ФГБОУ ВО “Чувашский государственный университет им.

И.Н. Ульянова”

Кафедра вычислительной техники

Лабораторная работа №5

по дисциплине

«Структуры и алгоритмы обработки данных»

Вариант 9.

Выполнил:

Студент гр. ИВТ-41-22

Иванов Владимир Сергеевич

Проверил:

Павлов Леонид Александрович

Чебоксары, 2024

Цель работы: изучить основные методы сортировки, получить практические навыки программирования задач сортировки, получить навыки экспериментальных исследований алгоритмов.

Подготовка к работе:

**Реализуемые в моем варианте алгоритмы:**

1. Простая сортировка вставками.

2. Сортировка бинарными вставками.

3. Шейкер-сортировка.

4. Быстрая сортировка (рекурсивный вариант).

5. Быстрая сортировка (итерационный вариант).

6. Простая сортировка выбором.

7. Сортировка подсчетом (перечислением).

8. Сортировка естественным двухпутевым слиянием.

9. Сортировка слиянием списков для простого двухпутевого слияния.

Алгоритмы на псевдокоде для каждого метода упорядочивания:

**1.** Простая сортировка вставками.

int comparisons\_1 = 0;

int swaps\_1 = 0;

void insertionSort(std::vector<int>& arr, int& comparisons\_1, int& swaps\_1) {

int n = arr.size();

auto start = std::chrono::high\_resolution\_clock::now(); // Запуск таймера

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

int key = arr[i];

int j = i - 1;

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

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

j = j - 1;

swaps\_1++; // Увеличение счетчика перестановок

comparisons\_1++; // Увеличение счетчика сравнений

}

arr[j + 1] = key;

swaps\_1++; // Увеличение счетчика перестановок

}

auto end = std::chrono::high\_resolution\_clock::now(); // Остановка таймера

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start); // Вычисление времени выполнения в наносекундах

std::cout << "Insertion sort" << " " << comparisons\_1 << " " <<

swaps\_1 << " " << duration.count()/std::pow(10,9) << std::endl;

}

**2.** Сортировка бинарными вставками.

int comparisons\_2 = 0;

int swaps\_2 = 0;

// Функция для бинарного поиска места вставки элемента в упорядоченном массиве

int binarySearch(const std::vector<int>& arr, int key, int low, int high, int& comparisons\_2) {

while (low <= high) {

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

comparisons\_2++;

if (arr[mid] == key)

return mid;

else if (arr[mid] < key)

low = mid + 1;

else

high = mid - 1;

}

return low;

}

// Сортировка бинарными вставками

void binaryInsertionSort(std::vector<int>& arr, int& comparisons\_2, int& swaps\_2) {

int n = arr.size();

auto start = std::chrono::high\_resolution\_clock::now(); // Запуск таймера

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

int key = arr[i];

// Используем бинарный поиск для нахождения позиции вставки

int j = binarySearch(arr, key, 0, i - 1, comparisons\_2);

// Перемещаем все элементы с позиции j до i на одну позицию вправо

for (int k = i - 1; k >= j; --k) {

arr[k + 1] = arr[k];

swaps\_2++;

}

// Вставляем key на найденную позицию

arr[j] = key;

}

auto end = std::chrono::high\_resolution\_clock::now(); // Остановка таймера

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start); // Вычисление времени выполнения в наносекундах

std::cout << "Binary Insertion sort" << " " << comparisons\_2 << " " <<

swaps\_2 << " " << duration.count() / std::pow(10, 9) << std::endl;

}

**3**.Шейкер-сортировка.

int comparisons\_3 = 0;

int swaps\_3 = 0;

void shakerSort(std::vector<int>& arr, int& comparisons\_3, int& swaps\_3) {

bool swapped = true;

int start = 0;

int end = arr.size() - 1;

auto start\_time = std::chrono::high\_resolution\_clock::now(); // Запуск таймера

while (swapped) {

swapped = false;

// Проход вперед

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

comparisons\_3++;

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

swaps\_3++;

std::swap(arr[i], arr[i + 1]);

swapped = true;

}

}

// Если нет обменов, массив уже отсортирован

if (!swapped)

break;

// Уменьшаем область, так как самый большой элемент уже находится на конце

--end;

