Министерство Образования Республики Беларусь

Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Информационных систем и технологий

**СТРУКТУРЫ ДАННЫХ**

**Сортировка записей**

**Лабораторные работы № 12, 13.**

Студент:

1курса 2 группы 1 подгруппы

Марушко Тимофей Фёдорович

Проверяющий:

Белодед Николай Иванович

**ОГЛАВЛЕНИЕ**

[**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 3**](#_Toc196208148)

[**1. АЛГОРИТМЫ. ВВЕДЕНИЕ. 3**](#_Toc196208149)

[**2. АСИМПТОТИЧЕСКАЯ СЛОЖНОСТЬ. 3**](#_Toc196208150)

[**РЕШЕНИЕ ОСНОВНЫХ ЗАДАЧ 12**](#_Toc196208151)

[**РЕШЕНИЕ ДОПОЛНИТЕЛЬНЫХ ЗАДАЧ 15**](#_Toc196208152)

Теоретические сведения

1. Алгоритмы. Введение.

Было подсчитано, что до четверти времени централизованных компьютеров уделяется сортировке данных. Это потому, что намного легче найти значение в массиве, который был заранее отсортирован. В противном случае поиск немного походит на поиск иголки в стоге сена.

**Алгоритмы сортировки** — это алгоритмы для упорядочивания элементов в массиве. Упорядочивание элементов в массиве необходимо для анализа данных или для дальнейшей работы с этим массивом уже других алгоритмов, например, алгоритм бинарного поиска работает только в заранее отсортированном массиве.

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти.

Для сортировки записей **в файле** необходимо считать данные из файла в память, выполнить сортировку и записать отсортированные данные обратно в файл. Для чтения из файла можно использовать оператор >> или функцию getline(). Для записи в файл используется оператор <<.

1. Асимптотическая сложность.

**Сложность алгоритмов** оценивают с помощью нотации О большое (Big O). О большое показывает, как сильно растут затраты на алгоритм, при увеличении количества элементов в массиве.

В анализе алгоритмов, мы часто рассматриваем лучший, худший и средний случаи времени выполнения. Давайте разберем каждый из этих случаев:

**Лучший случай (Best Case)**

Лучший случай времени выполнения алгоритма определяется как минимальное количество времени, которое потребуется алгоритму для выполнения задачи на входных данных. Этот случай редко рассматривается при анализе алгоритмов, так как он обычно представляет собой оптимистичный сценарий, который может не иметь практической ценности.

Пример: В алгоритме линейного поиска лучший случай происходит, когда искомый элемент находится в первой позиции массива. В этом случае алгоритму потребуется выполнить только одну операцию.

**Худший случай (Worst Case)**

Худший случай времени выполнения алгоритма определяется как максимальное количество времени, которое потребуется алгоритму для выполнения задачи на входных данных. Анализ худшего случая является важным, так как он предоставляет верхнюю границу на время выполнения алгоритма, что позволяет оценить, насколько плохо алгоритм может работать в самых неблагоприятных условиях.

Пример: В алгоритме линейного поиска худший случай происходит, когда искомый элемент находится в последней позиции массива или его нет в массиве. В этом случае алгоритму потребуется выполнить N операций, где N - размер массива.

**Средний случай (Average Case)**

Средний случай времени выполнения алгоритма определяется как среднее время выполнения алгоритма на всех возможных входных данных. Анализ среднего случая часто является наиболее полезным, так как он предоставляет представление о том, как алгоритм будет работать в "типичных" условиях.

Пример: В алгоритме линейного поиска средний случай можно оценить как выполнение N/2 операций, если предположить, что искомый элемент равновероятно может быть на любой позиции массива.

**Биг-Омега (Omega) и Биг-Тетта (Theta)** - это два других понятия, которые используются для оценки времени и сложности алгоритмов в С++.

