**ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА» (СПбГУТ)**

**Информационных технологий и программной инженерии (ИТПИ)**

**КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ (ПИИВТ)**

# Лабораторная работа №1.

«Алгоритмы сортировки: сравнительный анализ сортировок» ДИСЦИПЛИНА: «Алгоритмы и Структуры Данных»

Выполнил Студент группы ИКПИ-34

Кураш Д.И.

2025 г.

# Постановка задачи

Целью работы является ознакомление с алгоритмами сортировки линейных и нелинейных структур и методикой оценки эффективности алгоритмов, сравнение алгоритмов по времени, затрачиваемому на их выполнение.

# Описание программы

Для решения поставленной задачи были реализованы два алгоритма сортировки: шейкерная сортировка (функция ShakerSort) и быстрая сортировка (функция quickSort). Затем эти алгоритмы были запущены 10 000 раз на массивах случайных чисел в диапазоне от 0 до 10 000, которые генерировались с помощью функции getRandomNumber. После измерения времени выполнения 10 000 сортировок с помощью функции Sorts были построены графики, отражающие зависимость суммарного времени работы алгоритмов от размера входных данных. Программа была разработана на языке C++ в среде разработки Microsoft Visual Studio.

# Алгоритмы сортировки

# Шейкерная сортировка

Перестановка элементов в шейкерной сортировке выполняется аналогично той же в пузырьковой сортировке, т. е. два соседних элемента, при необходимости, меняются местами. Пусть массив требуется упорядочить по возрастанию. Обозначим каждый пройденный путь от начала до конца последовательности через Wi, где i – номер пути; а обратный путь (от конца к началу) через —Wj, где j – номер пути.

Тогда после выполнения Wi, один из неустановленных элементов будет помещен в позицию справа, как наибольший из еще неотсортированных элементов, а после выполнения —Wj, наименьший из неотсортированных, переместиться в некоторую позицию слева. Так, например, после выполнения W1 в конце массива окажется элемент, имеющий наибольшее значение, а после —W1 в начало отправиться элемент с наименьшим значением.

Данный

Худшее время: *O(n2)*

Среднее время: *O(n2)*

Лучшее время: *O(n)*

**Реализация на С++:**

void ShakerSort(vector<int> arr)

{

int size = arr.size();

int leftMark = 1;

int rightMark = size - 1;

while (leftMark <= rightMark)

{

for (int i = rightMark; i >= leftMark; i--)

if (arr[i - 1] > arr[i]) swap(arr[i], arr[i-1]);

leftMark++;

for (int i = leftMark; i <= rightMark; i++)

if (arr[i - 1] > arr[i]) swap(arr[i], arr[i - 1]);

rightMark--;

}

}

**Быстрая сортировка**

Быстрая сортировка 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).

Худшее время: *O(n2)*

Среднее время: *O(n\*log n )*

Лучшее время: *O(n\*log n )*

Затраты памяти: *O(n)*

**Реализация на С++:**

vector<int> quickSort (vector<int> &*arr*) {

if (arr.size() < 2) return arr;

int base = arr[0];

vector<int> left, mid, right;

for (int i = 0; i < arr.size(); i++){

if (arr[i] < base) left.push\_back(arr[i]);

else if (arr[i] > base) right.push\_back(arr[i]); else mid.push\_back(arr[i]);

}

left = quickSort(left);

right=quickSort(right);

left.insert(left.end(), mid.begin(), mid.end()); left.insert(left.end(), right.begin(), right.end());

return left;

}

**Сравнение работы алгоритмов**

Разберем работу обоих алгоритмов на наборах входных данных: Пусть нужно отсортировать массив {21,25,9,17}

**Шейкерная сортировка:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 21 | 25 | 9 | 17 |  |
| 21 | 9 | 25 | 17 |  |
| 21 | 9 | 17 | 25 | R3 |
| 9 | 21 | 17 | 25 | L1 |
| 9 | 17 | 21 | 25 | R2 |
| 9 | 17 | 21 | 25 | L2 |

*Рис. 1 – Пример работы алгоритма шейкерной сортировки*

Шаг 1: Инициализация границ

* **Левая граница (Left)**: 0
* **Правая граница (Right)**: 4 (поскольку индексация начинается с 0)

Шаг 2: Первый проход (слева направо)

* Обходим массив от **Left** до **Right**.
* Если текущий элемент больше следующего, то меняем их местами.
* **Флаг обмена (SwapFlag)**: Если обменов не было, то массив уже отсортирован.