// Проход назад

for (int i = end - 1; i >= start; --i) {

comparisons\_3++;

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

swaps\_3++;

std::swap(arr[i], arr[i + 1]);

swapped = true;

}

}

// Увеличиваем область, так как самый маленький элемент уже находится в начале

++start;

}

auto end\_1 = std::chrono::high\_resolution\_clock::now(); // Остановка таймера

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_1 - start\_time); // Вычисление времени выполнения в наносекундах

std::cout << "Shaker sort" << " " << comparisons\_3 << " " <<

swaps\_3 << " " << duration.count() / std::pow(10, 9) << std::endl;

}

**4,5.** Быстрая сортировка

int comparisons\_4 = 0;

int swaps\_4 = 0;

// Функция для разделения массива по опорному элементу

int partition\_1(std::vector<int>& arr, int low, int high) {

int pivot = arr[high]; // Опорный элемент

int i = low - 1; // Индекс элемента, меньшего опорного

auto start\_time = std::chrono::high\_resolution\_clock::now(); // Запуск таймера

for (int j = low; j <= high - 1; ++j) {

comparisons\_4++;

// Если текущий элемент меньше или равен опорному

if (arr[j] <= pivot) {

// Увеличиваем индекс меньшего элемента и меняем местами элементы

++i;

swaps\_4++;

std::swap(arr[i], arr[j]);

}

}

// Помещаем опорный элемент на правильное место

swaps\_4++;

std::swap(arr[i + 1], arr[high]);

return i + 1;

}

// Функция быстрой сортировки

void quickSort\_1(std::vector<int>& arr, int low, int high) {

if (low < high) {

// Разделяем массив и получаем индекс опорного элемента

int pi = partition\_1(arr, low, high);

// Рекурсивно сортируем элементы до опорного элемента и после него

quickSort\_1(arr, low, pi - 1);

quickSort\_1(arr, pi + 1, high);

}

}

// Функция быстрой сортировки

void quickSort\_2(std::vector<int>& arr, int low, int high, int& comparisons, int& swaps) {

std::stack<std::pair<int, int>> stack;

stack.push({ low, high });

while (!stack.empty()) {

int start = stack.top().first;

int end = stack.top().second;

stack.pop();

int pivotIndex = partition\_2(arr, start, end, comparisons, swaps);

// Если есть элементы слева от опорного, добавляем их в стек

if (pivotIndex - 1 > start)

stack.push({ start, pivotIndex - 1 });

// Если есть элементы справа от опорного, добавляем их в стек

if (pivotIndex + 1 < end)

stack.push({ pivotIndex + 1, end });

}

}

6.  Простая сортировка выбором.

int comparisons\_6 = 0;

int swaps\_6 = 0;

void selectionSort(std::vector<int>& arr, int& comparisons\_6, int& swaps\_6) {

int n = arr.size();

auto start = std::chrono::high\_resolution\_clock::now(); // Запуск таймера

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

// Находим индекс минимального элемента в оставшейся части массива

int minIndex = i;

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

comparisons\_6++;

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Обменяем текущий элемент с минимальным

if (i != minIndex) {

std::swap(arr[i], arr[minIndex]);

++swaps\_6; // Увеличиваем счетчик перестановок

}

}

auto end = std::chrono::high\_resolution\_clock::now(); // Остановка таймера

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start); // Вычисление времени выполнения в наносекундах

std::cout << "Simple Selection sort" << " " << comparisons\_6 << " " <<

swaps\_6 << " " << duration.count() / std::pow(10, 9) << std::endl;

}

7. Сортировка подсчетом (перечислением).

int comparisons\_7 = 0;

int swaps\_7 = 0;

void countingSort(std::vector<int>& arr) {

auto start\_time = std::chrono::high\_resolution\_clock::now(); // Запуск таймера

int maxElement = \*std::max\_element(arr.begin(), arr.end());

int minElement = \*std::min\_element(arr.begin(), arr.end());

int range = maxElement - minElement + 1;

// Создаем временный вектор для подсчета вхождений каждого элемента

std::vector<int> count(range, 0);

// Подсчитываем вхождения каждого элемента

for (int num : arr) {

++count[num - minElement];

}

// Строим отсортированный массив из подсчетов

int index = 0;

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

for (int j = 0; j < count[i]; ++j) {

arr[index++] = i + minElement;

swaps\_7++;