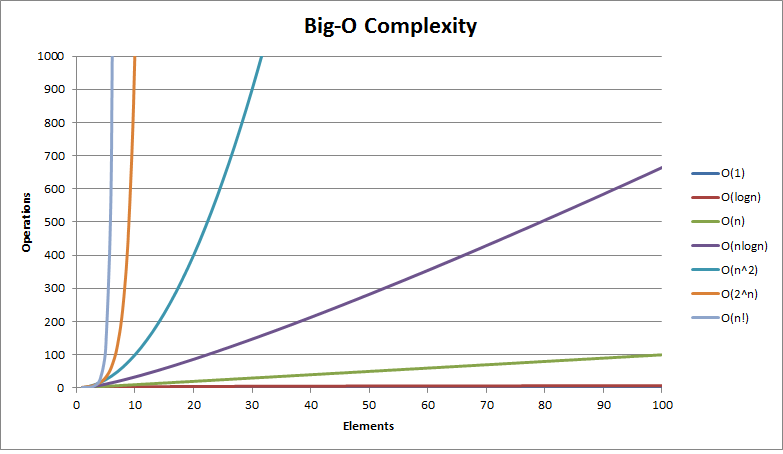
Биг-Омега (Omega) говорит о нижней границе времени выполнения алгоритма. Если мы говорим, что время выполнения алгоритма составляет Ω(n), то это означает, что в наихудшем случае алгоритм будет выполняться не быстрее, чем n единиц времени.

Биг-Тетта (Theta) говорит о верхней границе времени выполнения алгоритма. Если мы говорим, что время выполнения алгоритма составляет Θ(n), то это означает, что в наихудшем случае алгоритм будет выполняться не медленнее, чем n единиц времени и не быстрее, чем n единиц времени.

Важно отметить, что биг-Омега и биг-Тетта могут быть одинаковыми для одного и того же алгоритма, но они также могут быть разными. Поэтому, когда мы говорим о времени выполнения алгоритма, мы должны учитывать и биг-Омега, и биг-Тетта.

Например, если мы знаем, что алгоритм имеет сложность Ω(n) и Θ(n^2), это означает, что в наихудшем случае алгоритм будет выполняться не медленнее, чем n единиц времени, но может быть медленнее, чем n^2 единиц времени. В то же время, в лучшем случае алгоритм будет выполняться не быстрее, чем n^2 единиц времени, но может быть быстрее, чем n^2 единиц времени.

На картинке ниже рассмотрены различные способы роста алгоритмов.



Самыми популярными и частоиспользуемыми алгоритмами сортировки являются: пузырьковая сортировка, быстрая сортировка, сортировка выбором, сортировка вставками, сортировка слиянием.

Ниже представлены лучшие, худшие и средние случаи времени выполнения алгоритмов.



Так же есть шейкерная сортировка, сортировка расчёской, сортировка Шелла, гномья сортировка и многие другие, но используются намного реже, чем представленые выше.(по этой ссылке можно подробнее почитать про другие алгоритмы сортировок: [Описание алгоритмов сортировки и сравнение их производительности / Хабр (habr.com)](https://habr.com/ru/articles/335920/))

**Принцип «разделяй и властвуй»**

При подходе «разделяй и властвуй» задача делится на мелкие подзадачи, каждая из которых решается независимо. При их делении на еще более мелкие подзадачи в конце концов настает момент, когда дальнейшее деление невозможно. Эти мельчайшие «атомарные» подзадачи и решаются. Решения всех подзадач в итоге объединяются, и получается решение исходной задачи.

В целом подход «разделяй и властвуй» рассматривается как трехэтапный процесс:

1. Программа разбивается на независимые подзадачи(части), которые не пересекаются между собой.
2. Каждая подзадача решается отдельно и независимо от других частей.
3. Из отдельных решений подзадач строится решение исходной задачи.

Рассмотрим основные алгоритмы сортировки:

**Сортировка выбором (Selection sort)**

Для того, чтобы отсортировать массив в порядке возрастания, следует на каждой итерации найти элемент с наибольшим значением. С ним нужно поменять местами последний элемент. Следующий элемент с наибольшим значением становится уже на предпоследнее место. Так должно происходить, пока элементы, находящиеся на первых местах в массивe, не окажутся в надлежащем порядке.



(нажми два раза, чтобы оживить картинку)

void SelectionSort(int\* a, int n)

{

int smallest\_id;

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

smallest\_id = i;

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

if (a[j] < a[smallest\_id])

smallest\_id = j;

}

//меняем местами элементы

swap(a[smallest\_id], a[i]);

}

}

**Пузырьковая сортировка (Bubble sort)**

При пузырьковой сортировке сравниваются соседние элементы и меняются местами, если следующий элемент меньше предыдущего. Требуется несколько проходов по данным. Во время первого прохода сравниваются первые два элемента в массиве. Если они не в порядке, они меняются местами и затем сравнивается элементы в следующей паре. При том же условии они так же меняются местами. Таким образом сортировка происходит в каждом цикле пока не будет достигнут конец массива.