Шаг 3: Второй проход (справа налево)

* Обходим массив от **Right** до **Left**.
* Если текущий элемент меньше предыдущего, то меняем их местами.
* **Флаг обмена (SwapFlag)**: Если обменов не было, то массив уже отсортирован.

Шаг 4: Уменьшение границ

* Уменьшаем **Right** на 1, так как максимальный элемент уже на своём месте.
* Увеличиваем **Left** на 1, так как минимальный элемент уже на своём месте.

**Новые границы:**

* **Left**: 0
* **Right**: 3

Шаг 5: Повторение проходов

* Повторяем шаги 2 и 3 до тех пор, пока **Left** не станет больше или равно **Right**.

**Продолжение сортировки:**

Шаг 6: Завершение сортировки

* Если **Left** больше или равно **Right**, то массив отсортирован.

**Окончательный результат:** (9, 17, 21, 25)

**Быстрая сортировка:**

Жёлтым будем отмечать опорный элемент, относительно которого идет разделение.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | Опорный |  |
| 21 | 25 | 9 | 17 |  |
| 9 | 25 | 17 | 21 |  |
| 9 | 17 | 25 | 21 |  |
|  | Опорный |  |  | Опорный |
| 9 | 17 |  | 25 | 21 |
|  |  |  | 21 | 25 |
|  |  |  |  |  |
| 9 | 17 | 21 | 25 |  |

*Рис. 2 – Пример работы алгоритма быстрой сортировки*

*Шаг 1: Выбор опорного элемента*

* *Выберем первый элемент массива в качестве опорного: 17.*

*Шаг 2: Разделение массива*

* *Разделим массив на три части:*
  + *Элементы меньше опорного:****(9)***
  + *Элементы равные опорному:****(-)***
  + *Элементы больше опорного:****(25, 21)***

*Шаг 3: Рекурсивная сортировка*

* *Рекурсивно применяем алгоритм быстрой сортировки к частям массива, которые меньше и больше опорного элемента.*

***Рекурсия для части меньше опорного:***

* *Выберем опорный элемент для части****(9)****:****9****.*
* *Разделим на три части:*
  + *Меньше****9****:****(-)***
  + *Равно****9****:****(9)***
  + *Больше****9****:****(-)***
* *Рекурсивно сортируем части:*
  + *Для****(21,25)****: выберем опорный****21****.*

***Рекурсия для части больше опорного:***

* *Выберем опорный элемент для части****(25, 21)****:21.*
* *Разделим на три части:*
  + *Меньше****21****:****(-)***
  + *Равно****21****:****(21)***
  + *Больше****21****:****(25)****.*

*Шаг 4: Объединение отсортированных частей*

* *Объединяем отсортированные части:*
  + *Меньше опорного:****(9)***
  + *Равно опорному:****(17)***
  + *Больше опорного:****(21, 25)***
* *Объединённый массив:****(9, 17, 21, 25)***

**Сравнение работы алгоритмов:**

В результате работы обоих алгоритмов получили отсортированный массив. Алгоритм шейкерной сортировки массива длинной 4 элементов содержит в себе 4 шага, в то время как быстрая сортировка массива длинной 4 элементов содержит всего 5 шагов. Исходя из полученных данных, можем сказать, что на маленьком наборе данных сортировки по скорости почти не уступают друг другу, в то время как на массивах больших данных быстрая сортировка показывает себя куда лучше шейкерной сортировки.

# Результаты работы

Замеры производительности алгоритмов проводились на значениях массива в диапазоне от 0 до 10 000. Каждый алгоритм запускался 10 000 раз на массивах разной длины: 10, 100, 200 и так далее до 600. В результате измерений были получены времена работы каждого алгоритма в секундах для массивов длины N. На основе этих данных был построен сглаженный график, отражающий зависимость времени работы алгоритмов от размера входных данных.

