ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 5

«Реализация алгоритмов сортировки»

Выполнил работу:

Мавров Артём

Академическая группа №J3112

Принято:

Ассистент (квалификационная категория "ассистент"), Дунаев Максим

Санкт-Петербург

2024

Введение

Цель лабораторной работы — на практике изучить алгоритмы сортировок.

Задачи – изучить алгоритмы сортировок, реализовать их, посчитать асимптотику и используемую оперативную память алгоритмов, протестировать и наглядно изобразить результаты.

**Теоретическая подготовка:**

Для реализации этой лабораторной работы потребовалось:

* Базовые знания языка C++
* Умение оптимизировать работу алгоритма и расход памяти

**Реализация:**

* Реализация Pancake sort
* Реализация Odd-even sort
* Реализация Timsort
* Реализация тестирующей системы

Экспериментальная часть

**Testing**

Для тестирования и построения графиков работы всех алгоритмов сортировки была разработана программа (приложение 1). В ней представлены функции для генерации лучшего, среднего и худшего случая для всех сортировок, запуска тестов необходимое количество раз и необходимого объёма, а также с определённым шагом.

#include <iostream>

#include <vector>

#include <string>

#include <random>

#include <fstream>

#include <chrono>

#include <algorithm>

#include <windows.h>

#include "lab\_5\_sort\_1.cpp"

#include "lab\_5\_sort\_2.cpp"

#include "lab\_5\_sort\_3.cpp"

using namespace std;

string buffer;

void generate\_middle\_test(int& massive\_size, int& number){

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dist(numeric\_limits<int>::min(), numeric\_limits<int>::max());

ofstream test;

string file\_way = buffer + "/tests/test\_" + to\_string(number) + ".txt";

test.open(file\_way);

test << massive\_size << endl;

for (int i = 0; i < massive\_size; i++) {

test << dist(gen) << endl;

}

test.close();

}

void generate\_best\_test(int& massive\_size, int& number){

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dist(numeric\_limits<int>::min(), numeric\_limits<int>::max());

ofstream test;

string file\_way = buffer + "/tests/test\_" + to\_string(number) + ".txt";

test.open(file\_way);

test << massive\_size << endl;

vector<int> vec(massive\_size);

for (int i = 0; i < massive\_size; i++) {

vec[i] = dist(gen);

}

sort(vec.begin(), vec.end());

for (int i = 0; i < massive\_size; i++) {

test << vec[i] << endl;

}

test.close();

}

void generate\_worst\_test(int& massive\_size, int& number){

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<int> dist(numeric\_limits<int>::min(), numeric\_limits<int>::max());

ofstream test;

string file\_way = buffer + "/tests/test\_" + to\_string(number) + ".txt";

test.open(file\_way);

test << massive\_size << endl;

vector<int> vec(massive\_size);

for (int i = 0; i < massive\_size; i++) {

vec[i] = dist(gen);

}

sort(vec.rbegin(), vec.rend());

for (int i = 0; i < massive\_size; i++) {

test << vec[i] << endl;

}

test.close();

}

void run\_test(int& number) {

string file\_way = buffer + "/tests/test\_" + to\_string(number) + ".txt";

ifstream test;

test.open(file\_way);

int vec\_size;

test >> vec\_size;

vector<int> vec(vec\_size);

for (int j = 0; j < vec\_size; j++) {

test >> vec[j];

}

test.close();

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

odd\_even\_sort(vec);

auto finish = chrono::high\_resolution\_clock::now();

auto duration = chrono::duration\_cast<chrono::microseconds>(finish - start);

double final\_time = duration.count() / 1e6;

vector<int> check\_vec(vec.size());

check\_vec = vec;

sort(vec.begin(), vec.end());

if (check\_vec != vec) {

cout << "Wrong answer" << endl;

return;

}

cout << final\_time << endl;

}

void testing\_with\_step() {

for (int massive\_size = 1e4, number = 1; massive\_size < 1e6; massive\_size += 1e4, number++) {

generate\_middle\_test(massive\_size, number);

run\_test(number);

}

}

void testing() {

cout << "Quantity of tests: ";

int quantity;

cin >> quantity;

cout << endl << "Massive size: ";

int massive\_size;

cin >> massive\_size;

cout << endl;

for (int number = 1; number <= quantity; number++) {

generate\_middle\_test(massive\_size, number);

run\_test(number);

}

}

int main() {

char temp\_buffer[MAX\_PATH];