Изображение выглядит как снимок экрана, черный, черно-белый, Прямоугольник

Контент, сгенерированный ИИ, может содержать ошибки.



(нажми два раза, чтобы оживить картинку)

void BubbleSort(int\* a, int n)

{

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

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

if (a[i] > a[j]) {

//меняем местами элементы

swap(a[i], a[j]);

}

}

}

}

**Сортировка вставками (Insertion sort)**

При сортировке вставками массив разбивается на две области: упорядоченную и и неупорядоченную. Изначально весь массив является неупорядоченной областью. При первом проходе первый элемент из неупорядоченной области изымается и помещается в правильном положении в упорядоченной области.  
На каждом проходе размер упорядоченной области возрастает на 1, а размер неупорядоченной области сокращается на 1.  
Основной цикл работает в интервале от 1 до N-1. На j-й итерации элемент [i] вставлен в правильное положение в упорядоченной области. Это сделано путем сдвига всех элементов упорядоченной области, которые больше, чем [i], на одну позицию вправо. [i] вставляется в интервал между теми элементами, которые меньше [i], и теми, которые больше [i].

Изображение выглядит как снимок экрана, черный

Контент, сгенерированный ИИ, может содержать ошибки.



(нажми два раза, чтобы оживить картинку)

void insertionSort(int data[], int lenD)

{

int key = 0;

int i = 0;

for (int j = 1; j < lenD; j++) {

key = data[j];

i = j - 1;

while (i >= 0 && data[i] > key) {

data[i + 1] = data[i];

i = i - 1;

data[i + 1] = key;

}

}

}

**Сортировка слиянием (Merge sort)**

При рекурсивной сортировке слиянием массив сначала разбивается на мелкие кусочки - на первом этапе - на состоящие из одного элемента. Затем эти кусочки объединяются в более крупные кусочки - по два элемента и элементы при этом сравниваются, а в результате в новом кусочке меньший элемент занимает место слева, а больший - справа. Далее происходит слияние в ещё более крупные кусочки и так до конца алгоритма, когда все кусочки будут объединены в один, уже отсортированный массив.

Изображение выглядит как снимок экрана, черный, черно-белый, Прямоугольник

Контент, сгенерированный ИИ, может содержать ошибки.



(нажми два раза, чтобы оживить картинку)

void mergeSort(int data[], int lenD)

{

if (lenD > 1) {

int middle = lenD / 2;

int rem = lenD - middle;

int\* L = new int[middle];

int\* R = new int[rem];

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

if (i < middle) {

L[i] = data[i];

}

else {

R[i - middle] = data[i];

}

}

mergeSort(L, middle);

mergeSort(R, rem);

merge(data, lenD, L, middle, R, rem);

}

}

void merge(int merged[], int lenD, int L[], int lenL, int R[], int lenR) {

int i = 0;

int j = 0;

while (i < lenL || j < lenR) {

if (i < lenL & j < lenR) {

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

merged[i + j] = L[i];

i++;

}

else {

merged[i + j] = R[j];

j++;

}

}

else if (i < lenL) {

merged[i + j] = L[i];

i++;

}

else if (j < lenR) {

merged[i + j] = R[j];

j++;

}

}

}

**Быстрая сортировка (Quick sort)**

Быстрая сортировка использует алгоритм "разделяй и властвуй". Она начинается с разбиения исходного массива на две области. Эти части находятся слева и справа от отмеченного элемента, называемого опорным. В конце процесса одна часть будет содержать элементы меньшие, чем опорный, а другая часть будет содержать элементы больше опорного.

Изображение выглядит как снимок экрана, черный, Прямоугольник, черно-белый

Контент, сгенерированный ИИ, может содержать ошибки.



