort(arr, choose2);

finish = clock();

break;

case 3:

start = clock();

insertSort(arr, choose2);

finish = clock();

break;

case 4:

start = clock();

mergeSort(arr,0, choose2-1);

finish = clock();

break;

case 5:

start = clock();

quickSort(arr, 0, choose2 - 1);

finish = clock();

break;

case 6:

start = clock();

hybridQuickSort(arr, 0, choose2 - 1);

finish = clock();

break;

case 7:

start = clock();

radixSort(arr\_int, choose2);

finish = clock();

break;

default:

return -1;

}

time\_taken =((double) (finish - start)) / CLOCKS\_PER\_SEC;

printf("Time taken %lf\n", time\_taken);

if (choose3 == 1) {

if (choose1 != 7) {

if (checkSorted(arr, choose2) == 1) printf("Correct");

else printf("Not correct");

}

else {

if (checkSorted\_int(arr\_int, choose2) == 1) printf("Correct");

else printf("Not correct");

}

}

if (choose1 != 7) delete\_random\_arr(arr);

else delete\_random\_arr\_int(arr\_int);

}

sort.h

#pragma once

#ifndef SORT\_H

#define SORT\_H

double\* make\_random\_arr(int n);

int\* make\_random\_arr\_int(int n);

double\* make\_random\_arr\_sort(int n);

int\* make\_random\_arr\_sort\_int(int n);

double\* make\_random\_arr\_unsort(int n);

int\* make\_random\_arr\_unsort\_int(int n);

void delete\_random\_arr(double\* arr);

void delete\_random\_arr\_int(int\* arr);

int swap(double\* a, double\* b);

void bubbleSort(double\* arr, int elements);

void selectSort(double\* arr, int end);

void insertSort(double\* arr, int end);

void merge(double\* array, int left, int mid, int right);

void mergeSort(double\* array, int left, int right);

int partition(double\* array, int low, int high);

void quickSort(double\* array, int low, int high);

void hybridQuickSort(double\* array, int low, int high);

int getMax(int arr[], int n);

void countSort(int arr[], int n, int exp);

void radixSort(int arr[], int n);

int checkSorted(double\* arr, int n);

int checkSorted\_int(int\* arr, int n);

#endif

sort.cpp

#include <stdio.h>

#include <stdlib.h>

#include "sort.h"

#include <time.h>

double\* make\_random\_arr(int n) {

double\* arr = (double\*)malloc(n \* sizeof(double));

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

arr[i] = rand() \* (21.0 / 100.0);

}

return arr;

}

int\* make\_random\_arr\_int(int n) {

int\* arr = (int\*)malloc(n \* sizeof(int));

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

arr[i] = rand();

}

return arr;

}

double\* make\_random\_arr\_sort(int n) {

double\* arr = (double\*)malloc(n \* sizeof(double));

double first = rand() \* (21.0 / 100.0);

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

arr[i] = first + 2\*i;

}

return arr;

}

int\* make\_random\_arr\_sort\_int(int n) {

int\* arr = (int\*)malloc(n \* sizeof(int));

int first = rand() ;

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

arr[i] = first + i;

}

return arr;

}

double\* make\_random\_arr\_unsort(int n) {

double\* arr = (double\*)malloc(n \* sizeof(double));

double first =RAND\_MAX - rand() \* (21.0 / 100.0);

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

arr[i] = first - i;

}

return arr;

}

int\* make\_random\_arr\_unsort\_int(int n) {

int\* arr = (int\*)malloc(n \* sizeof(int));

int first = INT\_MAX/2;

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

arr[i] = first - i;

}

return arr;

}

void delete\_random\_arr(double\* arr) {

free(arr);

}

void delete\_random\_arr\_int(int\* arr) {

free(arr);

}

int swap(double\* a, double\* b) {

double temp = \*a;

\*a = \*b;

\*b = temp;

return \*a, \* b;

}

void bubbleSort(double\* arr, int end) {

size\_t swap\_count = 0;

for (int i = 0; i < end - 1; i++) {

int flag = 0;

for (int j = 0; j < end - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

swap\_count++;

swap(&arr[j], &arr[j + 1]);

flag = 1;

}

}

if (flag != 1) break;

}

printf("\n%d\n", swap\_count);

}

void selectSort(double\* arr, int end) {

int index;

for (int i = 0; i < end - 1; i++) {

index = i;

for (int j = i + 1; j < end; j++) {

if (arr[index] > arr[j]) index = j;

}

if (index != i) swap(&arr[i], &arr[index]);

}

}

void insertSort(double\* arr, int end) {

for (int i = 1; i < end; i++) {

double key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

void merge(double\* array, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

double\* L = (double\*)malloc(n1 \* sizeof(double));

double\* R = (double\*)malloc(n2 \* sizeof(double));

for (int i = 0; i < n1; i++) {

L[i] = array[left + i];

}

for (int j = 0; j < n2; j++) {

R[j] = array[mid + 1 + j];

}

int i = 0, j = 0, k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

array[k] = L[i];

i++;

}

else {

array[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

array[k] = L[i];

i++;

k++;

}

while (j < n2) {

array[k] = R[j];

j++;

k++;

}

free(L);

free(R);

}

void mergeSort(double\* array, int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(array, left, mid);

mergeSort(array, mid + 1, right);

merge(array, left, mid, right);

}

}

int medianOfThree(double\* array, int low, int high) {

int mid = low + (high - low) / 2;

if (array[low] > array[mid]) swap(&array[low], &array[mid]);

if (array[low] > array[high]) swap(&array[low], &array[high]);

if (array[mid] > array[high]) swap(&array[mid], &array[high]);

return mid;

}

int partition(double\* array, int low, int high) {

int pivotIndex = medianOfThree(array, low, high);

double pivot = array[pivotIndex];

swap(&array[pivotIndex], &array[high]);

int i = low - 1;

for (int j = low; j < high; j++) {

if (array[j] <= pivot) {

i++;

swap(&array[i], &array[j]);

}

}

swap(&array[i + 1], &array[high]);

return i + 1;

}

void quickSort(double\* array, int low, int high) {

if (low < high) {

int pivotIndex = partition(array, low, high);

quickSort(array, low, pivotIndex - 1);

quickSort(array, pivotIndex + 1, high);

}

}

void hybridQuickSort(double\* array, int low, int high) {

int threshold = 128;

if (high - low + 1 <= threshold) {

insertSort(array + low, high - low + 1);

}

else {

quickSort(array, low, high);

}

}

int getMax(int arr[], int n) {

int max = arr[0];

for (int i = 1; i < n; i++) {

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

void countSort(int arr[], int n, int exp) {

int\* output = (int\*)malloc(n \* sizeof(int));

int\* count = (int\*)malloc(10 \* sizeof(int));

for (int i = 0; i < 10; i++) {

count[i] = 0;

}

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

count[(arr[i] / exp) % 10]++;

}

for (int i = 1; i < 10; i++) {

count[i] += count[i - 1];

}

for (int i = n - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

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

arr[i] = output[i];

}

free(output);

free(count);

}

void radixSort(int arr[], int n) {

int max = getMax(arr, n);

for (int exp = 1; max / exp > 0; exp \*= 10) {

countSort(arr, n, exp);

}

}

int checkSorted(double\* arr, int n) {

for (int i = 1; i < n; i++) {

if (arr[i - 1] > arr[i]) {

printf("Not correct\n");

return -1;

}

}

return 1;

}

int checkSorted\_int(int\* arr\_int, int n) {

for (int i = 1; i < n; i++) {

if (arr\_int[i - 1] > arr\_int[i]) {

return -1;

}

}

return 1;

}