}

}

auto end\_time = std::chrono::high\_resolution\_clock::now(); // Остановка таймера

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_time - start\_time); // Вычисление времени выполнения в наносекундах

std::cout << "Counting sort" << " " << comparisons\_7 << " " << swaps\_7 << " " << duration.count() / std::pow(10, 9) << std::endl;

}

8. Сортировка простым двухпутевым слиянием.

int comparisons\_8 = 0;

int swaps\_8 = 0;

// Функция для слияния двух соседних отсортированных подсписков

void merge\_1(std::vector<int>&arr, int left, int mid, int right, int& comparisons, int& swaps) {

int n1 = mid - left + 1; // Размер первого подсписка

int n2 = right - mid; // Размер второго подсписка

// Создаем временные векторы для хранения элементов подсписков

std::vector<int> leftArr(n1);

std::vector<int> rightArr(n2);

// Копируем данные во временные векторы

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

leftArr[i] = arr[left + i];

}

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

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

}

// Инициализируем индексы для обоих временных векторов и исходного массива

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

// Слияние временных векторов обратно в исходный массив

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

comparisons\_8++;

if (leftArr[i] <= rightArr[j]) {

arr[k] = leftArr[i];

++i;

}

else {

arr[k] = rightArr[j];

++j;

}

++k;

}

// Копируем оставшиеся элементы leftArr (если они есть) обратно в arr

while (i < n1) {

arr[k] = leftArr[i];

++i;

++k;

swaps\_8++;

}

// Копируем оставшиеся элементы rightArr (если они есть) обратно в arr

while (j < n2) {

arr[k] = rightArr[j];

++j;

++k;

swaps\_8++;

}

}

// Функция для сортировки естественным двухпутевым слиянием

void naturalMergeSort(std::vector<int>& arr, int& comparisons, int& swaps) {

int n = arr.size();

int left, mid, right;

auto start\_time = std::chrono::high\_resolution\_clock::now(); // Запуск таймера

// Пока есть несортированные подсписки в массиве

while (true) {

left = 0;

// Находим начало и конец первого подсписка

while (left < n - 1) {

int i = left;

while (i < n - 1 && arr[i] <= arr[i + 1]) {

++i;

}

mid = i;

// Если мы достигли конца массива, значит, массив уже отсортирован

if (mid == n - 1) {

return;

}

// Находим начало и конец второго подсписка

while (i < n - 1 && arr[i] > arr[i + 1]) {

++i;

}

right = i;

// Выполняем слияние двух подсписков

merge\_1(arr, left, mid, right, comparisons, swaps);

// Переходим к следующему подсписку

left = right + 1;

}

}

std::cout << "jmfewikosjfveiowjfiowo" << std::endl;

auto end = std::chrono::high\_resolution\_clock::now(); // Остановка таймера

auto duration = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start\_time); // Вычисление времени выполнения в наносекундах

std::cout << "Natural merge sort" << " " << comparisons\_8 << " " <<

swaps\_8 << " " << duration.count() / std::pow(10, 9) << std::endl;

}

9. Сортировка слиянием списков для простого двухпутевого слияния.

int comparisons\_9 = 0;

int swaps\_9 = 0;

// Функция для слияния двух отсортированных списков

vector<int> merge\_2(const vector<int>& left, const vector<int>& right, int& comparisons, int& swaps) {

vector<int> merged;

int leftSize = left.size();

int rightSize = right.size();

int i = 0, j = 0;

// Слияние элементов из двух списков

while (i < leftSize && j < rightSize) {

comparisons\_9++;

if (left[i] <= right[j]) {

merged.push\_back(left[i]);

++i;

}

else {

merged.push\_back(right[j]);

++j;

}

}

// Добавление оставшихся элементов из левого списка

while (i < leftSize) {

merged.push\_back(left[i]);

++i;

swaps\_9++;

}

// Добавление оставшихся элементов из правого списка

while (j < rightSize) {

merged.push\_back(right[j]);

++j;

swaps\_9++;

}

return merged;

}

// Функция для сортировки слиянием

std::vector<int> mergeSort(const std::vector<int>& arr, int& comparisons, int& swaps) {

int n = arr.size();

// Базовый случай: если список пустой или содержит один элемент, он уже отсортирован

if (n <= 1)

return arr;