(нажми два раза, чтобы оживить картинку)

void QuickSort(int\* a, int left, int right)

{

int l\_hold = left, // левая граница

r\_hold = right, // правая граница

pivot = a[left]; // разрешающий элемент

while (left < right) // пока границы не сомкнутся

{

while ((a[right] >= pivot) && (left < right))

right--; // сдвигаем правую границу пока элемент [right] больше [pivot]

if (left != right) // если границы не сомкнулись

{

a[left] = a[right]; // перемещаем элемент [right] на место разрешающего

left++; // сдвигаем левую границу вправо

}

while ((a[left] <= pivot) && (left < right))

left++; // сдвигаем левую границу пока элемент [left] меньше [pivot]

if (left != right) // если границы не сомкнулись

{

a[right] = a[left]; // перемещаем элемент [left] на место [right]

right--; // сдвигаем правую границу вправо

}

}

a[left] = pivot; // ставим разрешающий элемент на место

pivot = left;

left = l\_hold;

right = r\_hold;

if (left < pivot) // Рекурсивно вызываем сортировку для левой и правой части массива

QuickSort(a, left, pivot - 1);

if (right > pivot)

QuickSort(a, pivot + 1, right);

}

Решение ОСНОВНЫХ ЗАДАЧ

**Условие:**

Реализовать функцию с выбором сортироки, которая будет использоваться для сортировки вашего массива структур по каким-либо параметрам(т.е. сортировать, например, по цене), выбор будет происходить среди следующих алгоритмов: *пузырьковая сортировка, быстрая сортировка, сортировка выбором, сортировка вставками, сортировка слиянием.*

**Решение:**

*//main.cpp*

*#include* <iostream>

*#include* <fstream>

*#include* <vector>

*#include* <string>

*#include* <sstream>

*#include* <chrono>

*#include* "sort\_algorithms.h"

using namespace std;

using namespace std::chrono;

*// Data structures*

struct Teacher {

int id;

string name;

string subject;

int experience;

};

struct Phone {

int id;

string number;

string model;

int year;

};

struct TeacherPhone {

int teacher\_id;

int phone\_id;

};

*// Functions for loading data*

vector<Teacher> load\_teachers(const string& *filename*) {

vector<Teacher> teachers;

ifstream file(filename);

string line;

*while* (getline(file, line)) {

stringstream ss(line);

Teacher teacher;

ss >> teacher.id >> teacher.name >> teacher.subject >> teacher.experience;

teachers.push\_back(teacher);

}

*return* teachers;

}

vector<Phone> load\_phones(const string& *filename*) {

vector<Phone> phones;

ifstream file(filename);

string line;

*while* (getline(file, line)) {

stringstream ss(line);

Phone phone;

ss >> phone.id >> phone.number >> phone.model >> phone.year;

phones.push\_back(phone);

}

*return* phones;

}

vector<TeacherPhone> load\_relations(const string& *filename*) {

vector<TeacherPhone> relations;

ifstream file(filename);

string line;

*while* (getline(file, line)) {

stringstream ss(line);

TeacherPhone relation;

ss >> relation.teacher\_id >> relation.phone\_id;

relations.push\_back(relation);

}

*return* relations;

}

*// Comparison functions for structures*

int compare\_teacher\_name(const void\* *a*, const void\* *b*) {

*return* ((Teacher\*)a)->name.compare(((Teacher\*)b)->name);

}

int compare\_teacher\_experience(const void\* *a*, const void\* *b*) {

*return* ((Teacher\*)a)->experience - ((Teacher\*)b)->experience;

}

int compare\_phone\_number(const void\* *a*, const void\* *b*) {

*return* ((Phone\*)a)->number.compare(((Phone\*)b)->number);

}

int compare\_phone\_year(const void\* *a*, const void\* *b*) {

*return* ((Phone\*)a)->year - ((Phone\*)b)->year;

}

*// Function for printing sort results*

void print\_sort\_results(const SortResult& *result*) {

cout << result.name << ": " << result.time << " seconds\n";

}

*// Function for printing teachers*

void print\_teachers(const vector<Teacher>& *teachers*) {

cout << "\nTeachers list:\n";

cout << "ID\t\tName\t\tSubject\t\tExperience\n";

*for* (const auto& teacher : teachers) {

cout << teacher.id << "\t\t" << teacher.name << "\t\t" << teacher.subject << "\t\t"

<< teacher.experience << "\n";

}

}

*// Function for printing phones*

void print\_phones(const vector<Phone>& *phones*) {

cout << "\nPhones list:\n";

cout << "ID\t\tNumber\t\tModel\t\tYear\n";

*for* (const auto& phone : phones) {

cout << phone.id << "\t\t" << phone.number << "\t\t" << phone.model << "\t\t"

<< phone.year << "\n";

}

}

*// Function for displaying teacher information*

void display\_teacher\_info(const vector<Teacher>& *teachers*, const vector<Phone>& *phones*, const vector<TeacherPhone>& *relations*) {

cout << "\n=== TEACHER INFORMATION ===\n";

cout << "Total number of teachers: " << teachers.size() << "\n\n";