GetCurrentDirectory(MAX\_PATH, temp\_buffer);

buffer = string(temp\_buffer);

testing();

}

(приложение 1: Тестирующая программа)

**Pancake sort**

**Реализация, подсчёт асимптотики:**

#include <iostream>

#include <vector>

#include <algorithm> //для reverse

using namespace std;

int find\_max(const vector<int>& input\_vec, const int& finish) { //O(N)

int ind\_max = 0;

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

if (input\_vec[i] > input\_vec[ind\_max]){

ind\_max = i;

}

}

return ind\_max;

}

void pancake\_sort(vector<int>& input\_vec) { //O(N) в среднем

for (int i = input\_vec.size() - 1; i > 0; i--){

int ind\_max = find\_max(input\_vec, i);

if (ind\_max != i){

reverse(input\_vec.begin(), input\_vec.begin() + ind\_max + 1);

reverse(input\_vec.begin(), input\_vec.begin() + i + 1);

}

}

}

//Итого: O(N^2) в худшем, среднем и лучшем т.к. на каждом шаге алгоритм ищет максимальный элемент. Но стоит заметить, что алгоритм будет работать быстрее при уже отсортированном массиве, т.к. не потребуется производить перевороты.

(приложение 2: реализация Pancake sort)

**Подсчёт памяти:**

Алгоритм использует лишь незначительные переменные, занимающие очень малое количество памяти. Общая пространственная сложность O(1).

**Odd-even sort**

**Реализация, подсчёт асимптотики:**

#include <iostream>

#include <vector>

#include <utility> //для swap

using namespace std;

void odd\_even\_sort(vector<int>& input\_vec) {

bool sort\_flag = false;

while (!sort\_flag) { //O(N) в среднем

sort\_flag = true;

for (int i = 0; i < input\_vec.size() - 1; i += 2) { //O(N / 2)

if (input\_vec[i] > input\_vec[i + 1]) {

swap(input\_vec[i], input\_vec[i + 1]);

sort\_flag = false;

}

}

for (int i = 1; i < input\_vec.size() - 1; i += 2) { //O(N / 2)

if (input\_vec[i] > input\_vec[i + 1]) {

swap(input\_vec[i], input\_vec[i + 1]);

sort\_flag = false;

}

}

//Итого: O(N^2) в худшем (массив отсортирован в обратном порядке) и среднем (массив случайно распределён), O(N) в лучшем (массив уже отсортирован)

(приложение 3: реализация Odd-even sort)

**Подсчёт памяти:**  
Алгоритм использует лишь незначительные переменные, занимающие очень малое количество памяти. Общая пространственная сложность O(1).

**Timsort**

**Реализация:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <cmath>

using namespace std;

const int RUN = 32;

void insertion\_sort(vector<int>& arr, int left, int right) {

for (int i = left + 1; i <= right; i++) {

int key = arr[i];

int j = i - 1;

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

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

j--;

}

arr[j + 1] = key;

}

}

void merge(vector<int>& arr, int left, int mid, int right) {

int len1 = mid - left + 1, len2 = right - mid;

vector<int> left\_arr(len1);

vector<int> right\_arr(len2);

for (int i = 0; i < len1; i++) {

left\_arr[i] = arr[left + i];

}

for (int i = 0; i < len2; i++) {

right\_arr[i] = arr[mid + 1 + i];

}

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

while (i < len1 && j < len2) {

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

arr[k] = left\_arr[i];

i++;

} else {

arr[k] = right\_arr[j];

j++;

}

k++;

}

while (i < len1) {

arr[k] = left\_arr[i];

i++;

k++;

}

while (j < len2) {

arr[k] = right\_arr[j];

j++;

k++;

}

}

void timsort(vector<int>& arr) {

int n = arr.size();

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

insertion\_sort(arr, i, min((i + RUN - 1), (n - 1)));

}

for (int size = RUN; size < n; size = 2 \* size) {

for (int left = 0; left < n; left += 2 \* size) {

int mid = min((left + size - 1), (n - 1));

int right = min((left + 2 \* size - 1), (n - 1));

if (mid < right) {

merge(arr, left, mid, right);

}

}

}

}

(приложение 4: реализация Timsort)

**Подсчёт асимптотики:**

В лучшем случае, если массив уже отсортирован, алгоритм будет иметь сложность O(N). В среднем и худшем случае каждый рун сортируется за O(RUN^2) а общее время сортировки рун будет O(N), так как длина всех рун в сумме равна. Поскольку руны сливаются попарно, каждая итерация требует O(N). Число итераций (высота дерева слияния) — O(log\_(N/RUN))=O(log\_N).