*Рис. 3 – График зависимости времени работы алгоритма от длины массива*

По графику можно заметить, что шейкерная сортировка имеет квадратичную зависимость. Алгоритм быстрой сортировки работает заметно быстрее при большой длине массива. Тем не менее при достаточно маленьких входных данных можно заметить можно заметить, что шейкерная сортировка отрабатывает быстрее.

# Выводы по работе

В ходе проведенных исследований были получены следующие результаты:

1. Изучены, описаны и реализованы алгоритмы сортировки пузырьком и быстрой сортировки в среде разработки Microsoft Visual Studio.
2. Проведены замеры длительности работы алгоритмов на разных массивах входных данных, различной длины.
3. Построены графики зависимости времени работы алгоритма от длинны входного массива, по которым не трудно заметить, что быстрая сортировка работает намного быстрее шейкерной.

**Приложения**

**Реализованная программа на c++**

#include <iostream>

#include <vector>

#include <ctime>

using namespace std;

#define N 10000

// Сортировка вставкой

vector<int> insertionSort(vector<int> nums) {

for (int i = 1; i < nums.size(); i++) {

int k = i;

while (k > 0 && nums[k - 1] > nums[k]) {

swap(nums[k - 1], nums[k]);

k--;

}

}

return nums;

}

// Быстрая сортировка

vector<int> quickSort(vector<int> nums) {

if (nums.size() < 2) return nums;

int base = nums[0];

vector<int> left, mid, right;

for (int i = 0; i < nums.size(); i++) {

if (nums[i] < base) left.push\_back(nums[i]);

else if (nums[i] > base) right.push\_back(nums[i]);

else mid.push\_back(nums[i]);

}

left = quickSort(left);

right = quickSort(right);

left.insert(left.end(), mid.begin(), mid.end());

left.insert(left.end(), right.begin(), right.end());

return left;

}

// Функция возвращающая случайное целое число в диапазоне [min, max]

int getRandomNumber(int min = 0, int max = 10000) {

return rand() % (max - min + 1) + min;

}

//Шейкерная сортировка

void ShakerSort(vector<int> arr)

{

int size = arr.size();

int leftMark = 1;

int rightMark = size - 1;

while (leftMark <= rightMark)

{

for (int i = rightMark; i >= leftMark; i--)

if (arr[i - 1] > arr[i]) swap(arr[i], arr[i-1]);

leftMark++;

for (int i = leftMark; i <= rightMark; i++)

if (arr[i - 1] > arr[i]) swap(arr[i], arr[i - 1]);

rightMark--;

}

}

// Функция запускающая тест при некотором n (количестве элементов) для сортировок

void Sorts(int n) {

cout << "Sort for n = " << n << "\n";

// Заполняем массив

vector<int> ar;

for (int i = 0; i < n; i++) ar.push\_back(getRandomNumber());

//Результаты шейкерной сортировки

cout << "Shaker sort\n";

double t1 = clock(); // <- Функция clock() возвращает текущее процессорное время в тиках (количество тактов процессора с момента запуска программы)

for (int i = 0; i < N; i++) ShakerSort(ar);

double t2 = clock();

cout << N << " sort for " << (t2 - t1) / CLOCKS\_PER\_SEC << endl;

cout << "Mediana time: " << (t2 - t1) / CLOCKS\_PER\_SEC / N << endl; // <- Среднее время выполнения одной сортирвки

//Результаты для быстрой сортивроки

cout << "Quick sort\n";

t1 = clock();

for (int i = 0; i < N; i++) quickSort(ar);

t2 = clock();

cout << N << " sort for " << (t2 - t1) / CLOCKS\_PER\_SEC << endl;

cout << "Mediana time: " << (t2 - t1) / CLOCKS\_PER\_SEC / N << endl;

}

int main() {

// Счетчик для генерации случайных чисел

srand(time(0));

// Настраиваем вывод чисел с плавающей точкой

cout << fixed; // Устанавливаем фиксированный формат

cout.precision(3); // Устанавливаем точность до 3 знаков после точки

// Запускаем тесты для сортировок на n = 10, 100, 200, ..., 1000

Sorts(10);

cout << "\n" << endl;

for (int n = 100; n <= 1000; n += 100) {

Sorts(n);

cout << "\n" << endl;

}

return 0;

}