*if* (teachers.empty()) {

cout << "No teacher data available.\n";

*return*;

}

*// Display all teachers*

cout << "All teachers:\n";

*for* (size\_t i = 0; i < teachers.size(); ++i) {

cout << i + 1 << ". " << teachers[i].name << " - " << teachers[i].subject

<< ", " << teachers[i].experience << " years\n";

*// Find phones for this teacher*

cout << " Phones: ";

bool has\_phones = false;

*for* (const auto& relation : relations) {

*if* (relation.teacher\_id == teachers[i].id) {

*for* (const auto& phone : phones) {

*if* (phone.id == relation.phone\_id) {

cout << phone.number << " (" << phone.model << ", " << phone.year << ") ";

has\_phones = true;

*break*;

}

}

}

}

*if* (!has\_phones) {

cout << "No phones assigned";

}

cout << "\n";

}

}

*// Function for displaying phone information*

void display\_phone\_info(const vector<Phone>& *phones*, const vector<Teacher>& *teachers*, const vector<TeacherPhone>& *relations*) {

cout << "\n=== PHONE INFORMATION ===\n";

cout << "Total number of phone records: " << phones.size() << "\n\n";

*if* (phones.empty()) {

cout << "No phone data available.\n";

*return*;

}

*// Display all phones*

cout << "All phone records:\n";

*for* (size\_t i = 0; i < phones.size(); ++i) {

cout << i + 1 << ". " << phones[i].number << " (" << phones[i].model << ", "

<< phones[i].year << ")\n";

*// Find teachers for this phone*

cout << " Assigned to: ";

bool has\_teachers = false;

*for* (const auto& relation : relations) {

*if* (relation.phone\_id == phones[i].id) {

*for* (const auto& teacher : teachers) {

*if* (teacher.id == relation.teacher\_id) {

cout << teacher.name << " ";

has\_teachers = true;

*break*;

}

}

}

}

*if* (!has\_teachers) {

cout << "No teachers assigned";

}

cout << "\n";

}

}

*// Function for sorting teachers*

void sort\_teachers(vector<Teacher>& *teachers*) {

int choice;

cout << "\nSelect sorting criteria for teachers:\n";

cout << "1. By name\n";

cout << "2. By experience\n";

cout << "Your choice: ";

cin >> choice;

vector<void\*> teacher\_ptrs;

*for* (auto& teacher : teachers) {

teacher\_ptrs.push\_back(&teacher);

}

SortResult result;

*switch* (choice) {

*case* 1: {

cout << "\nSelect sorting algorithm:\n";

cout << "1. Bubble Sort\n";

cout << "2. Quick Sort\n";

cout << "3. Selection Sort\n";

cout << "4. Insertion Sort\n";

cout << "5. Merge Sort\n";

cout << "Your choice: ";

int algo\_choice;

cin >> algo\_choice;

*switch* (algo\_choice) {

*case* 1:

result = bubble\_sort(teacher\_ptrs, compare\_teacher\_name);

*break*;

*case* 2:

result = quick\_sort(teacher\_ptrs, compare\_teacher\_name);

*break*;

*case* 3:

result = selection\_sort(teacher\_ptrs, compare\_teacher\_name);

*break*;

*case* 4:

result = insertion\_sort(teacher\_ptrs, compare\_teacher\_name);

*break*;

*case* 5:

result = merge\_sort(teacher\_ptrs, compare\_teacher\_name);

*break*;

*default*:

cout << "Invalid choice!\n";

*return*;

}

print\_sort\_results(result);

print\_teachers(teachers);

*break*;

}

*case* 2: {

cout << "\nSelect sorting algorithm:\n";

cout << "1. Bubble Sort\n";

cout << "2. Quick Sort\n";

cout << "3. Selection Sort\n";

cout << "4. Insertion Sort\n";

cout << "5. Merge Sort\n";

cout << "Your choice: ";

int algo\_choice;

cin >> algo\_choice;

*switch* (algo\_choice) {

*case* 1:

result = bubble\_sort(teacher\_ptrs, compare\_teacher\_experience);

*break*;

*case* 2:

result = quick\_sort(teacher\_ptrs, compare\_teacher\_experience);

*break*;

*case* 3:

result = selection\_sort(teacher\_ptrs, compare\_teacher\_experience);

*break*;