// Разделение списка на два подсписка

int mid = n / 2;

std::vector<int> left(arr.begin(), arr.begin() + mid);

std::vector<int> right(arr.begin() + mid, arr.end());

// Рекурсивное применение сортировки слиянием к двум подспискам

left = mergeSort(left, comparisons, swaps);

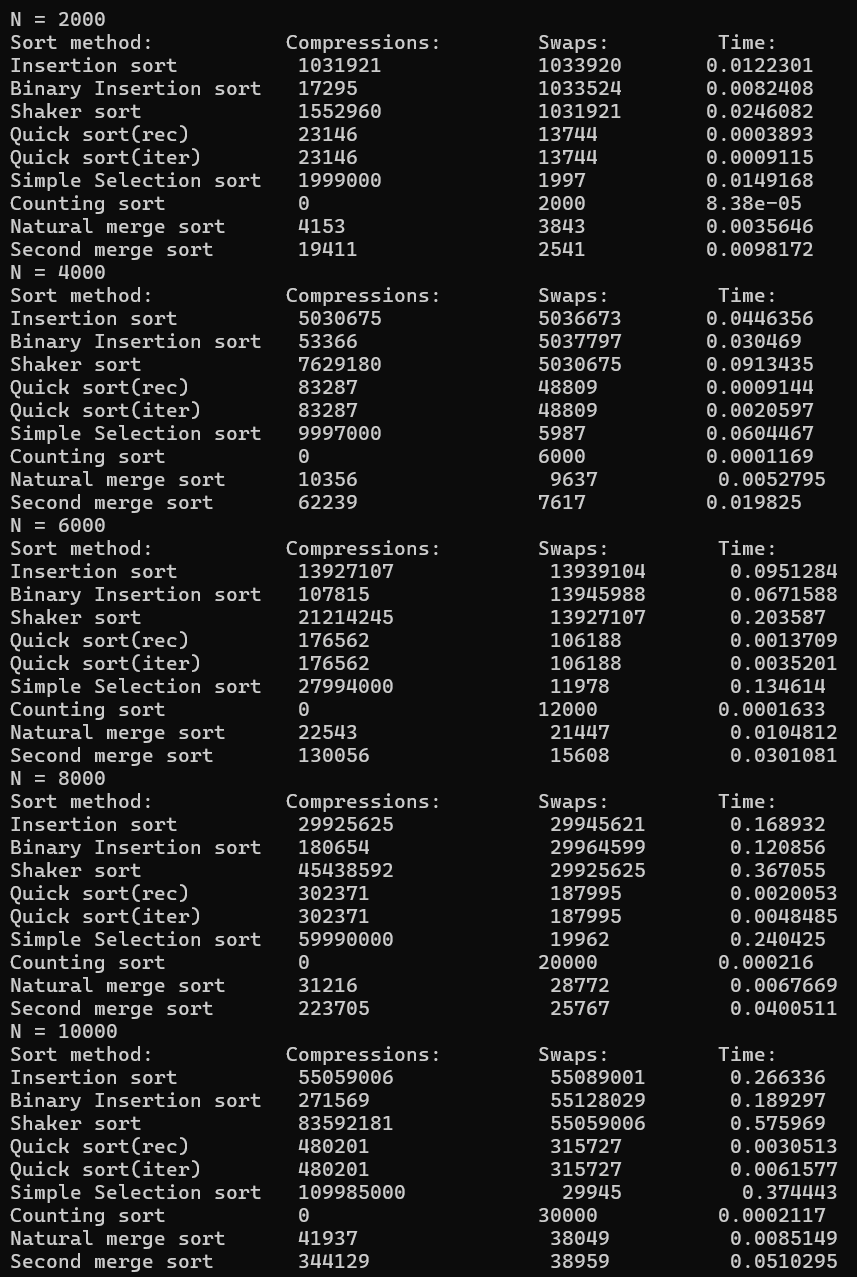
right = mergeSort(right, comparisons, swaps);

// Слияние отсортированных подсписков

return merge\_2(left, right, comparisons, swaps);

}

Скриншоты результатов выполнения программы:



Вывод: Изучил основные методы сортировки, получил практические навыки программирования задач сортировки, получил навыки экспериментальных исследований алгоритмов.