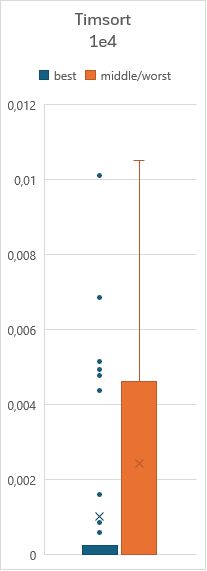
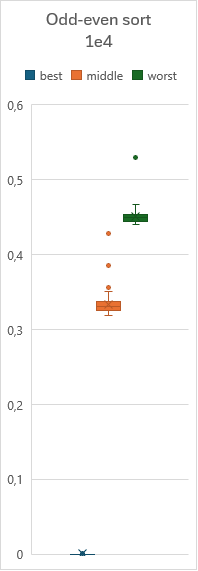
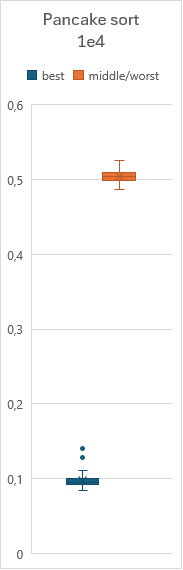
**Подсчёт пространственной сложности:**

Временные массивы для слияния требуют O(N) памяти, так как каждый элемент массива копируется в буфер. Также алгоритм отслеживает руны и их размер, что требует O(log\_N) памяти для хранения информации о слиянии. Таким образом пространственная сложность O(N + log\_N).

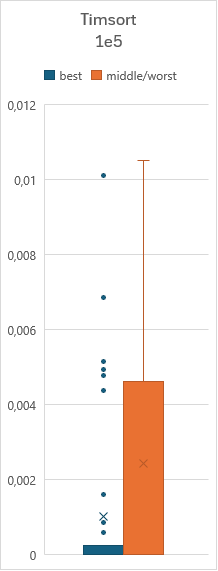
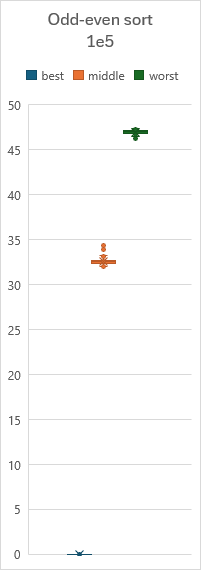
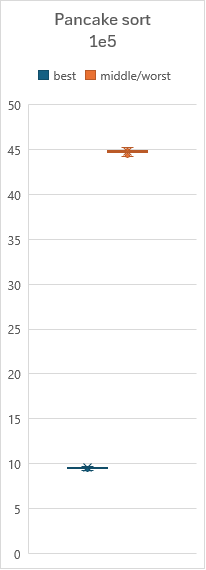
**Графики алгоритмов**

(приложение 5: Линейный график с 1000 до 100000 с шагом 1000)

\*В требованиях к отчёту были указаны ограничения с 1000 до 1000000 с шагом 1000, при этом должна быть реализована как минимум одна сортировка со сложностью O(N^2). Эти ограничения невыполнимы т.к. итоговая сложность >O(332833500000000), что будет выполняться крайне долго. В следствии этого было принято решение опустить нижнею границу до 100000, что вполне достаточно для формирования выводов.



(приложение 6: box plot 1e4)



(приложение 7: box plot 1e5)

**Вывод**

Ожидаемая асимптотика примерно сошлась с реальным временем. Выбросы на графиках незначительны и объяснимы не идеальностью оборудования, на котором производилось тестирование.

Pancake sort предсказуемо оказалась самой неэффективной. Заключается это в сложности алгоритма, поиск максимального элемента и переворачивание подмассива крайне неэффективен. На реальных задачах сортировка не имеет применения.

Odd-even sort долгий алгоритм который, но с малой пространственной сложностью. Может быть применим при малом и/или почти отсортированном массиве и необходимостью экономии памяти.

Timsort значительно быстрее других алгоритмов, но с большей пространственной сложностью. Является самым универсальным из представленных алгоритмов, подходит для больших массивов когда нет надобности экономить память.