*case* 4:

result = insertion\_sort(teacher\_ptrs, compare\_teacher\_experience);

*break*;

*case* 5:

result = merge\_sort(teacher\_ptrs, compare\_teacher\_experience);

*break*;

*default*:

cout << "Invalid choice!\n";

*return*;

}

print\_sort\_results(result);

print\_teachers(teachers);

*break*;

}

*default*:

cout << "Invalid choice!\n";

*return*;

}

}

*// Function for sorting phones*

void sort\_phones(vector<Phone>& *phones*) {

int choice;

cout << "\nSelect sorting criteria for phones:\n";

cout << "1. By number\n";

cout << "2. By year\n";

cout << "Your choice: ";

cin >> choice;

vector<void\*> phone\_ptrs;

*for* (auto& phone : phones) {

phone\_ptrs.push\_back(&phone);

}

SortResult result;

*switch* (choice) {

*case* 1: {

cout << "\nSelect sorting algorithm:\n";

cout << "1. Bubble Sort\n";

cout << "2. Quick Sort\n";

cout << "3. Selection Sort\n";

cout << "4. Insertion Sort\n";

cout << "5. Merge Sort\n";

cout << "Your choice: ";

int algo\_choice;

cin >> algo\_choice;

*switch* (algo\_choice) {

*case* 1:

result = bubble\_sort(phone\_ptrs, compare\_phone\_number);

*break*;

*case* 2:

result = quick\_sort(phone\_ptrs, compare\_phone\_number);

*break*;

*case* 3:

result = selection\_sort(phone\_ptrs, compare\_phone\_number);

*break*;

*case* 4:

result = insertion\_sort(phone\_ptrs, compare\_phone\_number);

*break*;

*case* 5:

result = merge\_sort(phone\_ptrs, compare\_phone\_number);

*break*;

*default*:

cout << "Invalid choice!\n";

*return*;

}

print\_sort\_results(result);

print\_phones(phones);

*break*;

}

*case* 2: {

cout << "\nSelect sorting algorithm:\n";

cout << "1. Bubble Sort\n";

cout << "2. Quick Sort\n";

cout << "3. Selection Sort\n";

cout << "4. Insertion Sort\n";

cout << "5. Merge Sort\n";

cout << "Your choice: ";

int algo\_choice;

cin >> algo\_choice;

*switch* (algo\_choice) {

*case* 1:

result = bubble\_sort(phone\_ptrs, compare\_phone\_year);

*break*;

*case* 2:

result = quick\_sort(phone\_ptrs, compare\_phone\_year);

*break*;

*case* 3:

result = selection\_sort(phone\_ptrs, compare\_phone\_year);

*break*;

*case* 4:

result = insertion\_sort(phone\_ptrs, compare\_phone\_year);

*break*;

*case* 5:

result = merge\_sort(phone\_ptrs, compare\_phone\_year);

*break*;

*default*:

cout << "Invalid choice!\n";

*return*;

}

print\_sort\_results(result);

print\_phones(phones);

*break*;

}

*default*:

cout << "Invalid choice!\n";

*return*;

}

}

int main() {

*// Load data*

auto teachers = load\_teachers("teachers.txt");

auto phones = load\_phones("phones.txt");

auto relations = load\_relations("foreigingkeys.txt");

cout << "Data loaded successfully\n";

int choice;

*do* {

cout << "\nMenu:\n";

cout << "1. Sort teachers\n";

cout << "2. Sort phones\n";

cout << "3. Display teacher information\n";

cout << "4. Display phone information\n";

cout << "5. Exit\n";

cout << "Your choice: ";

cin >> choice;

*switch* (choice) {

*case* 1:

sort\_teachers(teachers);

*break*;

*case* 2:

sort\_phones(phones);

*break*;

*case* 3:

display\_teacher\_info(teachers, phones, relations);

*break*;

*case* 4:

display\_phone\_info(phones, teachers, relations);

*break*;

*case* 5:

cout << "Program terminated.\n";

*break*;

*default*:

cout << "Invalid choice!\n";

}

} *while* (choice != 5);

*return* 0;

}

*//sort\_algorithms.cpp*

*#include* "sort\_algorithms.h"

*// Comparison function for integers*

int compare\_int(const void\* *a*, const void\* *b*) {

*return* (\*(int\*)a - \*(int\*)b);

}

*// Comparison function for strings*

int compare\_string(const void\* *a*, const void\* *b*) {

*return* (\*(string\*)a).compare(\*(string\*)b);

}

*// Bubble sort*

SortResult bubble\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*)) {

auto start = high\_resolution\_clock::now();

*for* (size\_t i = 0; i < arr.size() - 1; ++i) {

*for* (size\_t j = 0; j < arr.size() - i - 1; ++j) {

*if* (compare(arr[j], arr[j + 1]) > 0) {

void\* temp = arr[j];

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

arr[j + 1] = temp;

}

}

}

auto end = high\_resolution\_clock::now();

*return* {duration<double>(end - start).count(), "Bubble Sort"};

}

*// Quick sort*

int partition(vector<void\*>& *arr*, int *low*, int *high*, int (\**compare*)(const void\*, const void\*)) {

void\* pivot = arr[high];

int i = low - 1;

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

*if* (compare(arr[j], pivot) <= 0) {

++i;

void\* temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

void\* temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

*return* i + 1;

}

void quick\_sort\_helper(vector<void\*>& *arr*, int *low*, int *high*, int (\**compare*)(const void\*, const void\*)) {

*if* (low < high) {

int pi = partition(arr, low, high, compare);

quick\_sort\_helper(arr, low, pi - 1, compare);

quick\_sort\_helper(arr, pi + 1, high, compare);

}

}

SortResult quick\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*)) {

auto start = high\_resolution\_clock::now();

quick\_sort\_helper(arr, 0, arr.size() - 1, compare);

auto end = high\_resolution\_clock::now();

*return* {duration<double>(end - start).count(), "Quick Sort"};

}

*// Selection sort*

SortResult selection\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*)) {

auto start = high\_resolution\_clock::now();

*for* (size\_t i = 0; i < arr.size() - 1; ++i) {

size\_t min\_idx = i;

*for* (size\_t j = i + 1; j < arr.size(); ++j) {

*if* (compare(arr[j], arr[min\_idx]) < 0) {

min\_idx = j;

}

}

void\* temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

auto end = high\_resolution\_clock::now();

*return* {duration<double>(end - start).count(), "Selection Sort"};

}

*// Insertion sort*

SortResult insertion\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*)) {

auto start = high\_resolution\_clock::now();

*for* (size\_t i = 1; i < arr.size(); ++i) {

void\* key = arr[i];

int j = i - 1;

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

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

--j;

}

arr[j + 1] = key;

}

auto end = high\_resolution\_clock::now();

*return* {duration<double>(end - start).count(), "Insertion Sort"};

}

*// Merge sort*

void merge(vector<void\*>& *arr*, int *left*, int *mid*, int *right*, int (\**compare*)(const void\*, const void\*)) {

int n1 = mid - left + 1;

int n2 = right - mid;

vector<void\*> L(arr.begin() + left, arr.begin() + left + n1);

vector<void\*> R(arr.begin() + mid + 1, arr.begin() + mid + 1 + n2);

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

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

*if* (compare(L[i], R[j]) <= 0) {

arr[k] = L[i];

++i;

} *else* {

arr[k] = R[j];

++j;

}

++k;

}

*while* (i < n1) {

arr[k] = L[i];

++i;

++k;

}

*while* (j < n2) {

arr[k] = R[j];

++j;

++k;

}

}

void merge\_sort\_helper(vector<void\*>& *arr*, int *left*, int *right*, int (\**compare*)(const void\*, const void\*)) {

*if* (left < right) {

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

merge\_sort\_helper(arr, left, mid, compare);

merge\_sort\_helper(arr, mid + 1, right, compare);

merge(arr, left, mid, right, compare);

}

}

SortResult merge\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*)) {

auto start = high\_resolution\_clock::now();

merge\_sort\_helper(arr, 0, arr.size() - 1, compare);

auto end = high\_resolution\_clock::now();

*return* {duration<double>(end - start).count(), "Merge Sort"};

}

*//sort\_algorithms.h*

*#ifndef* SORT\_ALGORITHMS\_H

*#define* SORT\_ALGORITHMS\_H

*#include* <vector>

*#include* <iostream>

*#include* <string>

*#include* <chrono>

using namespace std;

using namespace std::chrono;

struct SortResult {

double time;

string name;

};

SortResult bubble\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*));

SortResult quick\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*));

SortResult selection\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*));

SortResult insertion\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*));

SortResult merge\_sort(vector<void\*>& *arr*, int (\**compare*)(const void\*, const void\*));

*#endif*